AN ABSTRACT OF THE THESIS OF

Chandrasekhara K. Reddy for the degree of Doctor of Philosophy in Computer Science presented on June 30, 1998.

Title:

Learning Hierarchical Decomposition Rules for Planning:
An Inductive Logic Programming Approach.

Abstract approved: ______________________________

Prasad Tadepalli

Artificial Intelligence (AI) planning techniques have been central to automating a gamut of tasks from the mundane route planning and beer production to the ethereal image processing of space-ship images. Of all the planning techniques, hierarchical-decomposition planning has been the technique most employed in industrial-strength planners. Hierarchical-decomposition planning is performed by recursively decomposing a planning task into its subtasks, until the decomposition results in primitive tasks which can be directly achieved by executing the primitive actions.

Hierarchical-decomposition planning is knowledge intensive; it exploits knowledge of the structure and the constraints of a planning domain, to decompose a task into subtasks. Because dependence on human experts for this knowledge leads to knowledge-acquisition bottleneck, machine learning of this domain-specific knowledge becomes important. There exist two opportunities for learning in the context of hierarchical-decomposition planning. One is to learn how a planning task decomposes into subtasks. The other is to learn control knowledge to choose among various decompositions for a task, depending upon situations. In this dissertation,

the focus is on the former; more specifically, we focus on learning rules for task or goal decompositions.

Goal-decomposition rules (d-rules) decompose goals into a sequence of subgoals under certain conditions. These are a special case of hierarchical task networks (HTNs). The methodology we used for learning d-rules is to map d-rules to Horn clauses, and, thus, transform the problem of learning d-rules to learning Horn clauses. We developed provably correct algorithms for learning Horn clauses. Our algorithms are based on a "generalize-and-test" method, where inductive least-general generalization of positive examples is followed by pruning of irrelevant literals by asking queries or performing self-testing. We implemented systems that are founded in the theoretical algorithms, and tested the applicability of the systems in two planning domains—a robot navigation domain and an air-traffic control domain. One of these systems, ExEL, learned from solved problems and expert-answered queries. The other, LeXer, learned from unsolved but ordered problems, or exercises, and self-testing. The applicability of the theoretical algorithms developed for learning Horn clauses, however, transcends the learning of d-rules and even the learning of the more general HTNs.

©Copyright by Chandrasekhara K. Reddy

June 30, 1998

All rights reserved

Learning Hierarchical Decomposition Rules for Planning: An Inductive Logic Programming Approach

by

Chandrasekhara K. Reddy

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Completed June 30, 1998
Commencement June 1999

Doctor of Philosophy thesis of Chandrasekhara K. Reddy presented on June 30, 1998

APPROVED:

Major Professor, representing Computer Science

Head of Department of Computer Science

Dean of Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Chandrasekhara K. Reddy, Author

Approved by Committee:

Major Professor (Prasad Tadepalli)

Committee Member (Bella Bose)

Committee Member (Paul Cull)

Committee Member (Thomas Dietterich)

Graduate School Representative (Harold Parks)

Date thesis presented June 30, 1998

ACKNOWLEDGMENT

I used to wonder why people thank almost the whole world in their dissertations. Now I know why. The arduous task of completing a dissertation takes superhuman abilities to accomplish by oneself. I have benefitted from the generous help from scores of individuals. I am indebted to them all. I will take this opportunity to record my gratitude to a significant few. At the outset, I must admit that, in most cases, I will not be able to adequately express how I really feel.

My profound gratitude is to my guru, Prasad Tadepalli. He was the veritable fount of inspiration, insight, encouragement and advice throughout my Ph.D. studies. Prasad is the best listener I know. On the umpteen occasions I went to him with half-baked ideas, he listened to me with remarkable patience, however long it took, and helped me clarify the ideas. Prasad always expressed confidence in me, at times more confidence than I had on myself, and gave me enough rope to explore my own directions on my research. On occasions when I came back to him all tangled up, he was extremely generous with his time, ideas and insight. Thank you Prasad for these and for innumerable other things which were instrumental for the completion of this thesis!

I am grateful to Tom Dietterich. Prof. Dietterich was a great source of inspiration to me. I learned a great deal about machine learning from his encyclopaedic knowledge. Thank you Prof. Dietterich! Many thanks to Bella Bose. Prof. Bose is one of the kindest persons I know. Throughout my stay in OSU, he showed genuine interest in how things are going on in my life. I also thank Paul Cull. Prof. Cull, I enjoyed your theory classes immensely! Your sense of interest in all knowledge has indelibly rubbed off on me.

I thank Prasad, Profs. Bose, Cull and Dietterich for being eminent models to emulate in my life. I thank them and Prof. Parks for being on my dissertation committee.

I thank the members of the machine learning reading group, past and present, for being comrades in arms in making sense of the literature on machine learning, and helping me keep abreast with the developments in the areas outside the narrow area of my research.

I thank Roni Khardon for the discussions on the conference-paper versions of Chapters 4 and 5.

I thank Bernie and the other staff, past and present, of the computer science department for all their help throughout my stay at OSU.

I thank my friends Sriram, Vani, Maitreyee, Siva, Narasimha, Giri and Krishna in helping me go about my goal of attaining Ph.D. I thank the Satsang group, Prasad and Padmaja, Murali and Brinda for discussions on larger issues in life which help one have a global perspective. I thank Padmaja and Sraavya for graciously allowing me to steal their time with Prasad.

My deepest gratitude is to my parents and uncles. Their confidence in me and their hopes and sacrifices for me are the main reasons for my coming this far in life.

Last but not the least, I thank the love of my life, my wife Sivalakshmi, for all the demands (mostly unreasonable ones) I made of her. I thank her for forgiving (actually, forgetting) the countless times I kept unreasonable hours, the countless times when I went back on my promises because I had to be with "my other wife", the computer. Sivam, I can promise to you now, with reasonable confidence, that we have our whole lives ahead to redress this remiss!

TABLE OF CONTENTS

Page

Chapter 1: Introduction **1**

1.1 Planning 1

1.2 Need for Learning 3

1.3 The Learning Problem Addressed 4

1.4 Organization of the Dissertation 7

Chapter 2: Goal-Decomposition Rules and Planning **9**

2.1 Representation of Goal-Decomposition Rules 9

2.2 Planning With D-rules 12

2.2.1 Soundness and Completeness of the D-rule Planner 13

2.2.2 Other Representations Versus D-rules 16

2.3 Reactive Planning with D-rules 17

2.4 Hierarchical Task Networks (HTNs) and D-rules 19

Chapter 3: Mapping D-rules into Horn Clauses **22**

3.1 Preliminaries 22

3.2 D-rules to Horn Clauses 23

3.3 HTNs to Horn Clauses 27

Chapter 4: Learning Horn Definitions **31**

4.1 Introduction 31

4.2 Preliminaries 34

4.3 Learning Horn Definitions 36

4.3.1 Learning Problem 36

4.3.2 The Learning Algorithm 37

4.3.2.1 Strong Compactness of Non-recursive Horn Definitions 40

4.3.2.2 Proof of Learnability 42

TABLE OF CONTENTS (Continued)

Page

4.4 Learning D-rules via Learning Horn Definitions 46

4.5 Discussion 49

Chapter 5: Learning Acyclic Horn Programs **52**

5.1 Introduction 52

5.2 Preliminaries 54

5.3 Learning Horn Programs 56

5.3.1 The Learning Model 56

5.3.2 The Learning Algorithm 57

5.3.3 An Example 58

5.3.4 Learnability of AH_k 61

5.4 Discussion and Conclusions 71

Chapter 6: Learning D-Rules from Examples **74**

6.1 Introduction 74

6.2 Learning System 77

6.2.1 ExEL's Empirical Generalization Process 79

6.2.2 Guiding Learning with Domain Theory 81

6.2.2.1 Explanation-Based Pruning 81

6.2.2.2 Abstraction by Forward Chaining 82

6.2.3 Learning Subgoals 83

6.2.4 Summary of the Learning Algorithm 86

6.3 Experimental Results 88

6.3.1 Blocks World (BW) 88

6.3.2 STRIPS World (SW) 89

6.3.3 Air-Traffic Control (ATC) domain 90

6.4 Discussion 92

TABLE OF CONTENTS (Continued)

Page

Chapter 7: Learning D-Rules from Exercises **96**

7.1 Introduction 96

7.2 Learning from Exercises 98

7.2.1 Solving Exercises 100

7.2.2 Example Preparation 103

7.2.3 Generalization of d-rules 104

7.3 Experimental Results 106

7.3.1 STRIPS World (SW) 107

7.3.2 Air-Traffic Control Domain 108

7.4 Discussion 110

Chapter 8: Conclusions **115**

8.1 Contributions 117

8.2 Future Work 120

Bibliography **123**

LIST OF FIGURES

Figure		Page
1.1	An example d-rule in blocks world	4
1.2	A blocks-world problem and its solution	6
2.1	A d-rule for `on` in the Blocks World	10
2.2	A d-rule for `clear` in the Blocks World	10
4.1	DLearn: An algorithm to learn Horn definitions	39
5.1	A derivation of $mother(a, b)$, $mother(b, c) \rightarrow grandParent(a, c)$ from H.	55
5.2	PLearn Algorithm	59
6.1	Generalize-and-query algorithm for generalizing d-rules, given an example d-rule	80
6.2	Order-Subgoals procedure	85
6.3	Performance of ExEL in the SW domain	90
6.4	Performance of ExEL in the ATC domain	91
7.1	Algorithm for learning from exercises	99
7.2	DFS-based exercise solver	101
7.3	Exercise solver achieving the goal `land` using IDS with the current depth limit 2	103
7.4	Generalize-and-test algorithm for generalizing d-rules, given an example d-rule	105
7.5	Performance of LeXer in SW	108
7.6	Performance of LeXer in ATC	109

DEDICATION

To the loving memory of my *je'ji* (grandmother), who wanted me to become a doctor, for all her sacrifices and unconditional support! Given my dislike for dissecting animals, this is the best I could do, *je'ji*!

LEARNING HIERARCHICAL DECOMPOSITION RULES FOR PLANNING: AN INDUCTIVE LOGIC PROGRAMMING APPROACH

Chapter 1

INTRODUCTION

Planning is the process of finding a course of action to achieve a goal from a given situation. Planning is an important activity in our day-to-day lives. Because of its ubiquity and importance in the real world, the study of planning in Artificial Intelligence (AI), too, has been one of the oldest and most important subfields of AI. Planning in AI (henceforth, just "planning") involves, given an initial situation or state, a goal, and a set of operators, each of which is described in some language, finding a sequence of instantiations of operators (or actions) that takes the initial state into a state where the goal is true. An initial state together with a goal constitutes a planning problem.

1.1 Planning

The classical work on planning can be broadly categorized into three areas: (1) State-space planning, (2) Plan-space planning, and (3) Task-reduction or Hierarchical-decomposition planning. State-space planning can be characterized as searching in the space of states. Each node in a state space is a state describing a situation. Each edge is an action connecting the state in which the action takes place to the state

that results as a consequence of that action. A solution plan for a planning problem is, then, a path in the state space of the planner from the initial state to a state where the goal of the problem is satisfied. On the other hand, plan-space planning is a search through the space of plans. Each node is a plan. Each edge connects a plan to a refined version of the plan. Refinements could be such as adding or inserting a new action, ordering existing actions, or other ways of specializing a plan. Here the ordering of the actions need not be total. In a plan with partially ordered actions, two actions that are not relatively ordered can be executed in any order to attain the goal. Refinement planning is a framework that captures these two methods and variations on them (Kambhampati, Knoblock, & Yang, 1995; Kambhampati, 1997). Refinement planning aims to capture some aspects of hierarchical-decomposition planning as well.

In hierarchical-decomposition planning, planning is done by recursively decomposing planning tasks into subtasks, until decomposition results in primitive tasks which can be directly achieved (Russell & Norvig, 1995). Hierarchical-decomposition planning has been the most prevalent technique used in classical "industrial-strength" planners. SIPE (Wilkins, 1988), O-Plan (Currie & Tate, 1991) and their successor systems, and VICAR (Chien, Estlin, & Wang, 1997) are prominent examples of real-world planners. The main reason for the success of hierarchical-decomposition planning is its ease in capturing and effectiveness in utilizing domain-specific control knowledge. Although hierarchical-decomposition planning has been around for a while (NOAH (Sacerdoti, 1977), HACKER (Sussman, 1975), NONLIN (Tate, 1977)), many of the powerful ideas in this technique have only been articulated, formalized, and given a standard name only much later (Erol, 1995). This formalization is called Hierarchical Task Network (HTN) planning.

Recently, two other methods—Graphplan and SATplan—have been developed. These methods use techniques that are distinctly different from the classical methods. Graphplan (Blum & Furst, 1997) plans for a planning problem by constructing a planning graph that captures the constraints inherent in the problem, and then analyzing the graph. SATplan (Kautz & Selman, 1996) plans by converting a planning problem into a propositional satisfiability (SAT) problem and then exploiting fast satisfiability algorithms, such as GSAT, to solve the SAT problem. Planners based on these two methods have been shown to outperform the planners based on state-space and plan-space planning. However, they have not been shown to capture and utilize domain-specific control knowledge as effectively as HTNs. Recently, there have been preliminary efforts to combine representation power of HTNs with the computation power of SATplan, by converting HTNs into propositional representations (Mali & Kambhampati, 1998).

1.2 Need for Learning

Domain-independent planning has been known to be computationally hard (Erol, Nau, & Subrahmanian, 1995). Therefore, for planners to be practical, they need to utilize domain-specific knowledge. One source of this knowledge is human experts. Dependence on human experts, however, is not always feasible for reasons such as (1) high difficulty for experts to articulate their knowledge to be usable by planners; (2) inordinate cost factors in acquiring knowledge from experts; and (3) unavailability of experts for certain planning domains. This infeasibility has been summarized as the "knowledge-acquisition bottleneck." A more practical alternative is to employ machine learning. Acquiring domain-specific knowledge by machine learning has been a popular method in the context of state-space or partial-order planning (Minton, 1988; Kambhampati, Katukam, & Qu, 1996; Estlin, 1998). The problem of

goal: `on(?x, ?y)`

subgoals: `<clear(?x), clear(?y), put-on(?x, ?z, ?y)>`

conditions: `block(?x), block(?y), table(?z), on(?x, ?z)`

FIGURE 1.1: An example d-rule in blocks world

the knowledge-acquisition bottleneck is exacerbated for hierarchical-decomposition planning, because of its dependence on domain-specific knowledge for goal decomposition. However, there have hardly been any learning systems targeted for hierarchical planning.

There exist two opportunities for learning in the context of hierarchical-decomposition planning. One is to learn rules for decomposing planning task into its subtasks. The other is to learn control knowledge to choose among various decompositions for a task, depending upon situations. In this dissertation, the focus is on the former.

1.3 The Learning Problem Addressed

This dissertation focuses, more specifically, on learning rules for goal decompositions.

Goal-decomposition rules (d-rules) are a special case of HTNs. D-rules decompose goals into a sequence of subgoals under certain conditions. Unlike HTNs, d-rules do not allow partial orderings between subtasks or non-codesignation (inequality) constraints between variables. Consider the example of a d-rule in Figure 1.1 taken from a blocks-world domain. (In the following and in the rest of the thesis, we use symbols that start with a '?' to denote variables.) The example rule says that to achieve the goal of putting ?x on ?y (`on(?x, ?y)`), when ?x and ?y are blocks and ?z

is a table, clear block ?x (`clear(?x)`), clear block ?y (`clear(?y)`), and then put block ?x on block ?y (`put-on(?x, ?z, ?y)`). The subgoals may themselves have their own d-rules, unless they are primitive subgoals. A primtive subgoal is a primitive action of the domain. (It is also called a primitive operator.) In the example, `put-on(?x, ?z, ?y)` is a primitive action; so, it will not have a d-rule. Both `clear(?x)` and `clear(?y)` are subgoals, and they have their own d-rules. To achieve the subgoals, d-rules can be recursively applied until decompositions result in primitive subgoals.

Note that the subgoals `clear(?x)` and `clear(?y)` can, in fact, be achieved in either order as long as both are achieved before `put-on(?x, ?z, ?y)`. d-rules specify one strict order, and cannot specify the partial order. HTNs, on the other hand, can specify this partial order, by indicating only that `clear(?x)` and `clear(?y)` should be achieved before `put-on(?x, ?z, ?y)`, and mentioning nothing about the relative order of `clear(?x)` and `clear(?y)`. Using HTNs, we can also specify that the objects specified by the variables `?x, ?y` and `?z` have to be different.

For the example problem in Figure 1.2, a possible solution is the sequence of primitive actions ⟨`put-down(A, B, Table), put-down(B, C, Table), put-on(B, Table, A)`⟩.

The methodology used in learning d-rules is first mapping the problem to that of learning Horn clauses, and then developing algorithms for learning Horn clauses. Each of the algorithms for learning Horn clauses learns by a "generalize-and-test" method, where the algorithm generalizes to produce hypotheses, and each of the hypotheses is validated by asking a query of a teacher. We implemented two systems that are related to the algorithms for learning Horn clauses. The implementations use learning from successful problem-solving instances, as opposed to learning from failure. One system takes as input positive examples or pairs of planning problems and their solution sequences. The set of literals describing the initial state and

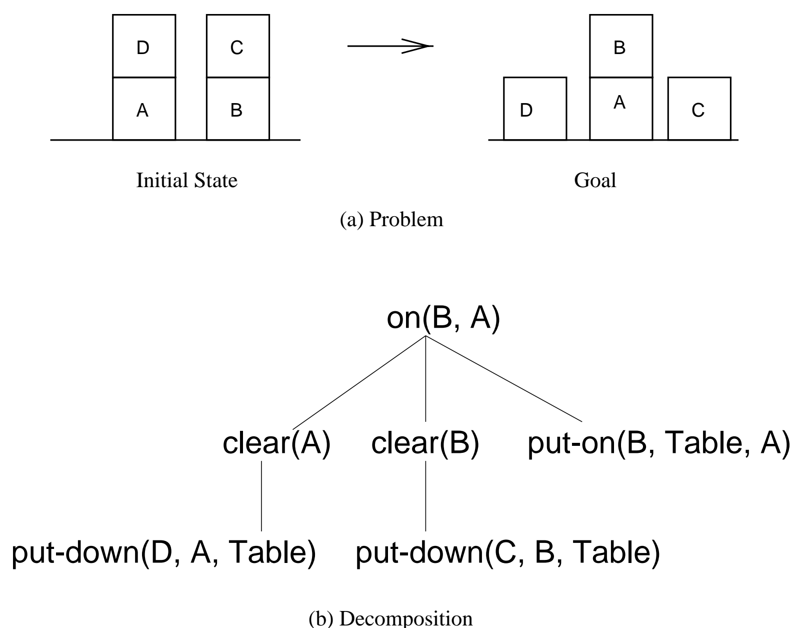

FIGURE 1.2: A blocks-world problem and its solution

the goal in Figure 1.2, paired with the sequence of primitive actions ⟨`put-down(A, B, Table)`, `put-down(B, C, Table)`, `put-on(B, Table, A)`⟩, for instance, is a positive example for a d-rule for the `on` goal. The other system learns from solving exercises or problems and subproblems supplied in the order of their difficulty. In this algorithm, the query-answering teacher is replaced by a self-testing process. In self-testing, the learner generates examples of a hypothesis d-rule it is working on and tests whether the decomposition suggested by the hypothesis d-rule works. This kind of examples can be called "near-miss" examples. They correspond to negative examples in other algorithms.

1.4 Organization of the Dissertation

The rest of the dissertation is organized as follows. Chapter 2 describes the representation of goal-decomposition rules and a planning architecture based on them. It also discusses how d-rules and the planning architecture can be extended to apply to dynamic domains (i.e., domain where the state of the world may change spontaneously or randomly).

Chapter 3 discusses the mapping between d-rules and Horn clauses. It also discusses how this mapping can be extended to HTNs. The idea is to apply algorithms for learning Horn clauses to learn d-rules and HTNs.

Chapter 4 formally describes the problem of learning Horn definitions. Further, it demonstrates the learnability of a special case of Horn definitions, namely non-recursive Horn definitions. A Horn definition is a set of Horn clauses, all of which have the same predicate symbol in their heads. A Horn definition is non-recursive if the head predicate symbol does not occur in the body of any Horn clause in the Horn definition. The result described in this chapter significantly extends the theoretical results in inductive logic programming.

Chapter 5 focuses on learning Horn programs. A Horn program is a set of Horn clauses, with possible interdependence between clauses—in the sense that the head predicate symbol of a clause may appear in the body of the same or a different clause. This chapter outlines the learning problem and presents learnability results for a restricted version of Horn programs, called acyclic Horn programs, that have polynomial-time forward-chaining procedures.

Chapter 6 details the implementation of learning d-rules from examples, in a system called ExEL. This implementation has foundations in the theoretical algorithm for learning Horn definitions. Chapter 6 also presents experimental results

of ExEL on the Blocks world,the STRIPS world, and air-traffic control domains. The experimental results suggest ExEL's effectiveness and applicability in learning d-rules.

Chapter 7 concerns with the implementation of learning d-rules from exercises, in a system called LeXer. In this chapter, we also explore connections between the algorithms for learning d-rules from exercises and learning Horn programs. Chapter 7 also presents experimental results of LeXer on the STRIPS world and air-traffic control domains. LeXer differs from ExEL in unburdening a teacher from having to supply solved problems and having to answer queries. LeXer solves the teacher-supplied exercises itself, and replaces queries by self-testing. The experiments suggest that LeXer can effectively lessen the burden of its teacher.

Chapter 8 presents conclusions, highlights the contributions of this dissertation, and identifies some future work.

Chapter 2

GOAL-DECOMPOSITION RULES AND PLANNING

In this chapter, we explain how hierarchical-decomposition planning can be done using goal-decomposition rules. First, we describe the syntax and the semantics of goal-decomposition rules and how they can be employed for planning. Next, we extend the representation of goal-decomposition rules to address the problem of reactivity that is needed to plan in dynamic domains—i.e., domains that have dynamic changes which are beyond those in the control of the planning agent. Finally, we show how goal-decomposition rules differ from Hierarchical Task Networks (HTNs).

2.1 Representation of Goal-Decomposition Rules

A decomposition rule (d-rule) is a 3-tuple $\langle g, c, sg \rangle$ that decomposes a goal g into a sequence of subgoals sg, provided that the condition c holds in the state where the d-rule is applied. Here, g is a single literal, c is a conjunction of literals, and sg is a sequence of literals with implicit left-to-right ordering. Literals are in function-free first-order calculus representation and are positive. Any variables appearing in the goal and the subgoal components of a d-rule must also appear in the condition component of the d-rule. The d-rules for the lowest-level subgoals each have a single observable action in the place of the subgoals.

Figure 2.1 is an example of a d-rule from the Blocks World for the goal `on`. This d-rule can be translated as follows: To achieve the goal of putting block `?x` on block `?y`, first achieve the subgoals that make the tops of the blocks `?x` and `?y` clear. Then

goal: on(?x, ?y)

subgoals: <clear(?x), clear(?y), put-on(?x, ?z, ?y)>

conditions: block(?x), block(?y), table(?z)

FIGURE 2.1: A d-rule for on in the Blocks World

goal: clear(?x)

subgoals: <clear(?y), put-down(?y, ?x, ?table)>

conditions: on(?y, ?x), block(?x), block(?y), table(?table)

FIGURE 2.2: A d-rule for clear in the Blocks World

achieve the primitive subgoal put-on to put the block ?x from on top of the table ?z to the top of the block ?y. Each of the non-primitive subgoals may have its own d-rule(s). For example, the subgoal clear(?x) for the goal on(?x, ?y) has the d-rule shown in Figure 2.2. The d-rule in Figure 2.2 says that to clear the top of the block ?x when block ?y is on block ?x, first clear the top of block ?y, and then put block ?y from the top of block ?x on to the table ?table.

The semantics of d-rules, more formally, with respect to planning is described in the following way. First, we need to mention what a primitive operator or lowest-level subgoal is. A primitive operator (or lowest-level subgoal) is a primitive STRIPS-style operator (Fikes, Hart, & Nilsson, 1972) with its standard semantics. Next, we define the notion of "complete decomposition" of a planning problem with respect to a set of d-rules.

Definition 2.1.1 *Let* $\langle S, G\rangle$ *be a planning problem, where* S *is the starting state and* G *is the goal of the planning problem. Given a set of d-rules* D *and a planning problem* $\langle S, G\rangle$*, a* **complete decomposition** *of* $\langle S, G\rangle$ *with respect to (w.r.t.)* D

- *if* G *is a lowest-level goal (or primitive operator), is* G*;*
- *otherwise, a sequence of complete decompositions of the planning problems* $\langle S_1, G_1\rangle$*,* $\langle S_2, G_2\rangle, \ldots, \langle S_n, G_n\rangle$ *w.r.t.* D*, where* $S_1 = S$*, and there is a d-rule* $\langle g, c, sg\rangle$ *in* D *and a substitution* θ *such that* $g\theta = G$*,* $c\theta \subseteq S$*, for* $i \in [1, n]$*,* $sg = \langle sg_1, \ldots, sg_n\rangle$*,* $G_i = sg_i\theta$*, and for* $j \in [2, n]$*,* S_j *is the state resulting after applying a complete decomposition of the subproblem* $\langle S_{j-1}, G_{j-1}\rangle$ *w.r.t.* D*.*

Figure 1.2 illustrates a problem in the Blocks World and its complete decomposition w.r.t. the d-rules in Figures 2.1 and 2.2.

The definition of complete decomposition gives an operational semantics of d-rules. To present declarative semantics of d-rules, we need to consider the state space formed by the start state S of a planning problem $\langle S, G\rangle$ and the primitive operators of a planning domain. A d-rule $\langle g, c, sg\rangle$ asserts that starting from any state S in which the condition c is satisfied, if each of the subgoals in sg is achieved one by one in sequence, then the goal g is true in the state that achieved the last subgoal in the sequence sg. We relate this to a path $S_0 = S, S_1, S_2, \ldots, S_t$ in the state space where S is the start state, and S_i, for $i \in [1, t]$, is the child of the state S_{i-1} in the state space. Then a d-rule says that for any path $S_0 = S, S_1, S_2, \ldots, S_t$ in the state space, if c is satisfied in the starting state S, and sg_i, where $i \in [1, n]$ and $sg = \langle sg_1, \ldots, sg_n\rangle$, is satisfied in a state S_j and sg_{i+1} is satisfied in S_k for $k \in [j, t]$, then g is satisfied in the state where sg_n is also satisfied after all other sg_i are satisfied.

In the next section, we will see how planning can be done employing d-rules. There we will also examine the soundness and the completeness of d-rules and a d-rule planner, by appealing to the above semantics.

2.2 Planning With D-rules

The d-rule-based planner for a domain takes as input a goal, a state from which the goal needs to be achieved, a set of d-rules, and the domain theory (comprising primitive-operator definitions, and background theory). The planner finds the d-rules that correspond to the goal, and picks a d-rule whose condition is satisfied in the current state. The chosen d-rule suggests a subgoal sequence. The suggested subgoals in the sequence are achieved one after the other in sequence. Each subgoal in turn becomes a goal, and has its own d-rules. So, the planner achieves goals in depth-first fashion, recursively, until the recursion bottoms out when a d-rule suggests only primitive operators as subgoals. Primitive operators are operators of the domain that are readily executable without recourse to a d-rule. The planner outputs the plan (the operator sequence) and the final state, in which the goal should be satisfied.

Since d-rules require their subgoal components to have only the variables in the condition and goal components, when the planner is trying to achieve a particular goal and the condition of a d-rule has been satisfied, the subgoals will all be ground. Moreover, the subgoals are ordered. This makes planning efficient. However, in general, there may be multiple d-rules for a goal, and their conditions may not be disjoint. This creates a choice for the planner. There are four design options here: (1) require that d-rules be disjoint; (2) if not, require that any d-rule choice for a goal achieve the goal; (3) allow backtracking over choice of d-rules; or (4) have control rules that know how to choose among different d-rules when there is a choice.

Allowing backtracking makes the planner inefficient. We shall assume that the set of d-rules the planner gets is "good"—i.e., any applicable d-rule achieves its goal. With good d-rules, planning is linear in the number of d-rule applications, and, hence, in the solution or plan length. The d-rule planner can be incomplete if the d-rules are not perfect; but it is always efficient.

Since efficiency is guaranteed by the planner, the issue of improving problem-solving translates to improving the coverage of the planner. An issue related to coverage is completeness of a planner. In the next section, we discuss the completeness of the d-rule–based planner, and also the validity (or soundness) of the plans it produces.

2.2.1 Soundness and Completeness of the D-rule Planner

First, we present the soundness of a set of d-rules and link it to the soundness of the D-rule planner described above. Then, we discuss the completeness of a set of d-rules and the D-rule planner.

We can make use of the declarative semantics of a d-rule with respect to a state space in defining soundness of a d-rule. We can define soundness of a d-rule as the following. A d-rule $\langle g, c, sg \rangle$ is sound if for any path $S_0 = S, S_1, S_2, \ldots, S_t$ in the state space, defined by the start state S of a planning problem $\langle S, G \rangle$, when c is satisfied in the starting state S, and sg_i, where $i \in [1, n]$ and $sg = \langle sg_1, \ldots, sg_n \rangle$, is satisfied in a state S_j and sg_{i+1} is satisfied in S_k for $k \in [j, t]$, then g is indeed satisfied in the state where sg_n is also satisfied after all other sg_i are satisfied. Then we can define soundness of a set of d-rules as the following. A set of d-rules is sound if every d-rule in the set is sound.

The above notion of soundness of a set of d-rules is restrictive because we can have a set of d-rules which produces valid plans, but each d-rule may not be sound.

That is, there can be a d-rule that may not satisfy the soundness condition on *all* paths of the state space, but satisfy the condition on those paths of the state space restricted by the other (higher-level) d-rules. The following definition of soundness relaxes the condition to cover this case.

Definition 2.2.1 *A set of d-rules is* **sound** *if for any planning problem a complete decomposition of the problem w.r.t. the set of d-rules is a valid plan for the problem. A plan is a* **valid plan** *for a planning problem* $\langle S, G \rangle$ *if the sequence of operators in the plan applied in order starting from* S *results in a state where* G *is satisfied.*

A d-rule planner is sound if for any planning problem a plan produced by the d-rule planner is a valid plan for the problem. Any d-rule planner that is consistent with the semantics of complete decomposition w.r.t. an input set of d-rules is sound iff the set of d-rules is sound. The D-rule planner described above is consistent with the semantics of complete decomposition w.r.t. a set of d-rules. Therefore, we can say that the D-rule planner is sound iff the input set of d-rules is sound.

Next, we discuss the completeness of a set of d-rules and the completeness of the D-rule planner w.r.t. a set of d-rules given as input.

Definition 2.2.2 *A set of d-rules is* **complete** *if for any planning problem that has a valid plan, there is a complete decomposition of the planning problem w.r.t. the set of d-rules that is also a valid plan.*

Because a set of d-rules D input to a d-rule planner could be incomplete, a d-rule planner's completeness should be defined relative to D. A d-rule planner is complete if for any planning problem for which there is a complete decomposition of the planning problem w.r.t. D that is a valid plan, then there is also a valid plan for the planning problem produced by the D-rule planner.

The D-rule planner described earlier does not consider all possible complete decompositions of a planning problem w.r.t. the set of d-rules D input to the planner, but assumes that any d-rule choice made achieves the goal. Once it makes a choice, it does not backtrack. Therefore, it can miss some complete decompositions of a planning problem that can produce a valid plan. Thus, the D-rule planner is not complete w.r.t. D.

However, if the d-rule set D has the "downward solution property", then the D-rule planner is complete w.r.t. D. The downward solution property (Russell & Norvig, 1995) when applied to d-rules says that if SP is the sequence of subproblems resulting from any d-rule, then there is a complete decomposition of the sequence of subproblems in SP w.r.t. D which solves the planning problem. When this property is satisfied of a d-rule set, then any d-rule application leads to a solution. Therefore, the D-rule planner can find a solution without backtracking.

Consider, for example, the d-rules for the Blocks-World domain in Figures 2.1 and 2.2. Given any starting state, when the goal is `on(?x, ?y)`, the d-rules clear the tops of the blocks `?x` and `?y` by unstacking the blocks on top of the blocks `?x` and `?y` onto the table, and then put the block `?x` on the block `?y`. Any sequence of complete decompositions, w.r.t. the two d-rules, of the subproblems corresponding to the subgoals `clear(?x)`, `clear(?y)` and `put-on(?x, ?z, ?y)`, can achieve the goal `on(?x, ?y)`. Thus, the two d-rules exhibit downward solution property. If, however, the d-rule for `clear` is modified such that it clears the top of a block `?x` by putting the blocks that are on top of `?x` on some other blocks, instead of putting them on table, then the downward solution property may not be guaranteed. This is so, because, in achieving `on(?x, ?y)`, after the subgoal `clear(?x)` is achieved, achieving the subgoal `clear(?y)` through the modified d-rule may clobber the already achieved

subgoal `clear(?x)` by putting a block that is on top of the block `?y` on top of the block `?x`.

2.2.2 *Other Representations Versus D-rules*

D-rules can be viewed as high-level operators for hierarchical-decomposition planning as well as representations of control knowledge for non-hierarchical or flat planners. In this section, we compare d-rules with macro operators and PRODIGY's control rules, the two most prominent methods for representing control knowledge.

A macro-operator is a sequence of operators, which may have been learned from a successful planning episode. It helps in reducing the search depth in planning. If they are used in addition to the primitive operators, as done by (Minton, 1988), they decrease the distance to a goal, but increase the number of operators the planner has to consider at each decision point. That is, this method can decrease the depth of the search tree, but increases the branching factor of its nodes. Otherwise, a planner could be incomplete. In any case, they do not have the hierarchical structure of d-rules. Moreover, learning macro operators in recursive domains is difficult because a macro-operator produced for N steps tends to be overly specific and cannot generalize for more than N steps—the so called Generalization-to-N problem (Shavlik, 1990; Subramanian & Feldman, 1990). PRODIGY's control rules, the other prominent method of representing control knowledge, select, reject or order applicable domain operators in each state (Minton, 1988). These rules help reduce the number of operators considered at each point, but do not affect the distance to the goal. They too, like macro-operators, are prone to the Generalization-to-N problem (Minton, 1988).

D-rules decompose a goal into a sequence of subgoals. Planning using subgoals has the complexity O($b^{D_{max}}$) where D_{max} is the distance between the farthest successive

subgoals, and b is the branching factor (the number of applicable operators) (Korf, 1987b). Without goal decomposition, the complexity is exponential in the distance between the start state and the goal, which is typically higher than D_{max}. When goals are recursively decomposed using d-rules into primitive actions and there is no backtracking over d-rule choice, problem solving is linear in the number of d-rule applications, which in turn is linear in the solution length. Thus, d-rules can reduce the planning effort more drastically than control rules or macro operators can.

2.3 Reactive Planning with D-rules

In dynamic domains where there can be changes that are not under the control of the planning agent, situations may change unexpectedly in the course of an execution of a plan, and the plan the agent has come up with may no longer be valid. A planner should be able to react and gracefully change its plans in the light of unanticipated changes. In the following, we extend the representation of d-rules and the d-rule planner to achieve reactive planning.

Consider a simplified scenario of planning for a long weekend in Oregon. Suppose our goal is to have fun (`have-fun`) during a long weekend. Further, suppose that this goal can be satisfied in two ways: out-door fun (`outdoor-fun`) and in-door fun (`indoor-fun`). Suppose out-door fun could be had by camping (`camping`) or picnicking (`picnic`). Naturally, we hope the weather will be dry and warm at the place and the time of our camping. To achieve our goal of having fun during a long weekend, suppose we choose the subgoal of out-door fun. To have out-door fun, suppose we choose to go camping. To go camping, suppose we planned all the steps needed. Now, between the time we start executing the plan and the time we finish the plan and achieve the goal of having fun, weather that was balmy and sunny could become damp, cold and cloudy suddenly (as is its wont in Oregon). We would

like the weather to be good throughout the execution of this plan for achieving the having-fun goal. We want to establish some "invariants," and require those invariants be true throughout the execution of the plan. If an invariant is violated, we need to replan for the goals the planner is still pursuing from the current situation. In the above scenario, we need to replan for the goal of having fun from the current state of our position, when this unexpected weather change occurred—maybe, by achieving the subgoal of having indoor fun.

To implement "invariants", we extend the representation of d-rules to include a *monitors* field (Veloso, Pollack, & Cox, 1998). The monitors field in a d-rule is a set of conjunctive conditions that are expected to be true throughout the execution of plans suggested by the d-rule. If not, the plan suggested by the d-rule is no longer guaranteed to achieve its intended goal. For example, a d-rule for the goal *outdoor-fun* could be the following.

goal:	`outdoor-fun`
condition:	`camping-site(?at), site-open-now(?at)...`
monitors:	`weather(?at, good)`
subgoals:	`camping(?at)`

The D-rule planner also needs to be modified to accommodate monitors. Recall that the D-rule planner plans in depth-first fashion—that is recursively decomposing a goal fully into a sequence of primitive operators that achieves the goal, before going on to decompose subsequent goals in a sequence of goals. Because we assume that any d-rule choice made is good (see Section 2.2), it is safe to perform a left-to-right depth-first refinement of the plan and to execute primitive operators as soon as they are encountered during the depth-first refinement process, as long as there are no dynamic changes to the planner's environment. To take care of dynamic changes

to the planner's environment, the D-rule planner maintains a list of monitors corresponding to all (sub)goals currently being achieved. Before each decomposition step, the D-rule planner checks the current state to see whether any of the monitors in the list of monitors is violated (due to unexpected changes). If not, the planner proceeds as usual. If a monitor is violated, recursion of the decomposition unwinds until and including the topmost goal, g, whose d-rule has its monitor violated. Replanning starts in the current state for achieving the main goal, which may be g or its ancestor in the decomposition hierarchy. In the above example, when the weather became inclement while we were executing the plan for achieving the goal `have-fun` by achieving the subgoal `outdoor-fun`, which in turn was being accomplished by the subgoal `camping`, the topmost (sub)goal whose d-rule has its monitor violated is `outdoor-fun`. Therefore, we replan for `have-fun` in the state where the weather is inclement, perhaps by trying a d-rule that suggests the subgoal `indoor-fun`.

Along with the checking for monitors, the planner checks whether any (sub)goal among the (sub)goals that are actively being tried for is achieved fortuitously, and, if so, unwinds the recursion past that (sub)goal, and proceeds with the next (sub)goal.

2.4 Hierarchical Task Networks (HTNs) and D-rules

Similar to d-rules, Hierarchical Task Networks (HTNs) also express goal-decomposition knowledge. Although HTNs have existed in one form or another since the early 1970's, the ideas behind HTNs have only been consolidated and formalized recently by Erol (1995). HTNs have tasks that correspond to goals and subgoals of d-rules. Primitive tasks in HTNs correspond to domain operators. Methods for a task in HTNs correspond to d-rules for a goal. The representation for methods in HTNs is more general than the representation for d-rules. HTN methods allow specifications of partial orders among tasks, whereas d-rules only allow total order among subgoals.

HTN methods specify non-codesignation (inequality) constraints, whereas d-rules do not. HTN methods can represent constraints such as what literals should be true between what points during the plan. This type of constraints enable us to specify between what points in a plan a subgoal is relevant and needs to be protected. Unlike HTNs, d-rules can only specify literals that need to be true (monitors) throughout the execution of a plan.

Accordingly, planners for HTNs are more complex than the d-rule planner. Nevertheless, d-rules and the d-rule planner capture the essential aspect of HTNs—task reduction or goal decomposition.

One aspect of task-reduction representations is that they are effective ways to represent control knowledge—more effective than control rules and macro operators. Task reduction helps a planner focus its search globally and before committing to specific primitive actions. Control rules and macro operators, on the other hand, aid the planner during search by making local decisions. HTNs and d-rules can naturally express looping or iteration by means of recursion. An example of looping is constructing a stack of blocks by placing one block at a time. State-space or plan-space planners have to resort to converting a goal with quantified variables representing objects, into a large conjunction of goals—e.g., UCPOP (Penberthy & Weld, 1992). The stacking example can be expressed as a d-rule in the following way:

```
     goal:  stack(?a, ?small-stack)
condition:  first(?b, ?small-stack), rest(?rest, ?small-stack)
 subgoals:  <stack(?b, ?rest), on(?a, ?b)>
```

The above d-rule says that to construct a stack of blocks with the block `?a` on top, construct the rest of the stack, and put the block `?a` on top of the top-most

block of the rest of the stack (block `?b`). It is hard to express this kind of iterative knowledge in the representations of either control rules or macro operators.

Chapter 3

MAPPING D-RULES INTO HORN CLAUSES

The rest of the dissertation concerns learning d-rules. In this chapter, we develop a mapping from the d-rule representation to Horn-clause representations. Thus, if we have learning algorithms for Horn clauses, we can utilize them for learning d-rules. This approach also helps us in exploiting some well-developed tools in the area of inductive logic programming (ILP) to learn d-rules. We also extend the mapping to transform HTN representations into Horn-clause representations.

3.1 Preliminaries

This section describes the terminology and the notation of logic we will be needing. For a more comprehensive description see standard text books on logic programming such as (Lloyd, 1987).

Definition 3.1.1 *A* **term** *is defined recursively as follows: (1) a variable is a term; (2) a constant is a term; and (3) if f is an n-ary function symbol and $t_1, t_2, \ldots, t_n$ are terms, then $f(t_1, t_2, \ldots, t_n)$ is also a term. A term is a* **ground term** *if it does not contain any variables.*

Definition 3.1.2 *If p is an n-ary predicate symbol, and $t_1, t_2, \ldots, t_n$ are terms, then $p(t_1, t_2, \ldots, t_n)$ is called an* **atom**. *A* **literal** *is an atom (positive literal), or a negation of an atom (negative literal).*

Definition 3.1.3 *A* **definite Horn clause** *(Horn clause or clause, for short) is a finite set of literals that contains exactly one positive literal. It is treated as a*

disjunction of the literals in the set with universal quantification over all the variables. The positive literal, if it exists, is called the head *of the clause. The set of negative literals is called the* body *of the clause. A Horn clause is* **non-recursive** *if the predicate symbol of the head literal of the Horn clause does not occur in the body of the clause. A* **unit Horn clause** *is a Horn clause with no negative literals and hence no body.*

We usually denote a Horn clause as $P \to q$, where q is the head of the clause and P is the set of all atoms appearing in the body of the clause. For example, the clause $\neg l_1, \neg l_2, \ldots, \neg l_n, q$ is denoted as $l_1, l_2, \ldots, l_n \to q$.

Definition 3.1.4 *A* **Horn definition** *is a set of Horn clauses where the heads of all clauses have the same predicate symbol.** *It is* **non-recursive** *if the head predicate symbol does not occur in any negative literal in any clause in the definition.*

Definition 3.1.5 *A* **Horn program** *or* **Horn sentence** *is a set of definite Horn clauses interpreted conjunctively.*

Definition 3.1.6 *A finite set* $\theta = \{v_1/t_1, \ldots, v_n/t_n\}$ *is called a* **substitution**, *where each* v_i *is a variable, and the variables are all distinct. Each* v_i/t_i *in* θ *is called a* binding. *A substitution is a* ground *substitution if all the terms in it are ground terms.*

3.2 D-rules to Horn Clauses

In this section, we describe how d-rules can be transformed into Horn-clauses.

Recall that a d-rule is comprised of three parts: goal, initial conditions, and a sequence of subgoals. To represent the notion of state, which is missing in the Horn

* A Horn definition is also called a *predicate definition.*

clause, we add special symbols to the literals that denote "situations". In particular, the first two parameters of each literal are new and denote the names of the situations in between which that literal must be true. The first parameter specifies the starting situation in which the literal must be true. The second parameter specifies the situation up to and including which the literal must be true. We call these two parameters of a literal the *situation parameters.* When we mean that a literal is true in a particular situation alone, that situation is mentioned in both the situation parameters of the annotation of the literal. When the situation parameters in a literal are different, it means that the literal is true throughout the duration between the situations represented by the situation parameters. In addition, the two situation parameters in a literal implicitly indicate that the first situation occurs before the second situation or that both the situations are the same. However, this in itself may not fully represent all the relative orderings between situations we want to specify. Therefore, to explicitly represent the relative ordering of two situations `Si` and `Sj`, we use a special predicate symbol `not-after` and add the literal `(not-after Si Sj)`, meaning that the situation `Si` does not occur after the situation `Sj`.

A d-rule can be declaratively read as follows: starting from a state that satisfies the initial conditions of the d-rule, if each of the subgoals is achieved one by one in sequence, then the goal of the d-rule would be true in the state that achieved the last subgoal in the sequence. This declarative reading makes the connection between d-rules and Horn clauses explicit. The goal of a d-rule corresponds to the head literal of the corresponding Horn clause and is true in the final situation. The initial conditions, which are conjunctions of positive literals, and the subgoals, which are single positive literals, when properly annotated with situation variables, correspond to the body of the Horn clause. In addition, we might need to add some `not-after` literals to constrain the relative orderings between different situation variables that

correspond to the different subgoals. One thing to note is that a d-rule does not say once a subgoal is achieved how long it should remain true. Another point is that the use of situation variables in Horn-clause form allows us to express partial ordering of subgoals. However, the subgoal component of a d-rule defines a total ordering of subgoals.

For example, the d-rule

```
      goal:  on(?x ?y)
  subgoals:  <clear(?x), clear(?y), put-on(?x, ?z, ?y)>
conditions:  block(?x), block(?y), table(?z)
```

translates into the following Horn clause.

```
block(?S0, ?S0, ?x), block(?S0, ?S0, ?y), table(?S0, ?S0, ?z),
clear(?S1, ?S2, ?x), clear(?S2, ?S3, ?y), put-on(?S3, ?S4, ?x, ?z, ?y),
not-after(?S0, ?S1) → on(?S4, ?S4, ?x, ?y)
```

The first three literals in the body of the Horn clause correspond to the initial conditions of the d-rule. Since these literals must be true in the initial state, they are given the situation parameters corresponding to the initial state (`?S0`). Next three literals in the body of the Horn clause correspond to the subgoals of the d-rule. They are given situation parameters such that they are true one after the other in a sequence of situations. The last literal in the body, explicitly states that the situation `?S0` does not come after the situation `?S1`. This, with the implicit orderings between the situations present in the other literals, implies that `?S0` is the initial state. Finally, in `?S4`, the goal literal is true, which is the head of the clause.

Note that it is possible to express partial orders using this notation by simply not specifying `not-after` relation between situations. For instance, if we would like to specify that `clear(?x)` and `clear(?y)` can be achieved in any order, we can replace the literal `clear(?S1, ?S2, ?x)` by `clear(?S1, ?S3, ?x)`, and add

`not-after(?S0, ?S2)`. By this, we say that `?S1` and `?S2` are not relatively ordered, but both of them are preceded by `?S0`, and succeeded by `?S3`. The ability of the Horn-clause notation to express partial ordering among subgoal literals raises some issues in learning d-rules via learning Horn clauses, since the semantics of d-rules say that subgoals are totally ordered. This issue is explored in Section 4.4.

Since the objective is learning d-rules via learning Horn clauses, training examples for d-rules should be converted to training examples for Horn clauses. A training example for learning d-rules has a sequence of states, `S0, S1,..., Sn` and a goal. An example can be viewed as a fully instantiated (ground) d-rule specifying the initial condition and a sequence of subgoals, with both including several irrelevant literals. It can then be converted into a Horn-clause as described above.

In particular, each state of the example is a set of positive literals describing the relationships between objects in a state. A state may have literals corresponding to subgoals achieved in that state. Along with a sequence of states the example has an instance of a goal that is true in the last state. Each state is given a situation number. In the corresponding Horn-clause form, each literal is annotated with situation parameters `Si` and `Sj` as its first two parameters, where `Si` and `Sj` represent a maximal duration in which the literal is true. For example, suppose the literal `clear(?x)` is true in `S0`, `S1`, `S2`, and `S3`, and again in `S5` and `S6`, but nowhere else. Then, its corresponding Horn-clause form would have only the literals `clear(S0, S3, ?x)` and `clear(S5, S6, ?x)`. Then, for each state `Si`, there is a set of literals comprising the literal `not-after(Si, Si)` and the literals `not-after(Si, Sj)` for each `Sj` such that $i < j \leq n$. These two sets of literals form the body of the corresponding Horn-clause example. The goal literal annotated with the situation number of the state in which the goal is true, becomes the head of the Horn-clause example.

So far, we have seen how one d-rule corresponds to one Horn clause. Next, we consider sets of d-rules. A goal can have multiple ways of achieving it, depending upon the initial state a planner starts from. These multiple ways correspond to multiple d-rules for a goal. A set of d-rules intended for a goal, then, corresponds to a set of Horn clauses with the same predicate symbol in all their head literals—also known as a Horn definition. If the predicate symbol of a goal literal does not appear in any of its d-rules' subgoals or conditions, then the corresponding Horn definition is a non-recursive Horn definition.

A set of d-rules for more than one goal corresponds to a set of Horn clauses or a Horn program.

3.3 HTNs to Horn Clauses

Methods in HTNs correspond to d-rules. A method is a 2-tuple $\langle \alpha, d \rangle$ where α is a non-primitive task (or goal/subgoal in our terminology), and d is a task network. A task network is of the form $[(n_1 : \alpha_1) \ldots (n_m : \alpha_m); \phi]$ where each α_i is a task, each n_i is a task label, and ϕ is a boolean formula. ϕ can contain the logic connectives such as conjunction, disjunction and negation, connecting literals of the following types:

- codesignation and non-codesignation constraints, such as a variable equals or does not equal a variable or a constant;
- ordering constraints between task labels, such as n_i precedes n_j;
- state constraints of the form (n, l), (l, n) and (n, l, n') where n and n' are task labels, and l is a literal. (n, l) means l is true in a state immediately after the end of task n; (l, n) means l is true in a state immediately before the start of

task n; and (n, l, n') means l is true in all states after the end of the task n and before the start of the task n';

- literals true in an initial state.

The declarative meaning of an HTN method $\langle \alpha, d \rangle$ for the task α, where the task network d is $[(n_1 : \alpha_1) \ldots (n_m : \alpha_m); \phi]$, is as follows. If the subtasks α_i labeled n_i are achieved obeying the constraint formula ϕ, then the task α is achieved.

As an example consider the Blocks-world example in Section 3.2. For the task `on(?x, ?y)` an HTN method is $\langle$`on(?x, ?y)`, $d\rangle$ where d is the task network

$[(n_1$: `clear(?x)`) $(n_2$: `clear(?y)`) $(n_3$: `put-on(?x, ?z, ?y)`); $((n_1$ precedes $n_3) \wedge (n_2$ precedes $n_3) \wedge$ $(n_1, \texttt{clear}(?\mathrm{x}), n_3) \wedge (n_2, \texttt{clear}(?\mathrm{y}), n_3) \wedge$ $(?\mathrm{x} \neq ?\mathrm{y}) \wedge (?\mathrm{y} \neq ?\mathrm{z}) \wedge (?\mathrm{x} \neq ?\mathrm{z}))]$.

The meaning of this method is as follows: To achieve the task `on(?x, ?y)`, achieve `clear(?x)` from situation S_1 till situation S_1', `clear(?y)` from S_2 till S_2', and `put-on( ?x, ?z, ?y)` from S_3 till S_3' such that the objects (table or blocks) denoted by `?x`, `?y` and `?z` are different, the situations S_1' and S_2' precede S_3, and once achieved `clear(?x)` and `clear(?y)` are true until S_3.

The declarative meaning of HTNs makes an explicit connection between HTN methods and Horn clauses. The task α of an HTN method corresponds to the head literal of the corresponding Horn clause. Achievement of subtasks in their situations and the constraint formula correspond to the condition or body part of the corresponding Horn clause. We consider in the following how to convert a method into Horn clauses, in detail.

First, Horn clauses contain only positive literals in the body. Hence, only HTN methods without negative literals can be converted into Horn clauses. Note, however,

that we do not consider non-codesignation constraints as negative literals. Next, bodies of Horn clauses cannot contain disjunctions. Therefore, we split a method containing disjunctions into multiple methods each containing no disjunctions (and no negations). Each non-disjunctive HTN method with only positive literals that has codesignation constraints can be converted into a method without codesignation constraints in the following way. For each codesignation constraint $v = x$, where v is a variable and x is either a variable or a constant, remove that constraint from the method, and replace every occurrence of v in the method by x. Henceforth, for the sake of brevity, such HTN methods without any disjunctions, negations and codesignation constraints are referred to as just HTN methods.

Now, we convert each constraint in an HTN method into a literal, in the following way. If it is a non-codesignation constraint, $v_1 \neq v_2$, then it becomes the literal `neq(`v_1`, `v_2`)`. For each task n_i, we associate two situations S_i and S_i' such that the task n_i is achieved starting from the situation S_i till the situation S_i', and add the literal `not-after(`S_i`, `S_i'`)` to the Horn-clause form. An ordering constraint of the form n_i precedes n_j is converted to the literal `not-after(`S_i'`, `S_j`)`. A state constraint of the form (n_i, l, n_j) where l is $p(t_1, t_2, \ldots, t_k)$ is converted to the literal $p(S_i', S_j, t_1, t_2, \ldots, t_k)$, meaning that $p(t_1, t_2, \ldots, t_k)$ is true starting from the situation S_i' where the task n_i ends up to the situation S_j where the task n_j begins. A constraint of the form (n, l), meaning that the literal l is true in the situation S' when the task n ends, where l is $p(t_1, t_2, \ldots, t_k)$, is converted to the literal $p(S', S', t_1, t_2, \ldots, t_k)$. A constraint of the form (l, n), meaning l is true in the situaton S when the task n commences, where l is $p(t_1, t_2, \ldots, t_k)$, is converted to the literal $p(S, S, t_1, t_2, \ldots, t_k)$.

The constraints of the last type specify literals that must be true in an initial state. In the Horn-clause equivalent, we add the literal `initial-state(`S_0`)`, and for

each literal that must be true in the initial state, $p(t_1, t_2, \ldots, t_k)$, we have the literal $p(S_0, S_0, t_1, t_2, \ldots, t_k)$. Each subtask element $(n_i : \alpha_i)$ of the list of subtasks for the HTN method $[(n_1 : \alpha_1) \ldots (n_m : \alpha_m), \phi]$, where $\alpha_i = p_i(t_1, t_2, \ldots, t_k)$, in its Horn-clause form, has a literal $p_i(S'_i, S'_i, t_1, t_2, \ldots, t_k)$, unless there is a state constraint $(S'_i, p_i(t_1, t_2, \ldots, t_k), S_j)$.

All the literals that we arrived at, using the above process, form the body of a Horn-clause version of the HTN method. The non-primitive task α of the method becomes the head of the corresponding Horn clause after being annotated with the final state: $p_t(S_f, S_f, t_1, t_2, \ldots, t_k)$, where α is $p_t(t_1, t_2, \ldots, t_k)$.

The Horn-clause form for the Blocks-world HTN method is the following.

```
clear(S1, S3, ?x), clear(S2, S3, ?y), put-on(S3, S3', ?x, ?z,
?y), not-after(S1, S1'), not-after(S2, S2'), not-after(S3, S3'),
not-after(S1', S3), not-after(S2', S3), neq(?x, ?y), neq(?y, ?z),
neq(?x, ?z), initial-state(S0) → on(S3', S3', ?x, ?y)
```

We do not have the literals `clear(S1, S1', ?x)` and `clear(S2, S2', ?y)` because the literals `clear(S1, S3, ?x)` and `clear(S2, S3, ?y)`, derived from the state constraints $(n_1, \texttt{clear}(?\texttt{x}), n_3)$ and $(n_2, \texttt{clear}(?\texttt{y}), n_3)$, cover them. Note that the partial order between `clear(?x)` and `clear(?y)` present in the HTN method is also reflected in the above Horn-clause form.

As in the case of d-rules, the set of Horn clauses corresponding to multiple methods for a task, forms a Horn definition. The set of Horn clauses corresponding to methods for multiple tasks forms a Horn program.

Chapter 4

LEARNING HORN DEFINITIONS

In Chapter 3, we have seen how d-rules and HTNs could be converted to Horn clauses. We noted there that d-rules for a simple goal and HTN methods for a single task correspond to Horn definitions in Horn-clause form. In this chapter, we consider learning non-recursive, first-order Horn definitions from entailment—where positive (negative) examples are clauses (not) implied by the target Horn definition. We show that this class is exactly learnable from equivalence and membership queries. It follows then that this class is PAC learnable using examples and membership queries. In Chapters 6 and 7, we will see how d-rules can be learned, based on the algorithm presented here.

4.1 Introduction

Horn clauses provide a popular way of representing relational or first-order knowledge. In this chapter, we consider learning Horn definitions—multiple Horn clauses with the same predicate in the heads of all clauses—in the *learning from entailment* setting (Frazier & Pitt, 1993; De Raedt, 1997). In this setting, the target concept is a Horn definition. A positive (negative) example is a Horn clause (not) entailed by the target. Learning Horn definitions is a fundamental problem both in Inductive Logic Programming (ILP) and in Computational Learning Theory. Since it is NP-hard to test membership in this concept class, it immediately follows that even non-recursive Horn definitions are hard to learn from examples alone, unless NP = P

(Schapire, 1990). Using only equivalence queries, single non-recursive Prolog clauses are learnable with restrictions such as determinacy and bounded arity (Cohen, 1995a; Džeroski, Muggleton, & Russell, 1992). Restricted versions of single recursive clauses are also learnable (Cohen, 1995b). However, learning multiple clauses or even slightly more general versions of either recursive or non-recursive clauses is shown to be hard without further help (Cohen, 1995c). Page showed that non-recursive Horn definitions with predicates having fixed arity and with the restriction that the clauses be "simple," i.e., only the variables and terms that occur in the head literal of a clause appear in the body of the clause, are learnable using equivalence and subset queries (Page, 1993). Here, we examine the learnability of a more general class.

We show that first-order non-recursive Horn definitions are exactly learnable from membership and equivalence queries with no other restrictions. In particular, the target concepts may have an arbitrary number of clauses with the number and the arity of the literals in each clause also being unbounded. The literals may also contain functions. Learning from equivalence and membership queries is one of the standard models (also called the "minimally adequate teacher" by Angluin (1988)) in the Computational Learning Theory literature. This is a natural model to consider when the learner has a choice of asking whether a given instance is positive or negative. Some languages such as deterministic finite state automata and propositional Horn sentences which appear not to be learnable from examples alone are learnable in this model. At the same time, it is a nontrivial model in that there are many languages, even apparently "simple" ones, such as arbitrary Boolean formulas, which are not learnable in this model (under some cryptographic assumptions). It is also known that for some languages such as DNF, membership queries do not help. Thus, learning a first-order language such as Horn definitions in this learning model is an important open problem (Angluin, Frazier, & Pitt, 1992). Most previous theoretical

work in ILP relies on the corresponding propositional algorithms, and hence does not really show the importance of using a first-order language. Our work is almost unique in that the hypothesis space we consider cannot be reduced to one that a propositional learner can learn efficiently. This is discussed in more detail in Section 4.5.

Our algorithm combines the ideas of several previous learning algorithms that use membership queries (Angluin, 1988; Frazier & Pitt, 1993; Haussler, 1989; Frazier & Pitt, 1995). It maintains a set of hypothesis clauses, each of which is subsumed by a corresponding target Horn clause. Given a new positive example, it either combines it with one of its hypothesis clauses producing a least general generalization (*lgg*) of the example and the hypothesis clause, or stores it as a new hypothesis clause. It uses membership queries to decide which hypothesis clause, if any, the example should be combined with. An example is combined with that clause which yields an *lgg* that is entailed by the target. The algorithm exploits the fact that there is at most one positive literal in each Horn clause, and that it cannot be resolved with any negative literals also in the Horn definition because the clauses are non-recursive and all have the same predicate symbol in their heads. These two facts imply that any clause which is entailed by the target must be subsumed by one of the clauses in the target—a property called "strong compactness." This guarantees that the membership queries, in effect, check whether a hypothesis clause is subsumed by a target clause. After combining the example with a hypothesis clause, the resulting *lgg* is pruned of redundant literals using membership queries. Without this step, the number of literals in the hypothesis clause can grow geometrically with each new example, exceeding any polynomial bound.

Learnability in our "exact-learning model" which uses equivalence and membership queries, implies learnability in the PAC-learning model (Valiant, 1984) which uses random examples and membership queries (Angluin, 1988).

The rest of the chapter is organized as follows: Section 4.2 presents some formal preliminaries about generalization of Horn clauses. Section 4.3 describes the learning problem, proves some properties of Horn definitions, describes the learning algorithm, DLearn, and proves its correctness. Section 4.4 shows how DLearn can be employed to learn d-rules. Section 4.5 concludes the chapter by relating our work to some previous work in this area and discussing its implications.

4.2 Preliminaries

In this section, we define and describe some more terminology from inductive logic programming that we need, in addition to the terminology in Chapter 3.

Definition 4.2.1 (Plotkin, 1970) *A clause D* **subsumes** *a clause E if there exists a substitution θ such that $D\theta \subseteq E$. We denote this as $D \succeq E$, and read it as D subsumes E or as D is more general than E.*

Definition 4.2.2 *If D and E are clauses such that $D \succeq E$, then a literal l in a clause E is* **relevant (irrelevant)** *w.r.t the clause D, if $D \not\succeq E - \{l\}$ ($D \succeq E - \{l\}$, respectively).*

Definition 4.2.3 *If D and E are two clauses such that $D \succeq E$, then a* **condensation** *of E w.r.t. D is a clause E' such that $E' \subseteq E$, $D \succeq E'$, and for any $l \in E'$, l is relevant w.r.t. D.*

For example, if $D = \{\neg p_1(x), p_2(y)\}$ and $E = \{\neg p_1(a), p_2(b), p_2(c), p_3(c)\}$, then $\{\neg p_1(a),\ p_2(b)\}$ and $\{\neg p_1(a), p_2(c)\}$ are the only condensations of E w.r.t. D.

Definition 4.2.4 (Plotkin, 1970) *Let C, C', C_1 and C_2 be sets of literals. We say that C is the* **least general generalization** *(lgg) of C_1 and C_2 iff (1) $C \succeq C_1$ and $C \succeq C_2$, and (2) $C' \succeq C$, for any C' such that $C' \succeq C_1$ and $C' \succeq C_2$.*

The first condition says that C is more general than C_1 and C_2, and the second says that any other clause more general than C_1 and C_2, is also more general than C. The existence and uniqueness of least general generalization of two clauses is shown by Plotkin (1970), and by Nienhuys-Cheng and de Wolf (1996).

Definition 4.2.5 (Plotkin, 1970) *A* **selection** *of clauses C_1 and C_2 is a pair of literals (l_1, l_2) such that $l_1 \in C_1$ and $l_2 \in C_2$, and l_1 and l_2 have the same predicate symbol, arity, and sign.*

If C_1 and C_2 are sets of literals, then $lgg(C_1, C_2)$ is $\{lgg(l_1, l_2) : (l_1, l_2)$ is a selection of C_1 and $C_2\}$. If l is a predicate, $lgg(l(s_1, s_2, \ldots, s_n), l(t_1, t_2, \ldots, t_n))$ is $l(lgg(s_1, t_1), \ldots, lgg(s_n, t_n))$. The lgg of two terms $f(s_1, \ldots, s_n)$ and $g(t_1, \ldots, t_m)$, if $f = g$ and $n = m$, is $f(lgg(s_1, t_1), \ldots, lgg(s_n, t_n))$; else, it is a variable $?x$, where $?x$ stands for the lgg of that pair of terms throughout the computation of the lgg of the set of literals.

For example, let C_1 be $\{\neg p_1(f(a), b), \neg p_2(a, c), p_3(b)\}$ and C_2 be $\{\neg p_1(f(c), d)$, $\neg p_1(b, a)$, $\neg p_2(c, c)$, $p_3(a)\}$. Then $lgg(C_1, C_2) = \{\neg p_1(f(?x), ?y), \neg p_1(?z, ?u)$, $p_3(?u)$, $\neg p_2(?x, c)\}$, where the variables $?x$, $?y$, $?z$ and $?u$ stand for the pairs of terms (a, c), (b, d), $(f(a), b)$ and (b, a), respectively.

Note that $|lgg(C_1, C_2)|$ can be as high as $|C_1| \times |C_2|$.

Lemma 4.2.1 *Let C_1, C_2 and C_3 be Horn clauses. Then $C_1 \succeq C_2$ and $C_1 \succeq C_3$ if and only if $C_1 \succeq lgg(C_2, C_3)$.*

Proof. The only-if part follows from the property (2) of the definition of least-general generalization. The if part follows from the transitive property of $\succeq$. □

We state the following fact explicitly, although it is straightforward, for it is useful later.

Proposition 4.2.1 *If $C_1 \succeq C_2$ then $C_1 \succeq C_3$ for any C_3 such that $C_2 \subseteq C_3$.*

4.3 Learning Horn Definitions

In this section, we first specify our learning problem. Next we describe the learning algorithm and then give the learnability result.

4.3.1 Learning Problem

Following the *learning from entailment* model, a ground Horn clause $body \rightarrow head$ is *in a* hypothesis H iff all models of H with respect to the literals in $body$ satisfies $head$. In other words, $body \rightarrow head$ is an instance of H iff $H \models (body \rightarrow head)$. Such an instance is a *positive example* of H. All other instances are *negative examples.*

Henceforth, Σ denotes the target concept in the hypothesis space.

Example 4.3.1 *This example illustrates the above definitions in a simplified version of an air-traffic control domain (see Section 6.3.3.*
$\Sigma = \{$

```
plane-at(?p,?loc), level(L1,?loc), free-runway(?r), short-runway(?r),
land-short(?p) → land-plane(?p);
plane-at(?p,?loc), level(L1,?loc), free-runway(?r), long-runway(?r)
→ land-plane(?p)
```

$\}$

The first clause in Σ gives the conditions under which a plane can land on short runways, that the plane should be at a level 1 location, that the plane is capable of landing on short runways, and that a short runway is free. The second clause is for long-runway landing. The following is a positive example of Σ (for the second clause):

```
plane-at(P737, 10), level(L1, 10), free-runway(R1), long-runway(R1),
short-runway(R2), wind-speed(high), wind-dir(south), free-runway(R2) →
land-plane(P737).
```

□

Before stating the learning problem, we define the queries we will need, following Angluin (1988).

Definition 4.3.1 *A* membership query *takes as input an instance x, and outputs yes if x is in Σ, and no otherwise. An* equivalence query *takes as input a hypothesis H, and outputs yes if H and Σ are logically equivalent; otherwise, it returns a* counterexample *from $H \oplus \Sigma$ —i.e., an instance that is in one but not in the other.*

The above combination of queries is called a "minimally adequate teacher" by Angluin. The learning problem in the exact learning model (Angluin, 1988) is as follows:

Definition 4.3.2 *An algorithm* exactly learns *a concept class $\mathcal{C}$ if for every concept $\Sigma \in C$, if it asks equivalence and membership queries, terminates in time polynomial in the size of Σ and the size of the largest counterexample, and outputs a hypothesis which is logically equivalent to Σ.*

In the rest of this section, we will be showing that the class of non-recursive Horn definitions is exactly learnable from equivalence and membership queries. Note that learning exactly does not mean learning a syntactically equivalent definition, but only a semantically equivalent one. In other words, there should be no counterexample for the learner's final hypothesis.

4.3.2 The Learning Algorithm

DLearn is an algorithm to learn non-recursive Horn definitions using equivalence and membership queries (Figure 4.1). DLearn makes use of an algorithm Reduce. The Reduce algorithm takes as input a Horn clause and generalizes it by eliminating literals from that Horn clause. It removes a literal from the Horn clause and checks

whether the resultant Horn clause is overgeneral. It can do this by substituting each variable in the hypothesis clause with a unique constant and asking a membership query. If it is overgeneral, the literal is retained; otherwise, it is eliminated to form a new, more general Horn clause.

DLearn starts with an hypothesis H that is initially empty. As long as H is not equivalent to the target concept C, the equivalence query returns an example e that is not included in H, and the algorithm modifies H to cover e. To include e in H, DLearn checks each Horn clause h_i of H whether generalizing h_i to cover e would not make the hypothesis overgeneral—i.e., whether $lgg(h_i, e)$ is in the target concept. If so, concluding that it has found the right Horn clause h_i to include e in, DLearn further generalizes $h = lgg(h_i, e)$, by removing irrelevant literals, i.e., those literals whose removal preserves the entailment relation between Σ and h. The entailment relation is checked by using the membership oracle on the result of skolemizing h (see Reduce in Figure 1). The assumption here is that the skolemizing process generates constants that are not already present in the target. DLearn finally replaces h_i in H by the new generalized h. If there is no h_i such that $lgg(h_i, e)$ is entailed by the target, it generalizes e using Reduce and makes it a new Horn clause of H.

Example 4.3.2 *Let* Σ *be* $\{\rightarrow q(f(f(?x))), ?x); p_1(?x, ?y), p_1(?y, ?z) \rightarrow q(?x, ?z); p_1(?x, ?y), p_2(?y, ?z) \rightarrow q(?x, ?z)\}$.
Let the first example be $e1$*:* $p_1(a, b), p_1(a, d), p_1(b, z), p_2(c, b), p_2(c, d), p_2(d, e) \rightarrow q(a, e)$.
Since H *is empty, the next step is* Reduce*(*$e1$*).*

In Reduce*:*

$\Sigma \models p_1(a, d), p_1(b, z), p_2(c, b), p_2(c, d), p_2(d, e) \rightarrow q(a, e)$*?*

yes*, so drop* $p_1(a, b)$*.*

$\Sigma \models p_1(b, z), p_2(c, b), p_2(c, d), p_2(d, e) \rightarrow q(a, e)$*?*

```
DLearn
(1)  Let Σ be the target concept.
(2)  H := {} /* empty hypothesis, initially */
(3)  m := 0 /* number of clauses in the hypothesis */
(4)  while equivalence(H, Σ) is not true and e is a counterexample do {
/* fix the clause in H for the example e */
(5)    if (m > 0) then, Let H be {h_1, h_2, ..., h_m}
(6)    found := false; i := 1
(7)    while (i ≤ m) and found is false do {
(8)      h := lgg(e, h_i)
(9)      if Σ ⊨ h then found := true; /* Member?(Skolemize(h)) implements Σ ⊨ h */
(10)     else i := i + 1
(11)   } /* i ≤ m */
(12)   if found = false then h := e; m := m + 1;
(13)   h_i := Reduce(h) /* further generalize h */
(14) }
(15) return H

Reduce(h)
(1)  h' := h
(2)  for each literal l in (the body of) h do
(3)    if Σ ⊨ h' − {l} then h' := h' − {l} /* Implemented by Member?(Skolemize(h' − {l})) */
(4)  Return h'.
```

FIGURE 4.1: DLearn: An algorithm to learn Horn definitions

no, *keep* $p_1(a,d)$. ...

Finally, $h' = p_1(a,d), p_2(d,e) \rightarrow q(a,e)$

$h_1 = p_1(a,d), p_2(d,e) \rightarrow q(a,e)$

Let the next example be e2: $p_1(a,b), p_1(a,d), p_1(b,z), p_2(c,b), p_2(c,d), p_2(d,e) \rightarrow q(a,z)$.

$lgg(h_1, e2) =$

$p_1(a, ?db), p_1(a,d), p_1(?ab, ?dz), p_2(?dc, ?eb), p_2(?dc, ?ed), p_2(d,e) \rightarrow q(a, ?ez)$

$\Sigma \models lgg(h_1, e2)$*?* no.

So, $\mathsf{Reduce}(e2) = h_2 = p_1(a,b), p_1(b,z) \rightarrow q(a,z)$.

With $e1$ and $e2$, we have seen how new hypothesis clauses get added to H. The next example, $e3$, shows how `DLearn` combines a new example with a clause in H.

e3: $p_1(r,s), p_2(s,t), p_1(r,u), p_2(u,v) \rightarrow q(r,t)$.

$lgg(h_1, e3) = p_1(?ar, ?ds), p_2(?ds, ?et), p_1(?ar, ?du), p_2(?du, ?ev) \rightarrow q(?ar, ?et)$

$\Sigma \models lgg(h_1, e3)$*?* yes.

$h_1 = \mathsf{Reduce}(lgg(h_1, e3)) = p_1(?ar, ?ds), p_2(?ds, ?et) \rightarrow q(?ar, ?et)$.

The next example, $e4$, shows how `DLearn` learns clauses with empty body.

e4: $p_1(a,b) \rightarrow q(f(f(a)), a)$

$lgg(h_1, e4) = p_1(?ar, ?ds) \rightarrow q(?arf, ?eta)$

$\Sigma \models lgg(h_1, e4)$*?* no.

$lgg(h_2, e4) = p_1(a,b) \rightarrow q(?af, ?za)$

$\Sigma \models lgg(h_2, e4)$*?* no.

$\mathsf{Reduce}(e4) = h_3 = \rightarrow q(f(f(a)), a)$.

... □

The literal-removal process of Reduce serves a critical purpose. Recall that the size of the *lgg* grows as a product of the sizes of the hypotheses being generalized. Unless the hypothesis size is limited, it can grow exponentially in the number of examples used to create the hypothesis. Reduce guarantees that the size of each clause in the hypothesis is bounded by the size of its corresponding target clause (see Lemmas 4.3.2 and 4.3.3 in Section 4.3.2.2).

4.3.2.1 Strong Compactness of Non-recursive Horn Definitions

In this section we describe a property of non-recursive Horn definitions, which is called *strong compactness* by Lassez, Maher, and Marriott (1988), and Page (1993), and relate this property to membership queries.

Strong compactness says that for non-recursive Horn definitions if we know that a clause is logically implied by a set of clauses Σ, then we can conclude that that clause is subsumed by a clause in Σ. The following lemma, in addition, says that the converse is true. This is useful to show later that each clause in the current hypothesis of our algorithm is always a specialization of some target clause.

Lemma 4.3.1 *Let Σ be a non-recursive Horn definition and h be a Horn clause which is not a tautology. Then $\Sigma \models h$ if and only if there exists a clause C in Σ such that $C \succeq h$. We call C the* **target clause** *of h and h the* **hypothesis clause** *of C.*

Proof. The if part follows from the fact that Σ is interpreted as a conjunction of its clauses. The only-if part is a direct consequence of the Subsumption Theorem (Kowalski, 1970). We give a brief sketch of the proof here. Since $\Sigma \models h$ and h is not a tautology, there must be a non-trivial proof of h from the clauses of Σ. However, since the head predicate symbol of the clauses in Σ does not appear in the body of any clause, there can be no chaining of the clauses in the proof of h. This implies that h must be subsumed by a single clause in Σ. □

If a clause h has variables, determining $\Sigma \models h$ is equivalent to determining whether all instances in h are also in Σ—which is the same as a subset query (Angluin, 1988). However, by substituting each variable in h by a unique constant—skolemization—we can form a fully ground clause that is an instance of h. Now, determining whether $\Sigma \models h$ is equivalent to asking whether $\Sigma \models \mathsf{Skolemize}(h)$. Asking whether $\Sigma \models \mathsf{Skolemize}(h)$ is the same as a membership query, since $\mathsf{Skolemize}(h)$ is ground. In effect, this membership query simulates a subset query, and it is answered *yes*, by Lemma 4.3.1, if and only if some clause in Σ subsumes h.

4.3.2.2 Proof of Learnability

In this section, we prove that DLearn exactly learns non-recursive Horn definitions, by exploiting their strong compactness property. Lemma 4.3.2 and Lemma 4.3.3 together show that Reduce guarantees that the sizes of the hypothesis clauses are bounded from above by the sizes of their corresponding target clauses. Lemma 4.3.2 also shows that Reduce does not over-generalize in the process.

Lemma 4.3.2 *If the argument h of* Reduce *is such that $\Sigma \models h$, then, at the end of* Reduce*, h' has a target Horn clause C_j—i.e., $C_j \succeq h'$. Moreover, h' in line 4 of* Reduce *is a condensation of h w.r.t. C_j.*

Proof. In the beginning of Reduce, h', which is the same as the argument h, is not overgeneral. h' is modified only when the modification still leaves the result inside Σ. That is, $\Sigma \models h'$. By Lemma 4.3.1, there exists a target Horn clause for h', say C_j, and $C_j \succeq h'$.

To show that h' in line 4 of Reduce is a condensation of h w.r.t. C_j, we need only to show that for any literal $l \in h'$, $C_j \not\succeq (h' - \{l\})$. Suppose that for some $l \in h'$, $C_j \succeq (h' - \{l\})$. Let h'' be the value of h' when l is considered for removal in the loop of lines 2—3. Since $h' \subseteq h''$, by Proposition 4.2.1, $C_j \succeq (h'' - \{l\})$. From Lemma 4.3.1, $\Sigma \models (h'' - \{l\})$. In that case, l would have been removed by line 3 of Reduce. But, $l \in h'$, a contradiction. Therefore, for any literal $l \in h'$, $C_j \not\succeq (h' - \{l\})$. □

Lemma 4.3.3 *If h' is a condensation of h w.r.t. C_j, then $C_j\theta = h'$ for some substitution θ. Moreover, $|h'| \leq |C_j|$.*

Proof. Suppose h' is a condensation of h w.r.t. C_j. Then there exists a θ such that $C_j\theta \subseteq h'$. Suppose $C_j\theta \subset h'$. Then, for some $l \in h' - C_j\theta$, $C_j\theta \subseteq h' - \{l\}$.

Hence, $C_j \succeq (h' - \{l\})$. This is a contradiction, since h' is a condensation w.r.t. C_j. Therefore, $C_j\theta = h'$. This implies that $|h'| = |C_j\theta| \leq |C_j|$. □

The following definition relates an example to a hypothesis clause and to a target clause.

Definition 4.3.3 *If $C_1, C_2, \ldots, C_n$ are the Horn clauses of the target concept Σ, and $h_1, h_2, \ldots, h_m$ are the Horn clauses in the hypothesis H, then the* correct hypothesis Horn clause *in H for an example e is a Horn clause h_i with the smallest index i such that for some $1 \leq j \leq n$, $C_j \succeq e$ and $C_j \succeq h_i$.*

Lemma 4.3.4 *In* DLearn*, suppose that e is a counterexample returned by the equivalence query such that e is covered by Σ, but not by H. Then* DLearn *includes e in a hypothesis Horn clause if and only if it is the correct hypothesis Horn clause in H for e.*

Proof. First the only-if part. DLearn includes e in h_i of H only if i is the smallest index such that $\Sigma \models lgg(e, h_i)$. Hence, by Lemma 4.3.1, $C_j \succeq lgg(e, h_i)$ for some C_j of C; and this is not true for any earlier h_i. Then, by Lemma 4.2.1, i is the smallest index such that $C_j \succeq e$ and $C_j \succeq h_i$. Therefore, if DLearn includes e in h_i of H, then h_i is a correct hypothesis Horn clause for e.

Now, the if part of the claim. Let h_i be the correct hypothesis Horn clause for e in H, which means that there is no hypothesis Horn clause h_k for e such that $k < i$. Then there exists a C_j of C such that $C_j \succeq e$ and $C_j \succeq h_i$. This implies, by Lemma 4.2.1, that $C_j \succeq lgg(e, h_i)$. By Lemma 4.3.1, $\Sigma \models lgg(e, h_i)$. Also, for $k < i$, $C_j \not\succeq h_k$, which implies $\Sigma \not\models lgg(e, h_k)$. Therefore, h_i is the first clause in the hypothesis H for which $\Sigma \models lgg(e, h_i)$. Then, by lines 7–13 in Figure 4.1, e is included in h_i by assigning the result of Reduce($lgg(e, h_i)$) to h_i. □

Lemma 4.3.5 *The following are invariant conditions of* DLearn*:*

1. Every Horn clause h_i in the hypothesis H has a target clause;

2. Every Horn clause C_j in the target concept Σ has at most one hypothesis clause.

Proof.

Proof of (1). For every Horn clause h_i in H, $\Sigma \models h_i$. This is true because (a) the input h to Reduce is checked to be such that $\Sigma \models h$, (by lines 7–13 of DLearn), and (b) by Lemma 4.3.2 the output of Reduce, which replaces h_i, preserves this condition. Therefore, by Lemma 4.3.1, h_i has a target Horn clause.

Proof of (2). First, we show that any new hypothesis clause added to H has a target clause distinct from the target clauses of the other hypothesis clauses in H. Next, we show that if two hypothesis clauses have distinct target clauses at the beginning of an iteration of the loop of lines 4–14, then they still have distinct target clauses at the end of the iteration.

Let h_i be the first hypothesis Horn clause in H for C_j. That is, there is no h_k such that $k < i$ and h_k is a hypothesis Horn clause in H for C_j. Another hypothesis clause $h_{i'}$ with the target clause C_j would have been added to H such that $i' > i$, only if there was a counterexample e belonging to C_j for which h_i is not the correct hypothesis Horn clause (by Lemma 4.3.4). That means $C_j \succeq e$ and $C_j \not\succeq lgg(h_i, e)$. This implies, by Lemma 4.2.1, $C_j \not\succeq h_i$. That is a contradiction, because h_i is a hypothesis Horn clause for C_j. Therefore, such a $h_{i'}$ cannot exist in H. That is, $h_{i'}$ could have been added only if it had a distinct target clause.

Let h_i be the clause in H that changes during an iteration of the loop of lines 4–14. Further, let for any target clause C_j of any other clause $h_{i'}$ in H, $C_j \not\succeq h_i$. That is, C_j is not a target clause for h_i. We show that even after the iteration, C_j is not a target clause of h_i. h_i can change in the lines 8 and 13. We need to show that both these changes maintain that $C_j \not\succeq h_i$. Since $C_j \not\succeq h_i$, $C_j \not\succeq lgg(h_i, e)$

(by Lemma 4.2.1). Therefore, line 8 maintains the property. In line 13, Reduce returns a subset of its argument. By the contrapositive of Proposition 4.2.1, $C_j \not\succeq$ Reduce$(lgg(h_i, e))$, thus maintaining the property. Therefore, h_i and any other $h_{i'}$ have different target clauses at the end of the iteration. □

Now to the main theorem on the exact learnability.

Theorem 4.3.1 *Non-recursive Horn definitions are exactly learnable using equivalence and membership queries.*

Proof. We prove this theorem by showing that DLearn exactly learns non-recursive Horn definitions.

Part 1 of Lemma 4.3.5 implies that for every h_i of H, there is a C_j such that $C_j \succeq h_i$. That means, H covers no example that is not covered by the target concept C. In other words, H is never over-general in DLearn. Therefore, every counterexample is an example that is covered by Σ, but not by H.

Each equivalence query guarantees that whenever DLearn gets a new example, it is not already covered by the hypothesis H. At the end of each iteration, before asking an equivalence query, by Lemma 4.3.4, DLearn guarantees that all the previous examples are covered by H. Each example, either modifies an existing hypothesis Horn clause (its correct hypothesis Horn clause) or adds a new Horn clause. The minimum change in H that is required to cover a new example is a change of a variable in its correct hypothesis Horn clause if one exists. That is, each new example, except the ones that add new Horn clauses, contributes at least one variable. Let n be the number of Horn clauses in a concept, l be the maximum number of literals in a clause in the concept, v be the maximum number of variables in a clause in the concept, and k be the number of literals in the largest counterexample. Because Reduce guarantees that each Horn clause in the hypothesis has at most as many literals as there are in its target Horn clause (by Lemma 4.3.2 and Lemma 4.3.3),

the number of variables in each Horn clause is at most v. Lemma 4.3.5 guarantees that H has at most n Horn clauses. Therefore, the total number of variables is at most nv. DLearn requires n examples to add each of the n Horn clauses in H. It requires at most nv examples to variablize all the Horn clauses in H. Therefore, DLearn requires $n(v+1)$ examples, and, hence, $n(v+1)$ equivalence queries.

Let m be the number of hypothesis clauses in the hypothesis H at any time. Then, for each of the base examples that forms a new Horn clause in H, DLearn asks at most m membership queries for deciding that there is no correct hypothesis Horn clause in H, and at most k membership queries to simplify and generalize using Reduce (because there are at most k literals in an example). Each new Horn clause has at most l literals (by Lemma 4.3.2). For each of the other examples, at most m membership queries are needed to determine a correct hypothesis Horn clause, and kl (which is the size of the lgg) number of membership queries to generalize using Reduce. Therefore, the total number of queries is at most $mn + kn + nv(m + kl)$, which is at most $n^2 + kn + nv(n + kl)$. This is also an upper bound on the running time of the algorithm. □

By the above theorem and the transformation result from the exact learning model to the PAC model (Angluin, 1988), we have the following.

Corollary 4.3.2 *Non-recursive Horn definitions are polynomial-time PAC-learnable using membership queries.*

4.4 Learning D-rules via Learning Horn Definitions

In Chapter 3, we have seen how d-rules and example d-rules can be converted to Horn clauses. In this chapter, so far, we have studied the DLearn algorithm for

learning Horn definitions from Horn-clause examples. In the following, we examine how DLearn can be utilized to learn d-rules from example d-rules.

One problem in utilizing the algorithm for learning Horn definitions to the task of learning d-rules is that a d-rule hierarchy for a planning domain, when converted into Horn clause notation, is a Horn program rather than a Horn definition. That is, literals appearing as heads also appear in the bodies of clauses.

Since a hierarchy of d-rules is equivalent to a Horn program, to learn a hierarchy of d-rules, we need a way to learn Horn programs. As far as we know, excepting the work of Arimura (1997) which learns Horn programs, most ILP methods that guarantee correctness are directed towards learning Horn definitions. In Arimura's work, among other restrictions, the clauses in a Horn program are required to be "simple"—that is, only the terms in the head of a clause can appear in the body of the clause. This is too restrictive for our purposes, because objects that do not appear in goal literal (which corresponds to the head literal in the Horn-clause form) can be tested in the condition part of a d-rule (which corresponds to the body in the Horn-clause form). Here, instead, we use the Horn-definition learning algorithm to learn Horn programs. (In the next chapter, we will study an algorithm for learning Horn programs directly.)

In Horn programs, since the head literal of a clause can appear in the body of another clause, the resulting interactions between clauses make Lemma 4.3.1 inapplicable. Hence, the Horn-definition learner cannot be used directly to learn Horn programs. However, if clauses for each head literal are learned separately, assuming that the other clauses are known, we can use the Horn-definition learner. Thus, d-rules can be learned for each goal separately—i.e., assuming that d-rules for lower-level (sub)goals are already known.

However, there is a glitch here. There is a dependency among the `not-after` literals in that they are transitive: `not-after(?Si, ?Sj), not-after(?Sj, ?Sk)` $\rightarrow$ `not-after(?Si, ?Sk)`. This makes the Lemmas 4.3.1 and 4.3.2 inapplicable. Following an insight in our work on Horn programs (Reddy & Tadepalli, 1998a), also reported in the next chapter, we order the `not-after` literals in the input h of Reduce such that the literals that match `not-after(?Si, ?Sj)` and `not-after(?Sj, ?Sk)` come earlier than the literals that are implied by them, such as `not-after(?Si, ?Sk)`. Hence, Reduce considers the literals for removal in that order. In this way, the output of Reduce will be a condensation, as was the case without the not-after literals. The idea here is that, in Reduce, if we removed `not-after(?Si, ?Sk)`, before we removed `not-after(?Si, ?Sj)` and `not-after(?Sj, ?Sk)`, the membership query would be answered *yes*, and we would remove `not-after(?Si, ?Sk)`. This may cause Reduce to leave the two literals `not-after(?Si, ?Sj)` and `not-after(?Sj, ?Sk)`, instead of just `not-after(?Si, ?Sk)`. If this were to occur, the output of Reduce might neither be a condensation nor be subsumed by a clause in the target. Ordering the removal of literals in the above manner overcomes this problem.

Another point to note is that learning d-rules via learning Horn clauses by the DLearn algorithm gives us d-rules that have partially ordered subgoals. Once a d-rule with partial ordering of subgoals is learned, we can output a d-rule with any total ordering that is consistent with the partial ordering. (Insisting that this method learn d-rules with totally ordered subgoals as the target makes it an intractable method. This is so for the following reason. Horn-clause translations of d-rules allow partial orderings among subgoals. When a query is asked by a Horn-clause learning algorithm if it has to be answered by a teacher having d-rules with totally ordered subgoals as the target, every total ordering of the literals in the Horn-clause must

be checked for consistency with the teacher's d-rule. The number of total orderings for a partial ordering is exponential in the number of subgoals.)

Taking the above point into consideration, with the change needed for `not-after` literals, it follows from Theorem 4.3.1 that d-rules (partially ordered subgoals) can be learned from example d-rules and membership queries. In Chapter 6, however, we will employ a variation of this algorithm for learning d-rules from examples.

4.5 Discussion

In this work, we have shown that first-order non-recursive Horn definitions are learnable utilizing a reasonable amount of time, and, hence, a reasonable number of examples and queries. As a special case, it follows that first-order monotone disjunctive normal forms (monotone DNF) formulas are PAC-learnable from examples and membership queries.

The learning setting we have used is that of learning from entailment. There is another setting of learning called *learning from interpretations* (De Raedt, 1997). In the setting of learning from interpretations, positive examples are models of the target, and negative examples are negative interpretations. It has been shown that propositional Horn programs are exactly learnable from equivalence and membership queries, in the learning from interpretation setting (Angluin et al., 1992), as well as in the learning from entailment setting (Frazier & Pitt, 1993). When membership queries are available, learning from interpretations reduces to learning from entailment. This is because we can convert every negative interpretation of the target into a a positive example in the entailment model (a Horn clause entailed by the target). Since every negative interpretation violates some Horn clause, we can prune all but one negative literal in the interpretation, while making sure that the result is still a negative interpretation. We can then convert that reduced negative inter-

pretation into a positive Horn clause by negating it. (This is explained more fully in Section 5.4.) Thus, our results imply that Horn definitions are learnable from interpretations as well.

Most of the positive results for PAC-learnability in ILP, so far, depend either on polynomial-time transformations of first-order clauses to propositional logic and propositional PAC-learning algorithms (Džeroski et al., 1992; Cohen, 1995a; Page & Frisch, 1992), or on exhaustively enumerating polynomially sized hypothesis spaces (Frazier & Page, 1993). Therefore, the classes considered were very restrictive. In comparison, the hypothesis space for the language class we consider is unbounded. Because the arity of the predicates is not constant, converting the first-order target to a propositional one yields an exponentially large target, which requires an exponentially large number of equivalence queries to learn. The negative results by Cohen (1995c) for PAC-learning interesting classes suggest that membership queries are necessary. Along with our work, the work by Page (1993) and the work by Haussler (1989) are significant efforts in making learning tractable using membership and subset queries.

The algorithm in Figure 4.1 is similar in spirit to an ILP system called CLINT (De Raedt & Bruynooghe, 1992), in the sense that they both are incremental and interactive. Like in our algorithm, CLINT uses queries to eliminate irrelevant literals. CLINT raises the generality of hypotheses by proposing more complex hypothesis clauses, whereas our algorithm uses the lgg.

Several pieces of research have used the lgg idea in different ways for the purpose of generalization. Haussler (1989) considers learning first-order conjunctive concepts. In the hypothesis language considered by him where the number of objects per scene is fixed, the lgg of two hypotheses is at most as big as one of the hypotheses, but is not unique. Haussler uses queries to select an lgg which is in the target. On the

other hand, in our case, the *lgg* of a set of hypotheses is unique, but its size grows exponentially with the number of hypotheses. We use queries to reduce the size of the hypothesis generated by the *lgg* (see Reduce procedure in Figure 4.1). Frazier and Pitt (1995) also use a pruning procedure similar to our Reduce procedure to limit the size of the *lgg* in learning descriptions in CLASSIC. GOLEM (Muggleton & Feng, 1990) is another system that uses *lgg* to generalize hypotheses. GOLEM mitigates the problem of combinatorial explosion due to the *lgg* in two ways: (1) by restricting the hypothesis language to ij-determinate Horn clauses which guarantee polynomial-sized *lgg*s; and (2) by using negative examples to eliminate literals from the hypotheses. In the case of the work by Page (1993), the simplicity and the fixed-arity restrictions make the size of the *lgg* polynomial in the sizes of the hypotheses being generalized.

The work we described here is one of the first to theoretically demonstrate that polynomial-time first-order learning is more powerful than propositional learning. We believe that structural domains such as planning and scheduling are inherently relational and are better suited to first-order learning. However, to build practical systems, we need to generalize our results in many directions. These include learning more general Horn programs, handling noise, probabilities and real numbers, and incorporating background theories. We will address the learnability of Horn programs in the next chapter.

Chapter 5

LEARNING ACYCLIC HORN PROGRAMS

In Chapter 3, we have seen how d-rules for multiple goals and HTN methods for multiple tasks can be mapped to Horn programs. In Chapter 4, we have seen how we can apply the algorithm for learning Horn definitions to learn Horn programs. In this chapter, we present a direct method to learn Horn programs from examples and queries. In Chapter 7, we will study learning d-rules from unsolved problems which are ordered according to their levels of difficulty. There, we will study the connections between the algorithm presented here and the method introduced in that chapter.

5.1 Introduction

Learning first-order Horn programs is an important problem in inductive logic programming with applications ranging from speedup learning to grammatical inference.

Here, we consider learning first-order Horn programs in the setting of learning from entailment (Frazier & Pitt, 1993; De Raedt, 1997). In learning from entailment, a positive (negative) example is a Horn clause that is implied (not implied) by the target. Results by Cohen (1995a, 1995b), Džeroski et al. (1992) and others indicate that classes of Horn programs having a single or a constant number of clauses are learnable from examples. Khardon (1996) shows that "actions strategies" consisting of a variable number of constant-size first-order production rules can be learned from examples. However, Cohen (1995a) proves that even predicting very restricted

classes of Horn programs (viz. function-free 0-depth determinate constant arity) with variable number of clauses of variable size from examples alone is cryptographically hard.

Frazier and Pitt (1993) first used the entailment setting for learning arbitrary propositional Horn programs. In addition to examples, they also used entailment membership queries ("entailment queries" from now on) which ask if a Horn clause is entailed by the target. Moving to first order representations, Frazier and Pitt (1993) showed that CLASSIC sentences are exactly learnable in polynomial time from examples and entailment queries. Page (1993) considered non-recursive Horn programs restricted to simple clauses and predicates of constant arity, and showed that they are learnable from examples and entailment queries. A *simple clause* is a clause where only the variables and terms that occur in the head literal of a clause appear in the body of the clause. Arimura (1997) generalized Page's result to acyclic (possibly, recursive) simple Horn programs with constant-arity predicates. In the previous chapter, we showed that non-recursive Horn definitions are learnable from examples and entailment queries. The result we present in this chapter applies to non-generative Horn programs, where the variables and the terms in the head are restricted to those in the body. We show that acyclic non-generative Horn programs with constant arity that have a polynomial-time subsumption procedure are learnable from examples and entailment queries when certain closure conditions are satisfied. In particular, the result applies to acyclic Horn programs with constant-arity determinate clauses.

Goal-decomposition rules are hierarchical in nature, as are Horn programs. One aspect of learning in hierarchical domains is the hierarchical order of literals (goals or concepts). In many systems, learning hierarchically organized knowledge assumes that the structure of hierarchy or the order of the literals is known to the learner.

Examples of such work include Marvin (Sammut & Banerji, 1986) and our system LeXer in Chapter 7, on the experimental side; learning from exercises by Natarajan (1989) and learning acyclic Horn sentences by Arimura (1997), on the theoretical side. Our algorithm also assumes that the hierarchical order of the literals is known.

The rest of the chapter is organized as follows. Section 5.2 provides definitions for some of the terminology we use. Section 5.3 describes the learning model and the learning algorithm, and proves the learnability result. Section 5.4 concludes the chapter with some discussion on implications and limitations of the work.

5.2 Preliminaries

In this section, we define and describe some of the terminology we use in the rest of the chapter, in addition to the terminology we used in Chapter 3 and 4. In the following, we use capital letters P and A, and their variants each to stand for a conjunction of literals; and small-case letters b, q, l and their variants each to stand for a single literal.

Definition 5.2.1 *A* **derivation** *of a Horn clause $P \rightarrow q$ from a Horn program H is a finite directed acyclic graph G such that there is a node q, there is no arc (q, r) in G, and for each node l in G, either $l \in P$ or if $(l_1, l), \ldots, (l_d, l)$ are the only arcs of G terminating at l, then $l_1, \ldots, l_d \rightarrow l = C\theta$ for some clause $C \in H$ and a substitution θ.*

For example, let H be $\{parent(x, y), parent(y, z) \rightarrow grandParent(x, z);\ mother(x, y) \rightarrow parent(x, y)\}$. Figure 1 shows a derivation of $mother(a, b), mother(b, c) \rightarrow grandParent(a, c)$.

Proposition 5.2.1 *In a derivation G of a clause $P \rightarrow q$ from a Horn program H, for any node l, either l is in P or $H \models P \rightarrow l$.*

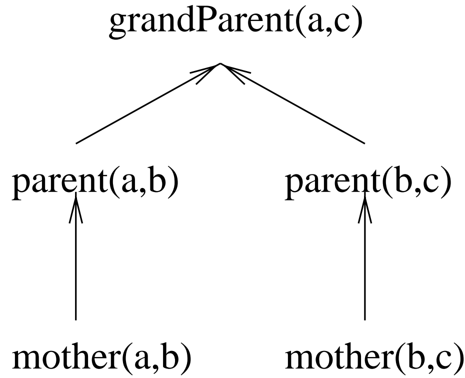

FIGURE 5.1: A derivation of $mother(a,b)$, $mother(b,c) \rightarrow grandParent(a,c)$ from H.

Let $\mathcal{P}$ be a set of predicate symbols, and $\mathcal{T}$ be a set of terms. Let $\mathcal{L}$ be a set of atoms defined using $\mathcal{P}$ and $\mathcal{T}$. Let $\mathcal{H}$ be a set of Horn programs using atoms in $\mathcal{L}$ only. If k is an integer, then $\mathcal{P}_k$ is a subset of P containing only those predicate symbols of arity k or less. Further, $\mathcal{L}_k$ is a set of atoms defined using $\mathcal{P}_k$ and $\mathcal{T}$, and $\mathcal{H}_k$ is a set of Horn programs using atoms in $\mathcal{L}_k$ only. In the following three definitions, we describe a class of Horn programs AH_k for which minimal models are of polynomial size.

Definition 5.2.2 (Arimura, 1997) *Let $\Sigma \in \mathcal{H}$. Then a binary relation* **supported by** *(denoted, $\succ$) over atoms in $\mathcal{L}$ w.r.t. Σ is such that (1) for all $P \rightarrow l \in \Sigma$, and for all $l_1 \in P$, $l \succ l_1$; (2) for all $l_1, l_2 \in L$ and every substitution θ, if $l_1 \succ l_2$, then $l_1\theta \succ l_2\theta$; and (3) if $l_1 \succ l_2$ and $l_2 \succ l_3$ then $l_1 \succ l_3$.*

Definition 5.2.3 *A Horn program Σ is* **acyclic** *over $\mathcal{L}$ if the relation $\succ$ over $\mathcal{L}$ w.r.t. Σ is terminating; i.e., for any $l \in \mathcal{L}$, there is no infinite decreasing sequence $l \succ l_1 \succ \ldots$.*

In the last example, H is acyclic because $grandParent(x,y) \succ parent(x,y) \succ mother(x,y)$ and there is no cycle formed by the $\succ$ relation.

Following Khardon (1998), we call a definite clause a *non-generative clause* if the set of terms in its consequent are a subset of the set of terms and subterms in its antecedent.

Definition 5.2.4 *If k is a constant, we define a Horn program $\Sigma \in \mathcal{H}_k$ to be in the class AH_k, if Σ is acyclic over $\mathcal{L}_k$, and each clause is either non-generative or has an empty antecedent.*

Definition 5.2.5 *Let $A \to b$ be a clause in a Horn program Σ, and $P \to q$ be a clause. Then, $A \to b$ is a* **target clause** *in Σ of $P \to q$ iff $A \to b \succeq P \to q$, i.e., for a substitution θ, $A\theta \subseteq P$, $b\theta = q$. We call $P \to q$ a* **hypothesis clause** *of $A \to b$.*

Definition 5.2.6 *For an antecedent P, q' is a* **prime consequent** *of P wrt Σ if $\Sigma \models P \to q'$, $q' \notin P$, and there is no $l \in \mathcal{L}$ such that $q' \succ l$, $\Sigma \models P \to l$ and $l \notin P$.*

In the last example, $parent(a,b)$ is a prime consequent of $mother(a,b)$, *mother*(b,c), but $grandParent(a,c)$ is not—since $grandParent(a,c) \succ parent(a,b)$.

5.3 Learning Horn Programs

In this section, we show that a subclass of AH_k is learnable, using the exact learning model (Angluin, 1988) in the entailment setting. Henceforth, $\Sigma \in AH_k$ denotes a target Horn program.

5.3.1 The Learning Model

In learning from entailment, an example is a Horn clause. An example $P \to q$ is a positive example of Σ if $\Sigma \models P \to q$; negative, otherwise. An *entailment query*

takes as input an example $(P \rightarrow q)$, and outputs *yes* if it is a positive example of Σ $(\Sigma \models P \rightarrow q)$, and *no* otherwise. An *equivalence query* takes as input a Horn program H and outputs *yes* if H and Σ contain (entail) exactly the same Horn clauses; otherwise, it returns a *counterexample* that is in (entailed by) exactly one of H and Σ. A *derivation-order query*, $\succ$, takes as input two atoms l_1 and l_2 in L and outputs *yes* if $l_1 \succ l_2$, and *no* otherwise. An algorithm *exactly learns* a Horn program Σ in AH_k in polynomial time from equivalence, entailment, and derivation-order ($\succ$) queries if and only if it runs in time polynomial in the size of Σ and in the size of the largest counterexample, and outputs a Horn program in AH_k such that the equivalence query answers *yes*.

5.3.2 The Learning Algorithm

In this section, we describe the learning algorithm, PLearn, shown in Figure 2. PLearn is similar to DLearn in that it always maintains a hypothesis H which is entailed by the target, so that every instance of H is also an instance of Σ and all counterexamples are positive. It is, however, more complicated than DLearn because the clauses in the target can be chained together in a derivation of a positive example.

Suppose that a counterexample $P \rightarrow q$ is given to the learner—see Figure 2. Every such counterexample has a derivation from the target theory, Σ. Since this derivation is not possible from the current hypothesis H, there is some clause used in the derivation that has not been learned with sufficient generality. The algorithm tries to identify the antecedent literals of such a clause, $c*$, in the target by expanding the derivation graph from its leaves in P toward the goal using the clauses in H. In other words, PLearn computes the minimal model (P'_f) of H implied by P ("closure" or "saturation") by forward chaining (line 4). To identify the consequent of $c*$, also called the "prime consequent" of P'_f, PLearn calls PrimeCons in line 5. PrimeCons

finds the prime consequent of P'_f by tracing the "supported-by" chain starting from q for a literal q_f not in P'_f, but that is directly supported by some of the literals in P'_f (lines 13–18). In line 6, PLearn makes use of Reduce to trim away "irrelevant" literals from the antecedent P'_f to form a new clause $P_f \rightarrow q_f$ that is also a counterexample to the hypothesis and is subsumed by a single target clause—see Lemmas 5.3.8, 5.3.1, and 5.3.2.

PLearn combines $P_f \rightarrow q_f$ with an "appropriate" clause $P_i \rightarrow q_i$ in H by computing the lgg (lines 7–9). It uses the entailment query to find an appropriate hypothesis clause by checking if the lgg is implied by the target (line 7). If no such clause exists in H, $P_f \rightarrow q_f$ is appended to H as a new clause (line 10).

One problem with this approach is that the size of the lgg is a product of the sizes of its two arguments. This causes the size of a hypothesis clause to grow exponentially in the number of examples combined with it in the worst case. We use a solution similar to the one employed by DLearn to avoid this. The antecedent literals of the clause after computing the lgg are again trimmed using Reduce so that the size of the resulting clause is bounded, while it is still subsumed by the target clause (lines 19–25). The result of Reduce then replaces the original hypothesis clause $P_i \rightarrow q_i$ it is derived from (line 9). After this step, only the antecedents of the target clause and some of their consequents remain in the resulting hypothesis clause—see Lemma 5.3.4. This process repeats until the hypothesis H is equivalent to Σ. The algorithm works for unit clauses (which have empty antecedents) without change.

5.3.3 An Example

As an example to see how PLearn works, consider $\Sigma = \{l_1(f(?x)), l_2(?x), l_3(?x) \rightarrow l_4(?x); l_1(f(?x)), l_2(?x) \rightarrow l_5(?x); l_4(?x), l_5(?x) \rightarrow l_7(?x)\}$ where f is a function symbol. Suppose $H = \{l_1(f(c)), l_2(c) \rightarrow l_5(c)\}$. Let the counterexample be

```
PLearn
Given equivalence, entailment queries, and the derivation order ≻
outputs a Horn program H such that equivalent?(H, Σ) is Yes.
(1)  H := {} /* empty hypothesis-clauses set */
(2)  while not equivalent?(H, Σ) do {
(3)      Let P → q be the counterexample returned
(4)      P'_f := {l : H ⊨ (P → l)} /* forward chaining */
(5)      q_f := PrimeCons(P'_f → q)
(6)      P_f → q_f := Reduce(P'_f → q_f)
(7)      if ∃P_i → q_i ∈ H such that Σ ⊨ P_g → q_g,
(8)                      where P_g → q_g is lgg(P_i → q_i, P_f → q_f)
(9)      then replace first such P_i → q_i by Reduce(P_g → q_g)
(10)     else append P_f → q_f to H
(11) } /* while */
(12) return H

PrimeCons(P → q) /* finds prime consequents */
(13) Let L be the set of all possible literals having only those terms that are in P
(14) q' := q;
(15) L' := {l : l ∈ L − P and Σ ⊨ P → l}
(16) while ∃l ∈ L' such that q' ≻ l
(17)     q' := l;
(18) return q'

Reduce(P → q) /* trims irrelevant literals */
(19) P' := P
(20) repeat
(21)    for each literal l in P' in sequence do
(22)       if Σ ⊭ (P' − {l}) → l and Σ ⊨ (P' − {l}) → q
(23)       then P' := P' − {l}
(24) until there is no change to P'
(25) return P' → q
```

FIGURE 5.2: PLearn Algorithm

$l_1(f(d)), l_2(d), l_3(d) \to l_7(d)$. In step 4, this clause does not change. In PrimeCons, since $l_7(d) \succ l_4(d)$ and $l_7(d) \succ l_5(d)$, $l_7(d)$ is not a prime consequent, but any one of $l_4(d)$ and $l_5(d)$ is. Suppose PrimeCons returns $l_5(d)$. Reduce eliminates $l_3(d)$ from the antecedent, because $\Sigma \models l_1(f(d)), l_2(d) \to l_5(d)$, and $\Sigma \not\models l_1(f(d)), l_2(d) \to l_3(d)$. Thus, $P_f \to q_f = l_1(f(d)), l_2(d) \to l_5(d)$. Combining this with the clause in H, we obtain $P_g \to q_g = l_1(f(?x)), l_2(?x) \to l_5(?x)$. Since $l_1(f(?x)), l_2(?x) \to l_5(?x)$ is entailed by Σ, the new H is $\{l_1(f(?x)), l_2(?x) \to l_5(?x)\}$.

Suppose the next counterexample is $l_1(f(c)), l_2(c), l_3(c) \to l_7(c)$. Then, $q_f = l_4(c)$, and $P'_f = \{l_1(f(c)), l_2(c), l_3(c), l_5(c)\}$. $P_f \to q_f = l_1(f(c)), l_2(c), l_3(c), l_5(c) \to l_4(c)$, since Reduce cannot remove $l_5(c)$, because it is implied by the other literals wrt Σ (line 22). The modified counterexample $P_f \to q_f$ cannot be combined with the clause in H, because the resultant $P_g \to q_g$ after lgg, $l_1(f(?x)), l_2(?x) \to$, is not entailed by Σ. Hence, it is appended to H to make $H = \{l_1(f(?x)), l_2(?x) \to l_5(?x); l_1(f(c)), l_2(c), l_3(c), l_5(c) \to l_4(c)\}$.

Suppose the next counterexample is again $l_1(f(c)), l_2(c), l_3(c) \to l_7(c)$. After line 4, $P'_f = \{l_1(f(c)), l_2(c), l_3(c), l_5(c), l_4(c)\}$. q_f now is $l_7(c)$, because it is a prime consequent of P'_f. After Reduce, $P_f = l_5(c), l_4(c)$. $P_f \to q_f$ cannot be combined with the clauses in H, because the resultant lgg's are not entailed by Σ. Again, $P_f \to q_f$ is added to H. This process continues until H and Σ are equivalent.

To bring out the nuances in Reduce, let us revisit the last part of the previous example. Consider the input $l_5(c), l_2(c), l_3(c), l_1(f(c)), l_4(c) \to l_7(c)$ to Reduce. Although $\Sigma \models l_1(f(c)), l_2(c), l_3(c) \to l_7(c)$, since $\Sigma \models l_1(f(c)), l_2(c), l_3(c) \to l_4(c)$ and $\Sigma \models l_1(f(c)), l_2(c) \to l_5(c)$, the literal $l_5(c)$ cannot be removed. This is because $l_5(c)$ is implied by the other literals $(l_1(f(c)), l_2(c))$ w.r.t Σ. The order in which the literals are removed in Reduce follows the derivation order: if $l_i \succ l_j$, then if l_i must be removed, it is removed after l_j is removed. This can be intuitively imagined in the fol-

lowing way. Consider a derivation tree for a counterexample, with the consequent literal on top and the antecedent literals at the bottom. The above process trims off the literals bottom-up in the tree up to the appropriate level, so that the resulting clause is subsumed by some clause in the target. In the above case, if Reduce removes $l_5(c)$ and leaves over $l_1(f(c)), l_2(c)$, the resulting clause $(l_1(f(c)), l_2(c), l_3(c), l_4(c) \rightarrow l_7(c))$ is not subsumed by any clause in Σ.

However, this means that Reduce leaves literals which are implied by the remaining literals, i.e., l cannot be removed from P' if $\Sigma \models (P' - \{l\}) \rightarrow l$ (line 22). Removing such literals could result in hypothesis clauses which are not subsumed by any target clause, as the following example illustrates. Let Σ be $\{l_1(a) \rightarrow l_2(a);$ $l_1(?x), l_2(?x) \rightarrow l_3(?x)\}$. Suppose the first counterexample is $l_1(a), l_2(a) \rightarrow l_3(a)$. Hence $P'_f = \{l_1(a), l_2(a)\}$ and $q_f = l_3(a)$ in line 6. If Reduce were to remove $l_2(a)$ from P'_f because $\Sigma \models l_1(a) \rightarrow l_3(a)$, it would end up with a clause that is not subsumed by any target clause. We would like to prevent such redundant hypothesis clauses so that their number is not too high compared to the number of target clauses. (This argument is formalized in Lemmas 5.3.5, 5.3.6 and 5.3.7.)

5.3.4 Learnability of AH_k

In this section, we prove that the PLearn algorithm in Figure 5.2 exactly learns a subclass of AH_k for which subsumption is of polynomial-time complexity. The plan of the proof is as follows: Through a series of lemmas, we first establish that every hypothesis clause learned has a target clause (Lemma 5.3.5). We then show that every target clause has at most one hypothesis clause (Lemma 5.3.7). Together, these two lemmas establish that the number of hypothesis clauses is bounded by the number of target clauses. We use this fact and the bounds of the sizes on the hypothesis clauses (established in Lemma 5.3.4) to show that PLearn learns successfully in poly-

nomial time (Theorems 5.3.1 and 5.3.2). We then define a specific hypothesis class that obeys the conditions of these theorems and prove that this class is learnable (Theorem 5.3.3).

Lemmas 5.3.1 and 5.3.2 show that PrimeCons with the input $P \to q$ finds a (prime) consequent q' of P such that $P \to q'$ is subsumed by a clause in Σ.

Lemma 5.3.1 *Let $P \to q$ be the input and q' be the output of* PrimeCons*. Assume that $q \notin P$ and $\Sigma \models P \to q$. Then, (1)* PrimeCons *terminates; (2) q' is a prime consequent of P wrt Σ.*

Proof. (1) Since Σ is acyclic, there is a terminating sequence $q \succ l_1 \succ l_2 \ldots$. Since the loop of lines 16–17 can only iterate as many times as the length of the sequence, PrimeCons terminates. (2) q' is such that $\Sigma \models P \to q'$, and $q' \notin P$ (by lines 15–17). Since q' is as in line 17 in the iteration immediately prior to the terminating iteration of lines 16–17, there is no l such that $q' \succ l$, $\Sigma \models P \to l$ and $l \notin P$. Thus, q' is a prime consequent of P wrt Σ. □

Lemma 5.3.2 *If q' is a prime consequent of P wrt Σ, then there is a clause $C \in \Sigma$ such that $C \succeq P \to q'$.*

Proof. Assume that q' is a prime consequent of P wrt Σ. Consider a derivation G of $P \to q'$ in Σ. Let $(l_1, q'), \ldots, (l_d, q')$ be the only arcs of G that terminate at q'. This implies that $q' \succ l_i$ for all $l_i \in \{l_1, \ldots, l_d\}$. It must be that every l_i is in P; otherwise, there is an l (viz. l_i) such that $q' \succ l$, $\Sigma \models P \to l$ and $l \notin P$—contradicting the assumption that q' is a prime consequent. Thus, $\{l_1, \ldots, l_d\} \subseteq P$. But, $l_1, \ldots, l_d \to q' = C\theta$ for some clause $C \in H$ and a substitution θ, following the definition of derivation. Thus, $C\theta \subseteq P \to q'$, implying that $C \succeq P \to q'$. □

The following definition and Lemmas 5.3.3 and 5.3.4 help show that Reduce, given a clause $P \rightarrow q$ as input, removes irrelevant literals from antecedent P, while maintaining q as a consequent.

Definition 5.3.1 *If A is a conjunction,* **closure** *of A with respect to Σ, denoted by κ_A, is defined as $\{l|\Sigma \models (A \rightarrow l)\}$.*

Lemma 5.3.3 *If q is a prime consequent of P w.r.t. Σ and $P' \rightarrow q =$ Reduce$(P \rightarrow q)$, then q is a prime consequent of P' also.*

Proof. Suppose that q is not a prime consequent of P'. This implies that there is another literal $q' \notin P'$ so that $\Sigma \models P' \rightarrow q'$ and $q \succ q'$. Consider a derivation of $P' \rightarrow q'$ that does not contain q. Since $q \succ q'$, there must be such a derivation. If q' is not in P, then q would not have been a prime consequent of P, since $q \succ q'$, and $\Sigma \models P \rightarrow q'$. If $q' \in P$, however, it would still have been in P', since $\Sigma \models P' \rightarrow q'$, and, by lines 22-23, only those literals that are not supported by P' are removed. Since we have a contradiction that neither $q' \in P$ nor $q' \notin P$, q must be a prime consequent of P'. □

Lemma 5.3.4 *If the input $P \rightarrow q$ to* Reduce *is s.t. q is a prime consequent of P wrt Σ, then the output $P' \rightarrow q$ is such that $P' \subseteq \kappa_{A\theta}$ where $A \rightarrow b$ is a clause in Σ and $A\theta \subseteq P'$ and $b\theta = q$.*

Proof. Since q is a prime consequent of P, by Lemma 5.3.3, q is a prime consequent of P' also. Then, by Lemma 5.3.2, there is a clause $A \rightarrow b \in \Sigma$, and a θ such that $A\theta \subseteq P'$ and $b\theta = q$. We now show that $P' \subseteq \kappa_{A\theta}$. Assume that there exists a literal in $P' - \kappa_{A\theta}$. Let $l \in P' - \kappa_{A\theta}$ be a least such literal so that there is no literal l' in $P' - \kappa_{A\theta}$ such that $l \succ l'$. Such a literal must exist, because Σ is acyclic. There are two reasons for l to remain in $P' - \kappa_{A\theta}$: either (a) $\Sigma \not\models (P' - \{l\}) \rightarrow q$ or (b)

$\Sigma \models (P' - \{l\}) \rightarrow l$. We disprove both the cases: (a) Since $A\theta \subseteq P'$, and l is not in $\kappa_{A\theta}$ and thus not in $A\theta$, $A\theta \subseteq (P' - \{l\})$. Therefore, $\Sigma \models (P' - \{l\}) \rightarrow q$. (b) The only other reason why l remains in P' is that $\Sigma \models (P' - \{l\}) \rightarrow l$. That means that $P' - \{l\}$ contains literals that imply l. There must be at least one such literal l' in P' that is not in $\kappa_{A\theta}$, or else $l \in \kappa_{A\theta}$, contradicting $l \in P' - \kappa_{A\theta}$. But then $P' - \kappa_{A\theta}$ contains literals l' such that $l \succ l'$, which contradicts the statement that there is no such l'. Thus, we disprove both the possibilities. Hence, $P' \subseteq \kappa_{A\theta}$. □

Lemmas 5.3.5, 5.3.6 and 5.3.7, below, show that PLearn only maintains correct clauses in H.

Lemma 5.3.5 *Every clause $P_i \rightarrow q_i \in H$ has a target clause.*

Proof. We first show that each $P_i \rightarrow q_i \in H$ is such that q_i is a prime consequent of P_i. Then, by Lemma 5.3.2, $P_i \rightarrow q_i$ has a clause $C \in \Sigma$ such that $C \succeq P_i \rightarrow q_i$.

We show that q_i is a prime consequent of P_i by induction on the number of times a clause at position i in H is updated. It is first introduced by line 10. By Lemmas 5.3.1 and 5.3.3, q_f is a prime consequent of P_f. This proves the base case. The other way a clause becomes a hypothesis clause is by line 9. The clause at position i in H ($P_i \rightarrow q_i$) is updated by line 9. As inductive hypothesis, assume that each $P_i \rightarrow q_i$ in H is such that q_i is a prime consequent of P_i, at the beginning of an iteration of the loop of lines 2–11 when position i in H is updated. Consider $P_g \rightarrow q_g = lgg(P_i \rightarrow q_i, P_f \rightarrow q_f)$. Suppose q_g is not a prime consequent of P_g, but q'_g such that $q_g \succ q'_g$ is. Let θ_f and θ_i be substitutions such that $P_g\theta_f \subseteq P_f$, $q_g\theta_f = q_f$, $P_g\theta_i \subseteq P_i$, and $q_g\theta_i = q_i$. Let $q'_f = q'_g\theta_f$ and $q'_i = q'_g\theta_i$. Since $q_g \succ q'_g$, by the definition of $\succ$ order, $q_f \succ q'_f$ and $q_i \succ q'_i$. Since q_f is a prime consequent of P_f, q'_f must be in P_f. Similarly, q'_i must be in q_i. Therefore, $lgg(q'_i, q'_f) = q'_g$ must be in P_g, contradicting the assumption that q'_g is a prime consequent of P_g. Hence, q_g is a

prime consequent of P_g. By Lemma 5.3.3 if $P_i \rightarrow q_i = \mathsf{Reduce}(P_g \rightarrow q_g)$, then q_i is a prime consequent of P_i. So by Lemma 5.3.2, $P_i \rightarrow q_i$ has a target clause. □

Lemma 5.3.6 *If* PLearn *combines a modified counterexample $P_f \rightarrow q_f$ with a clause $P_i \rightarrow q_i \in H$, then there is a target clause C s.t. $C \succeq P_f \rightarrow q_f$ and $C \succeq P_i \rightarrow q_i$. Further, there is no C' s.t. $C' \succeq P_j \rightarrow q_j$ and $C' \succeq P_f \rightarrow q_f$, for any $j < i$.*

Proof. PLearn combines $P_f \rightarrow q_f$ with $P_i \rightarrow q_i$ only if $\Sigma \models lgg(P_i \rightarrow q_i, P_f \rightarrow q_f)$. By Lemma 5.3.5, q_g is a prime consequent of P_g where $P_g \rightarrow q_g = lgg(P_i \rightarrow q_i, P_f \rightarrow q_f)$. By Lemma 5.3.2, there is a $C \in \Sigma$ such that $C \succeq P_g \rightarrow q_g$. Hence, $C \succeq P_i \rightarrow q_i$ and $C \succeq P_f \rightarrow q_f$. Since $P_f \rightarrow q_f$ is combined with $P_i \rightarrow q_i$, for any $j < i$, $\Sigma \not\models lgg(P_j \rightarrow q_j, P_f \rightarrow q_f)$. Therefore, there is no C' s.t. $C' \succeq lgg(P_j \rightarrow q_j, P_f \rightarrow q_f)$. Thus, there is no C' s.t. $C' \succeq P_j \rightarrow q_j$ and $C' \succeq P_f \rightarrow q_f$. □

Lemma 5.3.7 *Every clause $C \in \Sigma$ has at most one hypothesis clause.*

Proof. First, we show that any new hypothesis clause added to H has a target clause distinct from the target clauses of the other hypothesis clauses in H. Next, we show that if two hypothesis clauses do not have common target clauses at the beginning of an iteration of the loop of lines 2–11, then they still do not have common target clauses at the end of the iteration.

When $P_f \rightarrow q_f$ is added to H, by Lemma 5.3.6, for any clause H_i in H, there is no $C \in \Sigma$ such that $C \succeq H_i$ and $C \succeq P_f \rightarrow q_f$. Therefore, $P_f \rightarrow q_f$, a new clause added to H, has a target clause distinct from the target clauses of the other hypothesis clauses then in H. Next, at most one of H_i and H_j can change in an iteration of the loop. If neither changes, we are done with the proof. Suppose that H_i changes, without loss of generality. Let C be any target clause of H_j. Assume that H_i and H_j do not have a common target clause at the beginning of an iteration. Hence, C

is not a target clause of H_i. That is, $C \not\succeq H_i$. Let e be the counterexample for the current iteration. We first show that $lgg(H_i, e)$ does not have C as a target clause. Since $C \not\succeq H_i$, $C \not\succeq lgg(H_i, e)$. Therefore, C is not a target clause of $lgg(H_i, e)$. Let $lgg(H_i, e)$ be $P_g \rightarrow q_g$, and C be $A \rightarrow b$. Hence, for every θ, either $A\theta \not\subseteq P_g$ or $b\theta \neq q_g$. If $A\theta \not\subseteq P_g$, $A\theta$ is not a subset of any subset of P_g. Since Reduce outputs a clause with a subset of P_g as the antecedent and q_g as the consequent, $C \not\succeq$ Reduce$(lgg(H_i, e))$. Therefore, H_j and the new clause in position i, Reduce$(lgg(H_i, e))$, do not have a common target clause even at the end of the iteration. □

The following lemma shows that even after the modifications due to PrimeCons and Reduce, a counterexample remains a counterexample.

Lemma 5.3.8 *$P_f \rightarrow q_f$ as in line 6 of* PLearn *is a positive counterexample.*

Proof. First, we show that every counterexample $P \rightarrow q$, as in line 3, is a positive counterexample. Then, we argue that $P'_f \rightarrow q_f$ (lines 4 and 5) is also a positive counterexample. Finally, we show that $P_f \rightarrow q_f$ (line 6) is a positive counterexample.

Since, by Lemma 5.3.5, for every $H_i \in H$, there is a clause $C \in \Sigma$ such that $C \succeq H_i$, $\Sigma \models H$. Therefore, $P \rightarrow q$, as in line 3, is a positive counterexample. Since $P \subseteq P'_f$, $\Sigma \models P'_f \rightarrow q$. Since P'_f contains all and only those literals l such that $H \models P \rightarrow l$, for any literal $l' \notin P'_f$, $H \not\models P'_f \rightarrow l'$. Since q_f (by lines 5 and 15) is not in P'_f, $H \not\models P'_f \rightarrow q_f$. By line 15, $\Sigma \models P'_f \rightarrow q_f$. Therefore, $P'_f \rightarrow q_f$ is also a positive counterexample. Finally, since $P_f \subseteq P'_f$, $H \not\models P_f \rightarrow q_f$. By lines 6 and 22, $\Sigma \models P_f \rightarrow q_f$. Thus, $P_f \rightarrow q_f$ is a positive counterexample. □

Finally, Theorem 5.3.1 shows that PLearn exactly learns AH_k when forward chaining using H is of polynomial-time complexity. Theorem 5.3.2 identifies conditions on Σ such that PLearn returns an H for which the time complexity of forward chaining is polynomial.

Theorem 5.3.1 PLearn *exactly learns* AH_k *with equivalence,* $\succ$*, and entailment queries, provided that determining* $H \models P \rightarrow l$ *is polynomial in the sizes of* H *and* P*.*

Proof. By Lemma 5.3.8, $P_f \rightarrow q_f$ is a positive counterexample. For each counterexample, either a new antecedent is added (line 10) or an existing antecedent is replaced (line 9). In the latter case, the replaced clause $P_i \rightarrow q_i$ must be subsumed by the replacing clause $P' \rightarrow q_g$, since both lgg and Reduce generalize the original clause by turning constants to variables and dropping literals. On the other hand, the replaced clause must not subsume (and hence be different from) the replacing clause $P' \rightarrow q_g = \mathsf{Reduce}(P_g \rightarrow q_g)$. If not, that is if $P_i \rightarrow q_i \succeq P' \rightarrow q_g$, since $P' \rightarrow q_g \succeq P_g \rightarrow q_g \succeq P_f \rightarrow q_f$, $P_i \rightarrow q_i \succeq P_f \rightarrow q_f$. Since $P_i \rightarrow q_i \in H$, $H \models P_f \rightarrow q_f$—thus contradicting that $P_f \rightarrow q_f$ was a counterexample of H. Hence, the replacement at a position in H changes the clause at that position. The minimum change there can be is either a variablization of a constant or a removal of a literal.

Let n be the number of clauses, and s be the number of distinct predicate symbols in Σ. Further, let the maximum number of terms in any clause be t, and in any counterexample be t_e.

Since k is the maximum arity of the predicates in Σ, the maximum possible number of literals there can be using t terms is at most st^k. Hence, the maximum number of literals in κ_a, and therefore, by Lemmas 5.3.4 and 5.3.5, in each clause is at most st^k. This includes all literals and their variablized versions. Hence, we can consider variablization as removing a literal. Thus, we need at most st^k counterexamples for each clause. (This includes one base counterexample to introduce a clause into H.) By Lemmas 5.3.5 and 5.3.7, there are at most n clauses in H. Hence, we need at most nst^k counterexamples or equivalence queries. A call to PrimeCons from line 5

takes at most st_e^k entailment queries, because the literals we need to try as possible consequents are all in L, and $|L| \leq st_e^k$. PrimeCons is called once for each of the counterexamples.

For each of the nst^k counterexamples, the condition in line 7 is tested at most n times, which needs at most n entailment queries. Reduce is called with the argument $P'_f \rightarrow q_f$ once for each of the counterexamples, and with the arguments $P_g \rightarrow q_g$ for at most nst^k counterexamples. In Reduce$(P \rightarrow q)$, in $|P|$ iterations of the loop of lines 21–23, at least one literal is removed. So, this loop can be tried at most $|P|$ times. Each iteration of the loop of lines 21–23 takes two entailment queries. Therefore, Reduce$(P \rightarrow q)$ needs at most $|P|(|P|+1)$ entailment queries. Hence, Reduce$(P'_f \rightarrow q_f)$ needs at most $n_f = st_e^k(st_e^k+1)$ entailment queries. Since $P_i \rightarrow q_i$ and $P_f \rightarrow q_f$ are outputs of Reduce, the maximum possible number of literals in $P_g \rightarrow q_g = lgg(P_i \rightarrow q_i, P_f \rightarrow q_f)$ is at most s^2t^{2k}. Hence, Reduce$(P_g \rightarrow q_g)$ needs at most $n_g = s^2t^{2k}(s^2t^{2k}+1)$ entailment queries. Thus, the total number of entailment queries is at most $nst^k(st_e^k + n + n_f + n_g)$.

If determining $H \models (P \rightarrow l)$ takes $\mathcal{P}(n, l, t_e)$ time where $\mathcal{P}$ is a polynomial, then line 4 takes at most $st_e^k \cdot \mathcal{P}(n, l, t_e)$ time. In the rest, the number of entailment queries dominates the time. Hence, the time taken by PLearn is polynomial in n, s, l, v, t, and t_e. □

Definition 5.3.2 *Let $P \rightarrow q$ be a Horn clause. $P' \rightarrow q$ is called its* **antecedent expansion** *if $P \subseteq P'$ and P' contains only those variables in P. A class $\mathcal{C}$ of Horn sentences is closed under antecedent expansion, if every Horn sentence obtained by selecting a subset of its Horn clauses and replacing them with their antecedent expansions is also in C.*

Definition 5.3.3 *A* **subsumption algorithm** *takes a clause $A \rightarrow b$, a conjunction of literals P, and a ground substitution θ for the variables in b, and returns* true *if and only if $A\theta \succeq P$.*

Theorem 5.3.2 PLearn *exactly learns a subclass $\mathcal{C}$ of AH_k with equivalence, $\succ$, and entailment queries, provided that (a) $\mathcal{C}$ is closed under substitution and antecedent expansion and (b) the clauses $A \rightarrow b$ of the target concepts in $\mathcal{C}$ have a polynomial-time subsumption algorithm.*

Proof. By Lemma 5.3.4, each clause $P_i \rightarrow q_i \in H$ in PLearn has a target clause $A \rightarrow b$ and a substitution θ such that $A\theta \subseteq P_i \subseteq \kappa_{A\theta}$. Since the target class is closed under substitution and antecedent expansion, the hypothesis clauses have a polynomial-time subsumption algorithm. Hence, the forward-chaining step of computing the consequents of P in line 4 of PLearn can be done in polynomial time by repeatedly checking for a hypothesis clause $A \rightarrow b$ whose antecedent subsumes P after a substitution θ of the variables in b, and adding $b\theta$ to P. Hence, by the previous theorem, PLearn exactly learns $\mathcal{C}$. □

The following definition and theorem identify some syntactic restrictions on AH_k such that the resulting subclass satisfies the conditions of the previous theorem.

Definition 5.3.4 *Let P be a set of literals. A Horn clause $l_1, \ldots, l_n \rightarrow q$ is* ***i*-determinate** *w.r.t. P iff there exists an ordering $l_{o_1}, \ldots, l_{o_n}$ of $l_1, \ldots, l_n$ such that for every $i < j \leq n$ and every substitution θ such that $(l_{o_1}, \ldots, l_{o_{j-1}} \rightarrow q)\theta$ is ground and $\{l_{o_1}, \ldots, l_{o_{j-1}}\}\theta \subseteq P$, there is at most one substitution α for the variables in $l_{o_j}\theta$ such that $l_{o_j}\theta\alpha$ is ground and is in P.*[*] *We call such an ordering of the literals in*

[*] This definition strictly generalizes the standard definition of determinacy (Muggleton & Feng, 1990), in that a Horn clause (program) is determinate w.r.t. a set of literals p when it is 0-determinate w.r.t. P. i-determinacy should not be confused with ij-determinacy, or constant-depth fixed-arity determinacy, which is more restricted than determinacy.

*the clause an i-***determinate ordering** *w.r.t. P. A Horn program is i-determinate w.r.t. P iff each of the clauses in the program is i-determinate w.r.t. P.*

Theorem 5.3.3 *The class of i-determinate Horn programs in AH_k, denoted as $iDetAH_k$, is exactly learnable with equivalence, $\succ$, and entailment queries.*

Proof. First we show that $iDetAH_k$ is closed under substitution and antecedent expansion. Consider a target clause $(l_1, \dots, l_n \rightarrow q)$ for a target program in $iDetAH_k$, whose antecedent literals are sorted in the determinate order. Let $(l_1, \dots, l_n, l_{n+1}, \dots, l_m \rightarrow q)\beta$ be the target clause after antecedent expansion and substitution. We want to show the new clause to be i-determinate.

For every set of literals P, substitution θ, and j such that $i < j \leq m$ and $(l_1, \dots, l_{j-1})\beta\theta \subseteq P$ is ground, there is a substitution γ which is equivalent to applying β and θ one after another so that $(l_1, \dots, l_{j-1})\beta\theta = (l_1, \dots, l_{j-1})\gamma$ and $l_j\beta\theta = l_j\gamma$ for any l_j. Since the target clause satisfies i-determinacy, there must be at most a single ground substitution α for $l_j\gamma$, $j \leq n$, so that $l_j\gamma\alpha \in P$, which means that this is true for $l_j\beta\theta$ as well. Since the literals from l_{n+1} through l_m do not have any variables not already in l_1 through l_n, there is at most a single ground substitution for them as well. Hence, $(l_1, \dots, l_m \rightarrow q)\beta$ is also i-determinate.

Now we show that the clauses of the programs in $iDetAH_k$ have a polynomial-time subsumption algorithm. Given a set of literals P and a clause $l_1, \dots, l_n \rightarrow q$ (whose literals have an unknown determinate ordering), consider all possible subsets of $\{l_1, \dots, l_n\}$ of size i and less. Note that there are at most $O(n^i)$ such subsets. For each such subset, instantiate all the ki variables in that subset in all possible ways. If the total number of terms in p and Σ is t, this gives us t^{ki} different substitutions. For each such substitution, there is at most one substitution for the remaining literals in the clause. The order in which the remaining literals have to be substituted can be

determined by sequential search—apply the current substitution to each literal and pick the one that only allows one possible substitution for its remaining variables. This can be done in $O(n^2|P|)$ time. If the antecedent $l_1, \ldots, l_n$ subsumes P, then one of the considered subsets should yield a successful match. Hence, the total time for the algorithm is bounded by $O(n^i t^{ki} n^2 |P|)$, which is polynomial in all variables except k and i which are assumed to be constants.

Since the class $iDetAH_k$ satisfies the two conditions required by Theorem 5.3.2 for PLearn to be successful, the result follows. □

5.4 Discussion and Conclusions

In this chapter, we have shown the learnability of certain subclasses of acyclic k-ary Horn programs. More specifically i-determinate Horn programs in AH_k, are exactly learnable with equivalence and entailment queries. Unlike the work of Page (1993) and Arimura (1997), the programs we considered allow local variables in the antecedents. However, the clauses must be non-generative in that the set of terms and variables that occur in the head of the clause must be a subset of those that occur in the body of the clause. This is needed to constrain the forward-chaining inference step to finish in polynomial time; otherwise, it could become unbounded. It appears that simultaneously removing both the non-generative and simplicity restrictions could be difficult when functions are present, due to the unbounded nature of inference in that case.

Learning from entailment and *learning from interpretations* are two of the standard settings for first-order learning (De Raedt, 1997). In learning from interpretations, the learner is given a positive (or negative) interpretation for which the Horn sentence is true (or false). Interpretations can be partial in that the truth values of some ground atoms may be left unspecified. When membership queries are avail-

able, learning from entailment and learning from interpretations are equivalent for Horn programs. Hence we can use PLearn to learn from (negative) interpretations as follows. Given a negative interpretation, "minimize" it by removing the negative literals from it and asking membership queries. Since every negative interpretation must violate some Horn clause, this yields an interpretation with a set of positive literals $l_1, \ldots, l_n$ and at most one negative literal q_i. We can convert this into a positive counterexample for PLearn: $l_1 \wedge \ldots \wedge l_n \rightarrow q_i$. Similarly, if PLearn asks an entailment membership query on some clause, say, $l_1 \wedge \ldots \wedge l_n \rightarrow q_i$, we can turn that into a membership query on the interpretation $l_1, \ldots, l_n, \neg q_i$ after substituting a unique skolem constant for each variable in the clause. The answer to the entailment query is *true* iff the answer to the membership query is *false*.

One limitation of our algorithm is that it assumes that the *supported by* relation, $\succ$, is given. While this is a reasonable assumption in some planning domains, where it is known which goals occur as subgoals of which, it is desirable to learn this relation. Unfortunately, this seems to be difficult due to a number of problems. One of the main difficulties is that it is sometimes not possible to determine which, of the set of consequents of an antecedent, is the prime consequent. For example, consider the target $\Sigma : \{l_1(?x) \wedge l_2(?x) \rightarrow l_3(?x);\ l_1(?x) \wedge l_3(?x) \rightarrow l_4(?x)\}$. Given the counterexample $l_1(c) \wedge l_2(c) \rightarrow l_4(c)$, the literal $l_4(c)$ is not a correct consequent, but $l_3(c)$ is. Although Lemma 5.3.2 says that a prime consequent is the right consequent to choose, without knowing the order it is not clear how to identify it. Learning all possible clauses while maintaining all consequents also does not seem to work, resulting in spurious matches between some of these redundant clauses and counterexamples in some cases.

There is a problem in the way the acyclicity property and the supported-by relation are defined. They do not allow acyclic recursive Horn programs. For example,

according to Definition 5.2.2 and Definition 5.2.3, the following Horn program is not acyclic:

$$\{father(?x, ?y) \rightarrow parent(?x, ?y);$$
$$mother(?x, ?y) \rightarrow parent(?x, ?y);$$
$$parent(?x, ?y) \rightarrow ancestor(?x, ?y);$$
$$parent(?x, ?y), ancestor(?y, ?z) \rightarrow ancestor(?x, ?z)\}$$

If we substitute the same variable $?u$ for the variables $?x, ?y$ and $?z$ in the last clause, we have $ancestor(?u, ?u) \succ ancestor(?u, ?u)$—which violates the definition of acyclicity. In their usual meaning, the relations $father, mother$ and $ancestor$ are not reflexive. Therefore, in reasonable domains, we do not encounter literals such as $father(a, a)$ where a is a constant. Also, we do not expect the literals $father(a, b)$ and $father(b, a)$ both to be true in a reasonable interpretation. We call such reasonable interpretations "legal interpretations." Thus, it is reasonable to define supported-by and acyclicity with respect to a legal interpretation and a Horn program, instead of defining them with respect to just a Horn program. Then, the derivation-order query would answer queries on the ground literals with respect to a legal interpretation. Each example can be a Horn clause containing literals from a legal interpretation with respect to which the target Horn program is acyclic. In this scenario, the PLearn algorithm can be shown to learn Horn programs, such as the ones in the example above, which are recursive but are acyclic with respect to a set of legal interpretations.

In Chapter 7, we will see the connections and differences between the PLearn algorithm and the method we applied for learning d-rules from exercises.

Chapter 6

LEARNING D-RULES FROM EXAMPLES

In Chapters 3, 4 and 5, we have seen how d-rules can be learned via learning Horn clauses. In this chapter, we implement a system that learns d-rules from examples, where each input example is a pair consisting of a planning problem and its solution as a sequence of actions. The system is based on the ideas we developed in `DLearn`, in Chapter 4, which learns Horn definitions from Horn-clause examples.

6.1 Introduction

One of the goals of machine learning is to improve the problem-solving performance of systems with their own experience and from external teaching. One approach to this problem is to empirically learn heuristics or control knowledge from examples of successful and unsuccessful problem solving. For example, this is the approach adopted to learn heuristics for symbolic integration in LEX (Mitchell, Utgoff, & Banerji, 1983).

The purely empirical approach of LEX and other similar systems does not exploit the knowledge of the goals and the operators of the domain (i.e., "the domain theory") that the problem solver already has. Exploiting this prior knowledge could potentially expedite learning by finding more general rules from only a few examples. Explanation-based learning (EBL) is based on this observation. It applies the domain theory to explain why a particular problem-solving episode is successful (or

unsuccessful), and from that explanation, it extracts the conditions relevant to that success (Mitchell, Keller, & Kedar-Cabelli, 1986; DeJong & Mooney, 1986).

EBL was employed as the learning technique in LEX2 (Mitchell et al., 1983) as well as in many other learning systems including PRODIGY (Minton, 1988) and SOAR (Rosenbloom & Laird, 1986). Unfortunately, however, EBL does not always lead to efficient problem solving. Many EBL systems suffer from the "utility problem" identified by Minton as the problem of excessive cost of finding and using the appropriate learned knowledge in problem solving (Minton, 1988). One of the sources of the utility problem is the specificity of the explanations in EBL. To learn macro-operators, for example, the system has to explain why a particular operator sequence led to a goal. Since the preconditions of each relevant operator in the sequence are included in the condition extracted by the EBL process, if the operator sequence is long, the macro precondition is long as well. Such long preconditions lead to an expensive match process when the system attempts to use that learned macro.

The goal of speedup learning can be viewed as finding an efficient "target problem solver" by searching the hypothesis space of potential problem solvers. As in any induction system, it is possible to search this space efficiently only when the space is suitably constrained by a bias. This suggests that learning should begin with a hypothesized architecture for the problem solver which constrains the learner's hypothesis space. To eliminate the utility problem, the hypothesized architecture should only allow efficient problem solvers. Tambe, Newell, and Rosenbloom (1990) restrict the expressiveness of their knowledge representation method (chunking) to allow for efficient utilization of the learned knowledge. Here, we restrict the whole problem solvers to enable efficient problem solving.

In this chapter, we describe a learning system for a decomposition-based target problem-solving architecture. The architecture guarantees efficiency by relying en-

tirely on backtrack-free recursive goal-decomposition for problem solving. However, this target architecture also poses some difficulties for the learner. One of the important issues raised is that of *efficiently* finding a problem solver in the hypothesis space which is consistent with the examples. Unfortunately, this problem is hard for expressive hypothesis spaces that involve first-order rules, as we have seen in Chapters 4 and 5.

The system relies on two sources to overcome this difficulty. First, it uses the domain theory to simplify the examples by removing some irrelevant literals and to add some possibly relevant abstract literals. Second, it asks the teacher membership queries about specific instances, as done in Chapters 4 and 5. The domain theory helps simplify the hypothesis in focusing on only the relevant literals and in using abstract literals. Membership queries make it possible to ask about the relevance and generality of each literal in the example, and thereby determine which current hypothesis a given example belongs to, and how to combine it with that hypothesis.

We tested our system, ExEL, in three different domains—the blocks world, a variation of the STRIPS world, and a simplified air-traffic control world. In each of these domains, it found a target problem solver which has 100% accuracy on the test problems, with only a few random examples. Our results suggest that guiding empirical learning approaches by biasing them with a domain theory and providing them with suitable teacher oracles could be a successful approach to speedup learning in complex domains.

The rest of the chapter is organized as follows. Section 6.2 describes our system for learning d-rules. Section 6.3 describes the experimental results. Section 6.4 discusses our work in the context of the related work.

6.2 Learning System

In this section, we describe our learning system. For each domain, the input to the system is its domain theory and a teacher. The domain theory comprises the operators and the definitional axioms. The operators are specified in STRIPS style — i.e., with parameters, explicit preconditions, an add list and a delete list for each operator (Fikes et al., 1972). Axioms are definitions of utility functions (e.g., +, >, etc.), and of abstract or higher-level terms such as `toLeftOf`, `above`. The teacher provides training examples and answers the learner's queries. The task for the learner is to find a set of d-rules which are consistent with the solutions of the training problems.

All training examples are examples of successful problem solving, i.e., positive examples. Each example consists of a problem—an initial state and a goal—and its solution—the sequence of actions the planner has taken to reach the goal from the initial state. The following is a training example from the air-traffic control (ATC) domain.

⟨*problem* :

⟨*state* :

(at(p1, 10), type(p1, propeller), fuel(p1, 5),
cursor-loc(4), free(1), free(2),..., free(9),
free(11),..., free(15), runway-cond(wet),
wind-speed(high), wind-dir(south))

goal : land-plane(p1) ⟩

operator-sequence :

(jump(4, 10), select(10, p1), jump(10, 14), short-deposit(p1 R2)) ⟩

Observe that there is a correspondence between the components of an example and that of a d-rule. The goal of an example corresponds to the d-rule's goal, and

the initial state corresponds to the d-rule's conditions. Note that the subgoals are either eventually achieved by the actions in the action sequence or they are true in the initial state. Hence, the literals in the initial state and the literals in the subsequent states resulting from each action correspond to the d-rule's subgoals. Because a subgoal occurring in a state is achieved before a subgoal in a later state, inter-state literal ordering should be maintained for the purpose of learning subgoal order. This is done by treating the subgoals component as a sequence of sets that contain literals that are true in successive states. For the example above, the first set is the initial state *state*, the second set is *state* with `cursor-loc(4)` removed and `cursor-loc(10)` added—the result of the action `jump(4, 10)`—and so on.

The conditions component of a single d-rule is a conjunction of a set of positive literals. In Chapter 3, we have seen that multiple d-rules for the same goal that differ in their conditions and subgoals can be mapped to Horn definitions. In Chapter 5, we studied an algorithm for learning Horn definitions. Next, we will see how the learning algorithm for Horn definitions can be adapted to learn d-rules from d-rule examples.

As discussed in Chapter 4, Section 4.4, there is a problem in utilizing the algorithm for learning Horn definitions to the task of learning d-rules. A d-rule hierarchy maps to a Horn program rather than to a Horn definition. That is, literals appearing as heads also appear in the bodies of clauses. The resulting interactions between clauses violate the strong compactness property (Lemma 4.3.1). Hence, the Horn-definition learner cannot be used directly to learn Horn programs. As suggested in Section 4.4, we learn clauses for each goal separately, by assuming that d-rules for lower-level (sub)goals are already known.

6.2.1 ExEL's Empirical Generalization Process

For generalizing and learning d-rules, we employ an algorithm that is a slight variation of the Horn-definition learning (DLearn) algorithm. Although DLearn algorithm is usable for learning d-rules, by performing the mapping described in Chapter 3, we implemented a variant of it to exploit the specific optimization opportunities present in learning d-rules. In Chapter 3, we have seen that each literal in every d-rule (and d-rule example) is annotated with two situation parameters to keep track of state information. Moreover, we add `not-after` literals to keep track of the temporal ordering of situations. Instead, here, we separate the condition literals from subgoal literals of a d-rule. The sequence in which the subgoal literals occur is the sequence in which they need to be achieved while planning. For d-rules that are examples (or that are hypothesized), we separate initial-state literals, which are candidate condition literals, from subsequent state literals, which are candidate subgoal literals. In addition, the subgoal literals are maintained in a sequence. Then we use a separate procedure (Order-Subgoals) to learn an ordering among subgoals.

The algorithm is presented in Figure 6.1. It is quite similar to the DLearn algorithm. It takes as input the learned d-rules so far (d-rule hypotheses) and the current example d-rule. It outputs a new set of d-rules that is a generalization of the example d-rule and the previously learned d-rules.

It tries to include the example d-rule in each hypothesis d-rule. It does so by finding the *lgg* of the hypothesis d-rule with the example d-rule and querying whether the resultant hypothesis d-rule is a correct d-rule. The querying process carried out by Test-Query is essentially a membership query as in Chapter 4, but it has a slight variation when dealing with subgoal literals (cf. Section 6.2.3). If the query succeeds, the old hypothesis d-rule is replaced by the generalized d-rule. Otherwise, the example d-rule is added as a new d-rule in the set of hypothesis d-rules. In both

cases, the learned d-rule goes through further processing. First, the irrelevant literals in the generalized d-rule are pruned by Reduce. The Reduce procedure is the same as in Chapter 4, except for the slightly different Test-Query. After removing irrelevant literals, the hypothesis d-rule's subgoal order is refined by the Order-Subgoals procedure (cf. Section 6.2.3).

```
Generalize-Query(dRuleSet, egDrule)
 if there is a dRule in dRuleSet such that
          Test-Query(lgg(dRule, egDrule)) is successful
          /* correct hypothesis to generalize is found */
      replace dRule by Order-Subgoals(Reduce(lgg(dRule, egDrule)))
 else dRuleSet ← Reduce(egDrule) + dRuleSet
                    /* example becomes a new drule*/
 return dRuleSet

Reduce(lgg, dRuleSet)
 for each lit in lgg do
   if Test-Query(lgg − {lit}, dRuleSet) is successful then
     remove lit from lgg
 return lgg
```

FIGURE 6.1: Generalize-and-query algorithm for generalizing d-rules, given an example d-rule

So far, we have described empirical generalization from examples. Next, we describe the role of the domain theory in generalization.

6.2.2 Guiding Learning with Domain Theory

There are two problems if we directly generalize the examples: (1) there may be extraneous literals in the initial state of an example, which result in increased cost of generalization, in terms of learning time; (2) the literals in the example may not be at the right level of abstraction. Therefore, after the components of d-rules are identified in the examples, and before they are used in the inductive generalization process described in the previous section, the learner performs the following two steps to transform them. Both these steps are guided by the domain theory.

6.2.2.1 Explanation-Based Pruning

Examples may contain literals describing relations among objects that may be irrelevant to a target d-rule. For instance, in the STRIPS-world domain, if the goal is to navigate the robot from one room to another the relations involving objects that are in various rooms are irrelevant to the d-rules for this situation. Even if the objects are relevant, some of the relations between them may not be relevant.

The extraneous literals, unless eliminated, increase the generalization cost. Our solution is to first explain the training example using the domain theory, and then to collect the literals that are leaves of the proof tree that explains the example. This is basically like explanation-based generalization (EBG) without the generalization step. This solution limits the literals in the example to only the relevant ones. We do not use the generalization step of EBG, because (1) conditions of d-rules may need to be more specific than the result of the generalization step, and (2) the induction process does the necessary generalization anyway.

To illustrate the pruning process, consider the example at the start of Section 6.2. After the pruning step, all `free` literals are removed, except the literal `free(14)`,

because it so happens that only the literal `free(14)` is required among all the `free` literals to prove the preconditions of the primitive operators in the solution sequence. The literals `runway-cond(wet)` and `wind-speed(high)` are also removed, because it so happens that none of the actions requires these literals in preconditions of the operators.

6.2.2.2 Abstraction by Forward Chaining

If d-rules are specified using literals expressed at a higher level of abstraction than the terms in the operator's add and delete lists, they are more readable and concise. Furthermore, abstract terms may be used to cover different disjunctive cases. Because the conditions in d-rules are conjunctive, without abstract terms, we need a separate d-rule for each different case. Consider, for instance, adjacency of two rooms. A room, Room1, is adjacent to another room, Room2, if Room1 is to the north, south, east, or west of Room2. Without the abstract term for adjacency, we may need four separate d-rules — one each for each of the directions. Thus, abstract terms reduce the number of d-rules required for a domain.

Axioms in the domain theory help the planner in understanding these high-level terms by translating them into the planner's language. For example, one of the translations for the high-level term `can-land-short(?plane)` in the ATC domain is `type(?plane DC10)` $\wedge$ `wind-speed(low)`$\wedge$ `runway-cond(dry)`.

To introduce abstract literals into the learning example, we employ forward chaining. We take the example that is in the planner's low-level language, and apply forward chaining using the axioms. This would result in literals that are at the required levels of abstraction. The axioms are shallow—mostly one-level—rules; therefore, there is no danger of forward chaining introducing too many literals.

For our running example, after the pruning step, this step introduces the literals `can-land-short(p1)`, `wind-axis(north-south)`, `runway-loc(R2, 14)`, `runway-dir(R2, north-south)`, `short-runway(R2)` and `free-cursor()`.

6.2.3 Learning Subgoals

The subgoals of our d-rules do not contain any new variables besides the ones that are in the goal and the conditions. Since subgoals are learned after the goal and the conditions components are generalized, learning subgoals consists of selecting candidate subgoal literals and ordering them correctly. The subgoal literals are determined using membership queries on the candidate subgoal literals selected from the first example of a d-rule, which we call the "seed example." The remaining examples are used to refine the ordering of the subgoals learned from the seed example.

The literals which are true in the intermediate states when solving the seed example of a d-rule are candidates for subgoals. Out of these candidate subgoals, the extraneous ones are removed by asking a membership query for the sequence of remaining literals after temporarily removing each literal. While asking a query, the inter-state ordering of the subgoal literals is preserved, and intra-state ordering is ignored. In other words, the subgoal sequence of a hypothesis d-rule consists of a sequence of sets of literals, where the literals in each set were all true in the same step during the solution of the example. A queried subgoal sequence is *consistent with* a d-rule, if all subgoals in the d-rule are present in the queried sequence, and for every pair of subgoals in the d-rule, they either both occur at the same step in the queried sequence, or in the same temporal order as in the d-rule. In this case, the query is answered yes, and the literal which is removed is considered irrelevant and is permanently dropped. If the query is answered no, the removed literal is made a subgoal of the hypothesis d-rule and is included in the future queries.

The above procedure finds all the correct subgoal literals in the d-rule, but it does not necessarily fix their order, because more than one subgoal may have been achieved in the same step in the seed example. For instance, in the STRIPS-world domain, in the context of entering some room `A` from another room `B`, the door between `A` and `B` may already be open, thus obviating the application of the action `open-door`. Since this is unlikely to repeat in every example, the remaining examples are used to refine the order learned from the seed example.

Order-Subgoals **procedure.** The procedure is listed in Figure 6.2. Assume that the subgoal order in the current hypothesis d-rule is $\langle S_1, \ldots, S_n \rangle$, where $S_1, \ldots, S_n$ are sets of subgoals. A refinement might further split these sets, but never merges them. Consider a new example which is described by $L_0, \ldots, L_m$, which represent the sets of subgoal literals which are true respectively after $0, \ldots, m$ steps in the solution. At any step, the subgoals in the current set S_i are matched, after applying the substitutions generated during the generalization of the conditions part, with the literals L_j in step j of the example. Let α_i be the substitution for the hypothesis and α_j be the substitution for the example. If $S_i\alpha_i \cap L_j\alpha_j$ is not empty, S_i is split into two sets, those which match with the literals in L_j and those which do not. The pointer j in the example advances to $j+1$ and a match is attempted between literals of L_{j+1} and those in the second subset of S_i. Similarly, if the match is successful for all literals in S_i with some literals in L_j, then L_j is split into two sets, and the pointer i is advanced. A match is then attempted between the literals of S_{i+1} and those of L_j that have not been successfully matched. If $S_i\alpha_i \cap L_j\alpha_j$ is empty, that is none of the literals in L_j successfully match with any literals in S_i, only the j pointer is advanced.

The above algorithm works correctly if the subgoal literals of both the original hypothesis and the new example are consistent with the d-rule. In particular, we

```
Order-Subgoals(⟨S_1, ..., S_n⟩, α_i, ⟨L_0, ..., L_m⟩, α_j)
  i := 1; j := 0;
  while i ≤ n and j ≤ m do {
     if S_i α_i ∩ L_j α_j is not empty
     then Split S_i into S_i and S'_i where
               S'_i := {s | s α_i ∈ S_i α_i ∩ L_j α_j}, and S_i := S_i − S'_i;
          L_j := L_j − {l | l α_j ∈ S_i α_i ∩ L_j α_j}
          j := j + 1
     else i := i + 1;
  }
  return refined S_i's
```

FIGURE 6.2: Order-Subgoals procedure

rely on the assumption that the order of any two subgoal literals in any example is not reversed with respect to their order in the d-rule. Hence, we can proceed with the matching of the literals in the hypothesis and the example in a strict left-to-right order.

For example, consider that we are in a state `(cursor-loc(10), at(p1 10))` from which we want to reach the goal state `at(p1, 14)`. Given a sequence of actions that achieves this goal, the system might extract the subgoal sequence ⟨`(cursor-loc(10), at(p1, 10)) (selected(10, p1)) (cursor-loc(14)) (deposit-at(p1, 14))`⟩, after eliminating all the irrelevant literals using queries. Since the plane p1 and the cursor were in the same location 10, there was no need to move the cursor before selecting the plane in this example. Given another similar example with start state `(cursor-loc(7), at(p2, 8))` and goal `at(p2, 11)`, it splits the first subgoal set `(cursor-loc(10), at(p1, 10))` into two sets, and learns the

subgoal sequence ⟨`(cursor-loc(?x)) (at(?p, ?x)) (selected(?x, ?p)) (cursor-loc(?y)) (deposit-at(?p, ?y))`⟩, for the goal `at(?p, ?y)`.

6.2.4 Summary of the Learning Algorithm

In this section we summarize the learning algorithm. Recall that each example is presented by a problem and a sequence of actions. The learner starts by identifying in the examples the components corresponding to the d-rule, namely the goal, the condition, and the subgoals. Next, the learner transforms the conditions component by first removing the irrelevant literals using the explanation-based pruning step, and then by introducing higher-level terms using the forward-chaining step.

The transformed version of the example is then empirically generalized. Then the Generalize-Query algorithm in Figure 6.1 is invoked to find the matching hypothesis d-rule for the example d-rule. If one is found, it generalizes the example and that hypothesis d-rule, and ultimately replaces the hypothesis d-rule. If not, the example d-rule is added as a new d-rule to the set of hypothesis d-rules. In both cases, the subgoal ordering of the learned d-rule is refined as described in the previous section, before placing the learned d-rule in the set of hypothesis d-rules.

This learning process stops when the set of training examples is exhausted or when the desired performance is reached on a set of test examples.

To illustrate this algorithm, consider our running example to be the first training example. After the transformation step, explanation-based pruning, and forward chaining, it becomes the initial hypothesis, h_1:

```
g : land-plane(p1)
c :{ at(p1, 10), type(p1, propeller), fuel(p1, 5), free(14),
   cursor-loc(4), wind-axis(north-south), free-cursor(),
   can-land-short(p1), runway-loc(R2, 14),
   short-runway(R2), runway-dir(R2, north-south)}
```

sg :⟨(cursor-loc(10))(selected(10, p1))

(cursor-loc(14))(short-runway-deposit(p1, R2))⟩.

Suppose the next example e_1 after the initial transformation steps is

g : land-plane(p5)

c :{at(p5, 8), type(p5, DC10), fuel(p5, 5), free(16),

cursor-loc(3), short-runway(R4), free-cursor(),

can-land-short(p5), wind-axis(east-west),

runway-loc(R4, 16), runway-dir(R4, east-west)}

sg :⟨(... cursor-loc(8)...)

(... selected(8, p5)...) (... cursor-loc(16)...)

(... short-runway-deposit(p5, R4)...)⟩.

Since h_1 and e_1 match and the result passes the query, the new h_1 is

g : land-plane(?p)

c :{at(?p, ?x), type(?p, ?type), fuel(?p, 5), free(?y),

cursor-loc(?loc), wind-axis(?dir), free-cursor()

can-land-short(?p), runway-loc(?r, ?y)

short-runway(?r), runway-dir(?r, ?dir)}

sg :⟨(cursor-loc(?x)) (selected(?x, ?p))

(cursor-loc(?y)) (short-runway-deposit(?p, ?r))⟩.

Suppose the next example e_2 after the initial transformation steps is

g : land-plane(p9)

c :{at(p9, 10), type(p1, propeller), fuel(p9, 3), free(14),

cursor-loc(10), selected(10, p9), runway-loc(R2, 14),

wind-axis(north-south), can-land-short(p9),

short-runway(R2), runway-dir(R2, north-south)}

sg :⟨(... cursor-loc(14)...)

(... short-runway-deposit(p9, R2)...)⟩.

The example e_2 does not match with h_1 and the result does not pass the membership query, because e_2 does not have `free-cursor()`. Hence, e_2 is made into a new hypothesis d-rule for the goal `land-plane(?plane)`.

6.3 Experimental Results

The algorithm discussed so far has been implemented in Common Lisp as a system called ExEL. It has been tested using three different domains — the blocks world, a variation of the STRIPS world and a simplified air-traffic control domain. The objective is to see whether ExEL is able to learn d-rules in these domains efficiently, i.e., using a reasonable number of examples and queries. Another objective is to see how well the learned d-rules perform on the test set. As discussed earlier, because the d-rule problem-solver is guaranteed to be fast, the performance can be evaluated by its coverage rather than speed. The third objective is to see how well the system works on varied domains.

For the purpose of experiments, the teacher is implemented by providing a set of target d-rules for each domain. Training and test problems are randomly chosen. Each training problem and its solution provide several training examples for learning, for a problem may require several applications of a d-rule. Similarly a test problem tests several learned d-rules. The membership queries are answered by syntactic match with the teacher's d-rules.

6.3.1 Blocks World (BW)

The blocks-world domain has three d-rules. Problems are randomly generated with 5 to 10 blocks and a table, with a single goal. Each problem involves making some configuration of the blocks from some initial configuration. There are no multiple

d-rules for the same goal in this domain. ExEL takes at most 3 training problems for 100% performance on test problems.

6.3.2 STRIPS World (SW)

This is a minor variation of the STRIPS-world domain (Fikes et al., 1972); the configuration of the rooms in this domain is a grid, whereas in the standard STRIPS world it could be of any shape. The domain consists of rooms which are connected to each other by doors. The doors can be open or closed. Each room has zero or more boxes. There is a robot that can open doors, move around, and push a box from room to room. A typical goal for the robot is to move a box that is in some room to some other room. There are 16 d-rules for this domain. The goal of moving an object to a different room required six d-rules. It has subgoals such as moving from one room to another room, holding a box, releasing a box, opening a door, and closing a door. The subgoal for moving from one room to another room had six d-rules, and the rest of the subgoals had one d-rule each. Eight of these 16 d-rules are recursive.

The configuration has 12 rooms (4×3 grid) with 17 doors connecting them. There are also 5 to 10 boxes distributed among the 12 rooms. The robot is placed randomly in one of the rooms.

Figure 6.3 plots the number of training problems and their solutions versus the percentage of test problems ExEL successfully solves in SW. Both the training and the test problems are randomly generated. The training problems were for goals at all levels, whereas the test problems were for the goal of moving a box from some room to some other room, which makes use of all other (sub)goals. Each data point is the mean of 5 runs. The error-bars show one standard deviation on either side of

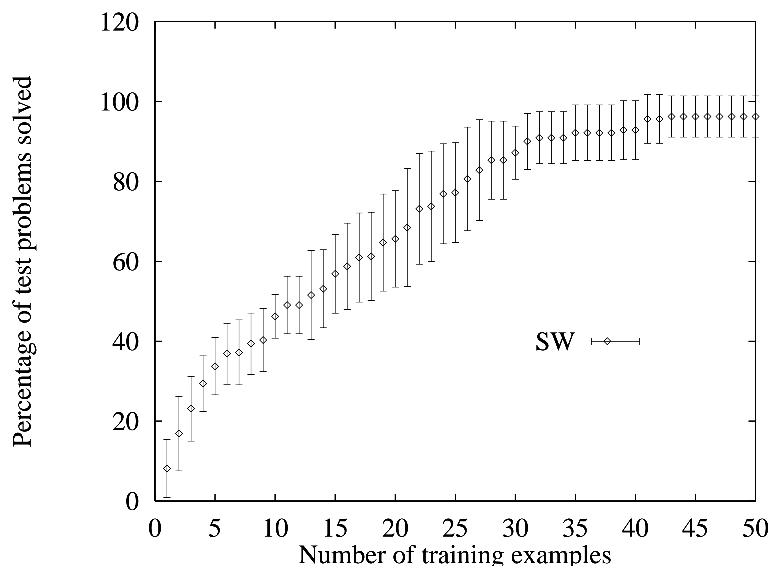

FIGURE 6.3: Performance of ExEL in the SW domain

the mean for these runs. The d-rules generated were useful for configurations other than the 4×3 grid ExEL learned from.

6.3.3 Air-Traffic Control (ATC) domain

This domain is a simplified version of the Kanfer-Ackerman air-traffic control task (Ackerman & Kanfer, 1993). The main task is landing a plane from any configuration. The task has a queue of incoming planes, holding patterns, and runways. The planes are accepted into the holding patterns, and then are landed on the runways. The type of conditions pose restrictions on landing a plane. The operators select a plane to land, deposit a plane either on a runway or in a holding position, or move the cursor on the screen. There are 13 d-rules for this domain, including multiple d-rules for some goals. The main goal of landing a plane from any holding pattern required three d-rules. Its subgoal to take the plane to the correct runway required four d-

rules. Depositing a plane on a runway needed three d-rules. Moving a plane between holding patterns, and its subgoals of moving the cursor and selecting a plane all required one d-rule each.

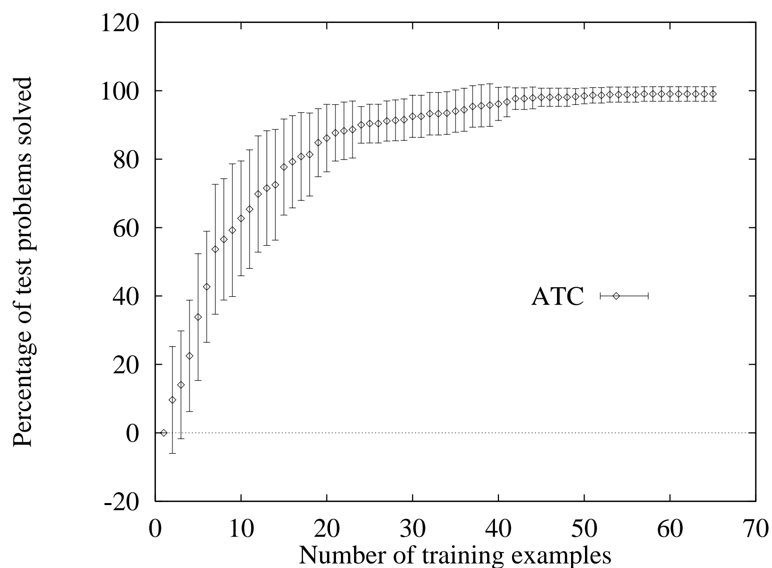

FIGURE 6.4: Performance of ExEL in the ATC domain

Figure 6.4 plots the number of training problems and their solutions versus percentage of test problems ExEL successfully solves in ATC. Both the training and the test problems are randomly generated. The training problems were for goals at all levels, whereas the test problems were for the toughest goal of landing a plane from any level onto a runway. Each data point is a mean of 20 runs. The error-bars show one standard deviation on either side of the mean for these runs.

6.4 Discussion

Our work is related to a number of different research areas including EBL, Inductive Logic Programming (ILP), and Computational Learning Theory (COLT). In this section, we explicate some of these connections and outline some future work.

Our work was initially motivated by the utility problem in EBL. Our approach to this problem is to constrain the problem solver to use its d-rules in a backtrack-free mode, and to allow the conditions of our d-rules to be expressed in high-level abstract terms which are common to all positive examples. The SteppingStones approach by Ruby and Kibler also learns subgoal sequences, which are similar to d-rules (Ruby & Kibler, 1991). However, as in EBL, their system learns by generalizing single examples. Comparing multiple examples of the same d-rule allows ExEL to eliminate redundant literals much more effectively, because different examples that use the same d-rule may not share the entire solution, and hence may not share many literals in their initial states. This allows for shorter and more general preconditions with better coverage and efficiency of match.

Our system uses membership queries to overcome the hardness problem of finding a conjunctive concept consistent with a set of positive examples. GOLEM, an ILP system, circumvents this problem by restricting the target class to be "determinate," which means that there is a unique binding to all the free variables of a literal in a clause given the bindings of all the previous literals (Muggleton & Feng, 1990). The class of determinate conjunctive clauses is proved to be PAC-learnable from examples and the input-output information of the literals when the arity of the literals and the depth of the dependency chain of the variables in the clause are constant (Cohen, 1995b). Unfortunately, however, these assumptions are too strong in practice.

Another ILP system, FOIL, starts from the most general hypothesis and progressively specializes it by restricting the hypothesis from covering negative examples (Quinlan, 1990). DOLPHIN (Zelle & Mooney, 1993), Grasshopper (Leckie & Zukerman, 1993), and SCOPE (Estlin, 1998) are systems that use FOIL for learning control knowledge for a problem solver. FOIL is difficult to adapt to our work because it needs negative examples. It is not possible to get relevant negative training examples in our learning-from-observation setting: d-rules do not have choice points for backtracking in case of failure, and only successful problem-solving instances are available. Moreover, unlike FOIL and GOLEM which process the examples in batch for their greedy hill-climbing, ExEL processes examples incrementally.

One advantage of FOIL and GOLEM, however, is that they do not require queries. In Chapter 7, we will show a way to eliminate queries by self-testing. In self-testing the learner generates actively its own negative or "self-critical" examples for its hypotheses and validates the hypotheses by testing the examples.

Hierarchical partial orders (HPOs) of Marsella (1993) also concerns recursive decomposition of problems. HPOs represent separate planning and execution orders for subproblems, whereas d-rules combine them into one. Moreover, these orders can be partial. Marsella's work is in the context of problem-solving, and not planning. The decompositions he considers are in terms of subproblems, and not subgoals.* He also considers learning HPOs, in two ways: learning from teacher-given solutions, the method he calls BU-PRL, and learning by solving problems, the method he calls PRL. BU-PRL is akin to ExEL. (The connection between PRL and our method is described in Chapter 7.) From a given solution (operator sequence), BU-PRL uses heuristics about the inter-dependency of operators and subproblems to generate new

* A problem (subproblem) is a pair of a starting state and a goal state, whereas a goal (subgoal) is a literal that should be satisfied in a state.

subproblems by coalescing operators and subproblems. It asks the teacher for solutions to the new subproblems. The idea is that this process builds a hierarchy of rules to decompose a problem into subproblems. These rules are generalized by making constants into variables. This kind of generalization is not robust, and not guaranteed to work always. There is no control over checking whether the subproblems generated are generally applicable. This results in a proliferation of subproblems and corresponding rules—a case of the utility problem. The learning mechanism is validated using only a "Tile-World" domain. It is unclear whether the heuristics used in generating the subproblems generalize to other domains.

In d-rules, the subgoal parameters are instantiated after the literals in the condition are satisfied in a state. However, in some domains, this may be a restrictive assumption; we may be able to ground subgoal literals only after further exploration. For example, in Marsella's Tile-World domain, to move a tile, call it the goal tile, from one square to another square, first the tiles on a path from the starting square to the destination square should be cleared out of the path, and then the goal tile has to be moved along the path. We do not know which tiles are to be cleared unless we know the path we are choosing for moving the goal tile. But, if we plan for finding a path to move the goal tile to its destination (but not execute the plan), then we will know which tiles are in the way and need to be cleared. Then, we can execute actions for clearing the obstructing tiles first, and then execute actions for moving the goal tile to its destination. Here, the planning order of the subproblems of clearing the obstructing tiles and moving the goal tile is different from their execution order. For this reason, Marsella separates the planning order from the execution order. While planning, the subproblems in a HPO are planned for in the planning order, to correctly instantiate all the subproblems. These instantiated subproblems are then tried in their execution order.

LEAP (Mahadevan, Mitchell, Mostow, Stienberg, & Tadepalli, 1993) is another system that learns methods for decomposing problems into subproblems. Although its underlying method can be extended to learn decompositions in planning, its main focus is problem decomposition. Unlike d-rules or HPOs, which use ordering for composing subgoals or subproblems, LEAP can consider any arbitrary "combinator" for composing subproblems. LEAP, however, expects a problem and the problem's subproblems and their combinator as an example. It verifies using some transformation rules (which are provided as part of the input domain theory) whether the decomposition supplied in the example is correct. Using transformation traces, it then identifies the relevant parts of the example needed for constructing a general decomposition method by performing constraint backpropagation. Then it uses the variable bindings generated during the verification process to generalize the resulting decomposition method. This process is basically the explanation-based learning process. The supply of subproblems makes the research problem in LEAP considerably easier in comparison to the problem addressed by ExEL and BU-PRL. Subgoals, in ExEL, and subproblems, in BU-PRL, need to be recovered from operator sequences. In a way, ExEL and BU-PRL are solving the plan-recognition problem by empirical generalization of several observed problem-solution pairs.

Chapter 7

LEARNING D-RULES FROM EXERCISES

The ExEL system in Chapter 6 learns d-rules from teacher-provided problem and solution pairs. In this chapter, we introduce the LeXer system, which learns d-rules from exercises—sequences of problems and subproblems that are in increasing of their difficulty.

7.1 Introduction

The research work in learning control knowledge falls under one of two extremes: supervised speedup learning and unsupervised speedup learning. In supervised speedup learning, the system is provided with examples or solved instances by a teacher (DeJong & Mooney, 1986; Mitchell et al., 1986; Shavlik, 1990). In unsupervised speedup learning, the system is given only problems without their solutions (Laird, Rosenbloom, & Newell, 1986; Minton, 1988). In this chapter, we explore a middle course of learning from exercises—useful subproblems of natural problems that are ordered in increasing order of difficulty.

Teaching problem-solving through exercises is a widely used pedagogic technique. A human teacher selects certain problems and orders them according to their level of difficulty to form a sequence of exercises. A human student starts by solving simple problems first; then attempts harder problems by applying the knowledge gained from solving the earlier problems; and then attempts still harder problems, and so on. This process continues until the student learns the concepts satisfactorily. Machine learning of problem-solving skill using exercises, apart from following this

pedagogic tradition, offers a compromise between supervised speedup learning and unsupervised speedup learning. Supervised speedup learning, although more efficient than unsupervised learning, places the burden of providing the solutions to the training problems on the teacher—usually a human. Unsupervised speedup learning, in contrast, expects the learner to solve the training problems, while unburdening the teacher. However, this is computationally hard for the learner because it lacks control knowledge and, hence, its only recourse is brute-force search. In the exercises approach, the teacher has the task of providing an exercise set—a sequence of problems ordered by difficulty. The learner has to solve the exercise problems using the bootstrapping method akin to the above-described method followed by a human student.

The exercises approach for speedup learning has been used for learning control rules (Natarajan, 1989). In this work, we use exercises for learning goal-decomposition rules (d-rules). The learning of control rules in (Natarajan, 1989) is essentially propositional, whereas learning d-rules is first-order or relational. Moreover, d-rules can be recursive, which complicates solving of exercise problems: to solve a particular goal, the learner should already know something about solving that very goal.

Our approach, implemented as a system called LeXer, follows two main steps: For each exercise, (1) an exercise-solver solves the exercise, and outputs the solution (the plan) and the subgoals used; and (2) a first-order inductive learner forms an example d-rule using the initial state as the d-rule condition and the subgoals as the d-rule subgoals, and then uses a "generalize-and-test" algorithm to include this example d-rule in the previous hypothesis d-rules. The generalization phase of the generalize-and-test algorithm finds the least general generalization (lgg) of the example d-rule with previously formed similar hypothesis d-rules. In the test phase, to ensure appropriate generalization, the algorithm generates examples of the hypothesized

d-rule and tests whether the examples are correct. This generalization algorithm is similar to ExEL's (Chapter 6), except that here the subgoal ordering is directly recovered from solutions of exercises, and queries are replaced by testing.

The rest of the chapter is organized as follows. Section 7.2 discusses the learning approach and the related algorithms. Section 7.3 discusses the results of experiments. Section 7.4 concludes with a discussion of previous work and related issues.

7.2 Learning from Exercises

Learning from exercises requires that the teacher give a sequence of problems ordered according to their level of difficulty: from least difficult problems which come first, to most difficult problems which come last. This is similar to the way exercises are presented at the end of a lesson in a textbook on mathematics. In the examples framework, on the other hand, the examples the learner gets are randomly chosen from a natural distribution. However, in the exercises framework, the burden of solving the input problems is on the learner, unlike in the examples framework where that burden is on the teacher.

We have not yet defined what a problem's level of difficulty means. Natarajan (1989) employed the solution length of a problem as the level of difficulty—i.e., problems with longer solution lengths are more difficult. This, however, does not work for d-rules. A d-rule, depending on the problem it is applied to, may produce varying solution lengths. Therefore, exercises are ordered, instead, according to a bottom-up examination of the goal-subgoal hierarchy of the domain. That is, an exercise related to a goal in the hierarchy is less difficult than an exercise related to goals that are higher in the hierarchy. This, however, does not work when there is recursion among goals, for there is no strict hierarchy then. In the case of recursion,

exercises for base cases are considered less difficult than the exercises for more general recursive cases.

```
LeXer
  dRuleSet ← {}
  ⟨state, goal⟩ ← Next-Exercise() /*next problem*/
  while ⟨state, goal⟩ is not FAIL do {
      depthLimit ← MIN-DEPTH
      done ← FALSE
      while depthLimit <= MAX-DEPTH and not done do {
          ⟨plan, subgs⟩ ← X-Solver(prob, dRuleSet, depthLimit, nil, nil)
          if ⟨plan, subgs⟩ is not FAIL then done ← TRUE
          else depthLimit ← depthLimit + 1
      } /* IDS */
      exampleDR ← Preprocess(⟨goal, state, subgs⟩)
      dRuleSet ← Generalize-Self-Test(dRuleSet, exampleDR)
      ⟨state, goal⟩ ← Next-Exercise()
  } /* all exercises are processed */
  return dRuleSet
```

FIGURE 7.1: Algorithm for learning from exercises

First we present an overview of the learner. (See Figure 7.1 for the pseudo-code.) The learner (LeXer) takes in exercises, which are in an order of increasing difficulty, as input. The goal-subgoal hierarchy of the domain can also be an optional input. This does not affect the correctness of the learner, but affects its efficiency. LeXer starts with an empty set of d-rules. It makes use of a routine **Next-Exercise**() to get the next exercise. An exercise is a problem—i.e., a state-goal pair. LeXer employs an exercise solver X-Solver that uses depth-first search (DFS). X-Solver utilizes LeXer's

own previously learned d-rules along with primitive domain actions to solve the exercises. With DFS there is a danger of the solver going too deep in a wrong direction. Therefore, LeXer uses iterative-deepening depth-first search (IDS)—i.e., starting with a depth limit and doing DFS until that depth limit is reached, and if no solution is found, increasing the depth limit and re-doing DFS with the new depth limit, and so on (Korf, 1987a). IDS also guarantees that the solution, if any, is reached with the fewest number of operator applications. The solver returns the solution plan that solves the exercise problem along with the subgoal sequence used to solve the exercise.

Once an exercise is solved, an optional preprocessing step (`Preprocess`) prunes irrelevant literals and introduces abstract literals using a domain theory. This step prepares an example for the next, generalization step.

Next, LeXer calls a first-order generalizer (Generalize-Self-Test) to include the new case in the learned set of d-rules. Generalize-Self-Test tries to include the new case in one of the existing d-rules by generalizing each d-rule and testing the validity of the generalized d-rule. If that is not possible, the new case forms the basis for a new d-rule.

In the following subsections, we describe the exercise solver, X-Solver, the preprocesser step, Preprocess, and the d-rule generalizer, Generalize-Self-Test, in detail.

7.2.1 Solving Exercises

To solve exercises in a domain, our exercise solver employs previously learned control knowledge—d-rules learned from the exercises seen so far—and primitive actions of a domain. Of course, to solve the initial exercises, only the primitive operators are used, for the learner has had little opportunity to learn yet.

```
X-Solver(prob, dRs, depth, prevPlan, prevSgs)
  ⟨state, goal⟩ ← prob
  if goal is reached then return ⟨prevPlan, prevSgs⟩
  else if depth > 0 then {
     operators ← dRs ∪ domainActions
     for each op ← operators do{
        if op's condition is satisfied then {
           if op is a d-rule then ⟨newState, plan⟩ ← D-rule-plan(prob, op, dRs)
           else /*op is a primitive action*/
             ⟨newState, plan⟩ ← execute(state, op)
           sgs ← prevSgs + op or op's goal /*append*/
           plan ← prevPlan + plan /*append*/

           newProb ← ⟨newState, goal⟩
           ⟨finalPlan, finalSgs⟩ ← X-Solver(newProb, dRs, depth − 1, plan, sgs)
                /* look deeper */
           if ⟨finalPlan, finalSgs⟩ is not FAIL return ⟨finalPlan, finalSgs⟩
        } /* if op's cond is...  */
     } /* for each */
     return FAIL /*no op is successful*/
  } /* else if depth */
  else return FAIL /* gone too deep!  */
```

FIGURE 7.2: DFS-based exercise solver

The exercise solver, X-Solver, employs fixed-depth depth-first search to solve the exercises. It uses, along with primitive actions of a domain, previously learned d-rules as operators for the search. Primitive actions of a domain are specified as STRIPS-style operators, and their execution is straightforward (Fikes et al., 1972).

In particular, first the solver checks whether the conditions of the action are satisfied. If yes, the current state is modified by adding and deleting, respectively, the add- and delete literals of the action. But, to execute d-rules, X-solver employs

the d-rule-based planner (D-rule-plan). The d-rule-based planner applies the d-rules suggested by the search. If the conditions of the d-rule to be executed are satisfied in the current state, using a set of bindings suggested by this match, the solver instantiates the subgoal sequence of the d-rule. Then, X-solver asks D-rule-plan to achieve the subgoal sequence. D-rule-plan has the primitive actions of the domain and the current d-rule base available to it. D-rule-plan, if successful, returns a plan applied in achieving the subgoal sequence, and the resultant state; else, it reports failure. In case of failure and when the conditions of the action or the d-rule are not satisfied in the current state, the solver considers alternative operators. If there are no more untried alternative operators for the current state, X-solver backtracks and considers alternate operators for the previous state. In case of successful application of the operator (action or d-rule), the resultant state is made the current state and the process continues till the goal is reached or the fixed depth limit is reached. See Figure 7.2 for the algorithm.

Let us consider an example from air-traffic control (ATC) domain (see Section 7.3 for details.). To land a plane which is at holding level 2, it has to be first moved to holding level 1 (node `n2` in Figure 7.3), and then be landed on a free runway (node `n4` in Figure 7.3). To execute the d-rule corresponding to `move-plane`, the d-rule planner achieves the subgoals—such as `move-cursor, select-plane,` and `deposit-plane`—suggested by the d-rule, and returns the resultant state and the sequence of actions. Similarly, for `land-plane`, the d-rule planner achieves its subgoals, and returns the resultant state and the sequence of actions. Presumably, the sequence of exercises is such that the d-rules corresponding to the goals `move-plane` and `land-plane` have already been learned. In this example, the successful solution path contains only d-rule operators and no primitive-action operators. X-solver finally outputs the instantiated goals of the d-rules tried—`move-plane(P,`

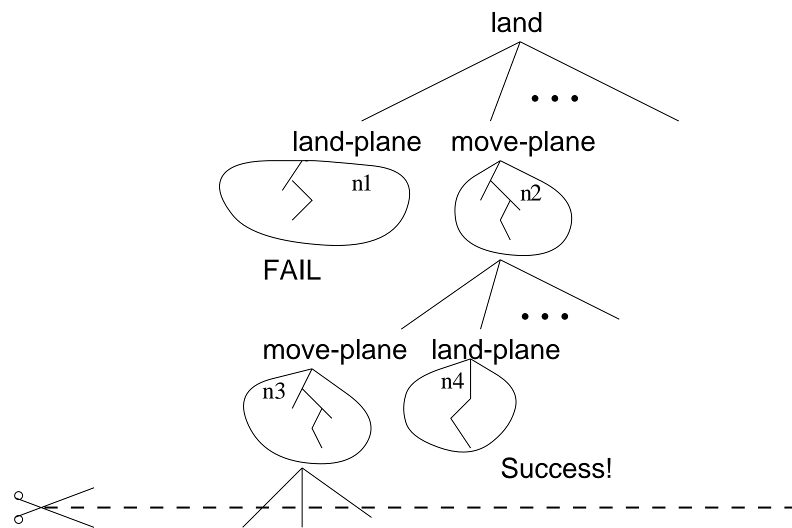

FIGURE 7.3: Exercise solver achieving the goal `land` using IDS with the current depth limit 2

L1) `land-plane(P)`—as the subgoals for a d-rule hypothesis for the main goal—`land(P)`—and the concatenation of the actions produced by each operator as the final plan.

If the learner is provided with the goal-subgoal hierarchy, X-Solve can exploit this to focus its search. In particular, it can limit the operator set used in the search to only those d-rules and primitive actions that address the subgoals of the current goal. In the above example, if the learner knows that `land` has only `land-plane` and `move-plane` as subgoals, then it can try only the instantiations of the d-rules of `land-plane` and `move-plane` as the operators, and ignore others.

7.2.2 Example Preparation

If X-Solver succeeds on an exercise, the resultant solved exercise forms an example of a d-rule: the goal and the initial state of the exercise problem correspond to the

example d-rule's goal and condition, respectively; the subgoal sequence output by the exercise solver for the exercise problem corresponds to the subgoal part of the example d-rule. This example d-rule (or simply, example) is subjected to the explanation-based pruning step and the step that introduces abstract literals by forward chaining, before the inductive generalization process. These two steps comprise Preprocess, and they are the same as the ones in Section 6.2.2. Explanation-based pruning removes irrelevant literals from examples, while forward-chaining step introduces abstract literals into examples.

For the example of Figure 7.3, the example d-rule, after the two steps, is

```
g:   land(P)
c:   (at(P, 10), type(P, prop), runway-cond(wet),
     free(1), hold-level(1, 10), cursor-loc(4),
     fuel(P, 5), wind-speed(high), wind-axis(NS))
sg:  ⟨move-plane(P, L1), land-plane(P)⟩
```

7.2.3 Generalization of d-rules

Given an example d-rule with all three components (goal, condition and subgoals), and a set of d-rules learned so far, the task is to form a new set of d-rules that cover the example. The sketch of the algorithm (Generalize-Self-Test) to accomplish this is given in Figure 7.4.

First, Generalize-Self-Test tries to include the example in one of the existing d-rules. If that is not possible, it creates a new d-rule with the current example as the basis. This algorithm is essentially the same as the Generalize-Query algorithm in Chapter 6, with two differences. One is that, since the example d-rule already has the necessary subgoals, and their relative order, here we need not do anything to order subgoals. Thus, we do not use Order-Subgoals or any other such procedure. The other difference is that instead of querying a teacher, it performs self-testing

```
Generalize-Self-Test(dRuleSet, egDrule)
  if there is a dRule in dRuleSet such that
           Test(lgg(dRule, egDrule)) is successful
           /* correct hypothesis to generalize is found */
       replace dRule by Reduce(lgg(dRule, egDrule))
  else dRuleSet ← Reduce(egDrule) + dRuleSet
                      /* example becomes a new drule*/
  return dRuleSet

Reduce(lgg, dRuleSet)
  for each lit in lgg do
    if Test(lgg − {lit}, dRuleSet) is successful then
      remove lit from lgg
  return lgg
```

FIGURE 7.4: Generalize-and-test algorithm for generalizing d-rules, given an example d-rule

(Test) to test the validity of its generalizations. This change is consistent with our intent to unburden the teacher. Also, assuming the availability of a teacher who knows the target concepts can be an unreasonable demand in certain domains.

The idea of self-testing is as follows. Given a hypothesis d-rule to test, the learner first generates a problem compatible with the condition of the hypothesis, and then decomposes the goal of the problem into the subgoals suggested by the hypothesis d-rule. Next, this sequence of subgoals is given to the solver to solve. The solver is supplied with the set of d-rules learned so far. If the solver is successful, then it generates another test problem and attempts to solve it, in a similar way. If the hypothesis d-rule is successful on a number of test problems, then the hypothesis d-rule is accepted as correct. If not, by concluding that the example d-rule and the learner's candidate rule for inclusion are not compatible, the learner returns FAIL.

This process is accomplished by Test in the generalization algorithm of Figure 7.4. The idea here is that if a hypothesis d-rule purports to be correct, then it must correctly decompose the goal of the problem that satisfies the condition part of the hypothesis d-rule into its subgoals. If the solver cannot solve the problem using the decomposition suggested by the hypothesis d-rule, then the decomposition is deemed incorrect. In such a case, the learner rejects the hypothesized d-rule.

The self-testing process in Reduce has a slight variation. Here, we remove a literal and check to see whether it was relevant by testing rather than querying. During this testing, we would like the examples generated by Test to be of the "near-miss" kind. That is, to remove a literal from the condition of a hypothesis d-rule, we would like test examples in which that literal is not true. Otherwise, the test example is not informative about the relevance of that literal for the hypothesis d-rule. When called from Reduce, Test generates such test examples, and tests them as before.

7.3 Experimental Results

We have implemented the algorithms discussed so far as a system called LeXer, in Common Lisp. LeXer has been tested on two domains: (1) A variant of the STRIPS world (SW), with recursive d-rules; and (2) the Kanfer-Ackerman air-traffic control (ATC) task. The purpose is to see how feasible and effective the approach of learning d-rules from exercises is. Since our d-rule-based planner is guaranteed to be efficient, we evaluate the performance of the system only by its coverage on test problems. In particular, we have measured the coverage of learned d-rules on randomly selected test cases against the number of exercises supplied. The test problems are natural problems that occur in a domain and are among the hardest, whereas the exercises are selected problems such that they span all levels of difficulty. The exercise set at each data point, except the first, includes the exercise set at the previous data

point. The test set is, however, fixed for all the data points. Each exercise set has an almost equal number of exercises for all the goals at all levels of difficulty. Moreover, the exercises in a set are ordered in increasing order of difficulty. To expedite the experiments, we have supplied the solvers with the goal-subgoal hierarchies in both domains, so that X-Solve can focus its search in solving the exercises.

7.3.1 STRIPS World (SW)

This domain is the same as the one used in ExEL's experiments. The tasks in the domain have 7 levels of difficulty. Basic tasks such as opening and closing doors, and holding and releasing objects are the least difficult. Going from a room to another room is recursive. For example, to go to a goal room that is to the left of the starting room, go to the right-neighbor room of the goal room, where going to the right-neighbor room may follow the same rule. Moving or pushing a box to a neighboring room, from a room, is at the next level. The most difficult tasks involve starting from one room, and pushing a box from a second room to a third room.

Figure 7.5 gives the learning curve: coverage versus number of exercises. The test set contained 10 randomly generated problems for the goal of moving a box from one room to another room. Each exercise set contained almost equal number of randomly generated problems for each level of difficulty. LeXer required 5 self-test examples. At the end of training, LeXer came up with 24 d-rules for SW. These d-rules are such that they are applicable to any-sized grid. These results demonstrate that learning from exercises is a feasible approach to learn recursive d-rules.

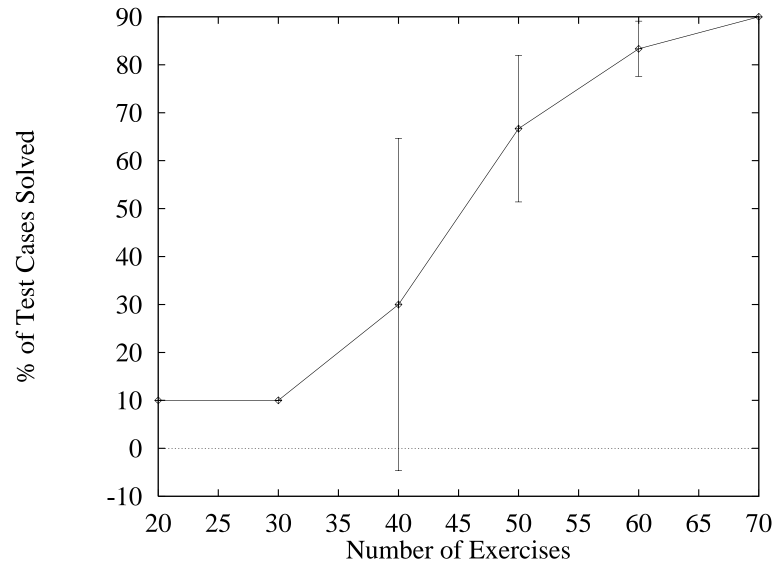

FIGURE 7.5: Performance of LeXer in SW

7.3.2 Air-Traffic Control Domain

This domain is the same as the one used in ExEL's experiments. The primitive actions in this domain are selecting a plane, moving the cursor between holding positions, and depositing a plane on a holding position or on a runway. The exercises that are the least difficult correspond to these basic tasks. There are 4 other gradations of difficulty: moving a plane from a holding level to the next holding level is the next gradation; landing a plane from the first holding level is the next difficult task; the most difficult exercises concern landing a plane without a runway being specified and from a holding level other than the first. The test problems were randomly generated with this task as the goal task.

Figure 7.6 gives the learning curve: coverage versus number of exercises. The test set contained 20 randomly generated problems for the goal of moving a box from one room to another room. Each exercise set contained almost equal number of

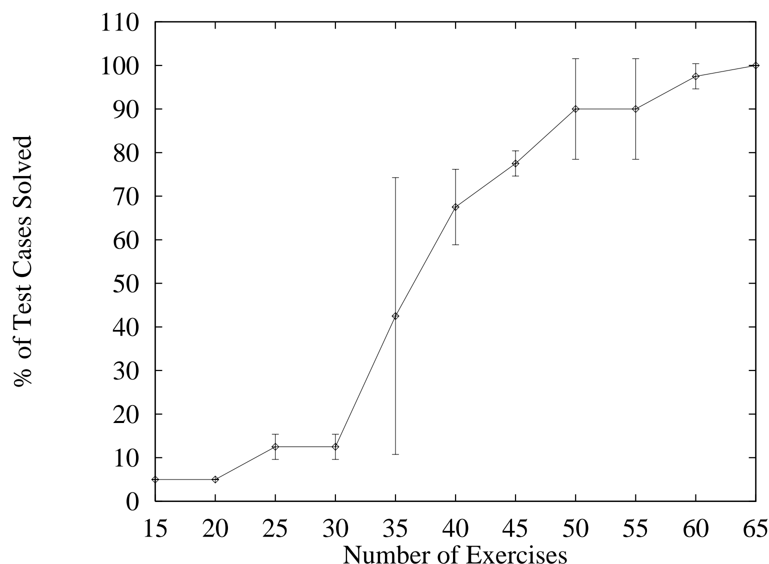

FIGURE 7.6: Performance of LeXer in ATC

randomly generated problems for each level of difficulty. LeXer required 4 self-test examples in this domain. LeXer came up with 14 d-rules for this domain at the end of training. They perform correctly on all 20 test cases. Here, too, each exercise set contained almost equal numbers of randomly generated problems for each level of difficulty.

The error bars in Figure 7.5 and Figure 7.6 indicate one standard deviation on either side of the mean. In both the domains, if an exercise cannot be solved within a maximum depth limit, it is abandoned, and no learning results from it. Maximum depth limits are set such that if an exercise cannot be solved, it is only due to the inadequacy of LeXer's learned d-rules. The inadequacy of LeXer's d-rules is due to the inadequacy of training. (In the plots, unsolved exercises are also counted when counting the number of exercises used by LeXer.) Some data points in both plots have large error bars. The reason for large error bars early on in both the plots is

that in some runs the exercises supplied resulted in LeXer learning good lower level d-rules, and thus enabling solving of and learning from higher-level exercises. On the other hand, in Figure 7.6, the large error bars later on are due to some runs getting unlucky: the randomly picked exercises had coincidences, in the sense that they were about the same plane or the same runway, which prevented the generalization of constants into variables in the d-rules learned from these exercises.

7.4 Discussion

We have shown that learning d-rules from exercises is a feasible and effective approach. The approach used in LeXer is directly related to speedup learning from examples and inductive logic programming (ILP). In the following, we position our work in the context of the related work in those two areas.

In the previous chapter, we have presented the ExEL system ,which learns d-rules from examples, i.e., problems and their solutions, unlike LeXer that learns from exercises. Another difference is as noted while discussing the Generalize-Self-Test algorithm. LeXer replaces queries with testing by generating "self-critical" examples.

Considering that planning without search-control knowledge is intractable, any unsupervised speedup learner may, in practice, need to behave like the learning-from-exercises systems to be successful. If an unsupervised learner gets a hard problem in the beginning, it may get bogged down in trying to solve the problem, unless there is a limit on the amount of time it can spend on a problem. As a result, learning may not be feasible, because there may not be enough of simple problems to learn from. If there are enough of simpler problems, then its behavior approaches that of learning from exercises. In any case, in the process, it may squander computing time and available training problems.

As we have seen in Chapter 3, a set of d-rules can be mapped to first-order Horn programs. In this respect, learning d-rules is an ILP problem. From (Cohen, 1995c), one can infer that this concept class is hard to learn without external help. Our system uses testing to learn this class of concepts. GOLEM, an ILP system, instead restricts the concept class to determinate clauses, which means in a clause there is a unique binding to all the free variables of a literal, given the bindings of all previous literals (Muggleton & Feng, 1990). In fact, this concept class with the further restriction of keeping the depth of the dependency chain of the variables in a clause constant, is shown to be PAC-learnable from examples (Cohen, 1995b; Džeroski et al., 1992). Unfortunately, these restrictions are too limiting on the expressiveness of d-rules to enable efficient planning. Therefore, we circumvent this problem by employing exercises and self-testing.

We have seen in Chapter 3 that a set of d-rules for multiple goals maps to a Horn program. In Chapter 5, we have discussed learnability of acyclic Horn programs using the PLearn algorithm. Here, we will explore the connection between PLearn and LeXer. The derivation order of literals for Horn programs corresponds to the order of difficulty for d-rules. Although learning is not explicitly ordered in PLearn, the process of finding prime consequents effectively serves this purpose. So, the next clause PLearn learns is a clause which can be constructed from literals derived from the clauses it already knows. Similarly, LeXer constructs a new d-rule whose subgoals are those that can be achieved using the existing learned d-rules. Another related similarity is in what the forward-chaining step of PLearn and the exercise-solving step (X-Solver) of LeXer accomplish. The forward-chaining step of PLearn introduces the literals that can be derived from the body of an example using the clauses learned so far. This ensures that the new clauses learned are as compact as, and are subsumed by, the clauses in the target program. The exercise-solving step

of in LeXer, similarly, introduces the subgoals that can be solved given the initial state of the exercise problem, using the d-rules learned so far. This makes the d-rules more compact. In both cases, these steps increase the generality or abstraction of the rules (clauses or d-rules) learned. PLearn, LeXer, and also DLearn, the Horn-definition learner, and ExEL, the example-based d-rule learner, all employ almost the same generalize-and-test method for the purpose of generalizing multiple examples. LeXer is an exception in that it employs self-testing instead of querying a teacher.

One other difference between our system and the above systems is in the way negative examples are generated.

DOLPHIN, Grasshopper and SCOPE consider the choices that did not lead to a goal as the negative examples of a “good” control rule. In our setting, the choices the exercise solver makes are over what subgoals to use. Since we obtain the subgoal literals of a prospective d-rule directly from the exercise solver, subgoal learning does not need any powerful inductive learner. So, the choice sequences of subgoals that did not lead to the goal are of no significant use for subgoal learning. However, they could possibly be used for learning the conditions part: each one is a negative example for a d-rule that has the matching goal and sequence of subgoals. This is because the sequence of subgoals in the example could not lead to the goal, because the example's conditions are wrong for the application of the subgoal sequence. However, instead of using these negative examples, LeXer generates self-critical examples for testing in an active manner, for the following connected reasons. First, not every hypothesis d-rule of the learner may have a negative example in the set collected during the search. Second, even if there are negative examples that address a hypothesis d-rule, they may not be near-miss examples. In LeXer, since the learner has control over which examples are generated, it can generate suitable near-miss examples.

The way the testing examples are generated and used is somewhat similar to the way Gil (1993, 1994) generates and uses experiments in learning operator models. Gil uses them to specialize overgeneral conditions, whereas we use test examples to generalize overspecific conditions. In our case, to eliminate a literal for generalizing a condition, we choose a literal that is already there in the condition. Gil, in contrast, needs to consider all possible literals and their various variablizations, to add to a condition that is to be specialized. To combat the explosive nature of this process, Gil employs heuristics to choose a literal to add.

The Marvin system by Sammut and Banerji (1986) also generates self-critical examples to validate its hypotheses in the context of concept learning. However, it depends on its teacher to tell whether the generated example of the hypothesis is an instance of the target concept. In effect, it is a membership query. As with LeXer, Marvin's learning is also ordered in increasing order of difficulty of the concepts to be learned. It depends on a single positive example to learn a concept. For generalization, it employs the ad-hoc process of turning constants into variables.

As promised in Section 6.4, we now compare Marsella's PRL system to LeXer. Recall that PRL learns Hierarchical Partial Orders (HPOs) to decompose problems into subproblems in a problem-solving framework (cf. Section 6.4). Like LeXer, PRL, too, takes as input problems without their solutions. However, PRL does not depend on ordering of problems to learn. It attempts to learn only when it fails to solve a problem by decomposing it into subproblems using existing problem-reduction rules. Failure occurs when there are no applicable reduction rules for the subproblem PRL is attempting. PRL then tries to solve the subproblem by bi-directional search. It expands partially to a "limited-depth" from the forward as well as the backward directions. It pairs each forward-frontier node with each backward-frontier node. It eliminates some pairs using some heuristics. The left-over pairs are

considered alternate hypotheses of a problem-reduction rule. It applies a critique which employs some further heuristics (such as preferring decompositions that allow independence between subproblems) to identify a hypothesis rule. This hypothesis rule is generalized by variablizing constants. One shortcoming PRL suffers from is that the number of pairings could be intractably high to even prune using heuristics, in some domains. Another shortcoming is in credit—or more accurately, blame—assignment for failure. PRL decides that the reason for a failure to solve a problem is a lack of reduction rules, rather than a possible fault with the existing rules. So, it goes on to learn new reduction rules including lower-level rules for the problem. Although the heuristic nature of the process does not justify that reason, PRL did not seem to be affected by this problem in learning for the Tile-World domain, because the reduction rules in the domain are of just two levels.

We discussed some of the representational advantages of HPOs over d-rules earlier in Section 6.4. On the learning front, PRL and BU-PRL employ a primitive generalization scheme which may not be robust across domains, whereas ExEL and LeXer makes use of the *lgg* method and other apparatus which are theoretically grounded and robust. An explanation for the quality of the generalization scheme in PRL and BU-PRL may be that Marsella emphasizes increasing the degree of independence between the subproblems generated in the learned HPO rules, as the main performance goal for PRL and BU-PRL, rather than the traditional goal of producing rules with high generality.

Chapter 8

CONCLUSIONS

In this chapter, we first summarize the research work in the dissertation, and then highlight contributions of this work. We end this chapter by looking back and identifying some interesting extensions to this work that can be tackled in future.

In the course of this dissertation, first we showed the need for learning control knowledge for hierarchical-decomposition planning. We did so by appealing (1) to the efficacy of hierarchical planning, (2) to the impracticality of total dependence on human experts for providing control knowledge, hence to the need for automatic learners, and (3) to the scarcity of such learners for hierarchical planning. We presented goal-decomposition rules (d-rules) as a way of representing hierarchical control knowledge. We then showed how planning can be performed based on d-rules, and how d-rules can be extended to be applicable for reactive planning. We also discussed how the d-rule representation for control knowledge is more advantageous than the other forms of representations such as control rules and macro operators. Next, in Chapter 3, we showed how d-rules can be seen as Horn clauses. There, we also showed that HTNs, a more general form of representation than d-rules for hierarchical planning, when restricted to positive literals, can be mapped to Horn clauses. We also observed that d-rules (HTN methods) for a single goal (task) can be seen as a set of Horn clauses with the same predicate symbol—or Horn definitions; and that d-rules (HTN methods) for multiple goals (tasks) can be seen as a set of Horn clauses—or Horn programs. The idea here was to exploit the techniques available in inductive logic programming (ILP) to learn d-rules for planning.

We then showed how Horn definitions can be learned tractably in the entailment setting from examples and membership queries. We noted that this result is applicable to the learning from interpretations setting also. In Chapter 5, we studied the learnability of Horn programs in the entailment setting, and showed that acyclic Horn programs restricted to those that have a polynomial-time forward-chaining procedure can be learned tractably from a combination of examples, membership queries, and derivation-order queries. We also identified syntactic restrictions on Horn programs that guarantee a polynomial-time forward-chaining procedure. As in the case of Horn definitions, this result is also applicable to the learning from interpretations setting.

In Chapters 6 and 7, we considered implementations of systems for learning d-rules. These systems have foundations in the theoretical algorithms developed for Horn definitions and Horn programs. ExEL system, described in Chapter 6, learns d-rules from examples, each of which comprises a planning problem and its solution as a sequence of actions. ExEL first converts an input example into the d-rule form. It then employs an incremental generalize-and-test algorithm to identify a correct hypothesis d-rule for combining with the example d-rule at hand. This algorithm uses the least-general-generalization (lgg) procedure to combine an example with a correct d-rule. The algorithm depends on membership queries to identify a correct hypothesis d-rule to generalize with. To keep the size of d-rules tractably small, literals are removed with the help of membership queries. This part of the algorithm is the same as the DLearn algorithm for learning Horn definitions. ExEL learns d-rules for one goal at a time, to learn d-rules for multiple goals. Also, ExEL explicitly refines the order of subgoals in a hypothesis d-rule.

In Chapter 7, we described a system, LeXer, that alleviates the burden, on the part of the teacher, of providing solved examples and answering learner's queries. LeXer takes as input a sequence of planning problems and subproblems that are

ordered in increasing order of difficulty. Except from having to order the problems, the teacher is freed from the tedium of solving all the problems, and from having to answer queries. LeXer solves each input problem by itself by performing an iterative-deepening depth-first search using previously learned d-rules as operators. From the solving process, it gathers a sequence of subgoals that achieve the goal of the problem being solved. Using this and the initial state of the problem, it constructs an example d-rule. It incorporates this example d-rule into the existing d-rules by employing the generate-and-test algorithm similar to DLearn, except that it replaces queries by self-testing. LeXer tests a hypothesis d-rule by generating a problem whose initial state satisfies the conditions of the d-rule and whose goal literal matches with the goal of the d-rule. It, then, asks the d-rule planner to solve the problem by attempting to solve the subgoals suggested by the decomposition in the hypothesis d-rule. If the d-rule planner cannot solve the problem with the suggested decomposition, then the hypothesis d-rule is considered incorrect. Otherwise, it generates another problem and tests again. This process is repeated until it is successful on a certain number of test problems, or until it fails. Also, in Chapter 7, we explored the many close similarities between learning from exercises in LeXer and the learning algorithm for acyclic Horn programs.

8.1 Contributions

We highlight here some of the contributions the thesis makes.

- We showed a mapping from d-rules to Horn clauses. We extended this mapping and showed how HTN methods with only positive literals can be transformed into the Horn-clause form. A consequence of these mappings is that learnability of Horn clauses implies learnability of d-rules and HTNs. There has been

a great body of past and current work devoted to learning first-order Horn clauses in the field of inductive logic programming (Muggleton, 1992; Lavrač & Džeroski, 1994; Bergadano & Gunetti, 1996; Nienhuys-Cheng & de Wolf, 1997). Thus, research work on learning d-rules and HTN methods can exploit techniques developed in ILP, in addition to the traditional speedup-learning methods (Reddy & Tadepalli, 1997a).

- In Chapter 4, we showed the learnability of first-order non-recursive Horn definitions from examples and membership queries (Reddy & Tadepalli, 1997c, 1998b). Earlier work on this front was limited to language classes that have hypothesis spaces of polynomial size. The language class we studied has unbounded size. Also, we do not assume that the alphabet of the language—such as of the predicate and function symbols—is known a priori. We also showed how d-rules can be learned by applying the algorithm for learning Horn definitions.

- In Chapter 5, we showed that acyclic first-order Horn programs that have polynomial-time forward-chaining procedure are learnable given the derivation order of the literals from examples and membership queries (Reddy & Tadepalli, 1998a). Earlier work was limited to learning language classes that either have their number of clauses limited or have the restriction that the clauses have no local variables. In our result, the number of clauses can be variable. We allow local variables in the body of a clause, but we restrict that the clauses to be non-generative—that is, the head of a clause cannot have terms that are not already present in the clause's body. This suits d-rule learning, in general speedup learning, well, because we are likely to operate on objects that satisfy a set of conditions.

- In Chapter 6, we demonstrated that control knowledge that is organized hierarchically and represented efficiently as d-rules is learnable from examples in a system called ExEL (Reddy, Tadepalli, & Roncagliolo, 1996). Each example comprises a planning problem and its solution as a sequence of actions. This could be perceived as making a flat state-space planner into a more efficient hierarchical planner. In this view, the learned knowledge helps speed up planning. There is another view. It has been argued that while building real-world planners one of the hard and tedious tasks is to acquire and encode HTN schemata (Chien, 1996). In this context, ExEL can be viewed as a tool for automatic knowledge acquisition.

- ExEL depends in two places on a domain expert or teacher. One is in expecting solved problems as input, and the other in expecting answers to its membership queries. To further automate the task of knowledge acquisition, these two expectations need to be removed. In the LeXer system, described in Chapter 7, we attempted to reduce this burden on a teacher (Reddy & Tadepalli, 1997b). LeXer does not expect solved problems as input. Instead, it expects as input a sequence of unsolved problems in increasing order of difficulty—that is, exercises. Further, instead of querying a teacher, LeXer tests its hypotheses by itself. There is, however, still some dependence on a teacher to order the problems. This is similar to the requirement by PLearn that the derivation order of literals be given for learning Horn programs. Lifting this requirement is, however, an open problem.

8.2 Future Work

We have seen in Chapter 2 that HTNs are more general than d-rules. Although we have theoretically indicated the learnability of HTN methods by way of translating HTN methods to Horn clauses, and showing the learnability of Horn clauses, we have not demonstrated it through implementations. The crucial issue here is in formulating a given example into the HTN notation. Once we have the examples in the HTN form, we can make use of the theoretical algorithms to implement learning algorithms for HTN methods.

Similarly, we have not dealt with learning monitors for d-rules for the purpose of reactive planning. The main issue here, too, is in obtaining good examples when the planner is in reacting setting. Another important issue is in implementing the self-testing process to answer queries. The feature of the learner generating self-critical problems, as in LeXer, cannot be implemented in a reactive setting, because all literals in a state cannot be set in the way the learner wants—for example, weather is not under the control of the learner.

In Chapter 1, we identified two opportunities for learning in the context of hierarchical-decomposition planning. One is to learn task decompositions, and the other is to learn to choose among various task decompositions. In this dissertation, we concentrated only on the former. For the latter, it appears that existing techniques such as in Prodigy (Minton, 1988) and SCOPE (Estlin, 1998) developed for state-space and plan-space planners are applicable. The validity of this conjecture, however, needs to be examined.

The ATC domain we used in the experiments in Chapters 6 and 7 is only a part of the task. The main task of the ATC domain is in fact an optimization task of landing as many planes as possible in a given time. This task requires

decisions such as which planes to land, and which runways to land on. We tackled this problem by encoding by hand preference rules that choose among alternative goals and subgoals. Learning such rules requires a learning mechanism that takes costs into account, such as reinforcement learning. Džeroski, Blockeel, and De Raedt (1998) address reinforcement learning in the context of first-order learning. Dietterich (1998) addresses reinforcement learning for hierarchical domains.* We presume that an appropriate combination of the ideas in these two methods should be suitable for learning preference rules.

In Chapter 5, we showed that acyclic Horn programs with some restrictions are learnable from examples, with the help of membership and derivation-order queries. One of the restrictions we impose is that the clauses be non-generative—that is, in each clause, only the terms and subterms that appear in the body of the clause appear in the head of the clause. Arimura (1997), on the other hand, shows learnability of Horn programs that are constrained so that in each clause only the terms and subterms that appear in the head of the clause appear in the body of the clause. The class addressed by him, called simple Horn programs, did not allow local variables, unlike our case, but had the other restrictions our class had. Independently of our work, Krishna Rao and Sattar (1998) showed that simple Horn programs with local variables are learnable. Although this class generalizes the class addressed by Arimura (1997), it is not more general than our class. Moreover, they assume mode declarations—indications that say whether a variable in a predicate is an input or an output variable—which we cannot assume in the context of our planning application. We believe that there is a more general class which generalizes the class addressed by us and the class addressed by Krishna Rao and Sattar (1998) that is learnable. Such

* In fact, the MAXQ hierarchical learning system described by Dietterich (1998) was used to learn non-first-order value functions for the ATC optimization task.

a more general algorithm may be applicable to a variety of tasks both in planning and in logic programming.

Another direction of future work is in establishing PAC-learnability of d-rules in the framework of learning from exercises. Natarajan (1989) first showed the PAC-learnability of control rules from exercises. Tadepalli and Natarajan (1997) extended this to the PAC-learnability of macro operators from exercises. The control rules addressed by Natarajan (1989) are essentially propositional. The macro operators addressed by Tadepalli and Natarajan (1997) are not recursive. The d-rules, on the other hand, are first-order representations and can be recursive. Thus, it would be significant to further extend the PAC-learnability from exercises to d-rules also. Some of the solutions we addressed in Chapters 4 and 5 for learning first-order representations and in Chapter 7 for learning recursive d-rules from exercises should be useful in establishing this result.

BIBLIOGRAPHY

Ackerman, P., & Kanfer, R. (1993). *Kanfer-Ackerman Air Traffic Control Task© CD-ROM Database, Data-Collection Program, and Playback Program.* Dept. of Psychology, Univ. of Minn., Minneapolis, MN.

Angluin, D. (1988). Queries and concept learning. *Machine Learning*, *2*, 319–342.

Angluin, D., Frazier, M., & Pitt, L. (1992). Learning conjunctions of horn clauses. *Machine Learning*, *9*, 147–164.

Arimura, H. (1997). Learning acyclic first-order horn sentences from entailment. In *Proceedings of the Eigth International Workshop on Algorithmic Learning Theory*. Ohmsha/Springer-Verlag.

Bergadano, F., & Gunetti, D. (1996). *Inductive Logic Programming: From Machine Learning to Software Engineering.* MIT Press, Cambridge, MA.

Blum, A., & Furst, M. (1997). Fast planning through planning graph analysis. *Artificial Intelligence*, *90*(1), 281–300.

Chien, S. (1996). Static and completion analysis for planning knowledge base development and verification. In *Proceedings of the Third International Conference on AI Planning Systems (AIPS-96).*

Chien, S., Estlin, T., & Wang, X. (1997). An argument for hybrid HTN/Operator planning. In *Proceedings of the Fourth Europian Conference on Planning.*

Cohen, W. (1995a). Pac-learning non-recursive prolog clauses. *Artificial Intelligence*, *79*(1), 1–38.

Cohen, W. (1995b). Pac-learning recursive logic programs: efficient algorithms. *Jl. of AI Research*, *2*, 500–539.

Cohen, W. (1995c). Pac-learning recursive logic programs: negative results. *Jl. of AI Research*, *2*, 541–573.

Currie, K., & Tate, A. (1991). O-plan: The open planning architecture. *Artificial Intelligence*, *51*(1), 49–86.

De Raedt, L. (1997). Logical settings for concept learning. *Artificial Intelligence*, *95*(1), 187–201.

De Raedt, L., & Bruynooghe, M. (1992). Interactive concept learning and constructive induction by analogy. *Machine Learning*, *8*(2), 107–150.

DeJong, G., & Mooney, R. (1986). Explanation-based learning: An alternative view. *Machine Learning*, *1*, 145–176.

Dietterich, T. (1998). The MAXQ method for hierarchical reinforcement learning. In *Proceedings of the 15th International Conference on Machine Learning*. Morgan Kaufmann.

Dietterich, T., London, B., Clarkson, K., & Dromey, G. (1982). Learning and inductive inference. In Cohen, P., & Feigebaum, E. (Eds.), *The Handbook of AI*, Vol. 3. Morgan Kaufmann, San Mateo, CA.

Džeroski, S., Blockeel, H., & De Raedt, L. (1998). Relational reinforcement learning. In *Proceedings of the 15th International Conference on Machine Learning; (and Proceedings of the 8th International Conference on Inductive Logic Programming)*. Morgan Kaufmann.

Džeroski, S., Muggleton, S., & Russell, S. (1992). Pac-learnability of determinate logic programs. In *Proceedings of the Fifth Annual ACM Workshop on Computational Learning Theory*, pp. 128–135.

Erol, K. (1995). *Hierarchical Task Network Planning: Formalization, Analysis, and Implementation*. Ph.D. thesis, University of Maryland, College Park.

Erol, K., Nau, D., & Subrahmanian, V. (1995). Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence*, *76*, 75–88.

Estlin, T. (1998). *Using Multi-Strategy Learning to Improve Planning Efficiency and Quality*. Ph.D. thesis, University of Texas, Austin, TX.

Fikes, R., Hart, P., & Nilsson, N. (1972). Learning and executing generalized robot plans. *Artificial Intelligence*, *3*, 251–288.

Frazier, M., & Page, C. D. (1993). Learnability in inductive logic programming: Some basic results and techniques. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*, pp. 120–127. AAAI Press.

Frazier, M., & Pitt, L. (1993). Learning from entailment: An application to propositional Horn sentences. In *Proceedings of the Tenth International Conference on Machine Learning*, pp. 120–127.

Frazier, M., & Pitt, L. (1995). Classic learning. *Machine Learning*, *25*, 151–194.

Gil, Y. (1993). Efficient domain-independent experimentation. In *Proceedings of the Tenth International Conference on Machine Learning*, pp. 128–134.

Gil, Y. (1994). Learning by experimentation: Incremental refinement of incomplete planning domains. In *Proceedings of the Eleventh International Conference on Machine Learning*, pp. 87–95.

Haussler, D. (1989). Learning conjunctive concepts in structural domains. *Machine Learning*, *4*, 7–40.

Kambhampati, S. (1997). Refinement planning as a unifying framework for plan synthesis. *AI Magazine*, *18*(2), 67–98.

Kambhampati, S., Katukam, S., & Qu, Y. (1996). Failure-driven search control for partial-order planners: An explanation-based approach. *Artificial Intelligence*, *88*.

Kambhampati, S., Knoblock, C., & Yang, Q. (1995). Planning as refinement search: a unified framework for evaluating design tradeoffs in partial-order planning. *Artificial Intelligence*, *76*, 167–238.

Kautz, S., & Selman, B. (1996). Pushing the envelope: Planning propositional logic and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pp. 1194–1201.

Khardon, R. (1996). Learning to take actions. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pp. 787–792.

Khardon, R. (1998). Learning first order universal horn expressions. In *Proceedings of the Eleventh Annual Conference on Computational Learning Theory (COLT-98)*.

Korf, R. (1987a). Depth-first iterative deepening: An optimal admissible tree search. *Artificial Intelligence*, *27*(1), 97–109.

Korf, R. (1987b). Planning as search: A quantitative approach. *Artificial Intelligence*, *33*, 65–88.

Kowalski, R. (1970). The case for using equality axioms in automatic demonstration. In *Lecture Notes in Mathematics*, Vol. 125. Springer-Verlag.

Krishna Rao, M., & Sattar, A. (1998). Learning from entailment of logic programs with local variables. In *Proceedings of the Ninth International Workshop on Algorithmic Learning Theory (To appear)*. Ohmsha/Springer-Verlag.

Laird, J., Rosenbloom, P., & Newell, A. (1986). Chunking in SOAR: The anatomy of a general learning mechanism. *Machine Learning*, *1*(1), 11–46.

Lassez, J.-L., Maher, M., & Marriott, K. (1988). Unification revisited. In Minker, J. (Ed.), *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann.

Lavrač, N., & Džeroski, S. (1994). *Inductive Logic Programming: Techniques and Applications*. Ellis-Horwood.

Leckie, C., & Zukerman, I. (1993). An inductive approach to learning search control rules for planning. In *Proceedings of the 13th IJCAI*, pp. 1100–1105.

Lloyd, J. (1987). *Foundations of Logic Programming* (2nd ed.). Springer-Verlag, Berlin.

Mahadevan, S., Mitchell, T., Mostow, J., Stienberg, L., & Tadepalli, P. (1993). An apprentice-based approach to knowledge acquisition. *Artificial Intelligence*, *64*, 1–52.

Mali, A., & Kambhampati, S. (1998). Encoding HTN planning in propositional logic. In *Proceedings of the Fourth International Conference on AI Planning Systems (AIPS-98)*.

Marsella, S. (1993). *Planning under the Restriction of Hierarchical Partial Orders*. Ph.D. thesis, Rutgers University, NJ.

Marsella, S., & Schmidt, C. (1993). A method for biasing the learning of nonterminal reduction rules. In Minton, S. (Ed.), *Machine Learning Methods for Planning*. Morgan Kaufmann, San Mateo, CA.

Minton, S. (1988). *Learning Search Control Knowledge*. Kluwer Academic Publishers, Boston, MA.

Mitchell, T. M., Keller, R. M., & Kedar-Cabelli, S. T. (1986). Explanation-based generalization: A unifying view. *Machine Learning*, *1* (1), 47–80.

Mitchell, T., Utgoff, P., & Banerji, R. (1983). Learning by experimentation: Acquiring and refining problem-solving heuristics. In Michalski, R., & et al. (Eds.), *Machine learning: An artificial intelligence approach*, Vol. 1. Morgan Kaufmann.

Muggleton, S. (Ed.). (1992). *Inductive Logic Programming*. Academic Press, New York, NY.

Muggleton, S., & Feng, C. (1990). Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory*, pp. 368–381. Ohmsha/Springer-Verlag.

Natarajan, B. (1989). On learning from exercises. In *Proceedings of the Second Workshop on Computational Learning Theory*, pp. 72–87. Morgan Kaufmann.

Natarajan, B. (1991). *Machine Learning: A Theoretical Approach*. Morgan Kaufmann, San Mateo, CA.

Nienhuys-Cheng, S.-H., & de Wolf, R. (1996). Least generalizations and greatest specializations of sets of clauses. *Jl. of AI Research*, *4*, 341–363.

Nienhuys-Cheng, S.-H., & de Wolf, R. (1997). *Foundations of Inductive Logic Programming*. Springer, New York.

Page, C. D. (1993). *Anti-Unification in Constraint Logics: Foundations and Applications to Learnability in First-Order Logic, to Speed-up Learning, and to Deduction*. Ph.D. thesis, University of Illinois, Urbana, IL.

Page, C. D., & Frisch, A. M. (1992). Generalization and learnability: A study of constrained atoms. In Muggleton, S. H. (Ed.), *Inductive Logic Programming*, pp. 29–61. Academic Press.

Penberthy, J., & Weld, D. (1992). UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of Third International Conference on Principles of Knowledge Representation and Reasoning*.

Plotkin, G. (1970). A note on inductive generalization. In Meltzer, B., & Michie, D. (Eds.), *Machine Intelligence*, Vol. 5, pp. 153–163. Elsevier North-Holland, New York.

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

Reddy, C., & Tadepalli, P. (1997a). Inductive logic programming for speedup learning. In DeRaedt, L., & Muggleton, S. (Eds.), *Proceedings of the IJCAI-97 workshop on Frontiers of Inductive Logic Programming*.

Reddy, C., & Tadepalli, P. (1997b). Learning goal-decomposition rules using exercises. In *Proceedings of the 14th International Conference on Machine Learning*. Morgan Kaufmann.

Reddy, C., & Tadepalli, P. (1997c). Learning Horn definitions using equivalence and membership queries. In *Proceedings of the 7th International Workshop on Inductive Logic Programming*. Springer Verlag.

Reddy, C., & Tadepalli, P. (1998a). Learning first-order acyclic horn programs from entailment. In *Proceedings of the 15th International Conference on Machine Learning; (and Proceedings of the 8th International Conference on Inductive Logic Programming)*. Morgan Kaufmann.

Reddy, C., & Tadepalli, P. (1998b). Learning Horn definitions: Theory and an application to planning. *New Generation Computing*, *17*.

Reddy, C., Tadepalli, P., & Roncagliolo, S. (1996). Theory-guided empirical speedup learning of goal-decomposition rules. In *Proceedings of the 13th International Conference on Machine Learning*, pp. 409–417. Morgan Kaufmann.

Rosenbloom, P., & Laird, J. (1986). Mapping explanation-based generalization onto SOAR. In *Proceedings of AAAI-86*. AAAI Press.

Ruby, D., & Kibler, D. (1991). Learning subgoal sequences for planning. In *Proceedings of AAAI-91*. AAAI Press.

Russell, S., & Norvig, P. (1995). *Artificial Intelligence: A Modern Approach*. Prentice Hall, Englewood Cliffs, NJ.

Sacerdoti, E. (1977). *A Structure for Plans and Behavior*. Elsevier/North-Holland, New York, NY.

Sammut, C. A., & Banerji, R. (1986). Learning concepts by asking questions. In *Machine learning: An artificial intelligence approach*, Vol. 2. Morgan Kaufmann.

Schapire, R. (1990). The strength of weak learnability. *Machine Learning*, *5*, 197–227.

Shavlik, J., & Dietterich, T. (1990). *Readings in Machine Learning.* Morgan Kaufmann, San Mateo, CA.

Shavlik, J. (1990). Acquiring recursive and iterative concepts with explanation-based learning. *Machine Learning*, *5*, 39–70.

Subramanian, D., & Feldman, R. (1990). The utility of EBL in recursive domain theories. In *Proceedings of AAAI-90* Menlo Park, CA. AAAI Press.

Sussman, G. (1975). *A Computer Model of Skill Acquisition.* Elsevier/North-Holland, New York, NY.

Tadepalli, P., & Natarajan, B. (1997). Learning to solve problems from exercises. in preparation.

Tambe, M., Newell, A., & Rosenbloom, P. (1990). The problem of expensive chunks and its solution by restricting expressiveness. *Machine Learning*, *5*, 299–348.

Tate, A. (1977). Generating project networks. In *Proceedings of IJCAI-77*, pp. 888–893.

Valiant, L. (1984). Theory of the learnable. *Communications of the ACM*, *27*, 1134–1142.

Veloso, M., Pollack, M., & Cox, M. (1998). Rationale-based monitoring for planning in dynamic environments. In *Proceedings of the Fourth International Conference on AI Planning Systems (AIPS-98).*

Wilkins, D. (1988). *Practical Planning: Extending the Classical AI Planning Paradigm.* Morgan Kaufman, San Fransisco, CA.

Zelle, J., & Mooney, R. (1993). Combining FOIL and EBG to speedup logic programs. In *Proceedings of the 13th IJCAI*, pp. 1106–1111.