EFFICIENT LEARNING OF RELATIONAL MODELS FOR SEQUENTIAL DECISION MAKING

BY THOMAS J. WALSH

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Michael L. Littman

and approved by

New Brunswick, New Jersey

October, 2010

ABSTRACT OF THE DISSERTATION

Efficient Learning of Relational Models for Sequential Decision Making

by Thomas J. Walsh

Dissertation Director: Michael L. Littman

The exploration-exploitation tradeoff is crucial to reinforcement-learning (RL) agents, and a significant number of sample complexity results have been derived for agents in propositional domains. These results guarantee, with high probability, near-optimal behavior in all but a polynomial number of timesteps in the agent's lifetime. In this work, we prove similar results for certain *relational* representations, primarily a class we call "relational action schemas". These generalized models allow us to specify state transitions in a compact form, for instance describing the effect of picking up a generic block instead of picking up 10 different specific blocks. We present theoretical results on crucial subproblems in action-schema learning using the KWIK framework, which allows us to characterize the sample efficiency of an agent learning these models in a reinforcement-learning setting.

These results are extended in an apprenticeship learning paradigm where and agent has access not only to its environment, but also to a teacher that can demonstrate traces of state/action/state sequences. We show that the class of action schemas that are efficiently learnable in this paradigm is strictly larger than those learnable in the online setting. We link the class of efficiently learnable dynamics in the apprenticeship setting to a rich class of models derived from well-known learning frameworks.

As an application, we present theoretical and empirical results on learning relational models of web-service descriptions using a dataflow model called a Task Graph to capture the important connections between inputs and outputs of services in a workflow, with experiments constructed using publicly available web services. This application shows that compact relational models can be efficiently learned from limited amounts of basic data.

Finally, we present several extensions of the main results in the thesis, including expansions of the languages with Description Logics. We also explore the use of sample-based planners to speed up the computation time of our algorithms.

Acknowledgements

I want to thank a few people for their contributions to this work and my career.

First, I want to thank my advisor, Michael Littman, who has been an extraordinarily patient and wise mentor throughout this process. I am eternally grateful for the faith he showed in me and the advice he imparted to me over the years. Michael's insights and sense of humor (puns and all) make his research lab a fantastic environment, and I am so proud to have been his student.

I also thank Alex Borgida, who was a mentor to me a Rutgers and worked with me on a number of different research projects, several of which are described in this thesis. Alex's openness to new research topics, wide breadth of knowledge, and endless supply of cookies were wonderful resources for me throughout grad school.

Also, thank you to the other members of my committee, Chung-chieh Shan and Roni Khardon, for helping to shape and clarify the ideas in this document. In particular, I want to thank Roni for his careful attention to detail throughout the whole thesis, which helped fix a number of errors and unclear statements from earlier versions.

Many portions of this thesis were expanded from earlier collaborative work, and while I have mentioned these papers in the corresponding chapters, I would like to thank Michael Littman, Alex Borgida, István Szita, Carlos Diuk, Kaushik Subramanian, and Sergiu Goschin, each of whom collaborated with me on these earlier works. I also thank Lihong Li, who worked with me on a number of publications and projects, and whose research echoes throughout this document. I would also like to thank my co-authors from other publications that were instrumental in my growth as a researcher: Bethany Leffler, Alex Strehl, Haym Hirsch, Ali Nouri, Fusun Yaman, Marie desJardins, and Rick Kuhn. A special thank-you goes to Marie desJardins, who was also my undergraduate advisor at UMBC. Her guidance and patience in those early years helped set me on the path I am on today.

While many of them have been mentioned above, I want to explicitly thank the members (past and present) of the RL^3 lab, with whom I have worked, laughed, and weathered the storms of grad school. To the original cast of Alex Strehl, Carlos Diuk, Bethany Leffler, Ali Nouri, and Lihong Li: you guys were there from the beginning and shared many of the moments of both exuberance and doubt that came in those early years. I can't think of a better group of people to be trapped in a van with on the way to Pittsburgh. And to the "next generation" of John Asmuth, Chris Mansley, Monica Babes, Michael Wunder, Ari Weinstein, Sergiu Goshin, and Kaushik Subramanian: thank you for your friendship and trips to "fancy lunch" over the last few years—I feel that this transition is leaving the lab in a high-reward state and that the future is very bright for all of you. Also, I want to specifically thank John, Ari, Chris, Sergiu, and Monica for carefully proofreading portions of this document.

Last but certainly not least, I'd like to thank my friends and family who have supported me throughout this whole endeavor. Specifically I'd like to thank my immediate family—my mother and father, Nancy and Tom, and my siblings Kathleen, Kenneth, and Susan for their moral support over the years.

Thank you all for helping to shape this document, and me as well.

Table of Contents

List of Tables

List of Figures

Chapter 1
Introduction

1.1 The Art of the State and the State of the Art

It has been called a “state” (Sutton & Barto, 1998), a “context” (Martino et al., 2006), a “situation” (McCarthy, 1963), and a myriad of other names, but whatever term you use, it’s a description of the important facts or impressions of the world that influence our decisions. Since this concept is central to the goals of Artificial intelligence (AI) (Russell & Norvig, 1995), it is no surprise that so much work has been done on creating better *representations* to encode states. But somewhat paradoxically, while early work in AI employed powerful generalized models steeped in logic (McCarthy, 1963), most of the modern works in the reinforcement learning (RL) (Sutton & Barto, 1998) subfield of AI instead use “bare bones” propositional representations. This thesis is primarily about RL algorithms that use more powerful representations, akin to the early AI models.

Traditional propositional representations of states and dynamics can model domains like the grid world in Figure 1.1 in a number of ways. For instance they might use

- A *discrete flat* state space where every square in the grid is its own state and there is no generalization on the dynamics across squares.
- A *continuous factored* state space where the state might be described as continuous coordinates of the agent.
- A propositional *discrete factored* state space over discrete x and y values with some known structure (like independence of x and y when moving), but where each possible x and y value needs to be learned about independently.

Each of these representations has benefits and drawbacks and each one might be the right representation for different tasks. The point here is not that there is one best representation, but that the engineering of states, and of domain descriptions in general, is really an art in

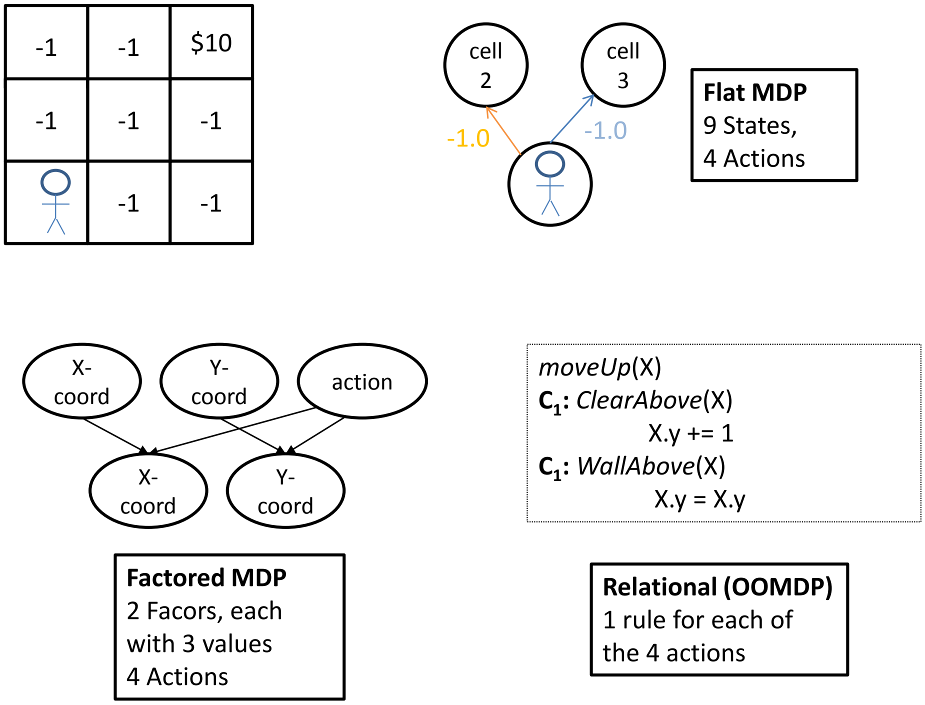


Figure 1.1: A simple RL grid world and some different representational views.

today's machine learning. The representations listed above are common in the literature, and many algorithms have been developed for using these models, but as we will soon see, they are fundamentally limited in their ability to compactly represent many domains. This thesis concentrates instead on *relational* representations that, in this case, could model dynamics like "moving up increases the y coordinate by 1" (bottom right in Figure 1.1), a kind of generalization not available to the "flat" propositional learners.

In the rest of this section, we describe some background on traditional reinforcement learning problems, relational representations, and our goal of melding the two. We cover most of the issues described below in far greater detail throughout the thesis, particularly the expanded background description in Chapter 2, but here we provide a brief overview of the main results of this document.

1.1.1 Exploration vs. Exploitation: A Classic Trade-off

One of the most prominent dilemmas in the reinforcement learning (Sutton & Barto, 1998) paradigm, is determining when an agent should perform actions that *exploit* its knowledge about its environment versus actions that *explore* the environment—better refining its model of the dynamics so it can later exploit a more complete model. For instance, suppose an agent in the simple grid world above knew about all the rewards in each of the cells except for the top-right cell, where it is uncertain what the reward is. The decision a learning agent faces in such a situation, is whether to exploit its current knowledge of the world by performing actions that avoid the unknown cell, *or* go to the top right corner and find out what the reward value is there. Notice there is a some risk here. If the top right corner actually had a reward of -10, all the steps going there would essentially be wasted. On the other hand, if the agent never went to the top right, it might consistently miss out on a reward of $+10$. The trade-off is non-trivial because in real domains, agents can't realistically be expected to explore every possible state of the world, and especially shouldn't if they have already found an optimal way to perform a task.

Making decisions about when to explore, and when to just exploit the known model is the crux of many works in reinforcement learning, especially in model-based RL. However, almost all such works have focused on the limited propositional representations described above. For instance, several works (Kakade, 2003; Strehl et al., 2009; Auer et al., 2007) have derived *sample complexity* bounds for flat MDPs. These results bound the worst-case number of steps where an agent is acting suboptimally or limit the amount of lost reward (the so-called regret) incurred over the agent's lifetime. However, in both cases, these (polynomial) bounds are based (among other factors) on the number of states in the domain. While this is acceptable for tiny grid worlds like the one in Figure 1.1, in the real world, such bounds are not as helpful since the number of ground states based on the agent's sensors may be extremely large.

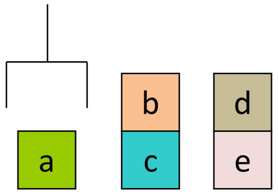
Moving up the generalization ladder, a number of works (such as Kearns & Koller (1999); Strehl et al. (2007)) have also derived sample complexity bounds for more general propositional models, such as Dynamic Bayesian Networks (DBNs) (Dean & Kanazawa, 1989), an example of which appears in the bottom left of Figure 1.1. In learning DBNs, the exploration-exploitation trade-off can be reformulated as one where the agent has to choose between exploiting the part of the Bayes net structure and parameters it is certain of, versus exploring and performing actions that will teach it more about the true underlying structure and parameters of the network. In this case however, the sample complexity bounds do not depend explicitly on the size of the state space. Instead, the bounds are only polynomial in the number of factors (assuming they have small in-degree), which is often logarithmic in the number of states. This thesis is mainly focused on deriving bounds that similarly scale sublinearly with the size of the state space, but in domains described using an even more powerful representation: relational models.

1.1.2 Relational Representations

While algorithms using propositional representations have performed admirably in the domains they are built for, their weak representational power limits the type of domains they can actually be applied in. Their main deficiency is that they cannot scale in large domains with repeated structure in the dynamics. For instance, consider the classic *Blocks World* domain illustrated in Figure 1.2. The domain consists of a set of $|\mathcal{O}|$ objects, specifically $|\mathcal{O}| - 2$ blocks, a hand, and a table. Blocks can be picked up and put down, or in some variants, moved from one block to another in one fell swoop. Either way, there is considerable repetition in the domain dynamics. Picking up block **a** has virtually the same effect as picking up block **b**; only the name of the block is different. In such cases, an agent that learns the dynamics of picking up *any* block X has a significant advantage over an agent that reasons about each individual object separately. But none of the propositional representations above make use of this structure. The number of states in the flat MDP model of blocks world grows exponentially with the number of blocks (there are over $400,000$ states with just 8 blocks). And the propositional DBN does not fare much better, because it needs to add factors for every ground proposition (On(**a**, **b**) and On(**b**, **c**) need to be separate factors), and most of these factors are highly inter-related (the in-degree of each factor grows with the number of objects), so both representations end up being exponentially large in the number of objects.

In contrast, *relational* representations exploit this repeated structure in the domain dynamics resulting in models whose size is independent of the number of objects in a domain. They do so by representing transitions with compact rules that we generally call *relational action schemas*. An example of such a rule is the stochastic STRIPS operator for the *move* action shown in Figure 1.2. This action description uses variables (X, From, To) to represent generic objects being moved. In this specific language, pre-conditions enforce type and state constraints that must hold for the action to work and Add/Delete lists describe the (again variablized) probabilistic effects of an action. By using variables, the effects of an action can be described regardless of the number of ground atoms in the domain, perfectly capturing the intuition about moving generic blocks described earlier.

In recent years, a number of efforts have been made to use such relational state and action



On(b, c), On(c, table), On(a, table), On(e, table), On(d, e), Clear(a), Clear(b), Clear(e)

move(X, From, To):

PRE: *On*(X, From) , *Clear*(X) , *Clear*(To) , *Block*(X), *Block*(To)

ω_1: (Pr = 0.9)
ADD: *On*(X, To) , *Clear*(From)
DEL: *On*(B, From) , *Clear*(To)

ω_2: (Pr = 0.1)
ADD:
DEL:

Figure 1.2: Blocks world and a Stochastic STRIPS operator for the *move* action. 10% of the time the action has no effect.

descriptions, mostly within the fields of *action schema learning* (such as Pasula et al. (2007)) and *relational reinforcement learning* (RRL) (Dzeroski et al., 2001). Work in action schema learning usually focusses on taking logs of experience data (either fully observable state-action trajectories (Pasula et al., 2007) or partially observable action trajectories (Shahaf, 2007; Zhuo et al., 2009)), with the goal of inferring the correct action models, such as the Stochastic STRIPS rule in Figure 1.2. In contrast, work in relational reinforcement learning usually has more traditional RL goals—instead of having a target model, an agent is judged based on the quality of its policy. Most RRL work has considered a situation where experience is collected using epsilon-greedy (randomized) or some other heuristic exploration (Dzeroski et al., 2001; Croonenborghs et al., 2007b; Lang et al., 2010). From there, algorithms from the Inductive Logic Programming (ILP) (Nienhuys-Cheng & de Wolf, 1997) community are used, usually to create relational representations of the value function itself (for instance a relational decision tree). While other methods have come closer to our own in trying to create MDP models with these ILP techniques (such as Croonenborghs et al. (2007b)), they are usually *batch* approaches—data is either already stored or naively collected. In contrast to a large portion of the literature in traditional reinforcement learning, and also in contrast to the emphasis on sample complexity results in behavioral cloning with relational models (Khardon, 1999b), the exploration-exploitation dilemma has not been explicitly addressed in these works, or where it has, only heuristics have been considered as solutions (Lang et al., 2010).

1.2 Bridging the Gap

This thesis's main objective is to bridge this RL/RRL disconnect by bringing traditional RL-style sample complexity analysis to sequential decision making domains where relational representations are used. The algorithms and constructive positive results in this document allow agents to learn generalized relational descriptions of otherwise large and unwieldy environments (like blocks world) and still perform efficient (polynomial in the parameters of the individual domain) exploration, limiting the number of steps needed to reach optimal behavior in a manner consistent with earlier efforts with propositional representations.

We also go beyond this traditional RL setting in later chapters to consider the sample complexity of learning action schemas in a setting (apprenticeship learning) where an agent has access not only to the environment, but also to a teacher that can show it examples of policies being enacted in the environment. We further consider apprenticeship learning in a real-world application of learning web-service task descriptions from traces of users performing the task (still a sequential decision making problem, but very different from traditional RL). In both cases we again show how to bound the amount of experience (in this case interactions with the teacher), needed to learn the relational models. This progression is expounded upon in the roadmap below, but in a rare attempt at brevity, we can describe the results of this thesis in one sentence as:

1.2.1 Thesis Statement

Compact relational models of actions, including action schemas and task-specific web service descriptions, can be efficiently learned in the online-learning and apprenticeship-learning paradigms, and used by agents that efficiently explore in corresponding sequential decision making domains.

1.3 A Roadmap for this Document

This section provides an outline of each chapter in this thesis, describing the different learning paradigms and sequential decision making problems considered in each chapter. We also summarize some of the major results of each chapter at a high level.

The thesis progresses roughly from the highly theoretical results to more applied work. This transformation can be seen clearly in the types of example and experiments conducted in these chapters. At first, a number of "toy domains" are used to showcase and empirically validate the algorithms introduced in each section. This is especially true in Chapters 3 and 4 where domains like Blocks World and a variant of Logistics World (Minton, 1988) are used in most of the experiments. This is partly done to showcase the algorithms in easy-to-understand domains, but also because we are working in a somewhat uninformative learning paradigm (Chapter 3) and using planners that are dependent on the (very large) size of the state space (both chapters, but relaxed in Chapter 6). In contrast, Chapter 5 is largely application driven, applying the learning paradigm from Chapter 4 and a more realistic relational representation to tackle a real-world web-service modeling challenge. Many of the examples in that chapter are built from real data and real-world third-party web services. While Chapter 6 mostly reverts back to the "toy domains" of the earlier chapters, the extensions utilized in this chapter are designed to push these earlier results towards use in real domains. These include the use of more complex action languages and better planners for tackling large state spaces efficiently. We now present more detailed summaries of each of the chapters.

1.3.1 Background Material

Chapter 2 introduces much of the background theory necessary for understanding the theoretical results in this work. Topics covered include basic reinforcement learning terminology and theory, basic model-based reinforcement learning algorithms like $R_{\max}$, and planning algorithms like Value Iteration. We also describe the exploration-exploitation dilemma in reinforcement learning and provide an extended treatment of the KWIK ("Knows What It Knows") framework (Li et al., 2008) for measuring the sample complexity of learning models in RL. While some of the related work in this thesis is covered on a chapter by chapter basis, Section 2.4 presents a detailed description of the historical and state-of-the-art methods from the field of relational reinforcement learning.

1.3.2 Online Action Schema Learning

Chapter 3 describes a plethora of efficient reinforcement learning algorithms for different sub-problems and types of dynamics for domains described with relational descriptions. Unlike later chapters, the analysis in this chapter considers an agent's experience to be confined to the traditional reinforcement learning channel; an online agent taking steps in the world and observing the changing state and rewards based on these actions. Because of this limited form of experience, there are several negative results in this chapter, but, the findings are the most comparable to traditional reinforcement learning results. That is, "sequential decision making" in this chapter refers exactly to the traditional RL notion of choosing an action to maximize possible long-term reward and solving the exploration-exploitation dilemma efficiently. This latter quality is evaluated by whether or not the algorithms meet the KWIK criteria as introduced in Chapter 2.

This chapter introduces terminology and definitions for a class of languages we call *relational action schemas*. This class covers a wide array of languages that describe domain dynamics with relational state descriptions, including traditional STRIPS and more advanced stochastic descriptions. An example action schema for the action *move(X, From, To)* in Blocks World appears in Figure 1.2. While we will not get into the technical details of the representation here, it is important to note that in general, action schemas are composed of (1) A parameterized action (*move*(X, From, To)), (2) some pre-conditions or conditions based on a relational description of the current state, (3) a set of effects (perhaps conditional effects) that describe how the action will change the state (the Add and Delete lists here), and (4) a probability distributions over each effect set determining the frequency of each effect when the action is invoked (in the example schema, the block is moved with probability 0.9 but nothing happens with probability 0.1).

For this class of languages, we investigate several sub-problems that involve learning only portions of the schemas while given others (for instance given the effects, how hard is it to learn the pre-conditions and the effect distributions? What about just the distributions?). Each of these sub-problems is in turn considered with different restrictions on the action schemas themselves (for instance if they are deterministic or stochastic, or if they have pre-conditions or conditional effects, and other restrictions like the size and type of the conditions). Each of these different learning-problem/domain-constraint pairings elicit a different architecture and set of component learning algorithms in the online setting, leading to a number of theoretical results. Some of the main theoretical results of Chapter 3 include:

- Learning the probability distributions over effects is a non-trivial problem due to "effect ambiguity" because multiple effects might look similar, making it impossible just to "count" the frequency of each one. But a KWIK linear regression algorithm provides an efficient solution to this learning problem.

- Large conjunctions are not efficiently learnable as pre-conditions or conditions governing effects. But tractable algorithms are possible for conjunctions with only $O(1)$ terms.

- Deterministic STRIPS effects are easily learnable, and more complex algorithms can be used to learn effects in a number of stochastic STRIPS settings, including one where the number of effects is known to be bound by a constant.

- A number of KWIK architectural solutions that combine component KWIK learners (like the linear regression or small condition learners above) can be used to solve various combinations of the sub-problems (like learning just the conditions and effect distributions when given the effects), and, with some assumptions, even the full action schemas are efficiently learnable using these KWIK architectures.

1.3.3 Apprenticeship Action Schema Learning

While Chapter 3 performed analysis of learning relational action schemas in the traditional (online) reinforcement learning setting, the class of domains that is learnable in this setting is limited by the types of models that are KWIK-learnable. Unfortunately, many classes of functions that are common with relational representations, such as conjunctions of more than a few terms, are not KWIK-learnable for online RL agents. To circumvent this deficiency, Chapter 4 considers learning such models in an *apprenticeship learning* protocol where an agent can learn from both its own experience *and* can receive *traces* (trajectories of ⟨state, action, reward⟩ tuples) of a teacher performing actions in the domain.

This new channel of experience slightly changes the type of sequential decision making problems being considered and the nature of the exploration/exploitation trade-off. Instead of trying to limit the number of suboptimal steps made by the agent during exploration, we instead need to limit the number of times the teacher will step in to show the agent how to perform a task. We do so by limiting the number of episodes where an agent executes a policy that is outperformed by the teacher in the same domain instance. Hence, the sequential decision making problem is still one of an agent choosing actions, but instead of trying to maximize long-term reward in the overall environment, it is just trying to take actions that do no worse than its teacher.

While this dependence on the teacher's policy may seem to be a drawback at first, one should note that if the teacher is optimal, the agent will also learn to be optimal, so in that case nothing would be lost versus the more limited traditional RL agents. Also, by essentially limiting the agent's online exploration, practical problems where an agent would not want to explore a vast expanse of territory, or even worse fall off a cliff, are approachable thanks to this guarantee. Moreover, we show that there is a much larger class of models that is learnable under these conditions than in the traditional RL setting. These theoretical improvements allow us to relax the restriction on the size of pre-conditions and conditions from Chapter 3 and have implications of interest to the larger RL community. The major results of this chapter are:

- We introduce a new learning framework called *Mistake Bound Predictor* which covers the classes of KWIK-learnable functions and Mistake Bound (Littlestone, 1988) learnable functions and many combinations of these classes.

- We present a general theoretical result connecting learnability in this model class to efficient behavior under the "do no worse than the teacher" criteria for the apprenticeship learning setting.

- Using these results, we can relax the size restrictions on learning *pre-conditions* in all the learning problems from Chapter 3, allowing the conjunction to reference all the domain literals.

- In the case of conditional effects, the size restriction can be similarly relaxed in the deterministic case where the effects are given by using a clever k-term DNF learning algorithm.

1.3.4 Apprenticeship Learning of Web Service Descriptions

While the previous two chapters dealt mostly with theoretical issues and toy problems, Chapter 5 is about a specific application of relational models in the apprenticeship setting. Specifically, we consider the problem of learning *web-service task descriptions* from traces of users performing a task. These tasks run the gamut from mundane to complex, including tasks for looking up and booking flights, filling in reimbursement forms for travel costs, and even buying birthday gifts for colleagues. Each task involves invoking a sequence of web services, which can be viewed as parameterized actions, similar in form to the action schemas used in the theoretical portion of this work.

But unlike those earlier examples, the sequential decision making problem in this setting is not so much choosing actions (the sequence of web services that need to be invoked is known to the agent), but rather choosing the correct parameters to those actions and making predictions about their outcomes. This is different than the traditional reinforcement learning setting, especially because we perform the analysis under the apprenticeship learning protocol. But what makes it relevant to this thesis is that (1) relational models are necessary here to represent the links between elements of the task (for instance realizing that the "departure city" for a flight booking should be the flyer's "home city"), and (2) there is a sequential component to this problem because we often need to choose inputs to invoke a service in order to gain information or because of requirements of a service used later in the task (for instance looking up flight information to make sure it fits a user's constraints before booking it).

While this chapter is focussed on practical matters, there are still some theoretical results reported based on the number of traces needed (similar to the criteria from chapter 4) to learn several classes of tasks. But a large portion of the chapter is also devoted to the problem of mining the relational models from real data, because in this case the "traces" provided are only assumed to be XML documents reporting the inputs and outputs of each service invocation (a sort of structured log of users performing each task). The main results of this chapter are as follows.

- We introduce a new relational representation for web-service tasks called a *task graph* which stores structural and semantic links between the inputs and outputs of services as well as between elements associated with different services.

- We link the learning of these representations to the apprenticeship learning results from the previous chapter by showing, under certain conditions, these task graphs are mistake-bound learnable from traces of users completing the tasks.

- We introduce several complex relations for selection of elements and rules for handling missing objects and lists, which need to be modeled in order to understand the tasks. We show that our task-graph representation and efficient learning algorithms can be adapted for use with these new challenges.

- We present empirical results demonstrating our system learning tasks from real world

	Sample Efficient Learning	Sequential Decision Making	Relational Representations
Sample Efficient Learning		PAC-MDP RL	PAC ILP
Sequential Decision Making	PAC-MDP RL		Relational RL
Relational Representations	PAC ILP	Relational RL	

Table 1.1: Pairwise combinations of the three main features of this thesis that have been considered in the literature. This thesis captures all three.

traces of XML documents showing clients invoking the services in the task. We show experiments where the tasks are composed of services from single providers and also multiple service providers (both Google and Amazon services used in different parts of a task). Our system performs remarkably well in these situations, often learning the full tasks with only a few traces.

1.3.5 Extensions

Chapter 6 explores several extensions of the theoretical (and more mainstream-RL) work presented in Chapters 3 and 4. First, where those chapters dealt almost exclusively with examples from the Stochastic STRIPS language, we show that these results can be expanded to another relational language from the literature (Object Oriented MDPs) that is covered by the relational actions schema family. We then investigate a class of action languages that go beyond the simple relational representations studied in the bulk of this work to allow for constructors from a powerful family of logical languages called Description Logics (Baader et al., 2003). This portion is meant to bridge the gap between simple relational languages like STRIPS and the more expressive Situation Calculus formalisms that allows full first order logic descriptions of states. We present results that link the efficiency of such algorithms to well-studied operations from the DL literature.

Finally, we present results concerning the use of sample-based planners for the large state spaces considered in this work. This investigation is meant to speed up the computation time of our algorithms, which hitherto have relied on planners that scale at least linearly with the size of the state space (which is usually exponential in the number of objects in relational domains). We show in this section that certain sample-based planners, which scale independently of the state space size, can be integrated into our learning framework without sacrificing our sample efficiency guarantees, thus expanding the practical reach of our algorithms. However, other than in this section, we generally treat the computational burden of planning as an orthogonal issue, instead concentrating on the sample efficiency of the learning algorithms.

1.4 Common Threads

Above, we laid out a roadmap for the rest of this work, and while we cover a wide variety of problems, we note here that there are several common threads running through the chapters. Specifically, all of the algorithms perform *sample efficient learning* of *relational representations* for *sequential decision making* problems. Combinations of these factors have occurred in the literature before (see Table 1.1), including work on efficiently learning policies in relational worlds (Khardon, 1999b,a). However, these three properties have not been extensively studied together as we attempt here. This is unfortunate because fast learning, generalized representations, and chaining actions together, are fundamental problems in AI and in our understanding of human decision making. While each factor can be studied in a vacuum, we contend that all three parts must be considered together to paint a true behavioral portrait. So without further ado...

Chapter 2

Background and Related Work

In this chapter, we cover a number of important frameworks, algorithms, and results that underpin the theoretical and empirical studies carried out in later chapters. We begin by describing some basic algorithms for reinforcement learning and frameworks for measuring their sample complexity. Later in the chapter, we recount prior work on modeling action dynamics with relational languages and then describe a branch of the lierature (RRL) that connects this work back to reinforcement learning.

2.1 Reinforcement Learning

In this section we describe the general reinforcement learning (Sutton & Barto, 1998; Kaelbling et al., 1996) (RL) framework, algorithms, representations, and some important theoretical results. We begin with a representation-agnostic description of the terminology and protocol that underlies RL. Following that, we describe basic model-based reinforcement learning, which is the class of algorithms considered throughout this work. We also describe planning algorithms that are used in model-based RL, with specific attention on Value Iteration. After this, we briefly discuss a "compact" propositional RL representation (factored MDPs) that uses generalization over the state space to make learning and planning more tractable in large, but structured domains. Finally, we discuss some issues and algorithms related to the exploration / exploitation trade-off in RL, a topic we cover in far greater detail in the next section.

2.1.1 General Reinforcement Learning and Markov Decision Processes

At the most basic level, a reinforcement learning agent interacts with its environment through the protocol illustrated in Figure 2.1. At timestep t, the environment has some state s_t and reports some observation o_t to the reinforcement learning agent as well as a special observation signal called the reward r_t, which may also be based on the agent's previous action. The agent

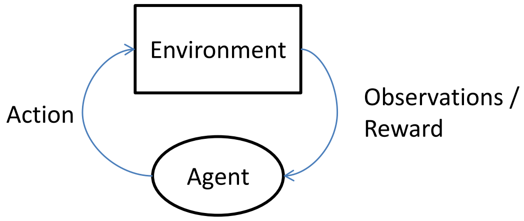


Figure 2.1: The online RL protocol.

then is able to perform an action a_t in the environment, which along with the environment's current state, will determine (perhaps stochastically) the environment's next state s_{t+1}, which in turn produces a new observation and the cycle repeats. This back-and-forth interaction continues over the agent's lifetime, which may be infinite, have some finite horizon T, or be episodic (see below). In this work, we consider this interaction as taking place over discrete timesteps with no latency in the signals, though continuous time RL has been studied in the literature (Puterman, 1994) and methods for dealing with latency in these signals have also been addressed (Walsh et al., 2009a). The agent's general goal during this interaction is not necessarily to reach a specific state, but instead to maximize some function of the reward sequence $r_0 ... r_t$ (see Section 2.1.2 below) collected over its lifetime.

Delving deeper into the "environment box", RL environments can be described using a general representation called a *Markov Decision Process* (MDP) (Puterman, 1994), defined as follows:

Definition 1. *A general* ***Markov Decision Process*** *(MDP) is described by a 7-tuple* $\langle S, A, R, T, \gamma, S_0, S_T \rangle$ *where* S *is a set of states the environment can be in,* A *is a set of actions an agent can take,* $R : S, A \mapsto Pr[[R_{min}, R_{max}] \in \Re]$ *is a reward function determining* r_t*,* $T : S, A \mapsto Pr[S]$ *is a transition function determining state* s_{t+1} *and* $\gamma \in [0, 1)$ *is a discount factor, which measures the degradation of future rewards.* $S_0 = Pr[S]$ *is a probability distribution over start states (determining* s_0*) and* S_T *is a set of* terminal *states for episodic environments as defined below.*

As an example of the basic MDP structures, in the simple grid-world in Figure 1.1, the set of possible states is simply all 9 cells that the agent can occupy, the set of possible actions are $A = \{Up, Down, Left, Right\}$, each of which have deterministic effects. Reward of +10 is received if the agent enters the upper-right square (which is also a terminal state), while $r_t = -1$ for all other state/action pairs.

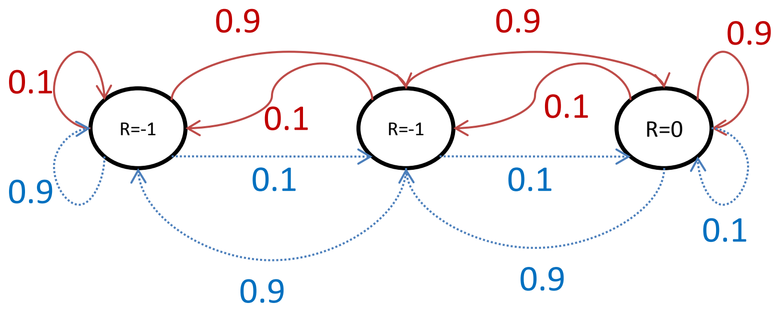


Figure 2.2: A simple Markov Decision process. The red (solid) action usually moves the agent to the right but with probability 0.1 moves it left. The blue (dashed) action does the opposite.

Throughout this work, we assume both T and R are stationary (time invariant), though we note that a portion of the literature has been devoted to non-stationary MDPs where these functions can shift (perhaps adversarially) over time (see Lane et al. (2007)). While the example above was deterministic, general MDPs can have non-determinism in both the transitions and the rewards. An example of a basic non-deterministic MDP is depicted in Figure 2.2. There are two actions in this MDP, a red (right) one and a blue (left) one, and each has a .9 probability of achieving its intended outcome, and with probability 0.1 the unintended outcome occurs.

In a flat MDP, the states S have no room for generalization—that is states s_i and s_j in S where $i \neq j$ have no inherent commonality and all that can be said about them is that they are potentially different in terms of T and R. We call such representations *flat* MDPs and when employing such a model, a reasonable representation for T and R is in a *tabular* form. That is, T and R can be stored simply as tables with unique indexes for every $\langle s, a \rangle$ pair. But such tables are potentially enormous in real-world domains . In this thesis, we will be investigating MDPs that contain far more structure, so that even with exponentially (in the domain factors) large $|S|$, *compact* models of T and R can be maintained without resorting to the tabular format.

When interacting with an MDP, an agent can be considered to be following a policy $\pi : S \mapsto Pr[A]$ with regard to its action selection. A policy is said to be *stationary* if the policy that the agent uses at any time t is the same as the policy it uses at $t + 1$. Furthermore, a policy is said to be *deterministic* if it maps each state to a single $a \in A$. The quality of a policy is determined by a function of the rewards it garners over the agent's lifetime, a subject we return to in Section 2.1.2.

Observability

Thus far, we have not discussed the observation function $\chi : S \mapsto Pr[Obs]$, which (perhaps stochastically) generates an observation for the agent from the world state. In general, if χ takes on any stationary distribution where the observability property below does not hold, the model becomes a Partially Observable Markov Decision Process (POMDP), where planning and learning can become intractable (Littman, 1996) without extra structure. In this work, we assume that the MDP is fully observable, meaning the following:

Definition 2. *A Markov Decision Process is* ***fully observable*** *if the set of observations Obs is a bijection onto the set of states S. That is, every state produces a unique observation.*

We will however, briefly return to this partially observable setting when discussing related work on relational RL models that are learned under these provably harsher conditions.

Episodic MDPs

A number of results in this thesis are for *episodic* MDPs where an agent interacts with the environment until reaching a *terminal state*, one of the $s_i \in S_T$, at which time the agent is reset into a new initial state as dictated by the initial state distribution S_0. We also often employ an empirical "hard cap" on the number of steps an agent can take in an environment, which we use simply to limit the length of experiments, though we return to the problem of "how long to let a bad agent run" when we study the apprenticeship learning setting in Chapter 4.

Factored MDPs

Above, we have described MDPs in their most general form, the so called flat MDP, which has a completely unstructured state space. That is, there is no correlation or similarity between the effects of an action (or the rewards) when applied in states s_i and s_j when $i \neq j$. But this representation does not conform to the way we normally view the world. For instance, if an agent finds that its alarm sensor triggers easily when it is out in the rain, the sensor will activate if its in the rain at night or during the day. The states are wildly different in terms of visibility, temperature, and other factors, but their commonality on a single *factor* gives us a way to compactly describe its transitions for the sensor observation, say: "With probability 0.8 the sensor activates when it is raining and it wasn't on before". Below, we review a compact representation for propositional MDPs (the classical factored MDP) that formalizes such intuitions. This thesis, however, deals with an even further generalized and compacted representation, relational MDPs, as introduced in Section 2.4.

A factored-state Markov decision process (fMDP) is an MDP, where a state s is the Cartesian product of m smaller propositional literals, or *factors* $\mathbf{S} = l_1 \times l_2 \times \ldots \times l_m$. A standard assumption (Kearns & Koller, 1999) in factored MDPs is that for each i there exist sets Γ_i of size $k = O(1)$ such that $s_{t+1}[i]$ depends only on $s_t[\Gamma_i]$ and a_t [1]. That is, each factor is dependent only on a small set of factor values from the previous timestep (whether the sensor was on or not and whether it was raining) and the last action. The transition probabilities are then $T(\vec{y} \mid s, a) = \prod_{i=1}^{m} T_i(l'[i] \mid s[\Gamma_i], a)$ for each $l, l' \in S$, $a \in A$. Under these conditions, the transitions can be compactly described using a *Dynamic Bayesian Network* (DBN) (Dean & Kanazawa, 1989) as shown in the bottom-left representation in Figure 1.1. An analogous assumption can be made for the reward function. Algorithms exist for planning in DBNs with known parameters (for example, Guestrin et al. (2003b); Hoey et al. (1999)), for learning transition probabilities (Kearns & Koller, 1999) and dependency structure (Diuk et al., 2009), and for learning their reward functions (Walsh et al., 2009b).

Intuitively, DBNs capture the structure of transition functions where states are described in terms of attribute-values for a set of features that is $log(S)$. This makes them a very powerful tool in RL because only this compact representation needs to be learned to know T (not the full exponentially sized tabular form). However, in domains that contain objects DBNs need to represent the changes to every object's attributes with different factors. So if instead of having one robot with a rain sensor, we had n such robots, the DBN representing each of their sensor values would contain n unlinked copies of the same structure. This is intuitively unappealing because we can describe this far more compactly with a "rule" about a generic robot's sensor dynamics as applied to all n robots. The relational MDPs used in this thesis will do just that.

2.1.2 The Value Function

A reinforcement learning agent's goal is to maximize some function of the sequence of rewards it sees while interacting with an environment. While a number of different functions have been studied in the literature, including average reward (Mahadevan, 1996), in this thesis we will consider maximization of *expected discounted reward* as calculated by the Value Function

[1] This can be relaxed to also allow members of Γ_i in the s_{t+1} to capture correlated effects.

(Puterman, 1994) induced by an MDP and a policy π.

$$V^{\pi}(s) = \sum_{a \in A} Pr(a|\pi)(R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V^{\pi}(s')) \tag{2.1}$$

This equation itself represents a telescoping sum of the expected future rewards starting from state s and following policy π thereafter. Specifically, for a sequence of states s_0, s_1... we have

$$V^{\pi}(s_0) = \sum_{t=0}^{\infty} \gamma^t r_t \tag{2.2}$$

It is known (Puterman, 1994) that using such a value function, there exists at least one *optimal policy* π^* such that $\forall s \in S$, $V^{\pi *}(s) = V^*(s) \geq V^{\pi}(s)$ and that there always exists one such policy that is deterministic. We can write this equation (known as the Bellman Equation (Bellman, 1957)) for V^{π^*} as

$$V^*(s) = R(s, \pi^*(s)) + \gamma \sum_{s' \in S} T(s, \pi^*(s), s')V^*(s'))$$

$$= \max_a R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V^*(s')) \tag{2.3}$$

We will often find it useful when discussing reinforcement learning algorithms that have to compare the values of different actions to consider a state-action version of the value function $Q^{\pi}(s,a)$ which is defined as:

$$Q^{\pi}(s,a) = R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V^{\pi}(s')) \tag{2.4}$$

Intuitively, $Q^{\pi}(s,a)$ simply represents the sum of expected discounted rewards for an agent starting at state s, taking action a, and afterwards following policy π. Analogously, the Q-values for actions followed by the optimal policy can be written as:

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V^*(s')) \tag{2.5}$$

Notice also that $V^*(s) = max_a Q^*(s,a)$ because the optimal policy can always be represented deterministically. Since the ultimate goal of a reinforcement learning agent is to select actions to maximize this quantity, it is important that we have algorithms to calculate these values given an MDP. The next section discusses such planning algorithms.

2.1.3 MDP Planning Algorithms

This section considers the general MDP planning problem defined as the following:

Definition 3. *The **MDP Planning Problem** with accuracy ϵ is, given an MDP, find a policy π that is ϵ-optimal, that is, $\sum_a \pi(s,a)Q^\pi(s,a) \geq V^*(s) - \epsilon$.*

Notice that this planning problem is different from the full reinforcement learning problem because in the planning case, T and R are already known, and don't need to be learned. Thus, planning is simply a computational process, no interaction is needed with the environment.

In this section, since we are only considering general MDPs without any other structure, we focus only on representation-agnostic planners which work by finding the fixed point of the set of linear equations represented by the optimal value function for all states. Perhaps the most intuitive method for finding such a fixed point is a Dynamic Programming solution to the MDP planning problem called *Value Iteration* (Puterman, 1994) as described in Algorithm 1.

Algorithm 1 Value Iteration

```
$\forall s, V(s) = 0$
$\delta = \infty$
while $\delta > \epsilon$ do
    $\delta = 0$
    for $s \in S$ do
        $oldV = V(s)$
        $V(s) = \max_a R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V(s')$
        $\pi(s) = \text{argmax}_a R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V(s')$
        if $||oldV - V(s)|| > \delta$ then
            $\delta = ||oldV - V(s)||$
        end if
    end for
end while
Return $\pi$
```

Intuitively, Value Iteration simply iterates through all of the states and updates their value functions based on the Bellman Equation for the policy currently considered optimal and the current values of the other states. This approach takes polynomial (in S and A) time per iteration, and the number of iterations before convergence is known to be pseudo-polynomial in the accuracy parameter ϵ and the discount factor γ (for a more complete analysis see Madani (2002)).

Other methods exist for planning in general MDPs, including linear programming (Bertsimas & Tsitsiklis, 1997) (which provides strong polynomial time guarantees) and Policy Iteration (Puterman, 1994). For most of the algorithms in this thesis that rely on a "general MDP planner" all of these methods can be thought of as interchangeable, as they are each essentially representation agnostic (everything is treated as a flat MDP). With that being said, several variants of Value Iteration are presented in this thesis that exploit MDPs with specific structure, so in those cases analogues to the structural modifications would have to be developed for each algorithm.

2.1.4 Model-based Reinforcement Learning Algorithms

Having covered the general MDP model and planning algorithms above, we can now return to the reinforcement learning problem. Here we describe the model-based reinforcement learning setting, the framework used in most of this document. We will briefly discuss model-free methods in a later section.

In model-based reinforcement learning, the agent maintains some model M_A of the environment's dynamics (T and R) as learned so far and uses this internal model to decide how to act next. This model is not necessarily (and in this thesis is almost never) a flat MDP or even the same model class as the domain's true encoding (if it has one). Model-based reinforcement learning algorithms alternate between two phases on each timestep (see Algorithm 2). In the first phase, the algorithm selects an action using a planner P (such as Value Iteration) based on some interpretation of its internal model ($M_A^{\mathcal{I}}$). These interpretations fill in the unknown (by some threshold described below) parts of the model so that it is usable to the planner. The exact scheme used in instantiating this interpretation has a critical effect on the behavior of the algorithm and is discussed at length in the next section and beyond.

In the second phase of model-based reinforcement learning, the agent receives feedback from the environment based on the action it took (a new observation and reward). This feedback is then used to update M_A and the process repeats on the next timestep. While this architecture is certainly elegant, a naïve implementation can falter, because the model interpretation involves filling in parts of M_A where there is little information to construct a perfect model. We now describe how strategies for filling in these gaps affects an agent's ability to explore and exploit its domain.

Algorithm 2 General Model-based RL

```
Agent knows $S$ (in some form), $A$, $\gamma$, $\epsilon$, $\delta$
Agent has access to planner $P$ guaranteeing $\epsilon$ accuracy
Agent has access to the environment $E$ which is representable as MDP $M$
initialize agent model $M_A$
$s_0$ = start state
**while** Episode not done **do**
    $a_t = P$.getAction($M_A^{\mathcal{I}}$)
    $\langle s_{t+1}, r_t \rangle = E$.Execute($a_t$)
    $M_A$.update($s_t, a_t, r_t, s_{t+1}$).
**end while**
```

2.1.5 Exploration / Exploitation in Reinforcement Learning

Intuitively, the exploration/exploitation dilemma describes a situation where an agent has a partially learned model with different degrees of certainty for different state-action pairs. In such a situation, the agent must decide whether it is best to *explore* the environment, that is travel to areas where it has uncertainty and learn more about the dynamics and rewards in that area, or *exploit* the areas where it has great certainty by doing the best actions it can with respect to the known model.

The most simple example of this dilemma is embodied in the k-armed bandit problem (Fong, 1995), which is a degenerate reinforcement learning environment with only a single state and k actions. Each of the actions can be thought of as a slot machine with a payoff distribution over $[R_{min}, R_{max}]$, but this distribution is unknown for each arm. By the pigeon-hole principle, at least one of these arms must be optimal, that is it has the highest expectation on reward. On every trial, since the distributions are stationary, the agent should pull this optimal arm. That is exploitive behavior, but in order to find that optimal arm, it needs to try the arms systematically, which is exploration. One common measure of the *sample complexity* of this problem (described more formally below) is the number of times the agent pulls a suboptimal arm.

The solution to this trade-off is non-trivial as behavior at either extreme can lead to disaster. If the agent pulls a suboptimal arm on the first try ($t = 0$), and receives reward $r_0 > R_{min}$, and always exploits with no further exploration, it will continue to pull this sub-optimal arm in perpetuity, resulting in an infinite sample complexity. Likewise, if the agent is constantly exploring, essentially ignoring all the empirical evidence it has gathered and just pulling arms at random, in the limit its sample complexity is also infinite. In the k-armed bandit problem, an optimal solution to this dilemma with respect to the PAC framework (described in Section 2.2) was derived by Fong (1995). This algorithm simply maintains upper confidence bounds on the reward function for each arm by storing the sample mean and calculating confidence bounds using Hoeffding's Inequality (Hoeffding, 1963). The agent then pulls, on each trial, the arm with the highest upper bound. This solution has a provably polynomial PAC sample complexity on the number of suboptimal arm pulls with probability $(1 - \delta)$.

With this simple, but intuitive, example in mind, we now formally define a notion of *sample complexity* for a reinforcement learning agent. Such definitions have taken on many forms in the literature (Kakade, 2003; Auer et al., 2007), but all of these definitions subscribe to some basic tenants.

- They have some failure probability δ, which gives the agent slack when learning in a stochastic (and adversarial) environment.
- They only require ϵ optimality, and demand polynomial dependence on ϵ.
- They bound either the number of steps where the agent acts using a policy π that has value less than $V^* - \epsilon$, or the expected loss in value associated with following such a policy.

In the first half of this thesis, we will use the definition of PAC-MDP (Strehl et al., 2009) and the associated KWIK model-learning framework (Li et al., 2008) to analyze the complexity of model-learning and reinforcement learning in a more general way. A topic for future study involves the adaptation of the results of this thesis to related sample complexity measures (such as regret (Auer et al., 2007)) mentioned above. Formally, a PAC-MDP bound on an agent's behavior is defined as follows.

Definition 4. *An algorithm $\mathcal{A}$ is considered **PAC-MDP** if for any MDP M, $\epsilon > 0$, $0 < \delta < 1$, $0 < \gamma < 1$, the sample complexity of $\mathcal{A}$, that is the number of steps t such that $V^{\mathcal{A}_t}(s_t) < V^*(s_t) - \epsilon$ is bounded by some function that is polynomial in the relevant quantities ($\frac{1}{\epsilon}$, $\frac{1}{\delta}$, $\frac{1}{(1-\gamma)}$ and $|M|$, where $|M|$ is a measure of the model complexity) with probability at least $1 - \delta$.*

In an attempt to meet this criteria, we now consider two *exploration strategies* and their analogous model interpretations for Algorithm 2. The first set of strategies is not sufficient for PAC-MDP behavior but is mentioned briefly because of its use in the literature. The second strategy is considered here only for the case where the flat MDP has $S = Poly(|M|)$, but we will revisit this strategy for more general (compact) MDPs in Section 2.3.

Greedy Interpretations

Greedy interpretations such as ϵ-greedy and Boltzman exploration (Sutton & Barto, 1998) make very strict interpretations of their learned models. For instance, the ϵ-greedy strategy is to use the maximum likelihood model given the current data for M_A, find its value function, and act greedily with respect to this model except for a certain proportion of the timesteps (ϵ) where the agent simply chooses a random action. These exploration techniques are often combined with *model-free* RL approaches like Q-Learning (Watkins, 1989), which simply does the following backup on every observed transition $\langle s, a, s' \rangle$:

$$Q(s,a) = Q(s,a) + \alpha(r(s,a) + \gamma \max_{a}' Q(s', a') - Q(s,a))$$

It has been shown (Whitehead, 1991) that in simple MDPs where a high reward is only reached by following a sequence of actions to a specific state, this dependence on randomness to reach unexplored regions can lead to an exponential number of suboptimal steps, and hence this exploration strategy is not PAC-MDP. Since greedy methods can lead to such exponential blowups in general (flat) MDPs, these results are easily reproducible when using relational MDPs. Thus, for the bulk of this thesis we instead focus on an optimistic exploration strategy, which we now describe for polynomial sized flat MDPs.

R-max for flat MDPs

Instead of the strict interpretation of the learned model considered by greedy exploration strategies, the R-max Algorithm (Brafman & Tennenholtz, 2002), as seen in Algorithm 3, takes an "optimism in the face of uncertainty" approach. The algorithm is a specific instantiation of the model-based RL template in Algorithm 2 and works by explicitly constructing an optimistic model based on a "known-ness" parameter m. Intuitively, state-action pairs that have been tried more than m times are considered to be *known* and their maximum-likelihood estimates can be used for T and R. The other states are considered "unknown" and their transitions are interpreted as taking the agent to an "R-max state" where maximum reward is gained in perpetuity. This potential over-valuation of the under-explored regions will lead the agent to act in a way so as to reach these areas, unless it can be fairly certain that better value can be obtained within the currently “known” MDP.

Algorithm 3 R-max (Brafman & Tennenholtz, 2002)

```
Agent knows $S$ (in some compact form), $A$, $\gamma$, $\epsilon$, $\delta$
Agent has access to planner $P$ guaranteeing $\epsilon$ accuracy
Calculate $m = Poly(S, A, \frac{1}{1-\gamma}, \frac{1}{\epsilon}, \frac{1}{\delta})$
$\forall s, a : count(s, a) = 0$
for Each step do
  Construct $M_A = \langle S \cup s_{max}, A, \widehat{R}, \widehat{T}, \gamma$ where $s_{max}$ is a trap state with reward $R_{\max}$ and is reached by any $< s, a >$ where $count(s, a) < m$
  Solve $V^*_{M_A}(s_t)$ and choose greedy action $a_t$
  $\langle s_{t+1}, r_t \rangle = E.nextState(s_t, a_t)$
  if $count(s_t, a_t) < m$ then
    Update $\widehat{T}$ and $\widehat{R}$ based on $\langle s_t, a_t, r_t, s_{t+1} \rangle$
  end if
  $count(s_t, a_t) = count(s_t, a_t) + 1$
end for
```

The analysis of the PAC-MDP sample complexity of R-max (Brafman & Tennenholtz, 2002) relies on showing that with $m = Poly(S, A, \frac{1}{\epsilon}, \frac{1}{\delta}, \frac{1}{1-\gamma})$, with probability $(1 - \delta)$ the flat MDP model reaches ϵ-accuracy with at most m suboptimal steps. The sample complexity of learning the model is then combined with known results about the value of a policy in two slightly different flat MDPs (Singh & Yee, 1994) to achieve a similar PAC-MDP bound.

Viewed this way, the exploration/exploitation dilemma in model-based reinforcement learning hinges on the efficient solution to a supervised model-learning problem where uncertainty needs to be quantified ($m - k$ more samples until this pair is known) or at the very least qualified (“I don’t know what happens in this state, but I know what this other one is like”). For flat MDPs, R-max does just that, and the analysis is fairly intuitive (each sample increases our certainty on the model and the agent is encouraged to explore unknown parts). But for compact representations like factored MDPs or the relational MDPs considered in this thesis, S is exponential in $|M|$. In such representations, the dependence of the algorithm on $|S|$ does not guarantee PAC-MDP behavior, and in fact guarantees that in most cases we won’t have efficient exploration, as the agent will continue to explore different state configurations that it ought to already know about (for example, the robot in the rain trying every street corner in the city to determine how its rain sensor acts there). In order to get a proper feel for the possible sample complexity results in a wider class of models, we now delve into the literature on traditional (supervised) computational learning theory.

2.2 Sample Complexity in Supervised Learning

Model learning can be thought of as a supervised-learning problem where an agent's interaction with the environment consists of an infinite sequence of predictions on inputs provided from the environment. Formally, the realizable supervised learning paradigm has the following general form:

- There is a (potentially infinite) set X of possible inputs and a (again possibly infinite) set Y of possible labels.
- There are accuracy and certainty parameters ϵ and δ.
- There is a stationary hypothesis h^* chosen from a hypothesis class H such that on every timestep t, $E[y_t] = h^*(x_t)$

There are now a large number of frameworks for measuring sample complexity in supervised learning (Valiant, 1984; Littlestone, 1988; Auer & Cesa-Bianchi, 1998; Helmbold et al., 2000; Li et al., 2008). Each of these places different assumptions on the way that each x_t is drawn from X (i.i.d or adversarially), what kind of predictions the learner has to make (for example, can the learner decline to make a prediction?), and what sort of interaction counts as a sample. One early, and widely used, frameworks for measuring sample complexity was PAC (Probably Approximately Correct) (Valiant, 1984), under which inputs x_t are drawn i.i.d. from an unknown (but stationary) distribution over X, and the sample complexity is the number of timesteps before, with probability $(1 - \delta)$, the learner can guarantee it is making ϵ accurate predictions ($||\widehat{y}_t - h^*(x_t)|| \leq \epsilon$). However, these i.i.d. assumptions on inputs are rarely encountered in the model-learning component of reinforcement-learning agents because their shifting policies during exploration do not produce i.i.d. samples. For instance in a factored MDP, disregarding certain factor combinations as unlikely based on the distribution of all states encountered so far may be incorrect, because further exploration may show these combinations to be part of a state that the optimal policy will visit many times. Below, we consider two different frameworks (MB and KWIK) that do not rely on i.i.d. assumptions, and instead assume that each x_t is drawn *adversarially* from X. These two frameworks will be instrumental in our study of sample complexity throughout this thesis.

2.2.1 Mistake Bound Learning

The Mistake Bound (MB) framework (Littlestone, 1988) is designed only for deterministic hypothesis classes where the true label is always provided by the environment. While this is clearly insufficient for learning stochastic MDP models, we will make use of this framework in the apprenticeship setting studied in Chapter 4, so we introduce it here and cover some of its properties and learnable classes as well as its insufficiency for autonomous model-based RL.

Formally, the mistake-bound protocol is described in Algorithm 4. For every input, the agent predicts a label, and if its prediction is wrong, it gets to view that input's true label. Notice that inputs are chosen adversarially by the environment (with no assumptions on the input distribution).

Algorithm 4 The MB Protocol

```
The agent knows the input space $X$, output space $Y$, and the Hypothesis class $H$.
The environment chooses $h^* \in H$ adversarially.
for Each input $x \in X$ chosen adversarially by the environment do
  The agent predicts $\widehat{y} \in Y$
  if $\widehat{y} \neq h^*(x)$ then
    The agent has made a mistake.
    $h^*(x)$ is revealed.
  end if
end for
```

A hypothesis class H is said to be efficiently learnable in the mistake-bound setting if the number of times the agent makes a mistake ($\widehat{y} \neq h^*(x)$) can be bounded by a polynomial function over problem's parameters. For instance, in a situation where an agent was given an input $x \in [0...k]$ and had to predict a deterministic output from $[0...k]$ for each x, an agent that simply memorizes each $x \mapsto y$ mapping could MB-learn the function with at most k mistakes. Here, we summarize a few results from the literature on mistake bound learning. Some of these classes will be used extensively in Chapter 4, where we go into further detail on their inner-workings and analyses.

- Conjunctions over n literals where the conjunction can contain any combination of 0's and 1's for each literal can be mistake bound learned (Littlestone, 1988; Kearns & Vazirani, 1994). The algorithm for doing so, which we call *MB-Con*, predicts $false$ as the label until it sees a positive example. After that, it maintains a set of literals $l_j \in L_H$ where $l_j = 1$ for every positive example it has seen before[2]. If $\forall l_{jt} \in L_H = 1$ in x_t, the agent

[2] For non-monotone conjunctions, the set of literals can be doubled to consider literals with0 values.

correctly predicts *true* , otherwise it defaults to *false* . Further details are provided in Section 4.2.1.

- k-term-DNF (disjunctive normal form of $k = O(1)$ terms). k-term-DNF are of the form $(l_i \wedge l_j \wedge ...)_1 \vee ... \vee (... \wedge ...)_k$, that is, a disjunction of k terms, each of at most size n. This class of functions is known to be MB learnable (Kearns & Vazirani, 1994) by creating a conjunction of new literals, each representing a disjunction of k original literals (for $k = 3$ we would have $l_{ijm} = l_i \vee l_j \vee l_m$), and then using the conjunction learning algorithm described above.

Unlike PAC, MB does not rely on an i.i.d. assumption over the inputs, so it is a little closer to what we need for the autonomous model learning in the reinforcement learning setting. However, MB results are still *insufficient* for model learning that induces PAC-MDP behavior because they fail the second criterion mentioned earlier: they do not keep track of model certainty. To see why this trait is vexing, consider an MDP that models a combination lock with n tumblers such that each tumbler can have values in $\{0/1\}$ and there are $n + 1$ actions, one to flip each tumbler, and one to "open" the lock. All rewards are -1 except for opening the lock with the exact combination (where $r = 0$). Notice that learning the combination in the MB setting can be done with a single mistake: the agent simply predicts "false" until it sees the combination. However, for an RL agent using this MB hypothesis as its model, the "default to false" predictions force it to assume every combination will result in the same outcome (not open), and that none of them will have a reward that is any better than the others. There is no explicit demarcation of what combinations have been tried before and which have not. Thus, the MB learner provides no guidance for exploration and the agent is left to thrash at random.

Thus, MB learning is not sufficient for PAC-MDP learning. This incompatibility is addressed in the following KWIK framework, which like MB, makes no assumptions about the distribution over the inputs, but unlike MB, explicitly keeps track of which predictions it can make with confidence, and which ones it does not know.

2.2.2 KWIK - A Framework for Model-Based Reinforcement learning

The KWIK framework (Li et al., 2008; Li, 2009) and associated protocol (Algorithm 5) are similar to the MB setting described above in that inputs are picked adversarially, but there are distinct differences:

Learner	Usage	Bound
Coin-Flipping	Binomial Distribution	$O(\frac{1}{\epsilon^2}\frac{1}{\delta})$
Dice-Learning	Multinomial (n outcomes) Distribution	$O(\frac{n}{\epsilon^2}\frac{n}{\delta})$
Enumeration	Hypothesis class H	$H-1$
Union	Combining k hypothesis classes	$(k-1)+\sum_i B_i(\epsilon/2,\delta/k)$

Table 2.1: Some simple KWIK-learnable classes and combination architectures for KWIK learners and their KWIK bounds for realizable hypothesis classes.

- Like the PAC framework, KWIK is designed for either deterministic or stochastic labeling as the general goal is to predict the expected value of $h^*(x)$. The labels provided to a KWIK learner may be noisy observations $z(x_t)$ such that over time the expected label $y_t = E[h^*(x_t)]$, and the agent's goal is to predict the correct expected value y_t.

- Now, instead of always having to predict a label from Y, the learner has the option of instead predicting a label or a unique symbol $\perp$ ("I don't know"), signaling the agent's uncertainty as to the true label for x_t

- Labels (potentially noisy) are only provided when the agent predicts $\perp$.

We will often describe the process of using this framework as simply *KWIK-learning.*

With these differences in the protocols, the definition of sample efficiency also changes in the KWIK framework.

Definition 5. *A model class is said to be* efficiently KWIK-learnable *(or just* KWIK-learnable *when the meaning is clear) if and only if with high* $(1-\delta)$ *probability: (1) All* $\widehat{y}_t$ *predictions that are not* $\perp$ *are* ϵ*-accurate, that is* $||\widehat{y}_t - E[h^*(x_t)]|| \leq \epsilon$ *(or in the discrete case* $\widehat{y}_t = E[h^*(x_t)]$*) and (2) the number of times* $\widehat{y}_t = \perp$ *is bounded by a polynomial function of the problem description.*

Similarly, we can classify learning problems based on our ability to KWIK solve them using the following definition.

Definition 6. *A learning problem is said to be* KWIK-solvable *if and only if there exists a KWIK algorithm for a model class (as stipulated above) that will make predictions for inputs in the learning problem in accordance with the KWIK criteria.*

Several simple KWIK learners and combination architectures for KWIK learners are listed in Table 2.1, along with their accompanying KWIK bounds from Li (2009).

The relationships between PAC, MB and KWIK are summarized as follows. Any KWIK algorithm for a deterministic hypothesis class can be turned into an MB algorithm by simply

Algorithm 5 The KWIK Protocol

```
The agent knows the accuracy parameter $\epsilon$ and $\delta$, the input space $X$, output space $Y$, and the Hypothesis class $H$.
The environment chooses $h^* \in H$ adversarially.
for Each input $x \in X$ chosen adversarially by the environment do
    The agent predicts $\widehat{y} \in Y$ or $\perp$ ("I don't know")
    if $\widehat{y} = \perp$ then
        $z(x)$ is revealed. This is a noisy observation of $h^*(x)$.
    end if
end for
```

predicting a random element of Y any time the KWIK learner was going to predict $\perp$. However, an efficient MB learner for a hypothesis class does not guarantee an efficient KWIK learner. A simple counter-example to this relationship is conjunction learning over n terms, as in the combination lock example described above. MB can learn the conjunction by exploiting asymmetry in the information content of the labeled examples. By defaulting to false, it is able to only "count" the highly informative positive examples in its sample complexity measure. KWIK learners are not afforded this luxury, because they must, with high probability, never make an incorrect prediction. Hence, when learning a conjunction, a KWIK learner can make $2^n - 1$ $\perp$ predictions, one for every new combination of literals, before it sees even one positive example. The relationship to PAC is then straightforward because all MB algorithms can be turned into PAC algorithms, but the reverse is not necessarily true (Blum, 1994).

As a "self aware" learning framework, KWIK lends itself to learning RL models (as elaborated on below) because, unlike PAC, it allows for adversarial inputs and unlike MB, KWIK learners explicitly admit when they are uncertain about portions of the model, providing exploration guidance.

2.3 KWIK-R-max, A general Algorithm for Sample Efficient RL

We now return to the full reinforcement learning problem and will show how to generalize the Flat-Rmax algorithm (Algorithm 3) to work for any model-class that is KWIK-learnable. We first describe the algorithm and give a sketch of the previous analysis of its PAC-MDP property. We then briefly list some domain classes that have been previously shown to be KWIK-learnable, and therefore PAC-MDP learnable. These results are proven in far greater detail by Li (2009) but are essential for the results described in the next chapter, and in Chapter 4 we will make a similar connection between a framework related to MB and efficient RL in the apprenticeship setting.

KWIK-Rmax (Algorithm 6) is a model based RL algorithm that generalizes the ideas of the original Flat Rmax (Algorithm 3) by replacing the “counts” on states with a reliance on KWIK model learners. Specifically, instead of expressly enumerating the state space (which could be exponentially large for compact models such as factored MDPs), the algorithm connects the MDP planner (such as Value Iteration) directly to the KWIK model learner KL, which can be queried by the planner to establish the actual state transitions. If, for a queried transition or reward, KL predicts $\perp$, this area of the state space is unknown (the analogue to $count(s, a) < m$ from Algorithm 3) and replaced by an Rmax trap-state transition.

Algorithm 6 KWIK-Rmax (Li, 2009)

```
Agent knows $S$ (in some compact form), $A$, $\gamma$, $\epsilon$, $\delta$
Agent has access to planner $P$ guaranteeing $\epsilon$ accuracy
Agent has KWIK learner $KL$ for the domain
$s_0$ = start state
while Episode not done do
    $M_A = KL$ with $\perp$ interpreted optimistically (Rmax state transition)
    $a_t = P$.getAction($M_A, s_0$)
    Execute $a_t$, view $s_{t+1}$, $KL$.update($s_t, a_t, s_{t+1}$).
end while
```

The following Theorem from Li (2009) establishes that this KWIK-Rmax algorithm is PAC-MDP for efficiently KWIK-learnable models. It also links the sample complexity of the RL agent directly to the sample complexity of learning the model, allowing us to concentrate on the latter when establishing PAC-MDP results.

Theorem 1. *(From Li (2009)): Let $\mathcal{M}$ be a class of MDPs with state space S and action space A. If $\mathcal{M}$ can be (efficiently) KWIK-learned by algorithms KL_T (for transition functions) and KL_R (for reward functions) with respective KWIK bounds B_T and B_R, then the KWIK-Rmax algorithm is PAC-MDP.*

The proof of the theorem (which is highly technical and can be found in detail in Li (2009)) relies on three properties of the model represented by KL: optimism, accuracy, and a bounded number of surprises. Optimism here means that with high probability, the value function for a partially learned model satisfies $V^*_{M_A} \geq V^*_M$, because of the Rmax-state transitions. Accuracy and bounded number of surprises (equivalently bounded $\perp$ predictions) follow from the two criteria for efficient learning in the KWIK framework. The proof also relies on a version of the *simulation lemma*, which establishes that if the learned model M_A is sufficiently accurate, behavior under policy π in M_A will net similar expected values to π executed in M. More formally, the lemma (originally due to Kearns & Singh (2002), though here a more specific version close to that from Li (2009) is presented) can be stated as:

Lemma 1. *Let M and M_A be two MDPs with the same S, A, and γ but the transition (T and T_A) and reward (R and R_A) functions are such that there exists two constants ϵ_T and ϵ_R, such that $||T(\cdot|s,a) - T_A(\cdot|s,a)|| \leq \epsilon_T$ and $||R(s,a) - R_A(s,a)|| \leq \epsilon_R$ where $||\cdot||$ is a 1-norm. Then for all $s \in S$, $||V(s) - V_A(s)|| \leq \frac{\epsilon_R + \gamma V_{max} \epsilon_T}{1-\gamma}$, where $V_{max} = \frac{R_{max}}{1-\gamma}$.*

This lemma can be extended to state/action value functions as well as the case where S is uncountable (proofs of both cases appear in Li (2009)). We will make use of the contrapositive of this statement (values for 1 MDP being very different from another means that one of their transition or reward entries are very different) in our theoretical results later on.

In summary, if a class of MDP models is KWIK-learnable, the KWIK-Rmax algorithm can be used along with the appropriate KWIK learner, and Algorithm 6 is guaranteed to be PAC-MDP. This result generalizes RL sample complexity results in a number of propositional domains, including compact models, that are known to be KWIK-learnable. These include Flat MDPs, where KWIK-Rmax simply devolves into the Flat-Rmax algorithm introduced earlier, and factored MDPs (Diuk et al., 2009; Walsh et al., 2009b). In the latter, the sample complexity bounds are not dependent on the number of states, rather they depend on the $O(log(S))$ number of factors, which is an exponential improvement over a naïve learner that would flatten such a state space and learn about every possible combination of factors. Several models of continuous dynamics (with $S \in \Re^n$) are also known to be KWIK-learnable (Strehl & Littman, 2007; Brunskill et al., 2008).

While these model classes are all of interest to the RL community and allow for generalization and tractable sample complexity in large state spaces, this thesis considers *relational* MDPs where even more generalization and compression is possible because the dynamics of individual objects are represented in terms of relations with variables. The next section introduces relational languages and presents some results from the field of relational reinforcement learning, though the exact formalism used in this work (relational action schemas) is presented later (Section 3.1).

2.4 Languages, Actions, and Learning for Relational Models

This section covers background material on the use of relational languages in reinforcement-leaning-like domains. Such representations allow us to model transitions with even more generality than even the propositional DBNs mentioned above. This advantage is gained by using action descriptions that contain variables, thereby representing transitions over generic objects rather than over each individual grounding. For instance, in the Blocks World domain illustrated in Figure 1.2, the relational Stochastic STRIPS rule in the diagram represents the pre-conditions and effects of performing the *move* action on any 3 objects. It covers moving **a** from **b** to **c** or moving **b** from the **table** to **a** or any other valid movement to a block. In contrast, a propositional DBN or flat MDP both need to be trained on each of these transitions separately.

The combination of relational representations and sequential decision making domains goes back to the early days of Artificial Intelligence research, and has gone through many transformations as the fields of machine learning, inductive logic programming, and reinforcement learning each came of age. One major landmark in this evolution was the establishment of the field of Relational Reinforcement Learning (RRL)(Dzeroski et al., 2001; van Otterlo, 2009), which blended (mostly model-free) RL algorithms and relational state/dynamics descriptions. While we cover a number of important and relevant works from this literature, a far more comprehensive history and portrait of this still maturing field can be found in the recent book by van Otterlo (2009). We begin by describing very early work with relational languages for action descriptions.

2.4.1 Early Approaches

The idea of using relational representations to model actions was used in many of the first theoretically grounded AI systems. These early works aimed to capture the most general version of the world possible, often incorporating full (and undecidable) first order logic. Perhaps the most famous of these frameworks is the Situation Calculus (McCarthy, 1963) (SC). The Situation Calculus is an axiomatized logical framework that allows state descriptions to be expressed in full first-order logic (quantifiers, disjunction, etc.) and transitions are calculated by applying successor axioms (such as $\mathit{paintGreen}(\mathrm{X}) \rightarrow \mathit{PaintedGreen}(\mathrm{X})$) to the current situation. These actions can also be described conditionally, based on full first-order logic formulas. Because of this, planning with the situation calculus or related languages often involves the use of a first order theorem prover, as it is possible to encode undecidable problems in this language.

In contrast, the more restricted STRIPS (Fikes & Nilsson, 1971) language describes transitions for actions without axioms and using variablized action descriptions made up of a list of pre-conditions, an Add-list, and a Delete-list similar to the blocks world example in Figure 1.2 (but with only deterministic outcomes). A STRIPS operator (a single action description) can only reference the variables in its action list and the action only *succeeds* (actually executes) if all the relations in the pre-condition list (with variable substitution based on the action's ground parameters on invocation) are true in the current state. If so, the relations in the Add-list are made true in the next state and the relations in the Delete list are made false. All other grounded relations in the current state retain their current truth values.

States in STRIPS domains do not contain logical formulas, and instead are encoded just as a set of grounded predicates that are currently true. Because of this, and the fact that planners must consider grounded action invocations, it is often remarked that "STRIPS is propositional", which is true from the planning perspective, but not true in terms of the model (which is what our learning agents will construct). That is, the STRIPS operators themselves represent transitions using variables, and can be applied to multiple propositional groundings (for example, the *pickup*(X) operator can explain *pickup*(**a**) or *pickup*(**b**)), so the language is indeed relational. Because it does not come anywhere near capturing the complexity of first-order logic, planning in STRIPS is more reasonable than in SC (STRIPS planning is P-Space Complete (Bylander, 1994)) and several heuristic planners perform well in many benchmark STRIPS domains (such as GraphPlan (Blum & Furst, 1995)).

Following the terminology of van Otterlo (2009), we can separate the families of languages derived from these earlier works into two classes: Action Formalisms and Action Languages. Action Formalisms are the languages derived from the likes of the situation calculus, such as the programming language GOLOG (Levesque et al., 1997). The tell-tale characteristic of action formalisms is their axiomatization of the world's dynamics, that is, a logical grounding for the progression of states, as encoded in the successor state and frame axioms (see (Reiter, 1991)) in SC. Because of this strong logical underpinning, Action Formalisms can support complicated (such as first-order) logical formulas in the state description. In contrast, Action Languages, like STRIPS or its stochastic variant, NID rules (Pasula et al., 2007), have no explicit axiomatization. States are usually represented with relations over ground objects (rather than having actual first-order formulas in the state), and the rules for transitions are somewhat less structured. While somewhat less rigorous, the restricted (and more manageable/tractable) state spaces associated with action languages and the freedom to model dynamics without resorting to logical axiomatization, make them a more practically reasonable and easier studied representation for integration with reinforcement learning. This thesis considers action languages, though we note this does not preclude the application of these ideas to action formalisms, and in fact we move slightly in this direction with our consideration of a (more close to axiomatized) language based on Description Logics in Section 6.2.

2.4.2 Inductive Logic Programming and Early Attempts at Action Schema Learning

While hard-coded relational action languages for representing static domain dynamics made their debut early on in the AI continuum (McCarthy, 1963), they were not very widely considered in the machine learning literature until the rise of the field of Inductive Logic Programming (ILP) (Nienhuys-Cheng & de Wolf, 1997) in the early 90's. ILP is a large field that studies the *supervised learning* of relational concepts. A full summary of ILP results and algorithms is well outside the scope of this thesis, but we mention a few algorithms and sample complexity results here because ILP has had such a strong influence on the field of relational RL (covered below) and because some of these supervised sample complexity results have a bearing on the model learning components of our algorithms in later chapters.

Inductive Logic Programming algorithms take a set of training data, which can be thought of as instances of grounded relational predicates (or rows in a relational database), with a label on each instance marking it as positive or negative. For instance, suppose we were trying to learn which states are goal states and which aren't, in blocks world (assuming goal states are instances where more than three blocks are in a stack). The instances in the training data might look like[3] $\{On$(a,b), On(b,c)$|+\}$, $\{On$(a,b), On(b,table), On(c,table)|-$\}$, and $\{On$(a,b), On(d,b), On(c,table)$|+\}$. From these and other examples, an ILP algorithm would be expected to infer the concept $Goal \leftarrow On(X,Y) \wedge On(Y,Z) \wedge Block(X) \wedge Block(Y) \wedge Block(Z)$.

In early ILP work, a number of sample complexity results were derived in the PAC framework. These included early results on learning specific concept types with restrictions on the constructors and structure of the definitions (Dzeroski et al., 1992) and even those encoded with

[3] Descriptions shortened for readability.

Description Logics (Cohen & Hirsh, 1994). A number of important PAC results (including a very fine separation of positive and negative learnability) in ILP as well as some elegant constructive algorithms came in a series of papers from Cohen (1995a,b). The positive algorithms in these works concentrated on representing the most specific concept that covered the training data seen so far and generalizing based on data, a tactic we employ in a number of algorithms in this thesis. However, the community has somewhat moved away from such sample complexity analysis in recent years in favor of studying heuristic learning algorithms in richer languages (inching closer to full First Order Logic).

Around the same time that ILP was emerging as a powerful field in machine learning, a number of researchers (Wang, 1995; Gil, 1994; Benson, 1996) began working on learning *action schemas*, a generalized term for the action languages (like STRIPS rules) shown above. We describe many of these in greater detail in Section 3.3, but we mention here that a number of these algorithms began by applying ILP algorithms directly to logged experience or experience provided by a "teacher" (for example Benson (1996)). Most of these algorithms used heuristic-search style methods and their goals were more towards learning specific "gold standard" models rather than optimal behavior. However, these action schema learners served as the first step towards using RL algorithms in concert with relational representations, and in many ways can be seen as primordial model-based relational RL algorithms. But interestingly, as the field of RRL developed (as chronicled below), the algorithms progressed[4] from these model-based systems to model-free approaches (the opposite of the traditional RL timeline in many ways).

2.4.3 Relational Reinforcement Learning

The field of relational reinforcement learning (RRL) (Dzeroski et al., 2001; van Otterlo, 2009) sprung into its current form following a series of talks and workshops in the late 90's culminating in the seminal journal paper entitled *Relational Reinforcement Learning* (Dzeroski et al., 2001). That work introduced the Q-RRL algorithm which used the model-free RL algorithm Q-learning as its learning backbone but connected it to a first order decision tree learner—using the tree learner as a function approximator $\widehat{Q}$ in the Q-learning backup:

$$Q(s,a) = Q(s,a) + \alpha(r(s,a) + \gamma \max_{a'} \widehat{Q}(s',a') - Q(s,a))$$

[4] Some might say digressed.

This generic backup with approximators had been used in Q-learning with Neural Networks and linear function approximators (for $\widehat{Q}$) (Sutton & Barto, 1998) and the use of relational decision trees is a technique common in ILP. But here for the first time, a function approximator specifically tailored to relational domains was employed in RL, and domains with sequential decision making characteristics were considered from an ILP-perspective. From the RL side, this was quite a leap because, all of a sudden, environments like blocks world (where Q-RRL was extensively tested) could be considered regardless of the number of blocks.

By classifying states based on Q-values, Q-RRL implicitly performed a form of *state abstraction* known as Q^*-irrelevance (Li et al., 2006). This form of abstraction is known to be "safe" in the learning setting in that treating states with the same Q-values under the optimal policy as being the same does not stop model-based or model-free algorithms from converging. Thus, a degree of the empirical success and the convergence guarantees for simple algorithms like Relational $TD(\lambda)$ (Kersting & Raedt, 2004) can be attributed to the use of this safe state abstraction. We note the another algorithm introduced in the original RRL paper, P-RRL, which used an unsafe abstraction known as policy-irrelevance (Jong & Stone, 2005) can fail to converge.

One major drawback to Q-RRL was that it had to recreate the entire decision tree on every step, forcing it to maintain a large number of instances and take an inordinate amount of computation time on each step. This problem was alleviated by the introduction of the RRL-TG algorithm (Driessens et al., 2001) which incorporated an *incremental* decision tree learner (TG) into the Q-learning architecture, greatly speeding up the algorithm. Variants of the RRL-TG algorithm dominated the RRL field for the next several years and achieved a modicum of empirical success. For example, an instance-based implementation of this algorithm (Driessens & Ramon, 2003) performed well in a diverse set of relational testbeds and in a larger empirical study of RRL techniques (Dzeroski, 2002) it was used to play video games such as Tetris and "Digger".

Meanwhile, the planning community was developing a number of powerful representations and new planning algorithms. For instance, Boutilier et al. (2001) introduced a version of Value Iteration that was shown to provably converge for general first-order MDPs (FOMDPs) written in a *stochastic* version of the Situation Calculus. This work then progressed to approximate linear programming (Sanner & Boutilier, 2005), approximate policy iteration (Fern et al., 2006), and linear function approximation (Sanner & Boutilier, 2006) for FOMDPs and work on a similar representation based on the Fluent Calculus (Großmann et al., 2002). By the middle of the 2000's, the stage was set for another batch of RRL and action schema learning works, focussed on more complex problems with more exotic planning and learning algorithms.

2.4.4 Later RRL Approaches

The next wave of RRL techniques used more exotic representations, better algorithms, or delved into other areas of AI. For example, a new instance-based RRL algorithm that used graph kernels to compare the structure of abstract states (Driessens et al., 2006) (as opposed to the older hand-tooled similarity metrics) saw success during this period. Work on model-based RRL (Croonenborghs et al., 2007b) also progressed, again using mostly random exploration. Larger environments were also tackled, including real time strategy games, with advanced planning techniques (Guestrin et al., 2003a; Balla & Fern, 2009).

One area that saw a great amount of progress due to synergy with RRL was the field of *transfer learning*, which is concerned with porting models, policies, or value functions between domain instances, a natural fit for (variablized) relational models. Algorithms for transfer in navigation tasks (Lane & Wilson, 2005) and general RRL tasks with model-based approaches (Walker et al., 2007), options (extended time/non-atomic actions) (Croonenborghs et al., 2007a), as well as environments where a teacher was available (Torrey et al., 2005) all achieved degrees of success in this fairly new field.

A number of new planning algorithms for relational and first order MDPs were also introduced (a good summary of the modern results appears in Sanner & Boutilier (2009)). These included advancements in First Order Decision Diagrams (FODDs) (Wang et al., 2008; Joshi et al., 2009), which provided a compact representation of the dynamics and value function for a FOMDP. Also, an advanced planner (Lang & Toussaint, 2009) for *Noisy Indeterministic Deictic* (NID) rules (Pasula et al., 2007), showed how to use an effective Bayesian Inference technique to produce approximate plans much faster than previously used more general (but slower) techniques like Sparse Sampling (Kearns et al., 2002).

In this renewed climate, the field of action-schema learning saw a reprisal in a number of challenging domains with much stronger languages than were used in the first incarnation. We cover these in more detail in the next chapter, but they included work on learning NID rules (Stochastic STRIPS with "noise" outcomes) (Pasula et al., 2007), learning with partial observability on the data (Shahaf, 2007; Yang et al., 2007), and learning hierarchical structures over the atomic action operators (Zhuo et al., 2009).

2.4.5 Excursions into Exploration

Perhaps because of the historical roots in inductive logic programming, which assumes a large database of facts as training data, the classic RL problem of exploration versus exploitation has received far less attention in the RRL community than it has in traditional RL. Still, there are some exceptions. One early work on RRL (Driessens & Dzeroski, 2002) showed how to introduce a rudimentary but common exploration technique (Boltzman exploration) into RRL-TG. In order to keep the exploration focussed, the system was bootstrapped with a number of expert traces (similar to the apprenticeship learning paradigm we use in Chapter 4). But the reliance on Boltzman exploration and Q-learning, a combination known to have negative sample complexity results even in propositional domains (Whitehead, 1991) is not sufficient for attaining the theoretical (or practical) guarantees that algorithms like KWIK-Rmax can make.

A later approach to integrating intelligent exploration and tree-based RRL came in the model based setting in the form of the MARLIE system (Croonenborghs et al., 2007b). Instead of representing the Q-function with a decision tree, MARLIE uses a set of relational decision trees to predict the transition and reward function for the domain. While MARLIE combines many of the same ingredients we use in this work (model-based RL, compact representations, and an approximate planner), the exploration in these domains is done in a greedy manner, without explicit demarcation of known and unknown regions, and so it does not give convergence guarantees. Instead, the "exploration" in this work is more a matter of speeding up the Q-learning backbone of RRL-TG by using a lookahead tree and learned model to propagate values faster.

Perhaps one of the closest RRL systems to those presented in this dissertation is the REX (Lang et al., 2010) algorithm, which uses an architecture based on E^3 (Kearns & Singh, 2002) to trade-off exploration and exploitation while learning NID rules. Like our $R_{\max}$-based techniques, REX uses optimism in the face of uncertainty to guide the agent. However, REX is designed for a more complicated language (with deictic references, noise outcomes, etc.) than those used in most of this thesis. This strong language and the planner used to reason about it (PRADA (Lang & Toussaint, 2009)) requires REX to approximate the uncertainty in states in a factored manner, resulting in good empirical exploration, though no sample-complexity bounds have been derived. However, the empirical validation of REX in many ways justifies many of the same techniques we employ in our own solutions (model-based RL and optimism in the face of uncertainty, among others).

In terms of sample complexity, there has been very little work in RRL on PAC-MDP bounds similar to those in mainstream RL, but there are again a few exceptions. Early work on relational MDPs (Guestrin et al., 2003a) considered more traditional PAC bounds for a scenario where agents are learning in a class of environments (what we later call a domain, like Blocks World) and a sample is defined as a single instance of this environment that the agent interacts with (Blocks World with 6 specific blocks). Thus, exploration is measured in terms of the number of worlds the agent is exposed to, not the number of steps it takes, and the derived PAC bounds are dependent on the distribution of worlds (which is not adversarial). In some ways, this measure is closer to the sample complexity metric we use in Chapter 4, where we consider the number of teacher interactions with the agent in the complexity bounds. But the passive learning scenario and distributional assumptions considered in this earlier work does not allow for a direct translation.

In recent years, PAC-MDP bounds have been derived for agents using the Object Oriented MDP language (Diuk et al., 2008), but this was only for a specific deterministic language that we connect our formalism to in Section 6.1.1. Also, our own work that has appeared in various forums (Walsh & Littman, 2008; Walsh et al., 2009b) established sample complexity and PAC-MDP bounds for specific languages and learning subproblems, some of which are presented in greater detail in Chapters 3 and 4. Other than these works, there has been very little consideration in the RRL community to the exploration-exploitation dilemma.

2.5 Moving Forward

This Chapter has summarized results in two hitherto disparate fields. On the one hand, we have seen the field of traditional (propositional) reinforcement learning expend considerable energy on the exploration-exploitation dilemma. On the other hand, we have seen that the representations studied in this still-emerging field of relational reinforcement learning are able to compactly model domains that fell the basic propositional representations used in traditional RL. Yet RRL has virtually ignored the exploration-exploitation dilemma, and in many ways has followed an opposing trajectory from traditional RL by drifting from clearly model-based approaches (action schema learning) to model-free approaches (like RRL-TG) that have known difficulties with exploration. This disconnect is unfortunate because both sides have valid points and advantages—sturdy theoretical footing and exploration strategies in RL and better representations and a stronger link to the seminal works of artificial intelligence in RRL. The rest of this thesis attempts, in one form or another, to bridge this divide, showing that we can use model-based RL with relational representations and still solve the exploration-exploitation trade-off efficiently in several different learning paradigms. In the next chapter, we present algorithms for doing so (and remark on their limitations) in the online RL setting.

Chapter 3

Online Action-Schema Learning

This chapter describes positive and negative results on the sample efficiency of learning a class of relational *action schemas*.[1] Our investigation of learning these models takes place within the context of the KWIK framework for a number of different dynamics settings and schema types. Specifically, we consider different combinations of determinism/stochasticity and preconditional/conditional effects and several different learning settings where parts of the schemas are known to the agent beforehand. These results comprise the first general theoretical study of the exploration/exploitation tradeoff for agents that use relational representations.

3.1 Terminology and the Representation

This section lays out the formal terminology for action schemas and introduces two example domains used throughout this work: the Blocks World and Paint-Polish domains. Both of these domains are best described relationally because they contain objects and parameterized actions that act in the same way on different objects in comparable conditions. For instance, picking up block **a** off the table is virtually the same as picking up block **b** off the table. Once one knows how to pick up a block, one can apply this knowledge to determine what will happen when picking up *any* block.

In contrast, representing these dynamics using a propositional structure, even a "generalized" one such as a DBN, would require factors for every possible ground literal (all combinations of *On*(X, Y) for all blocks X and Y). This number of factors is large, though not exponential if predicates have constant arity, *but* the number of parents for each factor will be very large (certainly not $O(1)$ as the standard assumptions for DBN learning require). For instance, the factor *On*(**a**, **b**) will depend on every other *On*(**a**, X) factor because these need to be considered to determine the success of the *move* action. We now describe a more compact, and more

[1] Portions of this chapter appeared earlier in joint work with Michael Littman, István Szita, and Carlos Diuk (Walsh & Littman, 2008; Walsh et al., 2009b).

intuitive, representation for such domains. Specifically, we introduce a class of languages called *relational action schemas* composed of relational conditions and effects, whose general form covers a wide variety of action languages.

3.1.1 Terminology

A *relational action schemas* can be thought of as a compressed version of the transition function (T) used in standard propositional MDPs. But, unlike the traditional MDP formalism, our schematic domain encodings describe dynamics at a conceptual, rather than a propositional level (for example, *travel*(X, Y) rather than *travel*(**paris**,**rome**). As covered in the previous chapter, a number of action-description languages have been proposed throughout the years, including STRIPS rules, OOMDPs, and even stronger (axiomatized) action formalisms such as the situation calculus or FOMDPs. In contrast, and somewhat in deference to this plethora of languages, in this work we try as best as possible to present results that are language agnostic, though the bulk of our results will be focussed on a Stochastic STRIPS language.

Because our action schemas form a compact representation of the transition function, it should be noted that these results do not cover the model-free approaches taken in the RRL community where relational state descriptions are associated with state or state-action values (V or Q). Rather, relational action schemas are built to represent the actual dynamics of the model, for use in a model-based reinforcement learning-setting.

The notion of *state* in these environments will be dependent on the particular encoding we consider, but we will generally consider the states to be composed of a set of fluents $f \in \mathcal{F}$ that can be true or false at each timestep. For instance, in STRIPS, predicates such as $At(\mathbf{a}, \mathbf{b})$ are the fluents. In this work, we will often use the term *literal* in place of “fluent” when describing a part of a formal description rather than a changing part of the state. For instance, we may describe an action *move*(X,Y,Z) as affecting the literal *On*(X, Y), and the instantiation of the action *move*(**a**,**b**,**c**) as affecting the *ground literal* *On*(**a**, **b**).

With this relational representation of a state, we can now turn our attention to defining the generalized dynamics of action-schema domains. Formally, we start with definitions of a domain instance.

Definition 7. *A* domain instance *is a* 6*-tuple* $\langle \mathcal{F}, \mathcal{F}_0, \mathcal{A}, R, \gamma, S_T \rangle$ *where* $\mathcal{F}$ *is a finite set of possible grounded fluents (over a set of available objects* $\mathcal{O}$*),* $\mathcal{F}_0 \subseteq \mathcal{F}$ *is the subset of fluents true initially,* $\mathcal{A}$ *is a set of action schemas as defined below,* R *is a reward function,* $R : S, a_G :\mapsto \Re$*,* *where S is a state comprised of grounded fluents and a_G is a grounded parameterized action as described below, γ is the discount factor, and S_T is a set of terminal states (described using $\mathcal{F}$).*

When describing environments, we may refer simply to a *domain*, with a similar definition, but without a defined initial or terminal states, and with only ungrounded fluents available (blocks world without a specific set of blocks). We now describe the dynamics of action schemas with different restrictions on the conditions and effects encoded in the dynamics. We will progress from the most simple encoding (unconditional deterministic schemas) to the most general form (conditional stochastic schemas). A summary of the six conditions considered and example actions encoded under these restrictions is presented in Table 3.1.

Definition 8. *A* deterministic unconditional action schema *is a pair $\langle a, \omega^a \rangle$ where a is a parameterized action with variables holding the parameter places and ω^a is an effect that describes the changes in the truth values of the fluent set.*

For example, in our Traveling domain examples (Table 3.1), a deterministic unconditional action schema for the action *travel* might be $a =$ *travel*(X,Y), $\omega^a :=$ *At*(Y), $\neg$*At*(X). In cases where the literals are attribute-value pairs, effects simply represent changes in the values. For instance, the action *moveUp*(Object) might result in the effect [$\omega^a :=$ Object.Ycoordinate = Object.Ycoordinate+1].

We can press beyond these basic deterministic descriptions and consider the case where the outcomes of the actions can be stochastic. For instance, the action *travel*(X, Y) may result in the effect *At*(Y) with probability .9 and the effect *At*(X) with probability .1. More formally:

Definition 9. *A* Stochastic Unconditional action schema *is a triple $\langle a, \Omega^a, \Pi^a \rangle$ where $a \in A$ is again a parameterized (with variables) action and Ω^a is a set of effects (ω_i^a) whose probability of occurrence is given by the corresponding $p_i^a \in \Pi^a$. Notice $|\Omega^a| = |\Pi^a| = n$ and $\sum_{i=1}^{n} p_i^a = 1$.*

Intuitively, when the action a is taken, exactly one of the effects from $\{\Omega^a\}$ occurs according to the probability distribution induced by Π^a.

Action schemas may also contain pre-conditions that govern whether the action *succeeds* or *fails.* For instance, in the travel example, *travel*(X, Y) may result in the distribution of effects above under the condition *HoldingTicket*(X,Y), otherwise the action will not have any effect. More formally:

Definition 10. *A* Pre-Conditional Stochastic Action Schema *is a 4-tuple $\langle a, c^a, \Omega^a, \Pi^a \rangle$ where a is again a parameterized action and Ω^a and Π^a represent the possible effects and distribution* *of effects* if the action succeeds. *An action is said to succeed if it is executed in a state whose literals $\mathcal{F}_s$ satisfy c^a (the pre-conditions are true). Otherwise the action* fails*, the agent receives a "failure signal" and no change is made to the current state.*

These actions with pre-conditions can be further generalized to a situation where there are multiple effect distributions, each corresponding to a different *condition*. For instance, under the condition $RainAt$(X) the distribution of effects for *travel*(X, Y) could be At(Y) with probability .7 and At(X) with probability .3, while non-raining situations will result in the distribution above. While the nature of these conditions will again be a function of the language used, we can generally describe this situation as being based on a set of conditions C^a governing which of many effect distributions ($\mathbf{\Omega^a_i}$) the current effect is drawn from. More formally:

Definition 11. *A* Conditional Stochastic Action Schema *is a 4-tuple* $\langle a, C^a, \mathbf{\Omega^a}, \mathbf{\Pi^a} \rangle$ *. a is again a parameterized (with variables) action, and* $\mathcal{C}^a$ *is a set of* ***mutually exclusive*** *conditions based on the set of fluents. For instance,* C^a *could be a set of conjunctions with variables drawn from the parameters of a. For each* $c^a_i \in C^a$*, there is a corresponding effect distribution* $\langle \Omega^a_i, \Pi^a_i \rangle$.

We note that the deterministic versions of these schemas involve the same condition structure but only a single outcome ($\Omega^a_i = \{\omega^a_i\}$) is linked to each condition. A summary of these different languages, with example schemas, is presented in Table 3.1. We note that these action schemas bear close resemblance to many specific languages in the literature. For instance, conditional stochastic action schemas with Add and Delete lists for effects and a special "noise" effect would be the same as NID rules (Pasula et al., 2007). Our schemas can also be seen as rules for defining the dynamics of basic First Order MDPs (FOMDPs) (Sanner & Boutilier, 2009) or PPDDL rules (Younes et al., 2005), though incorporating the complex (not just conjunction) conditions and universally quantified effects allowed in those languages are not considered in the bulk of this thesis.

3.1.2 Linking Action Schemas and general MDPs

In this section, we formalize the link between relational action schemas and the propositional MDP models that have been the workhorse of model-based reinforcement learning. We show that the more compact action-schema representation gives an agent far fewer (even exponentially fewer) parameters of T to learn and discuss different ways the reward function may be structured in these domains.

Setting	Parameters	Example
Deterministic Unconditional	$\langle a, \omega^a \rangle$	*travel*(X,Y): → *At*(Y), ¬*At*(X)
Stochastic Unconditional	$\langle a, \Omega^a, \Pi^a \rangle$	*travel*(X,Y): → *At*(Y), ¬*At*(X) (0.9), *At*(X) (0.1)
Deterministic Pre-conditional	$\langle a, c, \omega^a \rangle$	*travel*(X,Y): *HasTicket*(X,Y) → *At*(Y), ¬*At*(X)
Deterministic Conditional	$\langle a, C, \boldsymbol{\Omega}^{\mathbf{a}} \rangle$	*travel*(X,Y): *RainAt*(X) → *At*(X) *Sunny*(X) → *At*(Y), ¬*At*(X)
Stochastic Pre-Conditional	$\langle a, c, \Omega^a, \Pi^a \rangle$	*travel*(X,Y): *HasTicket*(X,Y) → *At*(Y), ¬*At*(X) (0.9), *At*(X) (0.1)
Stochastic Conditional	$\langle a, C, \boldsymbol{\Omega}^{\mathbf{a}}, \boldsymbol{\Pi}^{\mathbf{a}} \rangle$	*travel*(X,Y): *RainAt*(X) → *At*(Y), ¬*At*(X) (0.7), *At*(X) (0.3) *Sunny*(X) → *At*(Y), ¬*At*(X) (0.9), *At*(X) (0.1)

Table 3.1: Six dynamics settings for action schemas and example operators. Boldface indicates a set of sets (conditional case).

Action schemas as defined above can be thought of as a specific kind of transition-function decomposition. Such generalizations have been studied in the reinforcement-learning literature for general state classes (conditions) and outcomes (effects) (Sherstov & Stone, 2005). This previous work defines the transition function decomposition in the following way (with notational changes to make the mapping to our formalism easier to see): $T(s, a, s') = Pr[\eta(s, \tau(\kappa(s), a)) = s']$, $\kappa : S \mapsto C$, $\tau : C, A \mapsto Pr[\Omega]$, $\eta : S, \Omega \mapsto S$. Intuitively, κ is a "type" function that captures a general notion of the state (for example, "the floor is slippery"), τ maps types and an action to a probability distribution over "outcomes" (for example, "agent moves forward 2 units"), and η maps an effect and the current state to a next state. This formalism was previously used to define Relocatable Action Models (RAM) (Leffler et al., 2007) and here we show it can be used to describe an even broader class of models. To map our most general action-schema definition (the conditional stochastic case) to this formalism, we need only to slightly modify the definition of κ to be based on the action as well as the state. That is, we have $T(s, a, s') = Pr[\eta(s, \tau(\kappa(s, a))) = s']$ and $\kappa : S, A \mapsto C$. With that definition, κ captures the conditions of each action, τ captures the effect distributions ($\boldsymbol{\Omega}^{\mathbf{a}}$ and $\boldsymbol{\Pi}^{\mathbf{a}}$), and η describes a language dependent semantics of effects. For instance, in STRIPS, η represents the semantics of general Add and Delete lists.

What remains now is to connect the reward function for domains as described in Definition 7 to the general reward function $R(s, a)$. Ostensibly, this mapping is straightforward since any encoding of the reward function based on the fluents of a state and a grounded action is, by definition, determined by s and a. However, since the state space for a relational domain can be exponential in the number of objects, learning a reward function that, for instance, is only positive in a single state (one specific configuration of blocks world) will trivially lead to exponential sample complexity. If the reward function is not general, then the representation does not help us sidestep the size of the flat state space. This problem is not unique to relational representations; the same caveat must be heeded when employing DBNs. In the DBN case, a number of natural assumptions have been used to enforce structure on the reward function, leaving it compact enough to learn efficiently. Here, we list (though not exhaustively) some similarly small reward functions for action schema domains.

- **Environments with terminal states and rewards based on actions**: Most of the examples in this work simply attach rewards to each action schema. So, invoking that particular action results in a given reward (or expected value over rewards), regardless of the state. Note such schemas only make sense in environments with terminal states (otherwise there would be no need to learn the dynamics because all states would have the same value). A similar setup is possible where the rewards depend on whether the pre-conditions of the action hold, or which condition associated with a conditional action schema matches the current state. As long as the associated parameter (conditions, actions, etc.) *is learnable with polynomial samples* with respect to the domain description, the reward function will be polynomially learnable.

- **Reward based on a goal with existential variables**: While goals based on a specific configuration of the objects (a single state) can require an exponential number of samples to learn, rewards based on a goal such as $\exists X_1 ... X_k On(X_1, X_2) ... On(X_{k-1}, X_k)$, (a stack of $k = O(1)$ blocks in any order) is efficiently learnable. This situation can be seen as a special case of the "rewards linked to action conditions" case above as one can always define an action "Done" whose pre-conditions are only satisfied when the goal conditions are met. We note that later results in this chapter will show that a non-constant size conjunctive goal (for example, "stack all the blocks") is not polynomially KWIK-learnable.

- **Rewards based on a linear combination of a small number of fluents**: The reward function in a DBN is often assumed to be composed of a linear combination of the factor values and is solvable efficiently using a linear regression algorithm (Walsh et al., 2009b). With relational fluents, we can treat each fluent that is "on" in a given state as a 1 and the others as 0. If only a small number of these fluents contribute to the reward function, or if only a small number of literals in each action schema's scope contribute to R, then we can learn this mapping using the same linear regression tools. Note that this case also covers the case where a single fluent might need to be true to produce reward, as we will see later in a logistics example where a *done* action produces different reward when executed on an object that is not yet *Finished* (the weighting here is simply -1 on the *Finished* literal).

Other combinations are surely possible, but the main point is that if the reward function is compactly described, efficient learning is still possible. For the rest of this work, we assume that the reward function is provided to the agent ahead of time (which is natural in goal centered domains), so we will generally ignore issues of reward learning as they can be solved with similar techniques to the transition-learning component. We do note that the class of reward functions that is efficiently learnable expands in the next chapter when apprenticeship learning is considered, similarly to the expansion in the learnability of the transition function.

3.2 Example Language and Benchmark Problems

In this section, we introduce a compact RL domain encoding (Stochastic STRIPS with rewards) that can be described using the conceptual terminology outlined above. We also introduce several benchmark problems encoded in this language that will be challenging for different learning scenarios.

3.2.1 Deterministic STRIPS with Rewards

STRIPS domains (Fikes & Nilsson, 1971) are made up of a set of uniquely named[2] objects $\mathcal{O}$, a set of predicates P, and a set of parameterized actions A. A state s is defined as a conjunction of positive literals (predicates with parameters filled by members of $\mathcal{O}$). Actions have pre-conditions (**PRE** in the example tables) that are conjunctions over P with parameters filled by the arguments to the action (see Table 3.2). If the current state does not contain all of the elements of the pre-condition, attempting to invoke the action will result in a reported "failure signal" and the state will not change. Effects in STRIPS are specified by Add (**ADD**) and Delete (**DEL**) lists, which designate what predicates are added and deleted from the world

[2] We deal with some of the issues resulting from overlapping names in Chapter 5.

pickup(X, From): reward $= -1$
PRE: *On*(X, From), *Clear*(X), *EmptyHand*(), *Block*(X)
ADD:*Inhand*(X), *Clear*(From) **DEL:** *Clear*(X), *On*(X, From), *EmptyHand*()

putdown(X, To): reward $= -1$
PRE: *Inhand*(X), *Clear*(To), *Block*(To)
ADD: *On*(X, To), *Clear*(X), *Emptyhand*() **DEL:** *Clear*(To), *Inhand*(X)

putdowntable(X, To): reward $= -1$
PRE: *Inhand*(X), *Table*(To)
ADD: *On*(X, To), *Clear*(X), *Emptyhand*() **DEL:** *Inhand*(X)

Table 3.2: Deterministic Blocks World

state when the action occurs, and again variables in these pre-conditions must be linked to the action parameters.

An example deterministic STRIPS domain is the blocks world example in Table 3.2. Intuitively, the agent has 3 actions involving picking up a block (it can only hold one at a time), and putting the block down on another block (but only if it is at the top of a stack) or on the table (this last action is needed because a table is always clear, unlike a block). All of the actions result in a reward of -1, and it is assumed that some goal configuration (which may be very general, say a stack of at least 3 blocks), indicates the end of an episode and will have a value of 0. The use of rewards with STRIPS domains, while not part of the core language, has precedent in probabilistic planning competitions (Younes et al., 2005).

Throughout this work, we will also use more complex or misleading (to the learner) variants of this simple domain. These include the following mutations.

- Stochastic Blocks World: the operators above are deterministic—each action has only one possible effect if the pre-conditions are satisfied. But, it is easy to create a stochastic version of this environment (using a stochastic STRIPS variant described later) by having each action have a possibility of neither adding or deleting any fluents from the current state. For instance, *pickup* might have the outcome above with probability 0.8 and no effect with probability 0.2. Notice that because the failure signal is unique for when pre-conditions do not hold, this "do nothing" outcome will never be confused with a pre-condition failure.

- Blocks world with *move*(X, From, To) and *moveToTable*(X, From, To) actions. These larger-scope operators, which move a block from on top of block "From" to on top of block (or table) "To" are often used in other encodings of blocks world and are considered also in parts of this thesis. The main drawback to these extended actions is that the pre-condition lists for each action are longer than the encoding above (because the fluents associated with all 3 blocks need to be considered at once), and the larger arity of the actions can be detrimental to the learning and planning algorithms used in this thesis. However, in several cases this encoding is used to illustrate points in this work, in which case we will explicitly mention that these operators are in play.

- Blocks world with "Dummy" versions of *pickup* and *putdown*. This version of the domain uses the same operators as the stochastic case mentioned above, but adds in two new actions *dummyPickup* and *dummyPutdown* which have the same pre-conditions and effects as their counterparts above, but with the probabilities reversed. Again, when this domain is used in examples or experiments, we will explicitly indicate that the "dummy" actions are available.

3.2.2 Stochastic STRIPS with Rewards

Stochastic STRIPS operators generalize the deterministic representation above by considering multiple possible action effects specified by $\langle$ Add, Delete, Prob $\rangle$ tuples as in Table 3.3. Notice this is a specific grounding of the pre-conditional stochastic action schemas $\langle a, c^a, \Omega^a, \Pi^a \rangle$, where c is the pre-conditions, and each Add/Delete tuple is an effect $\omega_i^a \in \Omega^a$ with probability p_i^a drawn from Π^a.

Table 3.3 illustrates a variant of the classic Paint-Polish domain (Minton, 1988) in the Stochastic STRIPS setting. Intuitively, the world is comprised of objects that need to be painted and polished and then marked as finished. But, sometimes the action of painting the object scratches it, which requires more polishing (and perhaps repainting) before it can be finished. The environment also has a *shortcut* action that, with very low probability, paints and polishes an object in one fell swoop, but usually has no effect.

While environments like Blocks World and Paint-Polish demonstrate the effectiveness and power of relational representations, they also showcase opposite difficulties in the action-schema learning problem. In Blocks World, the actions have a fairly large arity (2 to 3 parameters) and the conditions for successful action execution are very strict, and therefore hard to discover, but the effects of actions are fairly straightforward to learn. In contrast, Paint-Polish world has fairly simple conditions, but its stochastic action effects are confusing to learn because often multiple effects can explain the same state transition. Each of these problems is dealt with in this chapter.

paint(X): reward $= -1$
PRE: none
ADD:*Painted*(X) **DEL:** none (0.6)
ADD:*Painted*(X), *Scratched*(X) **DEL:** *UnScratched*(X) (0.3)
ADD: none **DEL:** none (0.1)

polish(X): reward $= -1$
PRE: none
ADD: none **DEL:** *Painted*(X) (0.2)
ADD: *UnScratched*(X) **DEL:** *Scratched*(X) (0.2)
ADD: *Polished*(X), *UnScratched*(X) **DEL:** *Painted*(X), *Scratched*(X) (0.3)
ADD: *Polished*(X) **DEL:** *Painted*(X) (0.2)
ADD: none **DEL:** none (0.1)

shortcut(X): reward $= -1$
PRE: none
ADD: *Painted*(X), *Polished*(X) **DEL:** none (0.05)
ADD: none **DEL:** none (0.95)

done(X): reward $= 0$ if action succeeds and object was previously unfinished, else -1
PRE: *Painted*(X), *Polished*(X), *UnScratched*(X)
ADD: *Finished*(X) **DEL:** none (1.0)

The goal is reached when all the objects are *Finished*.

Table 3.3: Stochastic Paint/Polish World

3.2.3 Conditional STRIPS Operators

Finally, we consider a conditional version of Stochastic STRIPS, conforming to the conditional stochastic action-schema case, where each action has a set of conditions, each of which has its own distribution over the possible effects. In the STRIPS instantiation of this setting, each action has a set of conjunctions (over literals with the same scoping assumption as the preconditions from earlier), and the conjunction that matches the current state determines the distribution over the possible Add/Delete pairs that are invoked. An example of a domain with such conditions appears in Table 3.4. It is a version of the Paint-Polish domain above, but now each object can be either metal or wooden, and objects can be turned into metal objects using the *coatMetal* action. Metal objects have a far lower chance of scratching then their wooden counterparts when the *paint* action is invoked.

3.3 Related Work

Before defining our own learning algorithms, we summarize some other approaches to learning conditions, effects, and effect distributions. For the most part, the major separation between our work and these previous attempts is that our algorithms and analyses guarantee PAC-MDP

paint(X): reward $= -1$
c_1: *Wooden*(X)
ADD:*Painted*(X) **DEL:** none (0.1)
ADD:*Painted*(X), *Scratched*(X) **DEL:** *UnScratched*(X) (0.8)
ADD: none **DEL:** none (0.1)
c_2: *Metal*(X)
ADD:*Painted*(X) **DEL:** none (0.8)
ADD:*Painted*(X), *Scratched*(X) **DEL:** *UnScratched*(X) (0.1)
ADD: none **DEL:** none (0.1)

polish(X): reward $= -1$
c_1: none
ADD: none **DEL:** *Painted*(X) (0.2)
ADD: *UnScratched*(X) **DEL:** *Scratched*(X) (0.2)
ADD: *Polished*(X), *UnScratched*(X) **DEL:** *Painted*(X), *Scratched*(X) (0.3)
ADD: *Polished*(X) **DEL:** *Painted*(X) (0.2)
ADD: none **DEL:** none (0.1)

shortcut(X): reward $= -1$
c_1: none
ADD: *Painted*(X), *Polished*(X) **DEL:** none (0.05)
ADD: none **DEL:** none (0.95)

coatMetal(X): reward $= -1$
c_1: none
ADD: *Metal*(X) **DEL:** *Wooden*(X) (1.0)

The goal is reached when all the objects are *Painted*, *Polished*, and *UnScratched*.

Table 3.4: Metal Paint/Polish World

behavior by performing the model learning under the conditions of the KWIK framework. We also note that most previous work considered only a single representation, while many of our results will be extendible to multiple languages with common structure.

The problem of learning STRIPS operators with either pre-conditions or full conditional effects has been considered in a number of previous works. One of the earliest action-schema learners was EXPO (Gil, 1994), which was given an incomplete STRIPS-like domain description (missing some pre-conditions or effects of actions) with the rest being filled in through experience using operator refinement techniques. This application was more proactive than others in that it would "experiment" on operators, similar to the formalized active learning we present in this chapter. The OBSERVER system (Wang, 1995) also used a STRIPS-style language to represent operators and was trained with a mixture of both raw experience and grounded expert traces. Learning in OBSERVER involved maintaining a version space for the pre-conditions and effects of operators (a technique we modify in our own investigation) and the language used allowed for constants and conditions, among other constructs. The TRAIL system (Benson, 1996) used Inductive Logic Programming (ILP) to distill schemas from raw experience and a teacher. These systems were all deployed in the deterministic setting and did not provide the sample complexity guarantees that our learning agents have under these conditions. However, these early systems did demonstrate the insufficiency of using only "raw experience" to learn unbounded-size conjunctive (pre)-conditions, leading to a reliance on bootstrapping or expert traces. We will make use of a similar stream of experience in the next chapter.

The main body of work on learning stochastic STRIPS operators was due to Pasula et al. (2007), which introduced Noisy Indeterministic Deictic (NID) rules, which are similar to Stochastic STRIPS operators but with the following differences. Their operators were designed for the full conditional case and actually went beyond the standard STRIPS scoping assumptions by allowing for *deictic* references to objects that were not parameters to the action. Such references greatly expand the hypothesis space of potential action behavior and are generally not considered in this thesis, though we do discuss them in Sections 5.8 and 6.2. Another major difference was that NID rules modeled "noise" effects, which covered low probability transitions to any state (random changes not covered by the other effects). Such dynamics are both difficult to efficiently model (as we show in Section 3.5.2) and also somewhat philosophically vexing (since all outcomes in some way could be considered "noise"), so in this work we do not consider such ill-defined effects. Other than these key differences, NID rules are covered by our action-schema terminology. The learning algorithm espoused by this prior work was a heuristic search algorithm that attempted to maximize the log-likelihood of previously collected samples of action outcomes. While this algorithm showed empirical success in simulated domains, it did not provide the theoretical guarantees we seek in the online setting.

The REX algorithm (Lang et al., 2010) used the same NID representation, and actually the same operator learning algorithm, but guided the collection of samples using various measures of uncertainty and an E^3-inspired (Kearns & Singh, 2002) architecture. While this approach is closer to our own and their empirical results were very encouraging, no theoretical sample-complexity results are known for this algorithm.

In recent years, there has been a stronger concentration on learning relational models in the partially observable setting. Under these conditions, literals may be missing from state descriptions even though they are true—resulting in a version of open world semantics. This implicit information complicates both the learning (Shahaf, 2007; Zhuo et al., 2009) and the planning (Hoffmann et al., 2007) problems. For instance, schema learning with synthetic items (Holmes & Isbell, 2005) built deterministic operators for noisy and partially observable domains. The work on Simultaneous Learning and Filtering (SLAF) derived computational bounds for learning action-schema operators in different languages (including STRIPS) and under different sources of partial observability (Shahaf, 2007), but sample complexity was not considered. At the extreme of these approaches is the ARMS and HTN-Learn systems (Yang et al., 2007; Zhuo et al., 2009), which learn action schemas from plan traces that contain only initial states, actions, and goals, with no intermediate state information. In both systems, the data is then translated into a satisfiability problem and a heuristic solver is used to determine the likely operator definitions. Our work for the most part deals only with the fully observable scenario.

Another relation language is the Object Oriented MDPs (OOMDP) (Diuk et al., 2008) model, which mixes attribute-value style fluents with defined STRIPS-style predicates. We make the connection between our work and OOMDPs more explicit in Section 6.1.

3.4 Action Schema Learning Problems

Throughout this chapter, we consider the complexity of learning different portions of the action schemas described earlier. This section lays out some intuitive difficulties in this learning process and defines a number of sub-problems of overall schema learning that we will focus on in different parts of this chapter.

3.4.1 Intuitions on the Difficulties of Learning

In the learning setting, an agent will construct a model of schema dynamics based on its own experience. Such learning can be difficult, even in the relatively benign deterministic setting. Consider trying to update, for a given literal l, its membership in the pre-condition ("Pre" for short here), Add, and Delete lists based on an observed state transition $\langle s, a, s' \rangle$. Unfortunately, a single experience may not be enough to pin down l's role in any of these lists as we see in the *operator update rules* below that outline how to update a.PRE (Rule 1) and the effect lists (Rules 2-5) when an action succeeds.

1. $l \notin s$: *Fail*, $l \in s$: *Succeed*, or $l \in s$: *Fail*. None of these situations on their own imply $l \in a$.Pre or $l \notin a$.Pre.
2. **If** $l \notin s \wedge$ *Succeed* **Then** $l \notin a$.Pre.
3. **If** $l \notin s \wedge l \in s'$ **Then** $l \in a$.Add, $l \notin a$.Delete.
4. **If** $l \in s \wedge l \notin s'$ **Then** $l \in a$.Delete, $l \notin a$.Add.
5. **If** $l \notin s \wedge l \notin s'$ **Then** $l \notin a$.Add. No information as to whether $l \in a$.Delete
6. **If** $l \in s \wedge l \in s'$ **Then** $l \notin a$.Delete. No information as to whether $l \in a$.Add

Further complications arise with parameterized actions when the same object appears multiple times in an action's parameter list. For example, if the learner experiences the action a(**b**, **b**), which adds P(**b**), it is not clear whether P(X) or P(Y) (or both) should be inserted into a(X, Y).Add.

We propose avoiding such ambiguity by learning different *action-versions*, that is, a separate operator is learned for each pattern of matching parameters. For instance, a(**b**,**b**) and a(**b**,**c**) yield a_{11}(X) and a_{12}(X,Y), respectively, where the indexes represent the first occurrence in the parameter list of each unique identifier. The maximum number of versions for an action of arity m corresponds to the m^{th} Bell number (Rota, 1964), which is defined recursively as $B_m = \sum_{k=0}^{m-1} \binom{m-1}{k} B_m$. However, because m is considered a constant in our study, learning the schema for each version independently does not affect the worst-case sample complexity analyses of the algorithms studied. In practice, the number of action versions that are actually encountered is typically small. Also in more practical terms, it is possible to share information between action versions, for instance if $P(\mathrm{X}) \in a_{12}(X,Y)$.Add, then it can be inferred that $P(\mathrm{X}) \in a_{11}(X)$.Add. However, such reasoning is beyond the scope of this thesis and is not necessary for the theoretical contributions we have made. We assume throughout the rest of this work (except where explicitly considered in Chapter 5) that actions with the same object in multiple parameter locations are illegal.

Moving to the stochastic setting adds another wrinkle into the learning problem beyond the difficulties mentioned above. In the stochastic setting, effect probabilities cannot be learned in this domain simply by counting the number of occurrences of each Add/Delete tuple because in some states there is ambiguity as to what effect actually occurred. For instance, in Paint-Polish World (Table 3.3) when an object o_1 is already scratched but not painted, if one executes the $Paint(o_1)$ action, and the result is that the object is now scratched and painted, one cannot tell which of the first two possible effects occurred, though we know ω_3 (the "nothing changes" outcome) did not occur because o_1 is now painted.

Finally, when we introduce conditional effects the difficulty can increase because there is no longer a unique "failure" signal of whether a single conjunction was satisfied for a given action or not. Instead, the agent must infer which of the (currently being learned) conditions might have occurred based on the perceived effect (which is complicated by the problems of learning effects and distributions mentioned above).

3.4.2 Learning Sub-Problems

Just as action schemas have three main parameters (C^a, Ω^a, and Π^a), learning their dynamics can also be decomposed into three parts (as illustrated in Figure 3.1): learning the conditions, learning the effects, and learning the effect distributions. In this chapter, we will consider both the complete problem and several combinations of sub-problems assuming the other schema components are given. We will investigate each of these problems using the KWIK framework for model learning, which allows for very small inaccuracies (ϵ) in the learned model. As such, it will be helpful to have a notion of an ϵ-accurate *predictive* model of action schema dynamics, which we define here for the conditional case (the most specific of those presented earlier).

Definition 12. *An ϵ-accurate predictive model of an action schema $\mathcal{A}$ for action a in conditional domain D is any model $\widehat{M}$ (perhaps itself an action schema) that, when given a state s that is valid in a domain instance of D and composed of ground literals, can make a prediction over the probability of next states $Pr[S']$ such that $||Pr[\eta(S,\Omega_i^a)] - Pr[S'|\widehat{M}]|| \leq \epsilon$, where Ω_i^a is the*

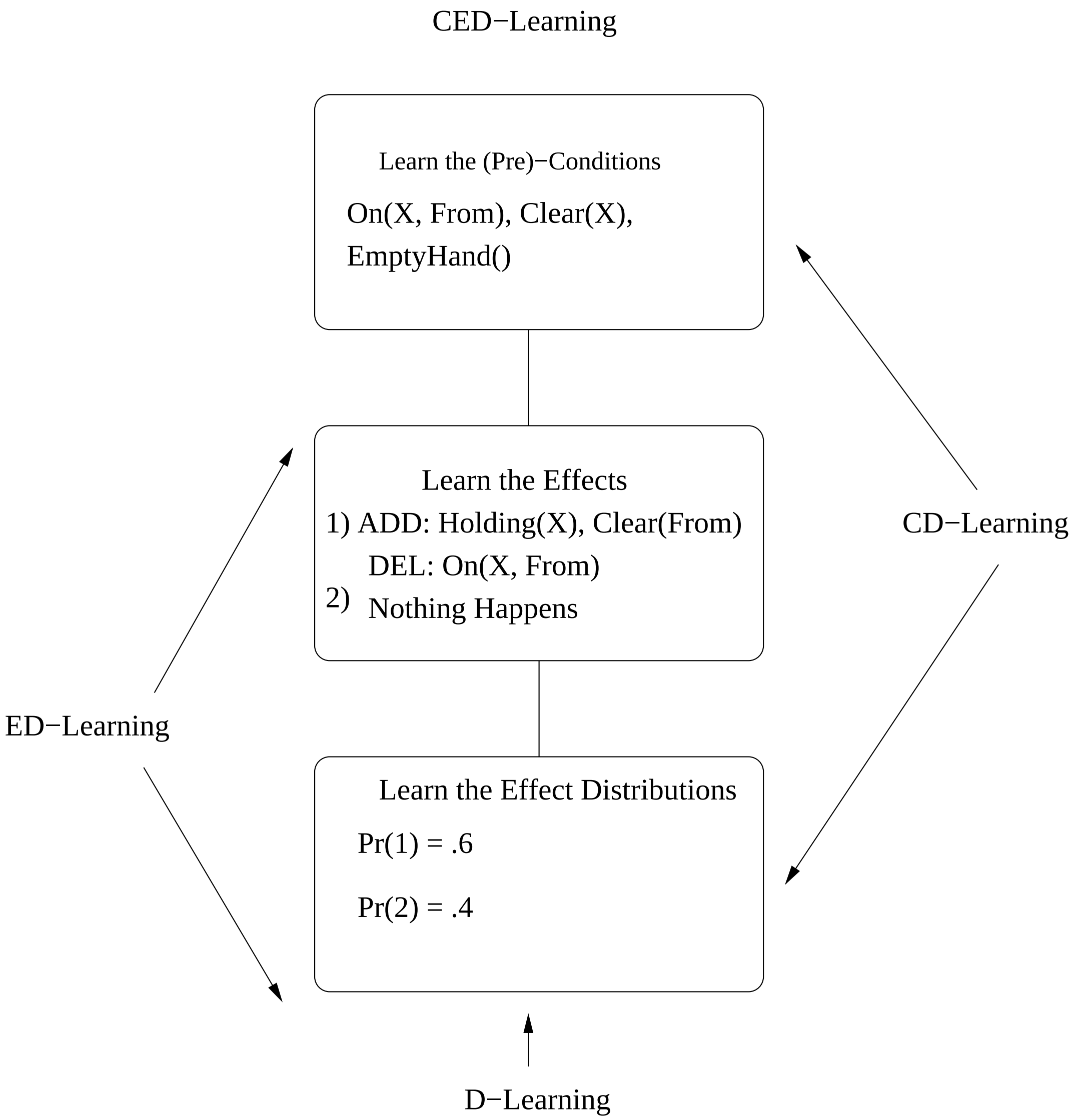


Figure 3.1: The CED-Learning problem and the decompositions considered in this work.

effect set associated with condition c_i such that $c_i[s] = true$ and η maps a state and effect set to a vector of next states, and $||\cdot||$ is an $\mathcal{L}_1$ norm over the next-state probability vector.

Intuitively, the only inaccuracy that is allowed is very small inaccuracies in the probability distributions on next states. Typically (though not necessarily), algorithms that satisfy this criterion will need to correctly model the conditions and effects (with small inaccuracies in the probabilities) in the learned model. Achieving such an ϵ-accurate model with high probability and a polynomial number of $\perp$ predictions will be sufficient for guaranteeing polynomial sample complexity in the KWIK framework, thus guaranteeing us the PAC-MDP behavior of the corresponding RL agents in these relational domains. More formally, the complete (Condition, Effects, and Distributions) learning problem is defined as [3]:

Definition 13. *The Condition, Effect, and Distribution (CED) Learning Problem is, given the set of literals L, actions A, reward function R, terminal (goal) states S_G, accuracy parameters ϵ and δ, and the ability to execute actions in the environment: for all actions $a \in A$, with probability $(1 - \delta)$ output an ϵ-accurate predictive model.*

In this chapter, we will analyze this learning problem and several sub-problems, using the KWIK framework to analyze the model-learning component and PAC-MDP to characterize the corresponding agent behavior. When the meaning is clear, we will refer to both the model learning and behavioral aspects as simply the CED-Learning problem. The sub-problems we consider in addition to the full problem are listed below. Each of the sub-problems, in addition to helping us solve the larger problem, has applicability in realistic scenarios where background information (like the possible effects but not their probabilities) are known. Our first sub-problem entails learning stochastic effects given the conditions (or in the unconditional case):

Definition 14. *The Effect and Distribution (ED) Learning Problem has all the inputs of the CED-Learning problem as well as the true set of conditions C^a for each action a. The agent must then, for all actions $a \in A$, with probability $(1 - \delta)$ output an ϵ-accurate predictive model.*

Decomposing this problem even further, we can consider an even smaller problem where the conditions and effects of the schemas are given, but the distributions of these effects must be learned. This problem can also be thought of as learning full stochastic operators given a traditional non-deterministic planning problem description (such as those considered in Kuter et al. (2008)).

[3] We provide the formal definitions here in the most general case (Conditional Stochastic Action Schemas) as the definitions are easily relaxed to the more restricted cases.

Definition 15. *The Distribution (D) Learning Problem has all the inputs of the ED-Learning problem as well as the true set of conditional effects* $\mathbf{\Omega}^a$ *for each action* a*. The agent must then, for all actions* $a \in A$*, with probability* $(1 - \delta)$ *output an* ϵ*-accurate predictive model.*

We also consider a sub-problem of interest to the robotics community where the effects themselves are given, but the conditions and the effect distributions must be learned. This occurs, for instance, when one has a robot that is known to sometimes move to its intended location, but sometimes slips to either side, and one doesn't know what governs this conditional behavior or what the probabilities are under different conditions.

Definition 16. *The Condition and Distribution (CD) Learning Problem has all the inputs of the CED-Learning problem, as well as the true set of conditional effects* $\mathbf{\Omega}^a$ *(but not the conditions themselves) for each action* a*. The agent must then, for all actions* $a \in A$*, with probability* $(1 - \delta)$ *output an* ϵ*-accurate predictive model.*

Both sub-problems in CD-Learning, finding the correct conditions and effect distributions, are non-trivial to solve. We now begin our investigation with the distribution-learning portion: given the effects and conditions for an action-schema, can we even learn the distribution over the effects in the online setting?

3.5 Learning Effect Distributions

In this section, we present a solution to the distribution learning problem (D-Learning) for relational action schemas. To concentrate on this particular sub-problem, we assume the conditions C^a and the associated effects $\mathbf{\Omega^a}$ are known. We will relax these assumptions in later sections when we consider larger sub-problems (like CD-Learning or ED-Learning). Our solution to the D-Learning problem (KWIK-LR) presented here will serve as a building block for solutions to the more complicated learning problems in the stochastic setting. Also, through this comparatively easier sub-problem, we will introduce concepts and strategies that will reappear throughout our investigation of online schema learning. These include: (1) the use of KWIK learners in action-schema learning, (2) the use and design of optimistic model interpretations in action schema domains, and (3) constructing full agent algorithms from KWIK learners, a planner, and these optimistic interpretations.

Below, we formalize the difficulties stemming from *effect ambiguity* and describe a naïve algorithm for dealing with this situation. We then show how a linear regression algorithms, and specifically a KWIK linear regression algorithm, can be used to solve this problem more efficiently than the naïve solution, and show how to integrate such an algorithm into a full learning agent by using different optimistic interpretations of the KWIK learner's predictions.

3.5.1 Effect Ambiguity and the Partition Algorithm

Learning the probabilities in D-Learning is non-trivial because for a given state-action pair $(s,\ a)$, the effects are partitioned into equivalence classes $E(s,a) = \{\{\omega_i, \omega_j, \omega_k\}, \{\omega_l, \omega_m\}, ...\}$ where each $e \in E(s,a)$ contains effects that are identical given state s. For instance, consider the *paint* action from the Paint-Polish domain (Table 3.3). The possible effects listed in the operator are that either the object is painted (ω_1), the object is painted and scratched (ω_2), or nothing changes (ω_3). But, suppose this action was applied to an object that was was already scratched, but not painted. In that case, there are only 2 possible outcomes of this action: either the object will be painted and scratched, or still just scratched (and unpainted). The reason is that the first two effects belong to the same equivalence class ($E(s,a) = \{\{\omega_1, \omega_2\}, \{\omega_3\}\}$), because applying either of the first two pairs of Add/Delete lists results in the same next state ($\eta(s,\omega_1) = \eta(s,\omega_2)$).

Consider trying to update the probabilities of the three effects after observing a transition where the formerly "just scratched" object becomes painted and scratched. The standard "counting" method (storing the frequency of each effect divided by the total number of observations of this action) for learning probabilities cannot be used in such a setting because it is unclear which effect's (ω_1 or ω_2) count we should increment. But, notice this sample is not entirely uninformative. While we cannot tell which of the first two effects occurred, we *do* know that the third effect did not occur (since the object came back painted), so *something* can be learned from this experience.

Based on this intuition, we can construct a naïve *Partition* algorithm as shown in Algorithm 7.[4] Intuitively, the algorithm simply maintains separate counts (ρ) of outcomes for every possible pairing of equivalence classes. For instance, in the *paint* example, the experience used to predict the probability of the initially scratched object being painted or scratched would be calculated only from samples where the first two effects mapped to the same state (but the third effect was different).

[4] This algorithm (and many others in this chapter) is presented here for the pre-conditional case. In the conditional case, one simply needs to consider the partitioning given an action and a condition rather than simply based on the action.

Algorithm 7 Partition

```
Input: $\epsilon, \delta, \eta : S, \Omega \mapsto S$
Output: On each step, a prediction of the probabilities of each next state.
Calculate $M = Poly(\epsilon, \delta, \Omega)$, the number of samples needed to accurately make a prediction for a class
for each $s$, $a$ do
  Let $E = E(s,a)$ //set of equivalence classes (a partitioning of effects)
  if $E$ has never been seen before then
    $\forall e \in E,\ \rho[E][e] = 0$ //Initialize counts ($\rho$)
  end if
  if $\sum_{e \in E} \rho[E][e] > M$ then
    $\widehat{Y} = \{\widehat{y}_i\} = \rho[E][e] / \sum_e \rho[E][e], \forall e \in E$
  else
    $\widehat{Y} = \perp$
  end if
  if $\widehat{Y} = \perp$ then
    Observe $s'$
    for $e \in E$ do
      if $\eta(s, \omega_e) = s'$ where $\omega_e$ is any effect in $e$ then
        $\rho[E][e] \mathrel{+}= 1$
      end if
    end for
  end if
end for
```

Proposition 1. *The Partition algorithm has a KWIK bound of* $O(A \sum_{i=2}^{\Omega} (S_2(\Omega, i)(\frac{i}{\epsilon^2} + \log(\frac{i}{\delta})))$*, where* S_2 *represents the Stirling number of the second kind (Sharp, 1968) (the number of* i*-sized partitions of* Ω*).*

Proof. (sketch) The proposition holds from an application of the "dice learning" KWIK bound (see Table 2.1 and Li (2009) for more information) on learning a multinomial distribution. Specifically, this bound is derived by setting $M = O(\frac{i}{\epsilon^2} \log(\frac{i}{\delta}))$ when there are i possible outcomes in a partition. The dice-learning bound itself is derived from applications of Hoeffding's inequality and a union bound. We note that the case where $i = 1$ is not considered because all the effects are in one partition (with outcome probability 1.0) and this bound is loose because it uses the dice learning bound when $i = 2$, a case that can be more tightly bounded using Hoeffding's inequality directly. □

There are several drawbacks to this approach. First, the KWIK bound is exponential in the number of effects, and while this number is often small (even $O(1)$), even in Paint-Polish world we see it as high as 5 (for the polish action), which makes the algorithm practically infeasible, though often not all equivalence-class partitions are reachable. Secondly, the Partition algorithm does not make use of the structure of the probability simplex associated with the domain dynamics. Specifically, since the probability of any equivalence class is equal to the sum of the probabilities of the effects it contains, the probabilities of many partitions can be derived from known probabilities in other partitions. For example, if there are four effects and the learner has established that $Pr[\omega_1] = 0.1$ and $Pr[\omega_2] = 0.1$, then for the equivalence class $\{\{\omega_1, \omega_2\}, \{\omega_3, \omega_4\}\}$, the learner can predict 0.2 and 0.8 as the correct probabilities even though the individual probabilities of ω_3 and ω_4 are unknown. In the next few sections, we show how to use an algorithm (KWIK-LR) that makes use of all this structure and has truly polynomial sample complexity (even when Ω is of polynomial size in the domain literals). We will use this superior algorithm in concert with optimistic interpretations (Section 3.5.4) and a full online agent algorithm (Section 3.5.5) to solve the D-Learning problem.

3.5.2 Viewing D-Learning as Linear Regression

An alternative view of the D-Learning problem is to treat the learning of the individual effect probabilities ($p_i^a \in \Pi^a$) as weights in a linear regression (LR) problem. In this section, we formalize this view and the construction of inputs and interpretation of outputs from a "black box" linear regression algorithm under these conditions. The section following this one will show how to replace this generic LR algorithm with a KWIK algorithm with polynomial sample complexity. Together, these two pieces give us a more efficient alternative to the naïve and inefficient Partition algorithm above.

We begin by reviewing some standard online linear regression terminology, using standard MATLAB-style notation ("," separating column entries, ";" for rows). We will concentrate on a basic linear regression problem without measures of uncertainty and without any regularization, features that are introduced in the next section. Let $X := \{\vec{x} \in \Re^n \mid \|\vec{x}\| \leq 1\}$, and let $f : X \rightarrow \Re$ be a linear function with weights $\theta^* \in \Re^n$, $\|\theta^*\| \leq M$, i.e. $f(\vec{x}) := \vec{x}^T\theta^*$. Fix a timestep t. For each $i \in \{1, \ldots, t\}$, denote the stored samples by $\vec{x}_i$, their (unknown) expected values by $y_i := \vec{x}_t^T\theta^*$, and their observed values by $z_i := \vec{x}_i^T\theta^* + \eta_i$, where the noise η_i is assumed to form a martingale, i.e., $E(\eta_i|\eta_1, \ldots, \eta_{i-1}) = 0$, and bounded: $|\eta_i| \leq S$. Define the matrix $D_t := [\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_t]^T \in \Re^{t \times n}$ and vectors $\vec{y}_t := [y_1; \ldots; y_t] \in \Re^t$ and $\vec{z}_t := [z_1; \ldots; z_t] \in \Re^t$. The *online linear regression problem* is simply for every new query point $\vec{x}$ that arrives, output $y_t = \vec{x}_t^T\theta^*$ When D_t is of full rank and has sufficient coverage to make accurate predictions we can calculate this prediction by $\widehat{y} = \vec{x}^T\theta$, where θ is the least-squares solution to the system.

With this machinery in hand, we can now craft a creative abstract solution to the D-Learning problem. The “trick” here is to represent each piece of experience in the data matrix with a series of binary indicator vectors, one for each equivalence class, and then mark the equivalence class that actually occurred in the output matrix with a 1. Formally, the algorithm for constructing these entries in D_t and $\vec{y_t}$ is presented in Algorithm 8. In words: each equivalence class contains one or more effects ($\omega_i, \omega_j ... \in \Omega^a$), and an indicator vector $\vec{x}$ is created where $x_i = 1$ if $\omega_i \in e_t$ (Line 5), otherwise $x_i = 0$. For instance, in learning the probabilities associated with the *paint* action, if the object was already scratched and then came back painted and scratched (effects ω_1 and ω_2 are confusable, but not ω_3), the new data vectors would be $[1, 1, 0]$ and $[0, 0, 1]$ but if the action was taken on an object that was not scratched and not painted, there could be no confusion so in that case 3 vectors would be added: $[1, 0, 0], [0, 1, 0]$, and $[0, 0, 1]$. Note that for a given state each equivalence class in $E(s, a)$ induces a unique (and disjoint) $\vec{x}$. *Each* of these is used to update the LR learner (Line 6), with an output (y) of 1 associated with the $\vec{x}$ that actually happened. The other equivalence class vectors that did not occur are still introduced into the data matrix, but the corresponding output entry in y_t is set to 0.

Algorithm 8 Transition Probability Learner for action a with LR

```
1: **Input:** set of effects $\Omega^a$ and a linear regression algorithm $LR^a$
2: **for** each current state $s_t$ **do**
3:    Take action $a$ and observe $e_t \in E(s_t, a)$
4:    **for** each equivalence class $e \in E(s_t, a)$ **do**
5:       Construct $\vec{x}$ where $x_j = 1$ if $j \in e$, else 0
6:       Update $LR^a$ with $\vec{x}$ and $y = 1$ if $e = e_t$, else $y = 0$
7:    **end for**
8: **end for**
```

Intuitively, linear regression is the correct tool for this task because each sample gives us a noisy indicator of the sum of the individual effect probabilities in the same equivalence class. For instance, consider 10 samples of the *paint* action all of which occurred under the partitioning $E(s, a) = \{\{\omega_1, \omega_2\}, \{\omega_3\}\}$, and 9 times out of the 10 the first (ambiguous) effect occurred. We cannot tell anything about $Pr(\omega_1)$ or $Pr(\omega_2)$ individually, but we do know their sum is close to 0.9 (and that $Pr[\omega_3] \sim 0.1$). At the highest level, we know that $\sum_{i=1}^{|\Omega|} Pr[\omega_i] = 1$. The *partition* algorithm from earlier inherently ignores each of these summation constraints.

The solution to such a linear system is a weight vector containing the frequencies of each individual effect with respect to y. For instance, in the *paint* case, any two positive numbers that sum to 0.9 are valid weights for θ_1 and θ_2 for now, but as samples are added where these effects are in different equivalence classes, the weights must respect the proportion of 1’s associated with each individual effect while not violating the summation constraint. Ultimately, the solution to such a linear system is exactly the unknown probability vector. In the case where all effects are unambiguous, the collected examples for a data matrix where each row of D_t contains only one 1 (essentially on each step of experience, an identity matrix of size $|\Omega|$ is appended to the data matrix), and the single 1 added to z_t appears in the row of the effect that actually happened. In that case, the least-squares solution garners the correct probabilities, since each weight θ_i must be proportional to the fraction of 1's in z in rows where effect ω_i has a 1. Finally, we note that this linear regression technique would be inappropriate in a schema that allowed for "noise" or "miscellaneous" outcomes such as those used by Pasula et al. (2007). In that work, conditional stochastic STRIPS operators were allowed to have a "noise" outcome where any possible change to the state space could occur. While theirs is a useful technique for capturing miscellaneous real-world dynamics, such an effect would always be a possible explanation for whatever outcome occurs, hence its associated column in the data matrix would be all 1's (since it is always ambiguous). In such a situation, a heuristic scoring function that tries to minimize the weights on the noise term (as was used in this related work) is likely a better solution. Finally, we note that a situation where "noise" simply referred to all outcomes except the other effects would be amenable to our solution, but could pose other problems when actually learning the effects.

With this intuition and using the algorithm above, we can make a formal connection between potential KWIK bounds for linear regression and KWIK bounds for the D-Learning problem.

Lemma 2. *If linear regression can be KWIK-learned with a sample complexity $B(n, \epsilon, \delta)$, then the D-Learning problem can be solved with a KWIK bound of $O(AB(\Omega, \epsilon, \delta))$, where $\Omega = \max_a \Omega^a$ and A is the number of actions. In the conditional case, the bound is $O(ACB(\Omega, \epsilon, \delta/C))$ where $C = \max_a C^a$ (the most number of conditions for an action) and $\Omega = \max_{ca} \Omega_c^a$.*

Proof. We concentrate here on the case where the KWIK learner makes a non-$\perp$ prediction as we have already assumed the number of $\perp$'s can be bounded by B above (we construct such an algorithm in the next section). In that case, by the properties of the KWIK algorithm, we know that the solution to any linear system we will create will (with probability $1-\delta$), be ϵ-accurate. Specifically, for an input x where a prediction $\widehat{y}$ is made, $|\widehat{y} - y| \leq \epsilon$.

The proof utilizes the fact that a generic LR algorithm produces $\widehat{\theta}$ such that $\widehat{\theta} = D^{\dagger}z$, where † indicates a pseudoinverse. First, we consider the case where each effect is completely unambiguous in all situations *or* any ambiguous effects do not appear separated anywhere (that is if ω_1 and ω_2 are ambiguous, nowhere do they appear alone or with other effects). In that latter case we can treat them as a single effect, so each $|e_i(s,a)| = 1$. In this unambiguous situation, each row of D is a unit vector $\vec{e}_i^T$. At timestep t, the number of rows in this unregularized form is $\Omega \times t$, but we will compact it in the next section to remove the factor of t. z contains $\Omega * t - t$ 0's and t 1's. Suppose t is large enough such that the proportion of 1's in z for each effect ω_i is within some ϵ_1 of $p_i \in \Pi$ so that an ϵ-accurate prediction can be made for any x_t. For instance, $\epsilon_1 = \epsilon/\Omega$ would satisfy this requirement. Finding a least-squares solution for this system means calculating $\widehat{\theta}$ such that:

$$\begin{aligned}\widehat{\theta} &= D^{\dagger} z \\ &= (D^T D)^{-1} D^T z \\ &= I * (1/t) c \\ &= \Pi \pm \epsilon_1\end{aligned}$$

where I is an identity matrix, and c_i is a count of the number of times a 1 appeared in z when $x_i = 1$. The last two steps comes from realizing that $D^T D$ is simply a diagonal matrix with entries of t (then inverted) and that $D^T z$ is the count of times a 1 appeared in z simultaneously with a 1 for each x_i. That is, each element of $\widehat{\theta}$ becomes the frequency of 1's for that effect divided by the total number of timesteps, which will be within some ϵ_1 of p_j to make ϵ-accurate predictions for the given input. Notice that if the LR algorithm were making predictions for an effect probability before these frequencies (with high probability) reached ϵ_1-closeness to the true Π, then in the limit (where the frequencies certainly approach Π), it would make inaccurate predictions on the weight vector, and thus would not be a KWIK algorithm. Also, note that any query for an equivalence class containing more than one effect (such as $\langle \omega_1, \omega_3 \rangle$) can be answered from this data matrix by just summing the corresponding $\widehat{\theta}_i$'s.

Now, let's fix the effects and consider the case where each row of the data matrix D contains at most k 1's (up to k effects are aliased), for $1 \leq k \leq |\Omega|$. Again, it is possible that the corresponding KWIK algorithm may not be able to make accurate predictions for all inputs, so we focus here on the case where the algorithm reports an ϵ-accurate output based on its $\widehat{\theta}$ weights. The question we must answer is whether the rows with the larger equivalency classes could cause the least-squares solution to deviate greatly from the true probabilities. But, notice that the frequency of a 1 in z corresponding to such a row aliasing $\omega_i ... \omega_j$ is simply $p_i + ... + p_j \pm \epsilon$ (at most). Therefore, setting each $\widehat{\theta}_i$ to p_i (as was the case for the unambiguous data rows) will satisfy the KWIK error tolerance for these entries as well. Notice that in some cases (where not all the singleton probabilities are recoverable because of ambiguity) there will be multiple solutions (not within the ϵ-threshold of one another) to certain queries x_t because the linear system does not yet have a unique solution, *one of which* is the true probabilities for these effects, but (if a prediction is made for vector $\vec{x}$ of k 1's), *all of which* sum to the total of the k true probabilities. For instance, in the extreme case, if all effects are always ambiguous, any probability vector summing to 1 is a valid $\widehat{\theta}$. Notice that if the true probability vector (again with some ϵ-tolerance) were *not* a least-squares solution to such a system, then there would be a different vector of weights $\widehat{\theta}$ such that $\sum_{\text{unique } d_i \in D} (d_i \cdot \widehat{\theta}_i - f_i)^2$ was minimized where f_i was the frequency of 1's in z for rows identical to d_i. But, if the system is actually making non-$\perp$ predictions, those frequencies must be in line with the true probabilities, so with high probability it cannot happen. In cases where the different possible solutions differ by more than ϵ, it will be incumbent on the KWIK algorithm to report $\perp$. □

We have thus established that when a KWIK linear regression algorithm makes ϵ-accurate predictions with high probability, it can be used to predict next state distributions based on effects, thereby solving the D-Learning problem. We now provide a detailed description of a KWIK linear regression algorithm that fulfills the requirements of the lemma with only a polynomial (in the number of effects) number of $\perp$ predictions, and also makes use of regularization.

3.5.3 Regularized KWIK-LR

Here, we describe a KWIK Linear Regression algorithm (KWIK-LR) that uses a compact representation of the data matrix (size Ω^2), and therefore makes the algorithm computationally tractable (as opposed to approaches that continue to increase the size of the data matrix with each instance). KWIK-LR, like all KWIK algorithms, admits when it is too uncertain to make a prediction, giving us a basis for performing exploration, which would not be the case with standard linear regression. Thus, an RL agent using KWIK-LR can learn the probabilities online and maintain the PAC-MDP guarantee.

Directly solving the linear system described above in the online case is problematic because: (1) if D_t is rank-deficient the least-squares solution may not be unique and (2) even if we have a solution, we have no information on its confidence. We can avoid problem (1) by using regularization, which certainly distorts the solution, but this gives us a measure of confidence: if the distortion is large, the predictor should have low confidence, and output $\perp$. On the other hand, if the distortion is low, then $\vec{z}_t$ and $\vec{y}_t$ must be very similar.

Let $A_t := [I; D_t^T]^T$. The solution of the system $A_t\theta = [(\theta^*)^T; \vec{y}_t^T]^T$ is unique, and equal to θ^*. However, the right-hand side of this system is unknown, so we use the approximate system $A_t\theta = [\vec{0}^T; \vec{z}_t^T]^T$, which has a solution $\widehat{\theta} = (A_t^T A_t)^{-1} A_t^T [\vec{0}^T; \vec{z}_t^T]^T$. Define $Q_t := (A_t^T A_t)^{-1}$. If $\|Q_t\vec{x}\|$ (which is directly proportional to the error $||\widehat{y} - y||$) is larger than a suitable threshold α_0, our algorithm will output $\perp$, otherwise it outputs $\widehat{y}$, which is guaranteed to be an accurate (with high probability) prediction. The intuition behind this choice is that in areas of the input space where high error might be encountered because of insufficient data, the agent requests a sample by predicting $\perp$, and this will help reduce the error in that subspace.

Algorithm 9 describes our method for KWIK-learning (as defined in Section 2.2.2) a linear model. Notice it avoids the problem of storing A_t and $\vec{z}_t$, which grow without bound as $t \to \infty$. The quantities $Q_t = (A_t^T A_t)^{-1}$ and $\vec{w}_t = A_t^T [\vec{0}^T, \vec{z}_t^T]^T$ are sufficient for calculating the predictions, and can be updated incrementally. The algorithm is KWIK by the following theorem (proof is presented in Walsh et al. (2009b) and the associated Technical Report):

Theorem 2. *Suppose that the observation noise is zero-mean and bounded: for all t, $E[\eta_t] = 0$ and $|\eta_t| \leq S$. Let $\delta > 0$ and $\epsilon > 0$. If Algorithm 9 is executed with $\alpha_0 := \min\left\{c_1\frac{\epsilon^2}{n}, c_2\frac{\epsilon^2}{\log\frac{n}{\delta}}, \frac{\epsilon}{2M}\right\}$, for bounded noise M and with suitable constants c_1 and c_2, then the number of $\perp$'s will be $B_{LR}(n, \epsilon, \delta) =$*

$$O\left(\max\left\{\frac{n^3}{\epsilon^4}, \frac{n\log^2\frac{n}{\delta}}{\epsilon^4}\right\}\right) \tag{3.1}$$

and with probability at least $1 - \delta$, for each sample $\vec{x}_t$ for which a prediction $\widehat{y}_t$ is made, $|\widehat{y}_t - f(\vec{x}_t)| \leq \epsilon$ holds.

This result is on par with the sample complexity of previous work on KWIK online linear regression (Strehl & Littman, 2007), and requires $\Theta(n^2)$ operations per timestep t, in contrast to their approach, which stored (and operated on) a matrix with as many rows as samples that have been collected. With this bound on the number of $\perp$ produced by KWIK-LR, we can state the following theorem.

Theorem 3. *The D-Learning problem can be solved with a KWIK bound of $O(AB_{LR}(\Omega, \epsilon, \delta/A)$ in the pre-conditional case and $O(CAB_{LR}(\Omega, \epsilon, \delta/CA))$ in the conditional case where B_{LR} is*

Algorithm 9 KWIK-LR

```
**Input:** $\alpha_0$
**initialize:** $t := 0$, $m := 0$, $Q := I$, $\vec{w} := \vec{0}$
**repeat**
    observe $\vec{x}_t$
    **if** $\|Q\vec{x}_t\| < \alpha_0$ **then**
        predict $\widehat{y}_t = \vec{x}_t^T Q\vec{w}$ *//known state*
    **else**
        predict $\widehat{y}_t = \perp$ *//unknown state*
    **end if**
    observe $z_t$
    $Q := Q - \frac{(Q\vec{x}_t)(Q\vec{x}_t)^T}{1+\vec{x}_t^T Q\vec{x}_t}$, $\vec{w} := \vec{w} + \vec{x}_t z_t$
    $t := t + 1$
**until** there are no more samples
```

defined in Equation 3.1 and C and Ω are defined as in Lemma 2.

Proof. (sketch) The theorem holds based on Lemma 2 and the Theorem 2 above, which respectively relate the bound on the D-Learning problem to the bound of KWIK-LR and explicitly state a polynomial KWIK-LR bound. □

3.5.4 Optimistic Interpretations in D-Learning

Using the KWIK learner above, we can KWIK-solve the D-Learning problem and create an associated PAC-MDP agent in an environment where the effect distributions are unknown using the KWIK-Rmax template in algorithm 6. This algorithm will insert an $R_{\max}$ state transition anywhere the current data matrix cannot be used to make an accurate probability prediction. But, this optimistic interpretation of the currently learned model is in some ways *too* optimistic. Consider the case where an agent is in stochastic blocks world and has a goal of stacking 4 blocks. The agent may have learned the probabilities for the *pickup* and *putdown* actions, but perhaps may not have learned the probabilities for the *putdownTable* action. While the probabilities on the effects (which are either to put the block on the table or do nothing) are unknown, no good can come from using this action in this situation, no matter what the real probabilities are. But, using the $R_{\max}$ heuristic, the agent will essentially think that its goal can be reached by just executing this action! The problem here is that the background knowledge, the known pre-conditions and effects, are not being properly leveraged in the optimistic interpretation. While this behavior still yields polynomial sample complexity bounds in the worst case, there is a much better solution in the best case that leverages the background knowledge in the problem without sacrificing the worst case bounds.

Our solution is the *Known-Edge Value Iteration* algorithm (Algorithm 10). The algorithm assumes that the model is represented by means of KWIK learners for each action, though other architectures are possible. The outer shell of the algorithm remains very much like traditional Value Iteration, but inside the main loop over states, the transition function is computed for every possible equivalence class (e) induced by the state-action on every iteration. For equivalence classes (outcomes) where the model reports that the transition probabilities are known, the reported values are used in conjunction with the effect represented by that class (ω_e on Line 11). On the other hand, for state-action pairs that have effects with known probabilities $K = \{\omega_i, \omega_j ...\}$ and unknown probabilities $U = \{\omega_k, \omega_l ...\}$, the effect in U that leads to the highest value next state is considered to have probability $1 - \sum_{\omega_i \in K} P(\omega_i)$ (Lines 15 through 16). Intuitively, at each iteration, for all the transition probabilities that are unknown, the "edge" (effect) leading to the next state with the highest value (based on the current estimate of the value function) is given all the probability mass not swallowed up by the known transitions. This change is designed to force Value Iteration to be "Pangloss", that is it considers the most optimistic of all models consistent with what has been learned *and* the known background information.

This "shifting of probabilities" during Value Iteration is a technique used in the model-based RL algorithm MBIE (Strehl & Littman, 2005). Like MBIE, Known-Edge value-iteration converges to an optimistic value function, which facilitates efficient exploration. That is, $\widehat{V}(s) \geq V(s) - \epsilon$ for all states s where $\widehat{V}$ is the calculated value function and V is the environment's true value function. To see why this statement is true, consider a state s where the probabilities have been shifted. In this case, there is some subset of truly reachable states S' such that $V(s) = \max_a R(s,a) + \gamma \sum_{s' \in S'} T(s,a,s')V(s')$. The calculated value function $\widehat{V}$ performs the summation over the same set of reachable states, but one state $\bar{s}' \in S'$ has a higher weight (larger $T(s,a,s')$). However, by definition, $\bar{s}$ has the highest $V(s')$ of any of the candidates. So, for all states, $\widehat{V}$ is optimistic. In practice, the behavior resulting from this algorithm can lead to better decisions than the Rmax heuristic, which ignores the background information (the known effects). For example, in the aforementioned blocks world example, where an $R_{\max}$ style planner attempted to learn a clearly useless *putdownOnTable* action, the Known-Edge variant will eschew this choice because the best outcome possible (in the stacking case it is the "nothing happens" outcome) will end up with probability 1.0 but will have a value worse than the *putDown* action, so it will never be explored.

Algorithm 10 Known-Edge Value Iteration

```
$\forall s, V(s) = 0$
$\delta = \infty$
while $\delta > \epsilon$ do
    $\delta = 0$
    for $s \in S$ do
        $oldV = V(s)$
        for action $a \in A$ do
            $knownProb = 0.0$
            $\forall e \in E(s,a)$, $T(s,a,\eta(s,\omega_e)) = 0.0$ //Get probabilities for known edges
            for equivalence class $e \in E(s,a)$ where $\Pi(e) \neq \perp$ do
                $T(s,a,\eta(s,\omega_e)) = \Pi(e)$
                $knownProb = knownProb + \Pi(e)$
            end for
            //Shift the unknown probability mass to an optimistic (but possible by $\Omega^a$) outcome.
            Let $e_{opt} = e \in E(s,a)$ s.t. $\Pi(e) = \perp$ and $V(\eta(s,\omega_e)) \geq V(\eta(s,\omega_{e'}))\forall\omega_{e'}\Pi(e') = \perp$
            $T(s,a,\eta(s,\omega_{e_{opt}})$ += $1.0 - knownProb$
        end for
        $V(s) = \max_a R(s,a) + \gamma \sum_{\omega \in \Omega^a} T(s,a,\eta(s,\omega))V(\eta(s,\omega))$
        $\pi(s) = \text{argmax}_a R(s,a) + \gamma \sum_{\omega \in \Omega^a} T(s,a,\eta(s,\omega))V(\eta(s,\omega))$
        if $||oldV - V(s)|| > \delta$ then
            $\delta = ||oldV - V(s)||$
        end if
    end for
end while
Return $\pi$
```

We note that this algorithm still enumerates the full state space, which may be necessary for exact planning, and therefore has computation bounds that are exponential in the size of the action-schema representations (particularly exponential in $\mathcal{O}$). While the exponential run time does not affect the sample complexity of the associated learning algorithm below, it can, in practice, introduce intractable computation just like vanilla Value Iteration. The benefit of Known-Edge VI is instead that it encourages more efficient exploration by the learning agent. In Chapter 6, we will introduce some alternative planning algorithms whose runtime does not scale exponentially in the state-space size.

3.5.5 An Efficient Online Agent for D-Learning

We now make use of KWIK-LR and Known-Edge Value Iteration in learning effect probabilities online when an agent is given the full action-operator specifications, except for the probabilities (the online D-Learning problem). In the parlance of the planning community, we are turning a non-deterministic action specification (Kuter et al., 2008) into a stochastic one, with the probability values acquired through learning. However, we add the additional constraint that the learning agent in this setting should be PAC-MDP. Algorithm 11 presents a PAC-MDP algorithm for this exact situation in the pre-conditional case.

Algorithm 11 Schema Transition Probability Learner

```
Input: $S$, $A$ (action schemas, sans $p_i$'s), $R$, $\alpha$
$\forall a \in A$, Instantiate a KWIK-LR learner (Algorithm 9) $LR^a(\alpha)$
for each current state $s_t$ do
    for each $e \in E(s,a)$, $s \in S$ and $a \in A$ do
        Construct $\vec{x}$ where $x_j = 1$ if $j \in e$, else 0.
        $\widehat{\Pi}(e)$ =Prediction of $LR^a(\vec{x})$ // can be $\perp$
    end for
    Perform Known-Edge Value Iteration on $\{S, A, E, \widehat{\Pi}, R\}$, to get greedy policy $\pi$
    Perform action $a_t = \pi(s_t)$, observe $e_t \in E(s_t, a_t)$
    for equivalence classes $e \in E(s_t, a_t)$ do
        Construct $\vec{x}$ where $x_j = 1$ if $j \in e$, else 0.
        Update $\mathcal{L}^{a_t}$ with $\vec{x}$ and $y = 1$ if $e = e_t$, else $y = 0$
    end for
end for
```

The learning portion of the algorithm uses the same basic structure as the generic LR probability learner (Algorithm 8), but with KWIK-LR as the linear regression component, allowing it to identify states that imbue equivalence classes with unknown transition probabilities. It then communicates this awareness of the uncertainty to the Known-Edge Value Iteration algorithm, which encourages exploration via optimism in the face of uncertainty without trying actions in states where knowing their true transition probabilities is not beneficial to the agent. This calculation yields an optimistic action a_t, which will eventually result in experience indicating one of the equivalence classes $e_t \in E(s_t, a_t)$ actually occurred. After that, the appropriate KWIK-LR learner is updated with the experience and the process repeats. Again, planning in the flat state space can require exponential computation time, but *learning* in this algorithm is done in a sample efficient manner, so we can formally claim that it is PAC-MDP, as shown in the following theorem.

Theorem 4. *In the online setting, Algorithm 11 is PAC-MDP, given the true conditions and effects for the action schemas describing the environment's dynamics.*

Proof. (sketch) The proof combines previous results from Theorem 1, Lemma 2, and Theorem 3. Specifically, Theorem 3 establishes the KWIK-learnability of the probability distributions using KWIK-LR, with help from Lemma 2. Together, these results establish a KWIK bound of $B(\epsilon, \delta) = O\left(\max\left\{\frac{\Omega^3}{\epsilon^4}, \frac{\Omega \log^2 \frac{\Omega}{\delta}}{\epsilon^4}\right\}\right)$, per action (with $\Omega = \Omega^a$). Using this linear KWIK bound on learning each probability distribution for the $|A|$ actions (and possibly C conditions for each action), we end up with a total of $O(CAB(\epsilon, \delta/CA))$ $\perp$ predictions throughout the learning

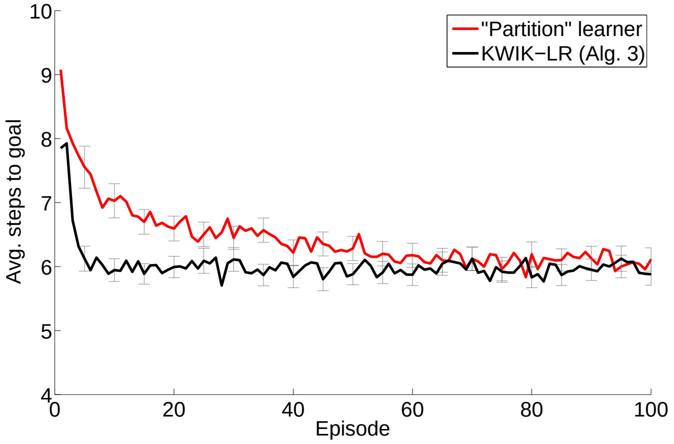


Figure 3.2: D-Learning in Paint-Polish world with 1 object and random start states, averaged over 1000 runs.

process. Theorem 1 establishes that such a KWIK learner using the KWIK-Rmax architecture will be PAC-MDP. The only change we have made there is the introduction of Known-Edge Value Iteration (KEVI). To see why this modification does not change the properties of KWIK-Rmax, first note that KEVI reduces to standard Value Iteration when all the parameters are known. In the case where there are unknown transitions, the KEVI "shifting" of probabilities is the same as that used in the PAC-MDP algorithm MBIE (see Strehl & Littman (2005), which also details the correctness of this shifting technique). The value function induced is optimistic as mentioned earlier, so it satisfies the portion of the KWIK-Rmax proof (optimism) responsible for efficient exploration while leveraging the known effect structure. So, Algorithm 11 given the conditions and effects of the correct action schemas will be PAC-MDP. □

We now present empirical evidence of the efficiency of Algorithm 11 in pre-conditional Stochastic STRIPS. We use the Stochastic STRIPS Paint-Polish world from Table 3.3 as our testbed. This specific instantiation has only one object but with random start states constructed from varying combinations of the *Painted*, *Polished*, and *Scratched* predicates. We empirically test Algorithm 11 against *Partition*. Figure 3.2 shows the results for the Paint/Polish domain averaged over 1000 runs with randomized initial states for each episode (both learners receive the same initial states). We see that Algorithm 11 learns much faster because it is able to share information between equivalence partitions though both algorithms eventually converge to the same (optimal) policy.

3.6 Learning Pre-conditions and Conditions

With our success above in learning effect distributions, we now turn our attention to the CD-Learning problem, where the possible effects $\mathbf{\Omega}$ are given but the distributions over those effects, and the conditions that govern which distribution is active, need to be learned. This formulation covers the likely real-world scenario where the effects of an agent's actions (walking, grasping, etc.) are well known, but how exactly those outcomes are affected by environmental conditions are unknown. We consider CD-Learning first in the conjunctive pre-conditional case and then in the conjunctive conditional case, both with STRIPS operators, though the results of this section extend to any language covered by our action-schema formalism that has KWIK-learnable conditions. We begin by showing that large conjunctions are not KWIK-learnable.

3.6.1 The Difficulty of Learning Conjunctive Pre-conditions

As covered earlier, STRIPS operators are composed of a pre-condition in the form of a conjunction over a subset of the domain fluents (literals) $\mathcal{F}$, and similar effects (Ω) consisting of Add and Delete lists. The variable arguments to these predicates must match variable parameters of the action itself. Thus, when actions have their arity bounded by a constant m and predicates P with arity bounded by a constant n, there are Pm^n literals to consider in each of the three operator lists (Pre-conditions, Add, and Delete). Note that if negated literals are considered in the pre-conditions the same encoding suffices with only a doubling of the possible predicates.

While this limited number of literals leads to tractable algorithms for learning other portions of the action schemas (as we will see later), we show that this is not generally true for the pre-conditions. The crucial problem is the limited feedback provided by the environment when the pre-conditions of the action do not hold. As stated in Definition 10, if an agent attempts to execute an action whose pre-conditions do not hold in the current state, the agent receives a "failure" signal and the state does not change. But, notice this failure signal does not stipulate what literals or combination of literals are missing from the current state. These *negative examples* are thus, highly uninformative. For instance, trying to *pickup* a block that is under another block will fail. But, from the learner's perspective, without positive examples it cannot tell if it was because the block was not *Clear* or was not a *Table* or some other combination of reasons.

In contrast, positive examples, where actions do succeed, are highly informative, because they identify a superset of the literals that are part of the pre-condition. Here, we give a negative result that, even in the deterministic case, an exponential (in the number of predicates P, and more generally in the number of literals) KWIK-bound can be reached when learning pre-conditions, because of overwhelming, uninformative, and unexpected negative examples. The example domain in the proof is a "combination lock" of D tumblers.

Theorem 5. *The CD-Learning problem is not efficiently KWIK-solvable in the pre-conditional (or conditional) case when conditions may be conjunctions of arbitrary size, even in the deterministic setting. In fact, a lower bound of $\Omega(2^L)$ $\perp$ predictions for environments with L literals can be established.*

Proof. We construct a "combination lock" environment with a single object, **lock**, and $2D+1$ predicates: *Unlocked*(X), $Zero_i$(X), and One_i(X), for $i \in [1, D]$ (note, $D = \frac{P-1}{2}$ in this environment with P predicates). There are $A = 2D + 1$ actions, one to set each predicate and falsify the other for pair i, and one to "open" the lock. More formally, the actions are of the form a_{0i}, a_{1i}, and *open*, again with $i \in [1, D]$. The schemas for these actions are:

a_{0i}(X):
PRE: $\emptyset$
ADD: $Zero_i$(X)
DEL: One_i(X)

a_{1i}(X):
PRE: $\emptyset$
ADD: One_i(X)
DEL: $Zero_i$(X)

open(X):
PRE: $\forall i \in [1, D]$ One_i(X)
ADD: *Unlocked*(X)
DEL: $\emptyset$

The goal is to reach a state where the grounded fluent *Unlocked*(**lock**) is true. As a result, an agent that is started in an initial state, say with all the $Zero_i$ fluents true, can fail (predicting $\perp$ but not opening the lock) in $2^D - 1 = 2^{\frac{P-1}{2}} - 1$ attempts to successfully invoke *open* (one for every possible state other than the one corresponding to the true combination). As a concrete example, starting from the all *Zero* state, a KWIK algorithm must predict $\perp$ for *open*'s outcome. Notice if it makes a non-$\perp$ prediction, then it could potentially be wrong (if the current state really was the correct combination) and therefore would fail the KWIK criteria because in this discrete deterministic setting any incorrect transition prediction leads to a potentially greater than ϵ error in predicting the next state. The same holds in every successive setting of the lock (000...01, 000...10, ...) until all but 1 combination has been tried, because at every step, the algorithm would risk a failure under the KWIK conditions if it claimed the lock would open when it would not. So, any succession of $2^D - 1$ unique combination settings where the lock does not open achieves the lower bound. Each failed plan is met only with a *fail* message, refuting only one hypothesis. □

This result demonstrates the intractability of KWIK-learning arbitrary length conjunctions, a known detriment to the KWIK protocol. The result is easily extendible to the conditional case *if* no restrictive assumptions (like having only 2 conditions) are made on the number and size of the conditions. As a result, we are faced with two choices, either limit the size of the conditions being learned, or change the way the agent and the environment interact so as to avoid the extremely uninformative negative examples that felled the agent above (all of those "unlock" failures). For the rest of this chapter, we will follow the first of these alternatives, assuming that the pre-conditions are defined by conjunctions of constant size ($O(1)$)[5]. However, in Chapter 4 we will consider a different learning paradigm (Apprenticeship Learning) where this size restriction can be relaxed (and the combination lock domain can be efficiently learned).

3.7 Learning Small Pre-conditions

In the case where the conjunctions are of size no greater than a constant $k = O(1)$, we can limit the number of unexpected negative examples the agent will encounter, leading to an efficient KWIK algorithm for learning this restricted class of pre-conditions. The "trick" here is that the hypothesis space for the pre-conditions is only exponential in k, so we can systematically prune the hypothesis space one by one and make only a polynomial number of $\perp$ predictions. For instance, if the combination lock above had only 3 tumblers, there are only D^3 combinations, where D is the number of digits on each tumbler. When the hypothesis space is enumerable in this way and the labels are deterministic ($z_t = y_t$ with no noise, which is the case with

[5] In the conditional case we will consider this assumption as well as an analogous assumption on the number of conditions.

pre-conditions in either the deterministic or stochastic schemas defined earlier) the generic enumeration algorithm (Algorithm 12) can be used to KWIK-learn any hypothesis class H with a bound of $|H| - 1$ (Li et al., 2008).

Algorithm 12 KWIK-Enumeration

```
Given: Hypothesis class $H$
$\widehat{H} = \{h \in H\}$
for each input $x_t$ do
    $Y = \{h(x_t)\}$ for all $h \in \widehat{H}$
    if $|Y| = 1$ then
        Predict $y \in Y$
    else
        Predict $\perp$
        Observe $y_t$
        Remove all incorrect hypotheses from $\widehat{H}$
    end if
end for
```

Returning to the case of learning conjunctions as pre-conditions for actions, we can re-write the enumeration algorithm in the following way to learn k-term pre-conditions.

Algorithm 13 STRIPS-Enumeration

```
Given: Action $a$ of arity $m$, Predicate set $P$ (max-arity $n$)
Output: for state/action pairs, predict one of $\{success, failue, \perp\}$
$PRE^a$ = Enumeration algorithm (Algorithm 12) for all the $O((Pm^n)^k)$ potential conjunctions for action $a$.
for each input state $s_t$ and ground action $a_t$ do
    Predict $Pre^{a_t}(s_t)$, observe $s'$ or failure
    Update $Pre^{a_t}$ with the observation.
end for
```

Intuitively, the algorithm predicts that an action will fail in a state only if every currently valid hypothesis says it will, and the same unanimity is required for predicting the action will succeed. Otherwise, $\perp$ will be predicted. In concert with an optimistic interpretation (covered below), the enumeration algorithm will drive the agent to explore states where it does not know if the pre-conditions of an action hold (trying various combinations on the lock), but never retrying disproved conditions, and never exploring to try a new condition if one that is known to work is already available (if the combination lock is already set to the correct combination and the agent knows this, it will not try others). The KWIK bound for this STRIPS-Enumeration algorithm is $O(A(Pm^n)^k)$, the size of the hypothesis space.

So, we now have a component for learning pre-conditions that are of constant size. Before combining it with our solution to the D-Learning problem, we must consider how to interpret such a model to facilitate efficient exploration (using the KWIK-Rmax architecture or instead a more clever algorithm like Known-Edge VI). There are several practical considerations for this design choice, as discussed in the next section.

3.7.1 Optimistic Schemas for CD-Learning

In our solution to the D-Learning problem, we introduced a heuristic optimistic interpretation of the KWIK learners for the effect distributions that leveraged the inherent background knowledge (the known effects) to improve the best case behavior of the learner, specifically moving the probability mass of the unknown effects to the best possible outcome. One could employ a similar trick naïvely in the CD-Learning case, but we now argue that doing so is not advisable, though a smarter combination of this heuristic with a KWIK-Rmax style optimism does result in a more usable interpretation than using only KWIK-Rmax.

The trouble with employing just Known-Edge Value Iteration in the CD-Learning problem is that the interpretation of every $\perp$ that originates from the pre-condition learner (STRIPS-Enumeration) needs to be interpreted as "this action will succeed here" in order to encourage the agent to actively explore the pre-condition hypothesis space. As a result, the planner that uses an optimistic interpretation will end up considering the application of effects and action parameters that are nonsensical and lead to otherwise unreachable states. For instance, in noisy blocks world, while an agent is learning the pre-conditions, the background knowledge about the effects might make it think it could pick up the table or put a block on top of itself. This assumption leads to an extremely large number of states for the planner to consider and is usually infeasible.

To combat this explosion without sacrificing the heuristic speed-up we saw in D-Learning with Known-Edge VI, we will use a combination of the Known-Edge and $R_{\max}$ techniques. Specifically, when the planner needs an optimistic interpretation of the model, we will introduce $R_{\max}$ state transitions anywhere a $\perp$ is coming from a learner that is uncertain if the pre-conditions in a state are sufficient for an action to occur, and will use Known-Edge VI anytime the uncertainty stems just from the effect distributions being learned. This optimistic interpretation will keep the planning space relatively small in most practical domains and again does not sacrifice the worst-case bounds of the naïve $R_{\max}$ interpretation of general KWIK-Rmax. This approach is used in the algorithm below for the pre-conditional case where the pre-condition and distribution learning are easily separable. In the conditional case, these signals will be more intertwined and so, depending on the implementation of the algorithms, standard KWIK-Rmax may be the best solution.

3.7.2 Efficient Pre-conditional CD-Learning

We can now make use of the modularity of KWIK-learning algorithms to combine KWIK-LR (for learning distributions) and STRIPS-Enumeration (for bounded length conjunctive pre-conditions) to make an efficient online algorithm for solving the CD-Learning problem for STRIPS, and more generally for any action-schema language where the pre-conditions are KWIK-learnable. The online RL algorithm pre-conditional CD-Learning is presented in Algorithm 14 using the mixed optimistic interpretations described above.

Algorithm 14 Pre-conditional CD-OnlineLearn

```
Given: Action set $A$ of max arity $m$, Predicate set $P$ of max-arity $n$, Effect set $\Omega^a$ for each $a \in A$
Initialize a copy of Algorithm 13 ($Pre^a$) for each $a \in A$ (more generally a KWIK learner for the pre-condition hypothesis class)
Initialize a copy of Algorithm 9 ($LR^a(\Omega^a)$) for each $a \in A$
for each episode do
    for $s_t$ = current state do
        If any of the KWIK learners have changed, construct a model interpreting any $Pre^a$ $\perp$ as an $R_{\max}$ transition and $LR^a$ used to determine transition probabilities
        Perform Known-Edge Value Iteration (Algorithm 10) on this model
        Execute the greedy action $a_t$
        Observe $\{success/failure\}$, $s_{t+1}$
        Update $Pre^{a_t}(success/failure)$
        Update $LR^{a_t}(s_t, a_t, s_{t+1})$
    end for
end for
```

Intuitively, Algorithm 14 maintains the consistent version space over pre-conditions using the enumerated possibilities in Algorithm 13, and uses Algorithm 9 to maintain the probabilities of each of the given effects for each action. Just as with Algorithm 11, Known-Edge Value Iteration is employed in this algorithm, but in an effort to constrain the state space considered by the planner, $R_{\max}$ transitions are used for uncertain pre-condition/state pairings. This construction is again done on a state by state basis, so a plan may be returned that contains an action to be executed where its pre-conditions actually fail. As we show in the next theorem, this sort of exploration helps the learner find a consistent model by testing conflicting hypotheses, ultimately achieving a polynomial sample complexity bound in the CD-Learning problem.

Theorem 6. *Algorithm 14 is PAC-MDP as the model learners KWIK-solve the CD-Learning* *problem in the stochastic (or deterministic) pre-conditional case when each action's pre-condition is a conjunction of no more than k terms for constant k, or some other KWIK-learnable hypothesis class.*

Proof. (sketch) The proof follows from the KWIK bounds for the component algorithms and the KWIK-Rmax Theorem. Specifically, there are a bounded number of $\perp$ predictions that can be made by each of the components that will lead to suboptimal behavior. Any of the $B_{LR}(\Omega, \epsilon, \delta/A)$ (Equation 3.1) such $\perp$ predictions by the KWIK-LR component that led to suboptimal behavior will result in experience for that specific action in a state/equivalence class were the probability distributions are currently unknown, just as in the D-Learning case. The other $O(APm^n)$ possible $\perp$ predictions that result in suboptimal behavior (from Pre^a) will result in at least one hypothesis (which incorrectly predicted an action could be executed from a given state) to be eliminated. □

As empirical evidence of the success of this algorithm, we present a comparison of Algorithm 14 against a flat MDP learner and Algorithm 11 for D-Learning, which is given the pre-conditions of actions. All 3 algorithms were run on stochastic blocks world with 3 blocks and the "dummy" actions (versions of pickup and putdown with the probabilities of the "nothing happens" outcome reversed from the two standard pickup/putdown actions) included. The number of blocks is kept low to accommodate the flat MDP learner and because we are still using exact planning algorithms (a situation dealt with in Section 6.3). Figure 3.3 displays the results. The KWIK CD-Learner takes around 20 episodes (with maximum of 15 steps per episode and a goal of stacking the 3 blocks) to learn the pre-conditions, but still greatly outperforms its flat MDP counterpart (which actually takes several hundred episodes to perform consistently well) because it is able to generalize its knowledge about moving blocks from one situation to another and reaches the same optimal behavior as the D-Learner. Note that the D-Learner learns much faster because many fewer examples are needed to just learn the effect distributions than to find the positive examples necessary for pre-condition learning (a combination-lock like problem, even with the bounded size pre-conditions).

We have shown that under assumptions on the size of conjunctive pre-conditions, we can efficiently solve the CD-Learning problem. In the next section, we consider the conditional case for stochastic operators.

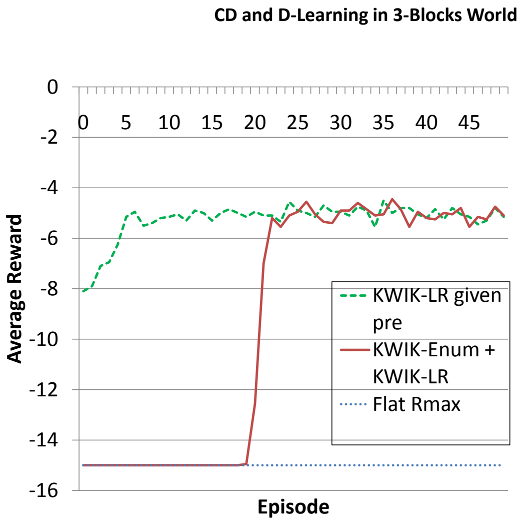


Figure 3.3: Learning pre-conditions and distributions in stochastic blocks world

3.8 Learning Conditions

We now turn our attention to the conditional case, where the effects (or distribution on effects) are determined by which of the $|C|$ conditions $c_1 ... c_{|C|}$ actually hold in a given state (denoted as $c_i[s] = \mathit{true}$). Recall that Definition 11 stipulates that these conditions may not overlap and must cover the entire reachable state space, so for each state, one and only one condition will "match" the state. The difficulty of learning conditions without any assumptions on the size or number of the conditions follows from Theorem 5, for the pre-conditional case with the following construction. Notice that it requires n conditions because there is no "else" clause in this construction.

Corollary 1. *Without restrictions on the conditions, the CD-Learning problem for conditional action schemas where conditions are conjunctions over the domain literals is not efficiently KWIK-solvable. In fact, again a lower bound that is exponential in the number of domain literals can be achieved.*

Proof. We can change the combination-lock example from Theorem 5 into a conditional action schema by creating one condition for the combination that opens the lock (say $\mathit{One}(\mathrm{X1}) \wedge \mathit{One}(\mathrm{X2}) \wedge \mathit{One}(\mathrm{X3})$) and D more conditions of size 1 that cover the possibility that tumbler i is set wrong. These extra D conditions are needed to represent the "else" behavior inherent to the pre-conditional case that is not explicitly representable in the conditional case. Notice that this problem is no more KWIK-solvable than the pre-conditional formulation—an agent is still required to try all 2^D combinations even though the domain encoding is linear in D. □

Despite this result, we can pursue a strategy similar to our approach in the pre-conditional case; we will work under assumptions about the size, and later the number, of the conditions. We begin by considering the cases (deterministic and then stochastic) where the sizes of the individual conditions are bounded by a constant $k = O(1)$. After that, we will also give positive (though less practical) results under the assumption that the conditions themselves can have non-constant size, but the *number* of conditions is restricted to be $O(1)$. Notice such restrictions do not allow for the construction of the combination lock, where D conditions are required, one of which is of size D.

Beyond the hardness discussed above, the conditional case presents difficulties for learning algorithms beyond those seen in the pre-conditional setting. Specifically, there is no longer a unique *failure* signal indicating whether or not a specific condition holds. For instance, in a deterministic version of the metal/wooden Paint-Polish world (where only metal objects can be painted without scratching), when the *paint* action is executed on an already scratched object, we receive no information about which condition actually "matched". Did the object become painted because it was wooden (or metal?), or some other condition (like being scratched already)? Note we cannot just refine a single effect's associated condition in this case because both the *Painted* and *Painted* $\wedge$ *Scratched* effects explain the observation. The problem becomes even more pronounced in the stochastic case where not only can multiple effects seem to have happened, but any one observation can appear to be from many different effect distributions. The following sections and algorithms deal with these problems in turn, starting with the deterministic case.

3.8.1 Learning Small Conditions in the Deterministic Case

In this section, we will work under the assumption that each condition will (like the pre-conditional results) have to be over only a small number of terms so that we can do a type of enumeration. For now, we will consider the deterministic variant of CD-Learning (where the effect distributions do not need to be learned) in order to highlight the challenges and solutions used in learning the conditions themselves. An example is a deterministic version of metal/wooden Paint-Polish domain where painting a wooden object causes it to become painted and scratched, while painting a metal object simply paints it. In this case, the agent must experiment to determine what conditions are causing the different effects. But, unlike the pre-conditional case, we cannot completely decouple the condition-learning piece (Enumeration) from the effects themselves, because there is no unique *failure* signal. Instead, the effects themselves are the only clues as to which hypothesis needs to be updated, but unfortunately these can be ambiguous (as in the painting of a scratched object).

In light of this symbiotic relationship, we will use a slightly more complicated (but still Enumeration-like) KWIK algorithm called *Active Union*, which is applicable in deterministic settings where enumerable conditions govern what other learners should be making predictions. More specifically, the algorithm will enumerate all the possible conditions (conjunctions of up to size k), many of which may match a current state, and merge their predictions if possible (otherwise predicting $\perp$). Active Union is a generalization of the enumeration algorithm, where conditions determine which other learners should be consulted to make a prediction. The original Union algorithm was first introduced by Li et al. (2008), but was used to combine predictions from all the component learners, not just a set of "active" components as determined by the conditions. The original algorithm is inapplicable in our setting as we do not want a component tied to the condition *Metal*(X) to make a prediction for a wooden object. This Active Union algorithm is described here in Algorithm 15.

Algorithm 15 Active Union

```
*Given*: KWIK Learners $\mathcal{L} = \{KL_1 ... KL_n\}$ and general Boolean functions conditions (like small conjunctions) $C = \{c_1 ... c_n\}$
**for** each input $x_t$ **do**
  $\widehat{KL} = \{KL_i | c_i(x_t) = true\}$ for all $KL_i \in \mathcal{L}$ and $c_i \in C$
  $Y = \{KL(x_t)\}$ for all $KL \in \widehat{KL}$
  **if** $|Y| = 1$ **then**
    Predict $y \in Y$ //could still be $\perp$.
  **else**
    Predict $\perp$
  **end if**
  **if** $\perp$ was predicted **then**
    Observe $y_t$
    **for** each $KL_i \in \widehat{KL}$ **do**
      **if** $KL_i(x_t) \neq \perp$ and $KL_i(x_t) \neq y_t$ **then**
        $\mathcal{L} = \mathcal{L} - KL_i$
        $C = C - c_i$
      **else if** $KL_i(x_t) = \perp$ **then**
        Update $KL_i$ with $x_t$ and $y_t$
      **end if**
    **end for**
  **end if**
**end for**
```

Intuitively, Active Union is initialized with a set of component KWIK learners, each of which has a corresponding Boolean formula (a condition). For every input x_t, those learners whose conditions are true for x_t become "active" learners and each makes a prediction. If these predictions are all the same, that prediction is made (note that if all the learners predict $\perp$, then that prediction is made). If the learners disagree, $\perp$ is predicted. Either way, if $\perp$ is predicted by the overall algorithm, a deterministic label is received (we will cover a stochastic version in the next section). This label is used to eliminate component learners (and their corresponding conditions) that are predicting incorrect labels (and no longer admitting $\perp$) and is also used to update learners that still declare uncertainty. We will use this algorithm for the deterministic conditional CD-Learning problem and later in the deterministic case where the effects also need to be learned, as well as a variant introduced later for the stochastic case. For this simple version above, we have the following KWIK result:

Lemma 3. *The Active Union algorithm (Algorithm 15) over* n *components has a KWIK bound of* $(n-1) + \sum_{i=1}^{n} B_i(\epsilon, \delta/n)$ *in the case where the hypothesis class is realizable by the* c_i *partitioning and where* ϵ *and* δ *are the desired accuracy and confidence parameters,* n *is the number of component learners and* B_i *is the KWIK bound of component learner* KL_i*.*

Proof. (sketch) The proof follows along the same lines as the original Union algorithm (Li, 2009), which we sketch here. Every time the algorithm makes a $\perp$ prediction, either one of the active component learners has made a $\perp$ prediction, or all the components made non-$\perp$ predictions but they were not identical. In the latter case, at least one component learner will be eliminated based on its wrong prediction. This learner cannot be applicable in other conditions because each c_i is static (though a whole c_i can be eliminated) and the true c_i's are not allowed to overlap unless they make the same prediction for a given x_t (hence the "realizable" caveat in the lemma). Thus, only $n-1$ $\perp$ predictions can come from such cases. The scenario in which some component learner is predicting $\perp$ is covered by the summation over individual bounds and a union bound is used to ensure the correct probability of failure δ, hence the stricter δ/n failure probability for the individual components. □

We note that the original Union algorithm allowed the outer learner to make a non-$\perp$ prediction if all of the $y \in Y$ were within ϵ of one another, in which case it predicted the median value of this set. A similar modification is easily done to Active Union with only a slight change in the worst-case bounds ($B(\epsilon/2, \delta/n)$ replaces the current B), but we do not make use of such functionality in this thesis (we use a different algorithm below in the stochastic case where such continuous-value accuracy is apropos).

In the deterministic CD-Learning setting, the effects are given, so the individual KWIK learners whose hypotheses are being combined in *ActiveUnion* are replaced simply by one *static* "learner" for each effect that always predicts that effect. We make $O(C^k)$ copies of each of these learners and link each one to a possible condition, thus enumerating the cross product of the small condition/known effects. More formally, a general solution to the conditional deterministic CD-Learning problem that uses the Active Union algorithm as its backbone is presented in Algorithm 16.

Algorithm 16 Conditional Deterministic CD-OnlineLearn

```
Given: Action set $A$ of max arity $m$, Predicate set $P$ of max-arity $n$, Effect set $\Omega^a$ for each $a \in A$
Construct static predictors $KL^a_\omega$, one for each possible effect of each action.
$\forall a \in A$ construct condition learners $CL^a$ by initializing a copy of Algorithm 15 with $C$ containing $|\Omega^a|$ copies of each possible conjunction of size $k = O(1)$ or less and attach each copy to a different $KL^a_\omega$.
for each episode do
  for $s_t$ = current state do
    If any of the KWIK learners have changed, construct a model interpreting any $CL^a$ $\perp$ as an $R_{\max}$ transition
    Perform Known-Edge Value Iteration or another $\epsilon$-accurate deterministic planning algorithm.
    Execute the greedy action $a_t$, Observe $s_{t+1}$
    Update $CL^{a_t}(s_t, s_{t+1})$
  end for
end for
```

Intuitively, the algorithm considers every possible condition/effect pairing for each action and the optimistic interpretation of the KWIK Active Union predictions (again using a KWIK-Rmax interpretation to keep the considered state space close to the actual reachable state space's size) drives the agent towards portions of the state space where different conditions can be experimented with to determine how they correlate to the known effects. This algorithm has the following property for domains described by conditional deterministic action schemas where the true conditions do no overlap in any reachable state and the conditions themselves are conjunctions of at most $k = O(1)$ terms.

Theorem 7. *Algorithm 16 is PAC-MDP for domains described by deterministic conditional action schemas where the conditions are conjunctions of size $O(1)$ or some other polynomially-enumerable hypothesis class.*

Proof. (sketch) The proof follows simply from the KWIK bounds in Lemma 15 and Theorem 1, which connects KWIK learning of a model of PAC-MDP behavior. Each sub-optimal step contributes towards eliminating some disagreement between one of the still possible conditions, which results in the elimination of one of these conditions for an action attached to a single effect. □

3.8.2 Learning Small Conditions in the Stochastic Case

We now present a KWIK algorithm that learns both the conditions and the effect distributions in the stochastic conditional CD-Learning problem, assuming that each condition is a conjunction of no more than $k = O(1)$ literals. As with the earlier stochastic pre-conditional CD-Learning case, the problem is decomposed into learning the conditions and effect distributions, *but* as in the deterministic conditional case above, we no longer have explicit feedback as to which condition was actually active in the last step. The stochastic case introduces a second obstruction to learning conditions. Not only do ambiguous effects not tell us which condition (if any) needs to be updated, but now the feedback we *do* see is stochastic. As an example, consider the stochastic version of Metal/Wooden paint-Polish world where the object's surface composition (metal or wooden) determines the distribution over possible effects, with metal objects having a lower (but still non-zero) probability of becoming scratched and painted than their wooden counterparts. In the earlier deterministic case, if an object was unscratched to begin with, we could tell exactly which condition/effect pairs needed to be eliminated (based on whether the object came back scratched or not), but in this case, a single piece of experience is not informative enough to eliminate one of these pairs—if the possible effect distributions involved scratching with probability 0.7 for wooden objects and probability 0.3 for metal objects, it is still possible to see a metal object get scratched, and unclear what pairing of distributions (the 0.7 and the 0.3 probabilities would each be predictions for different component learners) should be eliminated. Furthermore, in the CD-Learning problem these distributions are not given, and instead need to be learned online, so the architecture from the deterministic case needs to be changed.

The changes are made below by replacing Active Union with an algorithm called the Active k-Meteorologist Solver (AKMS), which is essentially a version of Active Union that can handle noisy observations. We also replace the "static" effect predictors from the deterministic case with KWIK-LR learners to learn the probability distributions over the given effect sets for each action.

The Active k-Meteorologist Solver

The difficulties described above require an algorithm that (1) considers multiple hypotheses for C and (2) does not commit to any condition set without overwhelming statistical evidence. We now describe such an algorithm, the Active k-Meteorologist Solver (AKMS). The algorithm is an extension of the original k-Meteorologist solver (Diuk et al., 2009), with an adaptation similar to our change to Union above—instead of all component learners making predictions, each component learner has an associated condition (Boolean formula) and only those learners whose conditions actually match the current input are consulted. The full algorithm appears in Algorithm 17.

Conceptually, AKMS is similar to Union in that it considers the predictions of several component learners and predicts $\perp$ when they disagree. But, there are two main differences. First, AKMS is built for continuous predictions and so when components make predictions that are close, they can be merged using a median prediction (the same change is possible for Active Union as discussed earlier). More importantly, AKMS does not eliminate hypotheses that make a single wrong prediction. Instead, it waits for sufficient statistical evidence to accrue, indicating that a component is (with high probability) making consistently incorrect predictions.

The conventional metaphor for understanding the meteorologist solver is that of deciding which weather forecasters in a given location make accurate predictions. That is, each of the component learners could be, at the most basic level, simply predicting the probability of precipitation every day. The algorithm keeps track of pair-wise statistics as to which meteorologist is more accurate than another. Once enough evidence is collected to show that two meteorologists are statistically different (line 20) , the further outlier is eliminated. Only when all the remaining meteorologists make predictions within ϵ of one another can a prediction be made with high confidence. AKMS simply extends this algorithm to the setting where not all the meteorologists make a prediction at every step. For instance, if one of the meteorologists was out sick on a given day, we should not ignore the predictions of the others. Notice the conservative approach of waiting for sufficient statistical evidence to eliminate a component learner is exactly what is necessary to deal with the noisy signals described above for stochastic conditional schema learning. Formally, AKMS has the following KWIK bound:

Lemma 4. *With $m = O(\frac{1}{\epsilon^2}\log\frac{k}{\delta})$, AKMS (Algorithm 17) has a KWIK bound of $O(\frac{n}{\epsilon^2}\log\frac{n}{\delta}) + \sum_{i=1}^{n} B_i(\frac{\epsilon}{8}, \frac{\delta}{n+1})$ where B_i is the KWIK bound on component learner KL_i.*

Algorithm 17 Active k-Meteorologist Solver (AKMS) [Extended from the work of Diuk et al. (2009)]

```
*Given*: KWIK Learners $\mathcal{L} = \{KL_1...KL_n\}$ and general Boolean conditions (like small conjunctions) $C = \{C_1...C_n\}$
For $1 \leq i, j \leq n$, $count_{ij} = 0, \Delta_{ij} = 0$
**for** each input $x_t$ **do**
    $\widehat{KL} = \{KL_i | C_i(x_t) = true\}$ for all $KL_i \in \mathcal{L}$ and $C_i \in C$
    $Y = \{KL(x_t)\}$ for all $KL \in \widehat{KL}$
    **if** $\forall y_i, y_j \in Y, ||y_i - y_j|| < \epsilon$ **then**
        $Y = \{$ the median value of $Y\}$
    **end if**
    **if** $|Y| = 1$ **then**
        Predict $y \in Y$ //could still be $\perp$.
    **else**
        Predict $\perp$
    **end if**
    **if** $\perp$ was predicted **then**
        Observe $z_t$
        **for** each $KL_i, KL_j \in \widehat{KL}$ **do**
            **if** $KL_i(x_t) \neq \perp \wedge KL_j \neq \perp \wedge ||KL_i(x_t) - KL_j(x_t)|| \geq \frac{\epsilon}{2}$ **then**
                $count_{ij} \mathrel{+}= 1$
                $\Delta_{ij} \mathrel{+}= (KL_i(x_t) - z_t)^2 - (KL_j(x_t) - z_t)^2$
                **if** $count_{ij} > m$ **then**
                    **if** $\Delta_{ij} > 0$ **then**
                        $\mathcal{L} = \mathcal{L} - KL_i$
                        $C = C - C_i$
                    **else**
                        $\mathcal{L} = \mathcal{L} - KL_j$
                        $C = C - C_j$
                    **end if**
                **end if**
            **end if**
            **if** $KL_i(x_t) = \perp$ **then**
                Update $KL_i$ with $x_t$ and $y_t$
            **end if**
        **end for**
    **end if**
**end for**
```

Proof. (sketch) The proof follows almost exactly the form as the proof of the original meteorologist solver (see Diuk et al. (2009) and Li (2009) for details). At a high level, the proof proceeds by counting the $\perp$ predictions from each component learner (the second term in the bound above) and the number of times $\perp$ will be predicted because of differing predictions before the components that are more consistently wrong are eliminated. The proof hinges on showing that the component learner(s) that are close to the true hypothesis have the lowest squared error on average and that, with high probability, only components with larger squared errors will be eliminated.

The only difference between the AKMS analysis and this result is that now not every component learner makes a prediction at each step. However, since only pairwise comparisons are ever used to eliminate component learners, and since the worst case bounds are derived considering the case where only one pair-wise disagreement occurs at a time, the worst case bounds are unchanged. □

Learning Small Conditions and Effect Distributions with AKMS

We now incorporate the active k-meteorologist solver into a revised version of Algorithm 16, now for the stochastic conditional CD-Learning problem. In this setting, we need to consider the form of the given effects a little more closely, because the effects for a conditional stochastic action schema are actually a set of effect sets $\mathbf{\Omega^a}$, with each $\Omega_i^a \in \mathbf{\Omega^a}$ being linked to a different condition. Two types of background knowledge seem plausible in this situation. We may be given the set of effect sets $\mathbf{\Omega^a}$ or just told what effects might occur in *any* condition for that action ($\Omega^a = \bigcup \Omega | \Omega \in \mathbf{\Omega^a}$). Since we can always turn the former form into the latter by "flattening" the effect sets for each action, we consider here the case where we are only given this flat set of effects, though we remark on an alternative architecture for the former case. Our algorithm will be able to deal with potential effects that do not occur under certain conditions (but are included in the flattened set) because the probability-learning component will set their likelihood to 0. With this background knowledge, Algorithm 18 efficiently solves the CD-Learning problem when conditions are guaranteed to be conjunctions of size $O(1)$ by using KWIK-LR learners (for learning effect distributions) as component learners for the meteorologist architecture (MET^a, which sorts out the correct conditions for each action).

Intuitively, M meteorologist sub-learners are considered for each action, each one for a pairing of a possible condition with an effect set. Each component is a KWIK-LR learner

Algorithm 18 Stochastic Conditional CD-OnlineLearn

```
Given: Action set $A$ of max arity $m$, Predicate set $P$ of max-arity $n$, Effect sets $\Omega^a$ for each $a \in A$
Construct $M = \sum_{i=0}^{k} \binom{P}{k}$ KWIK-LR predictors (Algorithm 9) $LR^a_i(\frac{\epsilon}{8}, \frac{\delta}{M})$, one for every possible condition.
$\forall a \in A$ Construct a meteorologist learner $MET^a$ by initializing a copy of Algorithm 17 with $C$ containing each of the $M$ possible conditions and using the KWIK-LR learners above as component learners.
for each episode do
  for $s_t$ = current state do
    If any of the KWIK learners have changed, construct a model interpreting any $MET^a$ $\perp$ as an $R_{\max}$ transition
    Perform Known-Edge Value Iteration.
    Execute the greedy action $a_t$, Observe $s_{t+1}$
    Update $MET^{a_t}(s_t, s_{t+1})$
  end for
end for
```

(Algorithm 9), which keeps a running estimate of the probabilities of each of the possible effects estimated from data from when the corresponding condition held (because component learners are only updated when their corresponding conditions are active). When the planner queries MET^a about executing $a(o_1...o_m)$ in s_t, all of the meteorologists whose condition c_i is satisfied (with substitutions based on $o_1...o_m$) by s_t are asked to predict the distribution of possible next states. Depending on the status of their accompanying KWIK-LR algorithm and the current state, they either report $\perp$ or a probability distribution and are combined using the meteorologist architecture. The agent uses these probability predictions to construct a model, plans with it, and then performs the recommended action and passes the experience to the appropriate meteorologists. This real experience activates the purging mechanism (Line 20 of Algorithm 17) to eliminate conditions whose component learners are making consistently bad predictions about the probability distributions of effects. In terms of exploration, the algorithm is driven by AKMS to areas of the state space where the possible conditions are separable (for example, trying to paint wooden or metal objects) and within those areas, actions whose probability distributions on effects are not known under these conditions are tested. Formally, we can state that the algorithm efficiently KWIK-solves the CD-Learning problem in the conditional stochastic setting when conditions are described by bounded conjunctions or some other enumerable and KWIK-learnable hypothesis class amenable to the AKMS architecture.

Theorem 8. *Algorithm 18 is PAC-MDP and KWIK-solves the CD-Learning problem in the conditional case when each condition is a conjunction of no more than $k = O(1)$ terms.*

Proof. The proof follows from the KWIK bounds on the learning components. Specifically,

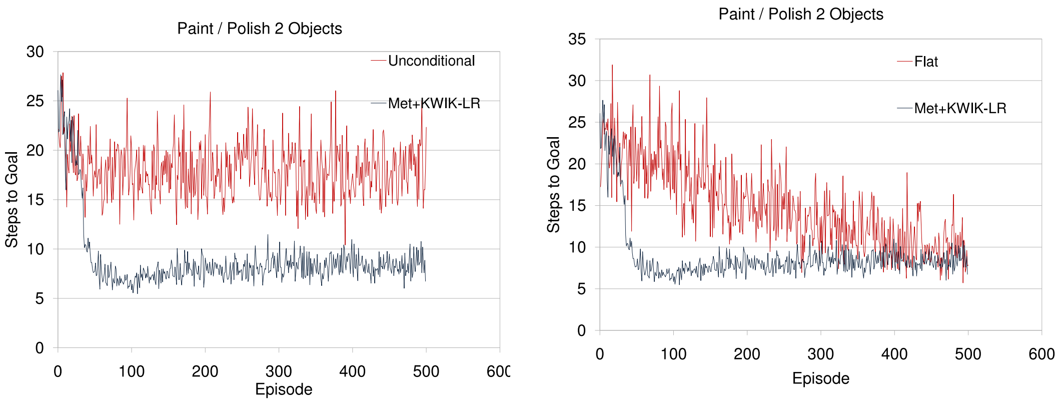


Figure 3.4: CD-Learning in Metal/Wooden Paint-Polish World with two objects. On the left, Meteorologist+LR is compared to a learner that does not learn conditions. On the right, Meteorologist+LR is compared to a Flat MDP learner.

since each KWIK-LR component learner runs with a bound of B_{LR} (Equation 3.1 with appropriate accuracy parameters as per the meteorologist) $\perp$ predictions, and using Lemma 4 to bound the number of $\perp$ predictions for all the MET^a learners, we end up with a total of $O(A\frac{L^k}{\epsilon^2}\log\frac{L^k}{\delta}) + \sum_{i=1}^{n} B_{LR}(\Omega, \frac{\epsilon}{8}, \frac{\delta}{n+1})$ $\perp$ predictions, where L is the number of domain literals in an action's scope (Pm^n in the STRIPS case). As the optimistic interpretations used have already been shown to be safe in translating KWIK algorithms to PAC-MDP behavior, the algorithm performs as required. $\square$

As a proof of concept, Figure 3.4 shows the algorithm above being used in the stochastic metal/wooden paint-polish domain with 2 objects. On the left, Algorithm 18 (MET+KWIK-LR) is run against a learner that tries to apply KWIK-LR directly without learning the conditions. This learner is unable to learn the benefit of coating the objects with metal and does not consistently reach optimal behavior. A flat MDP learner is also shown (on the right), which does not generalize over objects nor concentrating on the small conditions associated with each action, instead considering every combination of statuses of the two objects as a thoroughly independent state. Algorithm 18 performs dramatically better than this unstructured flat learner.

3.8.3 CD-Learning with a Small Number of Conditions

The positive KWIK results above were all derived under the assumption that the conditions themselves were small. However, in the conditional setting there is another natural assumption that yields a KWIK algorithm (although it is less practical due to large constants). Specifically, we consider the case where the conditions can be conjunctions over any number of literals, but the *number* of conditions is bounded by a constant ($|C^a| = k = O(1)$). Notice that the combination-lock example cannot be represented with such a restriction because conditional action schemas have no "else" block—there is no failure mode in the conditional case the way there was for the pre-conditional case. However, it is just this lack of expressiveness that will allow us to enumerate the possible hypotheses for the conditions. We note that such an else block can be accommodated in CD-Learning if the associated effect is always unique.

Specifically, we can capture all the possible conditions by enumerating the set of decision trees with k leaves. Each root-to-leaf path in such a tree represents a possible condition, and each leaf/label pair represents an effect. Generally, the number of decision trees with k (uniquely labeled) leaves that depend on ν attributes is $\nu^{k-1}(k+1)^{2k-1}$ (Pichuka et al., 2007), and in our case, ν is the number of domain literals L (in STRIPS $L = Pm^n$). Having thus enumerated the hypothesis space, we can utilize Active Union (Algorithm 15) to KWIK-solve the CD-Learning problem and induce PAC-MDP behavior as a direct consequence of Theorem 3 (which holds for any polynomially enumerable space), but now the bound is $AL^{k-1}(k+1)^{2k-1} = O(AL^{|C|})$, where $|C|$ is the maximum ($O(1)$) number of conditions. However, the rather large constant and the somewhat inelegant enumeration of trees calls the practicality of this approach into question. In Section 4.3.2, we will introduce an algorithm for a different learning paradigm (apprenticeship learning) that does not suffer from such a constant.

Moving to the stochastic case, we see that learning becomes more difficult because when a condition matches a state, an effect is chosen from the distribution $\langle \Omega_i^a, \Pi_i^a \rangle$, so the decision-tree enumeration needs to be augmented with a distribution learning component (for the effect distribution), and we can no longer eliminate possible trees based on a single sample. Again, we can use the template from the "small conditions" case to alleviate both of these troubles. That is, we can use AKMS (Algorithm 17) here, but instead of just enumerating the conditions, we again enumerate all decision trees with k leaves. Each of these possible trees is treated as a meteorologist (in the terminology of AKMS) and again KWIK-LR is used to learn the distribution associated with each possible condition. While the resulting bound is actually polynomial, it is again hampered by enormous constants. It is an open question as to whether a more efficient algorithm is possible without more assumed structure.

This last set of algorithms, which are clearly approaching the edge of tractability, complete our study of CD-Learning in the *online* setting. We will return to this problem in the next chapter where a more powerful learning paradigm will allow us to drop some of the restrictions used in this section.

3.9 Learning Effects and their Distributions

We now turn our attention to the ED-Learning problem, where the conditions are given (instead of having to be learned), but now both the effect distributions (Π^a or Π^a_c) and *the effects themselves* (Ω^a or $\mathbf{\Omega^a}$) need to be learned. In the original definition of ED-Learning (Definition 14), we did not stipulate the size of the learned action schemas, only that they be accurate. However, it seems reasonable to require these schemas to be as compact as possible. To ascertain the tractability of this goal, we consider the ED-Learning problem in the simple case where we are given a constant $k = |\Omega|$ that bounds the number of possible effects under any condition in the schemas. For instance, if $k = 1$, the schemas are deterministic.

Our results will progress from the simple deterministic setting to more difficult stochastic settings, culminating in a KWIK algorithm for general ED-Learning when $k = O(1)$. In all of these positive results, we give language-specific results for STRIPS because other languages may not have the same Add and Delete structure. We begin with the deterministic case, where STRIPS effects are surprisingly easy to KWIK-learn. Along the way, we show that computing a minimal action schema from incomplete data is NP-Hard, and that "greedy" approaches to learning effects can get caught in incorrect models, but our final algorithm sidesteps both of these difficulties by exploiting structure in the stochasticity of a schema and waiting until it has enough statistical evidence to commit to a specific model. Before considering such complications, we examine the simple deterministic case.

3.9.1 KWIK-learning Deterministic Effects

Unlike the stochastic setting, deterministic STRIPS effects can be inferred without considering multiple effects that might be producing the same outcome. In fact, the *operator update rules* from Section 3.4.1, which stated how to infer the role of a literal in an action's effects (for example, seeing a block *To* being no longer clear after a *putdown* action means that *Clear*(To) is in *putdown*'s Delete list). We can formally encapsulate these update rules in KWIK-DetEffectLearn (Algorithm 19) for learning deterministic STRIPS effects for an action[6].

[6] This algorithm can be trivially extended to the conditional case since conditions are given in ED-Learning.

Algorithm 19 KWIK-DetEffectLearn (STRIPS)

```
Given: Literals $L$
Output: Predict the next state or $\perp$
$\forall l \in L,\ Add[l] := Delete[l] := \perp$
for each input state $s$ with objects replaced with variables from $a$ do
  $\widehat{s} = \emptyset$
  for each Literal $l \in L$ do
    if $l \in s \wedge Delete[l] = \perp \bigvee l \notin s \wedge Add[l] = \perp$ then
      Predict $\perp$
      BREAK LOOP
    else if $l \notin s \wedge Add[l] = true \bigvee l \in s \wedge Delete[l] = false$ then
      $\widehat{s} = \widehat{s} \cup l$
    end if
  end for
  if No prediction made yet then
    Predict $\widehat{s}$
  else
    // $\perp$ was predicted
    Observe $s'$ (with variable substitution again based on $a$)
    for each literal $l \in L$ do
      if $l \notin s \wedge l \in s'$ then
        $Add[l] = true$
      end if
      if $l \notin s \wedge l \notin s'$ then
        $Add[l] = false$
      end if
      if $l \in s \wedge l \notin s'$ then
        $Delete[l] = true$
      end if
      if $l \in s \wedge l \in s'$ then
        $Delete[l] = false$
      end if
    end for
  end if
end for
```

Intuitively, the algorithm initially lists every literal in the scope of the action as being unknown in terms of its presence in the Add and Delete lists. Then, for every state given to the algorithm, it is asked to predict the next state after executing this action. If the Add and Delete lists contain uncertainty that would affect the next state (Line 7), then $\perp$ is predicted, otherwise s' is correctly predicted. Note that depending on the state, correct predictions can still be made even when there is uncertainty in irrelevant parts of the Add and Delete lists (for example, if *Table*(X)'s deletion is unknown, but is known not to be added, and is not true in s, it does not affect the prediction of s'). If $\perp$ is predicted, the next state is observed and unknown parts of the Add and Delete lists are filled in based on the observed changes, or lack thereof. For instance, consider the algorithm seeing its first state in deterministic Blocks World as having **a** on the table, **b** on the table, and **c** on **b**. The algorithm has seen no information thus far, so if asked to make a prediction for the *move*(**a**, **table**, **c**) action's effects, it must predict $\perp$. The algorithm would then get a piece of experience (gathered using the full RL algorithm in the next section) showing **a** being moved on top of **c**. This piece of experience would be used in lines 19 through 30 to update the Add and Delete lists of *move*(X, From, To) to contain information such as: Add[*On*(X,To)] $=$ *true* ; Add[*Table*(X)] $=$ *false* ; Delete[*Clear*(To)] $=$ *true* ; and Delete[*Block*(X)] $=$ *false* . In general, each piece of experience after a $\perp$ eliminates uncertainty for at least one literal in either the Add or Delete list. Formally, the algorithm has the following KWIK bound.

Lemma 5. *For deterministic STRIPS domains with P predicates of max-arity n and A actions of max-arity m, learning the effects of each action using Algorithm 19 has a KWIK bound of $O(APm^n)$*

Proof. The number of literals considered in each action's scope is Pm^n (for A actions of max arity m and P predicates of max-arity n), which is the number of predicates with unique variable permutations drawn from the action's variables. Each $\perp$ prediction must occur because either the Add or Delete list contains a $\perp$ prediction for at least one of these literals. So either $l \in s \wedge Delete[l] = \perp$ or $l \notin s \wedge Add[l] = \perp$. Either way, the observed s' will determine definitively if these should be set to *true* or *false* based on the rules in Lines 19 through 30 (since at least one of those *must* be true for each literal). Thus, each $\perp$ prediction removes at least one bit of uncertainty from the Add or Delete, with a maximum of $2Pm^n$ such $\perp$s for each action. □

We note that while this result is specifically for STRIPS domains, any deterministic action schema language where the effects are KWIK-learnable will be usable in the next section. For instance, deterministic OOMDPs with their arithmetic attribute-value changes are KWIK-learnable (Diuk et al., 2008) when the space of available changes is relatively small. We now consider how to translate such KWIK-learnable deterministic effect learning algorithms into full agents for the ED-Learning problem.

3.9.2 Optimistic Interpretations and Deterministic ED-Learning

As with other sub-problems, we have a choice of the optimistic heuristics to guide the learner. One choice is to simply use the KWIK-Rmax architecture and replace all transitions where the next state is predicted as $\perp$ with an Rmax state. However, this interpretation does not make use of the partially learned operators. That is, because KWIK-DetEffectLearn keeps track of the "known-ness" of each literal in each Add and Delete list, we sometimes know partial information about what literals are added or deleted, though maybe not all of them. In that case, the *optimistic Add/Del* heuristic has better best case behavior, if all the rewards and pre-conditions are based on positive values of literals (as is the case for pre-conditions in traditional STRIPS). This heuristic constructs "optimistic operators", where for every potential literal changed by an action, it is considered to be added if $Add[l] = (\perp \vee \mathit{true})$ and is considered to be deleted only if $Del[l] = \mathit{true}$. Under this heuristic, partially learned operators are thought to add as many literals as possible given prior evidence, and only delete those that have certainly been seen to be deleted earlier. STRIPS rules have pre-conditions based on monotone conjunctions, so if we assume that the reward function is similarly based only on monotone literals (and not their negation), this heuristic will always lead to higher valued states (since no action will be wrongly precluded and reward values will only be over-estimated).

Both heuristics have their benefits. As in the CD-Learning setting, the $R_{\max}$-style heuristic prevents us from having to enumerate a large number of states that may not even be reachable. On the other hand, when enough partial information is known about the Add and Delete lists, the optimistic-Add/Del heuristic avoids some useless exploration, though the worst case bounds are the same. One interesting melding of the two is to use the $R_{\max}$ heuristic early on and then switch to the more focussed optimistic-Add/Del when most of an operator is learned, hopefully avoiding the large state-space enumeration. In the following algorithm and our experiments, we simply use the $R_{\max}$ heuristic but we have seen empirical benefits to the other approach.

Our full algorithm for deterministic ED-Learning is presented in Algorithm 20. The algorithm is presented here for the conditional case, as it is the most general, but is easily adapted to the pre-conditional or unconditional cases by creating only 1 copy of KWIK-DetEffectLearn for each action. The algorithm works simply by using the KWIK predictions of KWIK-DetEffectLearn for each action and updating each one with the pre- and post- states of each action taken, which are of course translated into a variablized form (such as On(X, From) instead of On(**a**,**b**)) by simple substitution based on the action's variables. Agents using this algorithm are explicitly drawn towards areas of the state space where it will be able to definitively tell if a literal is added. For instance, if the agent has not learned whether *paint* actually paints an object, it will attempt the action on an object that is unpainted to determine whether the change actually happens. Formally, we can make the following statement about this algorithm in the deterministic ED-Learning setting.

Algorithm 20 Deterministic Conditional ED-OnlineLearn

```
Given: Action set $A$ of max arity $m$, Predicate set $P$ of max-arity $n$, Conjunctions for
  conditions $\mathcal{C}^a = \{c_1...c_n\}$ for each $a \in A$
$\forall a \in A$, $\forall c \in C^a$ Construct a copy of KWIK-DetEffectLearn (Algorithm 19) $KDE^a_c$ with
  $Pm^n$ literals (those in scope of $a$)
for each episode do
  for $s_t$ = current state do
    Construct a model interpreting any $KDE^a_c$ $\perp$ as an $R_{\max}$ transition (or use optimistic
      schemas)
    Perform Value Iteration using this model
    Execute the greedy action $a_t$, Observe $s_{t+1}$
    $c_t = c \in C^a$ such that $c(s_t) = true$
    Update $KDE^{a_t}_{c_t}(s_t, s_{t+1})$
  end for
end for
```

Theorem 9. *Algorithm 20 is PAC-MDP given conditions in domains encoded in deterministic STRIPS by KWIK-solving the ED-Learning problem for deterministic STRIPS schemas.*

Proof. The theorem follows from Lemma 5 and the KWIK-Rmax theorem because there are a bounded number of actions taken due to uncertainty in the model. Each such trajectory will lead to at least one observation that eliminates a $\perp$ prediction for at least one literal in the Add and Del arrays held by KWIK-DetEffectLearn. The total KWIK bound in this case is $O(ACPm^n)$ where C is the maximum number of conditions associated with an action. □

A small empirical demonstration of the algorithm is shown in Figure 3.5 for 4-Blocks World. The graph shows the average reward (over 10 trials) obtained by an agent using Algorithm 20

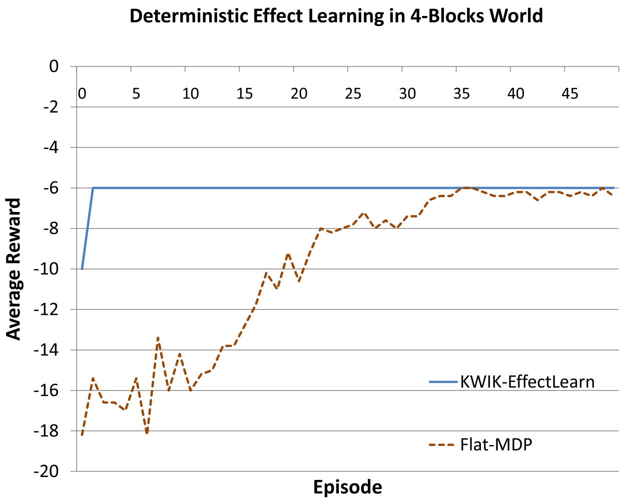


Figure 3.5: Learning deterministic effects in blocks world.

versus an agent that uses a flat MDP learner, which is unable to generalize across situations (not realizing that moving block **a** has anything to do with moving block **b**). As expected, the algorithm using relational action schemas learns all it needs to know about blocks world during the first episode while the flat MDP learner struggles for many episodes (each episode is capped at a step limit of 20) to uncover the true model.

3.9.3 Stochastic ED-Learning with "Signaled Effects"

We now begin transitioning to the stochastic ED-Learning problem as posed in noisy blocks world, or the difficult-to-learn Stochastic STRIPS rule in Table 3.5. Notice that the major difficulty for stochastic ED-Learning arises from effect ambiguity, which is complicated by the fact that the effects themselves are being learned. That is, for a given transition, not only may multiple real effects match this transition (as we saw in the D-Learning case), but because these effects may be partially learned, we may not be able to discern from a single example which effect should be updated. For instance, if we have seen A and B added at the same time, and then observe a transition from A to AB (only B was definitely added, though we can't tell for A), was that the same effect, or a different one?

In this section, we first consider the problem without this complication by assuming that an index number or other unique identifier of the effect that actually occurred is reported along with the next state, though we get rid of this restriction in the following sections. As an example,

ω_1: **Add**: B **Del**: C ($p_1 = 1/3$)
ω_2: **Add**: AB **Del**: D ($p_2 = 1/3$)
ω_3: **Add**: **Del**: C ($p_3 = 1/3$)

Table 3.5: A difficult set of Add and Delete lists to learn in the stochastic case (with simple propositional literals).

consider the noisy blocks world with the modification that if the *pickup* action actually picks up a block a 1 is reported (along with the next state showing the block in hand), and if the action instead resulted in no change, a 2 is reported along with the new state. This indicator will allow us to KWIK-learn the effects because we are no longer unsure how to "pair" effects from different samples (deciding how to group literals that were say, added on different timesteps and might belong to the same effect bin). Note that this extra amount of information also makes the distribution learning part of the ED-Learning problem easier to solve, though for different reasons. The new observations do not preclude 2 effects from being able to explain the same transition, but there can be no ambiguity at any given timestep because the actual effect index is reported. Hence, KWIK-LR can be replaced by a simple counting algorithm (the "KWIK dice-learner" from Table 2.1) in this case.

Under these conditions, we can use the following algorithm (Algorithm 21) with component KWIK learners for learning the effects and the probabilities. Intuitively, it just uses the deterministic effect learning algorithm to learn the Add and Delete lists for each indexed effect, and uses a simple KWIK dice-learning algorithm to determine the probabilities of each effect-set multinomial. Formally, it has the following property.

Algorithm 21 Stochastic "Signaled Effect" Conditional ED-OnlineLearn

```
Given: Action set $A$ of max arity $m$, Predicate set $P$ of max-arity $n$, Conjunctions for
  conditions $C^a = \{c_1 ... c_{|C^a|}\}$ for each $a \in A$, and a maximum number of effects per condition
  $|\Omega|$ with unique signals.
$\forall a \in A,\ \forall c \in C^a, w \in [1...|\Omega|]$ Construct a copy of KWIK-DetEffectLearn (Algorithm 19)
  $KDE^a_{cw}$ with $Pm^n$ literals (those in scope of $a$)
$\forall a \in A,\ \forall c \in C^a$ Construct a "dice-learner" $D^a_c$ to learn the effect probabilities
for each episode do
  for $s_t$ = current state do
    If any of the KWIK learners have changed, construct a model interpreting any $KDE^a_{cw}\perp$
      where $D^a_c(w) \neq 0$ or $D^a_c = \perp$ as an $R_{\max}$ transition
    Perform Value Iteration using this model
    Execute the greedy action $a_t$, Observe $s_{t+1}$ and effect index $i$.
    $c_t = c \in C^a$ such that $c(s_t) = true$
    Update $KDE^{a_t}_{c_t i}(s_t, s_{t+1})$ and $D^{a_t}_{c_t}(i)$
  end for
end for
```

Theorem 10. *Algorithm 21 is PAC-MDP and KWIK-solves the ED-Learning problem in the "signaled effect" setting where each $\omega \in \Omega_c^a$, has a unique index that is reported whenever that effect occurs.*

Proof. (sketch) The overall KWIK bound for this algorithm is $O(CA\Omega(Pm^n + \frac{1}{\epsilon^2}\log\frac{\Omega}{\delta})$. The first term comes from the effect learning component and the second from the dice-learning bound (see Table 2.1). Special consideration has to be given in this case to effects that never happen because we use the maximal number of effects to set up the learners, but there may be fewer than this number of effects under a given condition (for example, some conditions may produce deterministic effects while others have a distribution over 2 or more). This situation is taken care of in Line 6, which halts exploration due to the effect-learning component if the dice learner predicts probability 0 for this effect. The PAC-MDP property comes from the KWIK bound and the application of the KWIK-Rmax architecture. □

Unfortunately, most of the problems considered in this thesis, and most in the literature, do not fit this profile. We now consider the more general case, where such an indicator is not provided.

3.9.4 Negative Results for ED-Learning: Flawed Approaches

Turning our attention to the case of $k = O(1)$ general stochastic effects, we first describe a negative result on computing minimal action schemas from incomplete data. We then provide an example of the hazards of using a "greedy" approach to construct learned effects. These negative results are tempered in the next section, which shows how to KWIK-learn k stochastic effects in a non-greedy fashion and without constructing full schemas from incomplete data.

The Intractability of Computing Minimal Effects from Incomplete Data

We start by showing the difficulty of learning stochastic action schema effects in the batch setting from a log of data (transitions) that does not uniquely induce an action schema. This result will show that the *computation* required to infer the minimal effects from a log of experience data makes the problem NP-Hard. While others (Pasula et al., 2007) have reported similar results, the proofs are not widely disseminated, so we provide our own here. This result does not directly imply that we cannot KWIK-learn stochastic effects (in fact we show the opposite for $k = O(1)$ effects later), but provides good intuition on the difficulty of the problem. Formally, the problem we consider here is: given a set of M "experience data points" for an unconditional Stochastic STRIPS action a, where each piece of data is a tuple $\langle s, s' \rangle$ showing the state of the domain before and after the action, determine whether there is an effect-set Ω^a of size k such that every experience tuple is explained by at least one $\omega \in \Omega^a$ $(\eta(s, \omega) = s')$.

Theorem 11. *The ED-Learning problem for Stochastic STRIPS rules in the batch setting, with a log of M experience tuples for an action, is NP-Hard when the number of effects is larger than 2.*

Proof. The proof is by a reduction from graph coloring. We begin by defining some notation. As per *operator update rules* 3-6 from Section 3.4.1, every experience tuple indicates one of four transformations for every literal l in the state description, which we denote with four symbols (1,0,*,#), standing for (literal was added, literal was deleted, literal was already true and not deleted, and literal was already false and not added). Thus, given an ordering over the literals, we can write every experience tuple in terms of these four symbols. For instance, in a propositional domain instance with literals A,B,C,D, and E, the transition AB → ACE can be written as *01#1.

The k graph coloring problem, which we do the reduction from, is defined as: given a graph $\langle V, E \rangle$ with vertices V and edges $\langle v_i, v_j \rangle \in E$, determine if there is a mapping $\kappa : V \mapsto [1...k]$ (a "coloring" of the vertices) such that no two adjacent vertices have the same color (formally, $\langle v_i, v_j \rangle \in E, \rightarrow \kappa(v_i) \neq \kappa(v_j)$). This problem is known to be NP-Complete (Garey & Johnson, 1990).

The reduction from graph coloring to batch ED-Learning goes as follows. We will construct a log of $M = |V|$ experience tuples for a domain instance with $|V|$ literals. Each experience tuple et_i has a 1 in position i. All other positions in the tuple are filled with one of two symbols. If $\langle v_i, v_j \rangle \in E$, then position j is set to #, otherwise it is set to *. The interpretation of each experience tuple is that literal l_i definitely is added in this instance, and several other literals are shown not to be added with it (those that had edges in the graph coloring instance), and the others may or may not be in the Add list with l_i that generated this instance (*s).

All that needs to be shown now is that a valid solution to the constructed ED-Learning problem answers the original graph coloring problem and vice versa. This connection is completed by interpreting each inferred effect $\omega_i \in \Omega$ as a unique color $[1...k]$. A color k_i is assigned to vertex v_j if experience tuple $\langle s, s' \rangle_j$ can be explained by applying ω_i to s $(\eta(s, \omega_i) = s')$. If multiple effects could explain the transition, v_j can be assigned any of the corresponding colors.

This coloring will be consistent because each edge produced a 1/# pairing in experience tuple j, so no effect that explains this transition can apply to both vertices (because it would have to both add and not add each of the literals). In the other direction, if we have a solution to the graph coloring problem, and instances for ED-Learning constructed as described above, we can simply classify each experience tuple with effect ω_i corresponding to the color $i \in [1...k]$. These effects must be valid because each instance has 1 in position i, a # in any position corresponding to a neighbor of i, and a $*$ (which doesn't conflict with any other instance) in all other positions. □

While the previous theorem provides good intuition as to the intractability of learning effects, it only gives a negative result on the *computational* complexity of solving the ED-Learning problem with a specific strategy (constructing minimal effects from insufficient batch data). We now show that even if this computation is ignored, a greedy approach to learning stochastic effects (one that considers only a single minimal model) can falter. We will then show that a more careful algorithm that makes better use of the probabilistic structure and waits to commit to a model can efficiently solve the ED-Learning problem.

Problems for Greedy Approaches

While the result above shows the intractability of computing the minimal model from data when there are not enough samples to build probabilities of the effects, one may wonder how a "greedy" algorithm might fare. That is, consider a procedure that builds a single possible set of Add and Delete lists from available data (using an NP-oracle to do the graph coloring / effect inference) and then tries to learn the probability of these lists until evidence showed that the lists were wrong. Unfortunately, the following example shows such a strategy can lead the agent into a deadlock situation. The example uses the 3 real effects in Table 3.5 from earlier.

We will use the notation from the proof of Theorem 11, that is the four characters (1,0,*,#), standing for literal was added (1), literal was deleted (0), literal was already true and not deleted (*), and literal was already false and not added (#). Suppose the agent has collected the following samples from that operator and then chooses a "coloring" of them (grouping them into possible effect sets):

1 * * # (red)

* 1 * 0 (red)

* 1 # # (blue)

* * 0 * (blue)

1 * # # (green)

* # # * (green)

This experience could induce the following effects (although several interpretations are possible, they can all be made to fail in similar ways):

ω_1: **Add**: AB **Del**: D (Scapegoat)

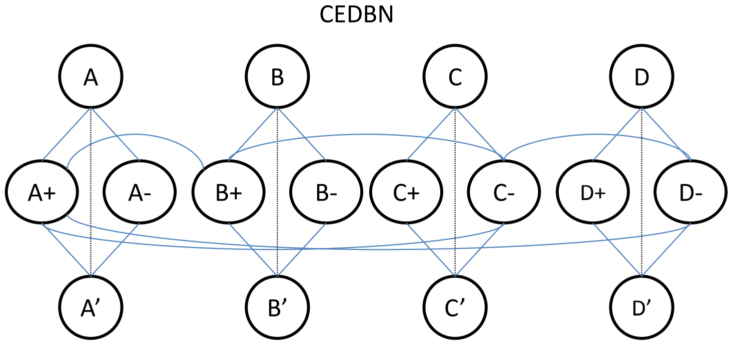
ω_2: **Add**: B **Del**: C

ω_3: **Add**: A **Del**: C (Wanted)

Now suppose the agent starts in a state where only B and D are true, and the goal is to get to a state where A and D are true. Unfortunately, outcomes from BD (assuming there's another action for resetting) will always be either "nothing happens" (the first and third real effects: B is added and C is deleted or just C is deleted), which will be accredited to the second learned bin, or A is added and D is deleted (blamed on first learned bin, the scapegoat). The agent may think that the third learned bin (labeled "wanted") will get it to the goal, but this outcome will never happen and yet it will never see single-step evidence from this state that the effects it has created are definitely wrong. Thus, using only a single hypothesis on Add and Delete lists before sufficient statistical evidence is collected can lead to a deadlock that requires backtracking. We now introduce an algorithm that avoids the pitfalls by making use of the special structure of the effect probabilities in the $|\Omega| = k = O(1)$ case.

3.9.5 Stochastic ED-Learning with only k Effects

So far, we have a positive result for the minimal ED-Learning problem when $|\Omega| = 1$ (deterministic), and for a benign (signaled effect) version of the stochastic effect case. But now we consider the case where the number of effects is greater than 1, but still small, specifically $|\Omega| = k = O(1)$. We will introduce an algorithm (KWIK-CorEffect, Algorithm 23) that KWIK-learns these dynamics, but because of the difficulties encountered in the negative results above,



Actual Operators

Add: B Del: C (1/3)
Add: AB Del: D (1/3)
Add: nil Del: C (1/3)

CPTs for A+, B+, C- and D-

key	Pr[A+=1]
A+ = 1	1/3

key	Pr[B+ =1]
A+ =1	1
A+ = 0	1/2
A+ = */#	2/3

key	Pr[C- =1]
A+ =1	0
A+ = 0	1
B+ = 1	1/2
B + = 0	1

key	Pr[D- =1]
A+ =1	1
A+ = 0	0
B+ = 1	1/2
B + = 0	0
C- = 1	1
C - = 0	0

B+=1,
C-=0
	1

B+=1,
C-=1
	0

Figure 3.6: Coupled effects represented using a Cross-Edged Dynamic Bayes Net (CEDBN). Intermediate nodes have CPTs up to k parents. For any given time-step, half the intermediate nodes are */# (will not affect the next state) and the others have values of 0 or 1. Dependencies flow down and to the right.

we will be using a much different internal representation than considered in the rest of this thesis. Essentially, because of the intractability of computing and maintaining a single Stochastic STRIPS model during learning, and because KWIK learners are only tasked with the problem of making predictions (not particularly outputting a specific model), we now introduce a graphical model for Stochastic STRIPS effects that leverages the probabilistic structure to compactly represent the valid hypotheses. We will use this model in subsequent sections to KWIK-solve the ED-Learning problem.

The Cross Edged Dynamic Bayes Net

The model we introduce for representing Stochastic STRIPS effects is a Cross Edged Dynamic Bayes Net (CEDBN) as illustrated in Figure 3.6 for the (difficult to learn) effects from Table 3.5. The use of similar graphical models for inducing probabilities of next states from relational operators has precedent in the planning community (Lang & Toussaint, 2009), but here we use this newly defined structure to make the learning process tractable.

A CEDBN for an action represents a transition with the following structure. Like a standard DBN, it has nodes for each factor's value in the state before and after the action executes (the top and bottom levels in the figure). While these are labeled with propositional factors in our diagram (for readability), in STRIPS each one of these nodes would correspond to a literal that is potentially affected by the action (such as *On*(X, From)), and their values are simply 1 or 0 depending on whether the literal is true or not. However, we will need an extra set of nodes to capture the dynamics of the Add and Delete lists as well as correlations between the factors.

To see why we need to capture such correlations, notice that in the simple operator in Figure 3.6, if we know that A is added, then we can guarantee that B is also added (since A is only added in ω_2). These correlations are captured in a CEDBN by introducing $2n$ (for n literals) linked *intermediate* nodes (the middle layer in Figure 3.6), which store the probabilities (in Conditional Probability Tables (CPTs), described later) for each literal being added or deleted (the A^+ and A^- nodes, respectively). The *cross-edges* within this intermediate layer encode statistical dependencies between these literals and an *ordering* (in our example, simply alphabetical) is placed on these intermediate nodes such that dependencies only flow one way. For instance, the link between A^+ and B^+ indicates that there is a statistical correlation (in this case deterministic) from A being added to B being added. We note that a similar structure has been used to capture correlations between factors in prior work (Hoey et al., 1999).

The values permissible for each intermediate node X^+ are a 1 (which indicates that literal was definitely added as X was 0 in the pre-state), or 0 if X was definitely not added (X was 0 in the pre-state and will remain so). Similar values are allowed for X^-. However, we also need to encode the transitions associated with the $*$ and $\#$ markers from the earlier section indicating the cases where we can't tell if X was added or deleted. Note that one of $\langle X^+, X^- \rangle$ must have this ambiguity in every state. Thus, we allow a third value for an intermediate node, labeled $*/\#$, indicating this ambiguity and we will later see that nodes with this value can actually be ignored when making next state predictions (because the probability of adding a literal that is already true is immaterial). The semantics of the intermediate node values with respect to the next state literals is simple once their values are known. If the intermediate node is labeled $*/\#$, it has no effect on the current value of literal X, but its "sister" node ($+$ or $-$) must be consulted. If it is labeled 1, then the literal X will have its value flipped (either off or on) and if it has a value of 0, then X will definitely maintain its current value. Since the $*/\#$ values are directly observable from the current state of X, the CPTs for X only needs to store the probability of X^+ (similarly X^-) having the value 0 or 1, but the number of entries in the CPT is dependent on the parent nodes (other intermediate literals earlier in the ordering). Note, if we could compactly capture the CPTs for the values of the intermediate nodes, we would have the full dynamics of the model.

In a standard DBN, the number of entries (rows) in each node's CPT is exponential in the in-degree of the node, because each different parent configuration could induce a different distribution. When this in-degree is $O(1)$, this learning is still tractable in the KWIK setting (Diuk et al., 2009), but a naïve interpretation of a CEDBN does not conform to this assumption, because a literal at the end of the ordering can have a statistical dependence on all the other literals (for instance D in the diagram). However, in the next section we show that only $k = O(1)$ of these edges actually need to be considered at any specific timestep (so only $O(n^k)$ entries are needed) when making a next-state prediction and subsequently bound the size of sufficient CPTs for this task. To see how this compaction can be done, and as a glimpse of how this model is useful in KWIK-learning, we first show how a fully instantiated CEDBN (completely learned, with no $\perp$ entries) with polynomial size CPTs can be used to predict a next state.

Next State Prediction with CEDBNs

Suppose we are given a current state and an action (with $|\Omega^a| = k = O(1)$) that can affect a set of n literals (as with the scoping of STRIPS actions by their parameters) and asked to predict a distribution over next states in the unconditional (ED) case. Suppose we also have a fully instantiated CEDBN as described above. We will now show that CPTs of size $O(n^k)$ for each literal are sufficient for a prediction algorithm to predict the next state distributions using the corresponding CEDBN. Specifically, the CPTs will have one row for every combination of "choice" literals that are ordered before it, that is every earlier literal that may have had a probabilistic prediction (and therefore made us consider at least one extra "effect bin"). The prediction procedure is laid out in Algorithm 22.

Intuitively, the algorithm starts with only a single effect (that says nothing is added and deleted) with probability 1.0 and an empty "key". The algorithm then looks at each intermediate node for each literal (in the given order) that might be added or deleted, and refines the possible effects, probabilities, and keys based on some of the intermediate nodes' CPT entries. If the intermediate node would not make a difference (line 9) (and isn't even observable) due to the current state configuration (A^+ is meaningless if A is already true), it sets the node's value to $*/\#$ and moves to the next intermediate node. Otherwise, the probability of that intermediate node having value 1 in each possible effect constructed so far is considered. Notice that the entry in this node's CPT is indexed using the *key* for that effect, not the effect itself. If the effect's key deterministically (probability 1.0) asserts the node will have value 1, the corresponding literal is added to the effect's Add or Delete list (depending on if it is X^+ or X^-). For instance, in the example if an effect has $A^+ = 1$ in its key, then $B^+ = 1$ with probability 1, so it is added to that effect. *However*, in that case the *key* for that effect is not changed, because the keys are meant to contain only a small amount of information necessary for differentiating this effect from others, and since the earlier literals deterministically determine this one, putting it in the key would be redundant.

The other case that needs to be considered is when the intermediate node has a non-zero (but non-deterministic) probability of taking on value 1. In that case, we have found a "branch" node, because it splits at least 2 effects that were aliased by the values of the previous intermediate nodes. For instance, in our example if $s[A] = 0$, the very first node A^+ has a 1/3 chance of having value 1.0 because there are no other literals in the order before it. Therefore, the initial (empty) effect has to be split into two cases, one where A is added, and one where it is not.

Algorithm 22 PredictState(s, L, M)

```
*Input*: Current state $s$, Ordered Literals $L = [l_1 ... l_n]$, CEDBN $M$ with CPT entries for each intermediate node $l_i^+$ for every $\binom{n}{k}$ $\{0/1\}$ values for intermediate $l_j$ where $j < i$
*Output*: The possible next states (maximum $k$) and their probabilities
$\omega_1 = \emptyset$
$\widehat{\Omega} = \{\omega_1\}$
$key(\omega_1) = []$
$\Pi[\omega_1] = 1.0$
**for** each intermediate node $x = l^+$ (or $l^-$) $\in L$ (in order) **do**
  // If its value should be $*/\#$, then mark it as such, move on
  **if** $x = l^+ \wedge s[l] = 1 \bigvee x = l^- \wedge s[l] = 0$ **then**
    Set $x$ to $*/\#$.
    NEXT $x$ //move on to next literal
  **end if**
  //Otherwise, for each effect constructed so far
  **for** $\omega_i \in \widehat{\Omega}$ **do**
    **if** $Pr_M[X = 1|key(\omega_i)] = \perp$ **then**
      Return $\perp$
    **else if** $Pr_M[X = 1|key(\omega_i)] = 1.0$ **then**
      //if deterministic and 1, place it in the effect, on to the next one
      Add $X$ (Add or Delete) to $\omega_i$
    **else if** $Pr_M[X = 1|key(\omega_i)] > 0.0$ **then**
      //If non-deterministic, split the two effects, update their keys and probabilities
      Construct $\omega_j$, $j = |\widehat{\Omega}| + 1$ with $\omega_j = \omega_i$
      $\Pi[\omega_j] = \Pi(\omega_i)$
      //Effect $\omega_j$ has $X$ added or deleted, $\omega_i$ does not
      Add $X$ (Add or Delete) to $\omega_j$
      //Update the corresponding keys
      Add $X = 0$ to $key(\omega_i)$
      Add $X = 1$ to $key(\omega_j)$
      //Use $X$'s CPT entry to update the two "split" effects' probabilities
      $\Pi(\omega_i)$*=$Pr_M[X = 0|key(\omega_i)]$
      $\Pi(\omega_j)$*=$Pr_M[X = 1|key(\omega_i)]$
      $\widehat{\Omega} \bigcup = \omega_j$
    **end if**
  **end for**
**end for**
$S' = []$
**for** each $\omega \in \widehat{\Omega}$ **do**
  Add $\eta(L, \omega)$ to $S'$
**end for**
return $\langle S', \widehat{\Pi} \rangle$
```

The algorithm accounts for this by "splitting" the effect it was considering into two effects and using the corresponding intermediate node to change the Add (or Delete) list of the new effects. Otherwise the two effects are identical for now. It also updates the probabilities of each bin based on the CPT entries at this node (for values 1 and 0) and updates the keys for both of these effects by adding in a reference to this intermediate node and its value (0 for one bin, 1 for the other). In summary, when considering just A^+, we would have created two possible effects (also two different next states): either A is added (next state $= \{A\}$), or not (next state $= \{\}$).

To see how such partial effects are expanded when considering more than one literal, consider the two partial effects just described. Suppose $s[B] = 0$, so we have to consider B^+ in both of the effects created in the first step ($\{A\}$ and $\{\}$). The first partial effect simply corresponds to the deterministic case for B^+ described above, so that effect is expanded to $\{A, B\}$ but the key remains simply $A^+ = 1$ and the probability remains 1/3. But the second case (the empty next state with $A^+ = 0$) can have B^+ being 1 or 0, so this effect is again split into $\{\}$ and $\{B\}$, with keys $\langle A^+ = 0, B^+ = 0\rangle$ and $\langle A^+ = 0, B^+ = 1\rangle$ respectively. The CPT entry for B^+ for the key $\{\}$ is 1/2, so the original effect's probability (2/3), is multiplied by 1/2 in both cases (since $1 - 1/2 = 1/2$), and we end up with 3 possible next states ($\{A, B\}$, $\{\}$, $\{B\}$), each with probability 1/3 . Notice that each of these will induce a deterministic value for the intermediate nodes corresponding to the deletion of C and D, so we can just add those in to get the true effects. The splitting of the effects described above is illustrated in Figure 3.7, which shows the progression of the effects and their keys as a succession of decision trees based on the non-deterministic intermediate nodes. Notice that the keys of the effects contain only the non-deterministic intermediate nodes that actually discriminate between effects given all the earlier intermediate node values. In fact, we can formalize this property in the following lemma.

Lemma 6. *Algorithm 22 never creates a key with more than than $k - 1$ values in it, and each key maps to an effect or a set of aliased effects, as determined by the given state s.*

Proof. We can prove this feature by induction on $k'_s = |\eta(s, \Omega^a)|$ for the CEDBN's action a. Intuitively, k'_s is the number of *discernible* effects of an operator from a state s. For $k'_s = 1$ (all effects look the same), a deterministic transition is ensured and no intermediate nodes will have non-deterministic CPT entries (otherwise there would be more than one effect), so only the initial (empty) key will exist, with size 0. We make the inductive hypothesis that for $k'_s = k-1$, each key can have no more than $k - 2$ values in it, and each key maps to the exact effects engendered by s and the Add and Delete lists. Now suppose we encounter a state with $k'_s = k$

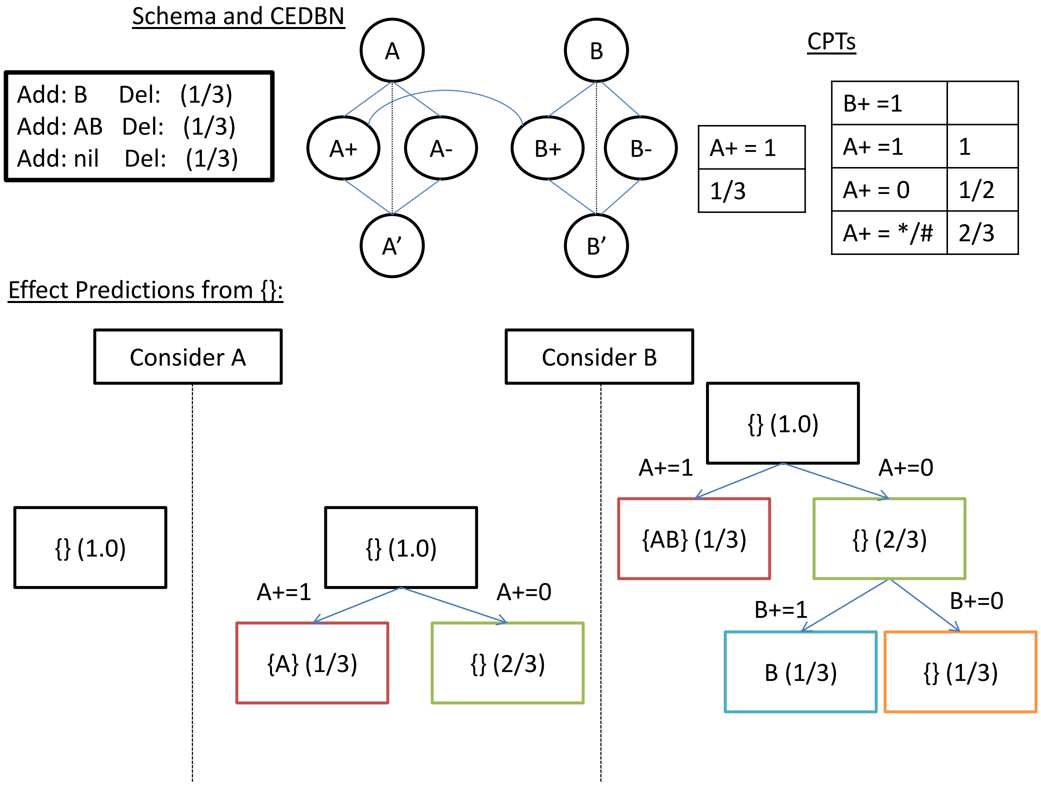


Figure 3.7: The predicted effects from a state of $\{\emptyset\}$ for a simple stochastic STRIPS schema/CEDBN with different combination of A and B (or nothing) being added. The effects are shown being constructed as decision trees. Effects themselves are in the leaves, keys are represented as paths from the root to a leaf, and splits occur anywhere a non-deterministic CEDBN entry is encountered. If the version of the schemas with C and D being deleted was used, these literals would appear in the leaves of the final tree (no more leaves are necessary for 3 effects).

discernible effects (all the true effects lead to unique next states). Since the algorithm creates keys incrementally, there is some step where we consider literal X and there are $k'' < k'_s$ keys, of maximum size $k'_s - 2$, and where if we ignored literal X and all subsequent literals then by the inductive hypothesis each of the k'' keys map exactly to a real effect *or* are aliased because of X (otherwise the hypothesis would be invalidated when X was set to $*/\#$). But since all k effects are discernible from s, X will split the remaining aliased effects, and since this is only a single split on one literal, it can only create a key of max size $k - 1$. Future literals will be deterministically placed in effects. Thus, no more keys (or extensions of current keys) will be created. □

Intuitively, the property above can be interpreted as saying that each key is analogous to a branch in a decision tree of depth no more than k. The correctness of the algorithm with CPTs of size $O(n^k)$ (based on the number of possible keys) for making predictions is shown in the following lemma.

Lemma 7. *CPTs each of size $O(n^k)$ (with n literals and k effects) are sufficient for making correct predictions about next states using Algorithm 22.*

Proof. Lemma 6 established that each key can only have up to size $k - 1$ and since the CPTs are accessed only by these keys, each one can only reach size $\sum_{j=0}^{k} \binom{n}{j-1} * 2$, because each key could induce 2 distributions on literal X (if X causes a branching in the current state for that key). What is left to be shown is that the algorithm establishes the correct probabilities for each predicted next state when using these keys.

We do so by showing that each branching literal establishes the correct probability distribution over the currently constructed effects (ignoring the literals that come after X). Notice first that at each node/effect pair in the main loops, the CPT entry that matches the key *exactly* (not partially) is used to get that literal's probability of being added/deleted. To see why no other entry could be correct for the prediction procedure, consider the alternatives.

If a larger key (one that contains more branching literals than the current key) actually described the effect more precisely, that would be the current key, so we need not worry about larger keys. Also, two CPT keys of the same size could not have matched the effect, because they would be identical or a larger key would have been constructed already. Finally, if we used a smaller key (say replacing $\langle A^+ = 1, B^+ = 1\rangle$ with $\langle A^+ = 1\rangle$, this key would alias 2 (or more) effects that were already split, and we would be using the sum of the effects' probabilities for the current literal. So the smaller key (which is essentially a prediction for higher up in the tree) would induce the wrong probability distribution.

Now we turn our attention to the literal X in the context established by s and under the partial effect represented by the key. If the literal's probability is 1 or 0, then all the effects aliased by the key agree on its addition/deletion, so there is no need to change the probability. If it is non-deterministic, then the "split" of the two effects establishes new probabilities because the probability of the formerly aliased effects is the product of the previous CPT entries, so the multiplication performed computes a valid probability.

$\square$

KWIK Learning a CEDBN

We now show that it is possible to KWIK-learn the compact CEDBN used in the state prediction problem above for each action in the stochastic STRIPS setting with $k = O(1)$ effects. The algorithm we use to do so is KWIK-CorEffect (KWIK for Correlated Effects) as described in Algorithm 23.

Intuitively, this algorithm simply uses the earlier prediction algorithm to make next state predictions for the given action. But now, whenever an intermediate node CPT entry reports $\perp$, KWIK-CorEffect updates the model with a sample. However, just executing an action may not provide information about the specific $\perp$ entry that caused this prediction. This deficiency occurs because executing the action results in one of k effects, not necessarily the effect corresponding to the key for the unknown entry. And even if we get the right effect (or aliased effects), we still need multiple samples to learn the CPT entry. That is the qualitative explanation of the m parameter in the algorithm, and the shifting of extremely low probabilities (to avoid trying to learn about an effect that has an extremely low chance of occurring). Formally, we bound the number of $\perp$ predictions by establishing a value for m as follows:

Theorem 12. *KWIK-CorEffect can KWIK-learn a CEDBN in the ED-Learning setting for a Stochastic STRIPS action with n literals in its scope with only $\tilde{O}(n^{k+2}\frac{1}{\epsilon^4})$ $\perp$ predictions.*

Proof. We consider the (worst) case where, at a given timestep, there is only a single $\perp$ prediction from the last ordered intermediate node X^-, corresponding to two aliased effects (indexed by a key of maximum size $k-2$). This $\perp$ corresponds to the probability that literal X will be deleted given the parent values in the key (corresponding to 2 aliased effects). If we were guaranteed that each sample would match this key, then determining this probability (of the node

Algorithm 23 KWIK-CorEffect for a single action in Stochastic STRIPS

```
*Input*: Ordered Literals $L = [l_1 ... l_n]$ (based on the action's scope), and $k$, the number of effects
*Output*: The possible next states (maximum $k$) and probabilities, given each input state
Construct a CEDBN $\widehat{M}$ with CPT entries for each literal for every combination of $k$ or fewer literal values chosen from the literals earlier in the ordering
Initialize all CPT predictions in $\widehat{M}$ to $\perp$
Initialize $count(X, key) = 0$ for all intermediate nodes $X$'s CPT entries.
**for** each input state $s$ **do**
  $Prediction = PredictState(s, L, \widehat{M})$
  **if** $Prediction \neq \perp$ **then**
    Predict $Prediction$
  **else**
    //Update $\widehat{M}$
    execute the action, observe state $s'$
    **for** each intermediate node $X = l^+(orl^-) \in \widehat{M}$ (in order) **do**
      //If its value should be $*/\#$, then mark it as such, move on
      **if** $x = l^+ \wedge s[l] = 1 \bigvee x = l^- \wedge s[l] = 0$ **then**
        Set $X$ to $*/\#$.
        NEXT $X$
      **end if**
      **for** each maximum sized key in $X$'s CPT that matches $\langle s, s' \rangle$ **do**
        **if** $Pr_{\widehat{M}}[X = 1|key] = \perp$ **then**
          Update the estimated CPT entry based on $s[X]$ and $s'[X]$.
          $count(X, key) \mathrel{+}= 1$.
          **if** $count(X, key) > m$ **then**
            Use the maximum likelihood estimate for the CPT values with this key, instead of $\perp$
            **if** $Pr[X = 0|key] * \Pi_{X' \in key} Pr[X'|\text{earlier parts of key}] < \epsilon$ (or $> 1 - \epsilon$) **then**
              Set the new CPT entries to 0 (1).
            **end if**
          **end if**
        **end if**
      **end for**
    **end for**
  **end if**
**end for**
```

taking values 0 or 1) would be equivalent to learning the probability of a biased coin coming up heads/tails, for which we would need $m = O(\frac{n^2}{\epsilon^2} \log \frac{n^k}{\delta})$ samples by Hoeffding's inequality, with probability of failure $\frac{\delta}{n^k}$, and where the n^2 term appears because we require accuracy of $\frac{\epsilon}{n}$ in each node since their CPT entries are multiplied together to make next state predictions. However, it may be that only a subset of the k effects will match this key. For instance, in our running example, if we are trying to update node B^+'s entry for the key $A^+ = 1$, we need to get a sample where A is actually added (ω_2 must occur). Thus, we need to consider a second point of failure where we are trying to learn a specific CPT entry but the effect corresponding to its key is not occurring. However, line 25 guarantees us that the probability of the aliased effects that correspond to this key is $\geq \epsilon$. So in the worst case, we need to be able to receive at least 1 sample of an effect with probability $\geq \delta' = \frac{\delta}{2mn^k}$, where the extra 2 is introduced to both this term and the m term probability since we have doubled the number of failure points. A loose bound on this quantity is simply the number of samples needed to estimate a probability distribution on such an effect within an ϵ tolerance (since we will need to see at least one such sample of it) with probability δ', or by Hoeffding's inequality again: $O(\frac{1}{\epsilon^2} \log \frac{1}{\delta'})$.

So we will set m as described earlier with the caveat that each of the m samples needed for a CPT entry may actually require another $\tilde{O}(\frac{1}{\epsilon^2})$ real samples to see the effect matching this key. Finally, we must not try to achieve this count if the key for this entry has extremely low probability (because further disambiguation will not be helpful). In that case, where an effect has sufficiently small probability (line 25), the node's value may be characterized as deterministic (for instance $Pr[X^+ = 1|key] = 1.0$, when it really should have been 0.99), but this inaccuracy is acceptable given the ϵ and δ parameters and Definition 12, which requires only an ϵ-accurate prediction of the next-state distribution. This result gives us the bound on the number of samples needed to learn one of n^{k-1} CPT entries for each of the $2n$ intermediate nodes. Putting it all together, the algorithm has a loose KWIK bound of $\tilde{O}(\frac{n^{k+2}}{\epsilon^4})$ (with the rest hidden in log terms). PAC-MDP agent behavior follows by combining the KWIK algorithms for each action and using the KWIK-Rmax architecture (Algorithm 6). □

While this result holds for $|\Omega| = k = O(1)$, it is an open question as to whether these results can be extended beyond this number of possible effects. While we saw in Section 3.5 that D-Learning could be accomplished in the presence of a polynomial (in the number of literals) number of effects, a similar result does not appear to hold in ED-Learning, because the Bayes net representation would have exponentially large conditional probability tables.

3.9.6 Non-minimal Effect Learning Given Possible Effects

The previous theorems give a fairly complete picture of the *minimal* ED-Learning problem, where the agent must learn the smallest possible effect list for each condition. However, it is possible to efficiently learn effects and distributions if we give the agent a relatively small (polynomial) number of effects that might be possible for each action. That is, we could allow the agent to learn about a much larger set of effects than the minimal, say by declaring a constant bound ξ on the size of each individual effect (allowing us to enumerate a polynomial but likely non-minimal number of effects). In this setting, we can use the KWIK-LR algorithm from the D-Learning case to learn which effects do not happen, because their individual weights will be set to 0 through enough experience. More formally, we can state the following extension of the results from the D-Learning setting.

Corollary 2. *Algorithm 11 KWIK-solves the ED-Learning problem when we are given a polynomial-sized superset of possible effects or each effect is an Add/Delete pair of no more than ξ terms for constant ξ.*

Proof. The more specific case with the ξ bounded effects can be turned into the superset case by just enumerating every possible effect, a set of size $O(L^{\xi})$. The proof follows the same form as Lemma 2 and Theorem 2 from the D-Learning setting, just with the extra effects needing to be explored enough to set their probabilities to 0, KWIK-LR provides a valid solution with a polynomial sample complexity of $O\left(\max\left\{\frac{\Omega^3}{\epsilon^4}, \frac{n\log^2\frac{\Omega}{\delta}}{\epsilon^4}\right\}\right)$, where Ω is the size of the given superset of possible effects. □

This concludes our discussion of the individual sub-problems in action-schema learning and we now summarize the results and show how to bring them together to solve the full CED-Learning problem.

3.10 The full CED-Learning Problem

In this section we put all the pieces together and describe cases where the full CED-Learning problem is solvable in the online reinforcement learning setting (KWIK solvable). Table 3.6 summarizes the results for the 4 major learning problems we have covered in this chapter (including CED-Learning) in each of the 6 domain settings from Section 3.1. Below we explain these results in more detail, describing how to meld the architectures for the various sub-problems in different settings. For the cases where pre-conditions need to be learned, we assume here that they are small, and note when analogous algorithms can be developed in the conditional case when the *number* of conditions is small. The cases where full CED-Learning is tractable in the online-learning setting include the following:

- **Deterministic Unconditional**- When there are no conditions, the CED-Learning problem devolves into the ED-Learning problem, so Algorithm 20 (KWIK-DetEffectLearn) can be used to solve the unconditional CED-Learning problem.

- **Deterministic Pre-conditional**- With pre-conditions and deterministic effects, we can adapt the algorithm for CD-Learning from the *stochastic* case (Algorithm 14) to one that learns the conditions and effects by replacing KWIK-LR with KWIK-DetEffectLearn. Thus, KWIK-Enumeration learns the pre-conditions just as before, but now the effects themselves are learned by KWIK-DetEffectLearn. Notice the architecture is still a parallel one. The two KWIK learners operate fairly independently, linked only in the planner, which can use a KWIK-Rmax heuristic or mix KWIK-Rmax (for unknown pre-conditions) and Optimistic Add/Delete Lists while learning effects. The total KWIK bound here in the STRIPS setting is $O(A(Pm^n)^k)$ where k is the maximum size of any pre-condition.

- **Deterministic Conditional**- Here, we can adapt the *deterministic* CD-Learning algorithm for the conditional case (Algorithm 16), that used Active-Union over several static effect predictors. The change here is to swap out the static effect predictors for copies of KWIK-DetEffectLearn. Notice unlike the pre-conditional case, the architecture here is a serial one: A copy of KWIK-DetEffectLearn must be attached to each potential condition under the Active-Union umbrella. The total KWIK bound here for the case where the conditions ($|c_i^a|$) are small is $A((L^k - 1) + L^k 3L) = O(L^{k+1})$ where L in the STRIPS case is Pm^n. Bounds are also derivable when the number of conditions is known to be small using the tree-enumeration procedure described earlier.

- **Stochastic k-Effect**- In this case, we adapt the *stochastic* CD-Learners, either Algorithm 14 (Enumeration and KWIK-LR) or Algorithm 18 (Meteorologist over many copies of KWIK-LR), and replace KWIK-LR in both cases with the algorithm for the ED-Learning setting with k effects (Algorithm 23) which learns both the effects and distributions.

- **Stochastic Signaled-Effect**- This follows the same form as the Stochastic k-Effect case, but instead of swapping in Algorithm 23, we swap in the linked versions of KWIK-DetEffectLearn and CoinFlipping from Algorithm 21 to learn the effects and distributions.

Setting	D-Learning	ED-Learning	CD-Learning	CED-Learning
Deterministic Unconditional	Given	DetEffectLearn	Given	DetEffectLearn
Stochastic Unconditional	KWIK-LR	KWIK-CorEffect$\star$	KWIK-LR	KWIK-CorEffect$\star$
Deterministic Pre-conditional	Given	DetEffectLearn	Enumeration*	Enumeration* and DetEffectLearn (in parallel)
Deterministic Conditional	Given	DetEffectLearn	Enumeration*	Union* and DetEffectlearn (multiple copies)
Stochastic Pre-Conditional	KWIK-LR	KWIK-CorEffect$\star$	Enumeration* and KWIK-LR (in parallel)	Enumeration* and KWIK-CorEffect$\star$ (in parallel)
Stochastic Conditional	KWIK-LR	KWIK-CorEffect$\star$	Meteorologist* and KWIK-LR (multiple copies)	Meteorologist* and KWIK-CorEffect$\star$ (multiple copies)

Table 3.6: The 6 learning settings and their KWIK solutions for the autonomous (online) case. A * denotes necessary assumptions about the size of conjunctive (pre)-conditions (or other formulas that are only KWIK-learnable when they are small). A $\star$ indicates the number of effects must be bounded by a constant k or some other special structure (like signaled effects) must be known.

- **Stochastic with a Superset of Effects-** In this case, since the effects are essentially given, and dealt with using KWIK-LR, we can use the algorithm from the CD-Learning cases (pre-conditional or conditional) with the larger effect set. The bounds are the same as the CD-Learning cases, but with the larger Ω.

One important point is that all the situations that involved learning conjunctions (either preconditions or conditions) required an assumption on the size of these conjunctions, specifically that the number of terms in each conjunction be $O(1)$. While this sufficed for several domains we considered in this chapter, many were right on the brink of tractability (specifically the Blocks World domains, which, depending on the encoding, can have 4 or 5 terms in the pre-conditions). But, the negative result in Theorem 5 shows this is an inherent limitation of the KWIK-learning paradigm, and likely of online learning itself. One has to try all the combinations on the lock. In the next chapter, we consider a different learning paradigm, apprenticeship learning, where these limitations will be absolved and thus larger domains will be learnable in a sample-efficient manner.

Chapter 4

Apprenticeship Learning of Action Schemas

The previous chapter presented results on the sample efficiency of learning action schemas in the online RL setting where an agent's interaction with its environment is limited to its own experience. While we were able to derive many positive results in this *autonomous* setting, there were necessary limitations concerning the size of the pre-conditions (c_i^a), specifically that the number of terms in each c_i^a was $O(1)$. Even when this size restriction holds, blindly enumerating all the possible conditions can lead to slow learning in practice. For instance, even the rather benign Paint-Polish domain, with maximum condition size of 3 (for the *Done* action) has $5^3 = 625$ possible conditions up to that size. Even worse, many domains may have the size of their conditions increase linearly with the number of available measurable predicates.

In this chapter, we consider a different RL setting *apprenticeship learning* where the $O(1)$ condition size restriction can be relaxed.[1] To do so, we will introduce a generalized definition and protocol for apprenticeship learning, as well as a new model-learning framework, Mistake Bound Predictor (MBP), which we will show is sufficient for efficient learning in this setting. We will link MBP to the MB and KWIK frameworks introduced in Section 2.2 and show that the class of learnable functions in this framework is strictly larger than what can be learned in either framework individually. With these results in hand, we will analyze versions of the CD-Learning problem in the apprenticeship setting.

4.1 Apprenticeship Learning: An Alternative Learning Protocol

Evidence in psychology (Shafto & Goodman, 2008) indicates that the availability of teachers unquestionably increase the speed and efficacy of learning in humans. Yet in the field of reinforcement learning (RL), which in many ways tries to mimic the human learning process, almost all learning agents gain experience solely by interaction with their environment—teachers are

[1] Portions of this chapter appeared earlier in joint work with Michael Littman, Kaushik Subramanian, and Carlos Diuk (Walsh & Littman, 2008; Walsh et al., 2010b).

not in the loop. In this section, we address this disconnect by proposing a generalized protocol for *apprenticeship learning* within the reinforcement-learning paradigm.

4.1.1 The Apprenticeship Learning Protocol

In contrast to the autonomous learning framework, we now consider a paradigm where the agent's experience is augmented with experience produced by a teacher and the criterion is to find a policy whose value function is nearly as good as, or better than, the value function induced by the teacher's policy. Previous work has used a number of different definitions and protocols for incorporating teachers into reinforcement learning. These include agents that attempt to learn behavior without observing rewards (Abbeel & Ng, 2004), agents that receive a log of teacher traces (including rewards) beforehand (Abbeel & Ng, 2005; Lin, 1991), those that have access to a teacher performing in each episodic domain instance they encounter (Khardon, 1999b), and those where a trace is actively requested (Walsh & Littman, 2008). Here, we define a generalized apprenticeship learning setting that is similar to the standard reinforcement learning protocol, but still adheres to the basic guidelines of these earlier approaches. Formally, we define the *Apprenticeship Learning Protocol* for episodic domains where each episode has a length of at most $H = Poly(|M|, |A|, R_{\max}, \frac{1}{1-\gamma})$ in Algorithm 24, where $|M|$ is a measure of environment complexity, as in the definition of PAC-MDP (Definition 4).

Algorithm 24 The Apprenticeship-Learning Protocol

```
The agent starts with $S$, $A$ and $\gamma$, a time-cap $H$ and has access to episodic environment $E$
The teacher has policy $\pi_T$.
for each new start state $s_0$ from $E$ do
  $t = 0$
  while The episode has not ended and $t < H$ do
    The agent chooses $a_t$.
    $\langle s_{t+1}, r_t \rangle = E.progress(s_t, a_t)$
    $t = t + 1$
  end while
  if the teacher believes it has a better policy for that episode then
    The teacher provides a trace $\mathcal{T}$ starting from $s_0$.
  end if
end for
```

Intuitively, the agent is allowed to interact with the environment, online just as in the standard RL context. But, unlike standard RL, at the end of an episode, the teacher can provide the agent with a trace of its own behavior starting from the original start state. The criteria the teacher uses to decide when to send a trace is left general here, but one specific test that has many desirable properties, is for the teacher to provide a trace if at any time t in the episode, $Q^{\pi_T}(s_t, a_t) < Q^{\pi_T}(s_t, \pi_T(s_t)) - \epsilon$. That is, the agent chooses an action that appears worse than the teacher's choice in the same state. Traces are of the form:

$$\mathcal{T} = (s_0, a_0, r_0), ...(s_t, a_t, r_t), ...(s_g, r_g)$$

where s_0 is the initial state, and s_g is a terminal (goal) state or some other state if the H cutoff is reached. The individual transitions in this trace must adhere to the environment's dynamics (the teacher cannot unilaterally pick next states). Notice that the trajectory begins before (or at) the point where the agent first acted sub-optimally, and may not even contain the state in which the agent made its mistake. This "single trajectory" approach is both practically necessary for domains without random access to states, and has characteristics essential for the trace to facilitate efficient learning.

Since the teacher's policy may not be optimal, this trace could potentially prescribe behavior worse than the agent's policy. We distinguish between these traces and their more helpful brethren with the following definition of *Valid Traces.*

Definition 17. *A **valid trace** (with accuracy ϵ) is a trace supplied by a teacher executing policy π_T delivered to an agent who just executed policy π_A starting from state s_0 such that $V^{\pi_T}(s_0) - \epsilon > V^{\pi_A}(s_0)$.*

Defining valid traces in this way allows agents to outperform their teacher without being punished for it. With deterministic policies, this definition means that at no time in a valid trace does the teacher prescribe an action that is much worse than any of the actions the agent used in that state. Note that when the teacher enacts optimal behavior ($\pi_T = \pi^*$) and the agent acts suboptimally, only valid traces will be provided. But if the teacher is using a suboptimal policy, it may provide many invalid traces, which the agent must eventually ignore. Either way, we will show that the number of times a teacher can provide a valid trace in many domains is bounded.

Specifically, we characterize the *efficiency* of learning analogous to the way the PAC-MDP framework (Definition 4) is used to characterize efficient behavior in the autonomous setting. We define PAC-MDP-Trace learning as follows:

Definition 18. *A reinforcement-learning agent is said to be* PAC-MDP-Trace *if, given accuracy parameters ϵ and δ, and following the protocol outlined in Algorithm 24, the number of valid traces (with accuracy ϵ) received by the agent over its lifetime is bounded by*

Poly$(|M|, |A|, R_{\max}, \frac{1}{1-\gamma})$ *with probability* $1 - \delta$*, Here,* $R_{\max}$ *is the maximum reward in the environment and where* $|M|$ *measures the complexity of the MDP's representation, specifically the description of* T *and* R*.*

This definition allows us to characterize the efficiency of an agent based on its behavior.

4.1.2 Related Work

Before going forward, we review a number of other works that have used similar apprenticeship learning frameworks and those that employed traces in the learning of action schemas. The idea of integrating a teacher into the learning process has been proposed in several different forms. In supervised learning, the labels of datapoints can be considered to originate from a teacher, but these labels are more an indirect communication—they do not necessarily correct an agent's mistakes. A more direct channel was proposed in the early computational learning theory literature, *equivalence queries* (Angluin, 1988) and later "teaching sequences" (Goldman & Kearns, 1992). While we produce some results linked to these as part of our study of model learning, our main concern is with apprenticeship learning in the RL setting.

Within RL, a number of different protocols and complexity measures have been described for apprenticeship learning. The one used in Algorithm 24 is an extension of the interaction described by Abbeel & Ng (2005) with the change that teachers no longer have to give all of their traces up front (making it similar also to the one used by Khardon (1999b)). Several other areas of RL with access to demonstrations have similarities to our problem setting but are ultimately different. The field of Inverse Reinforcement Learning (Abbeel & Ng, 2004; Syed & Shapire, 2007), also sometimes called apprenticeship learning, attempts to learn an agent's reward function by observing a sequence of actions (not rewards) taken by a teacher. In contrast to this interaction, our learners actually see samples of the transitions *and* rewards collected by the teacher and use this "experience" in a traditional model-based RL fashion. Recent work in Imitation Learning (Ratliff et al., 2006) took MDP instances and trajectories and tried to generalize these behaviors based on assumptions about the linearity of costs with respect to the feature space. Our results do not rely on any such relation.

The use of apprenticeship learning and traces in relational MDPs, and actions schemas in general, has many precedents, with a number of systems showcasing the empirical benefit of traces and teachers without laying out the theoretical foundations and provable separation from the online case that we do here. One of the earliest action schema learners, EXPO (Gil, 1994), was bootstrapped by an incomplete STRIPS-like domain description (missing some preconditions or effects of actions) with the rest being filled in through experience. This background knowledge can be viewed from our perspective as traces given prior to any autonomous exploration, essentially biasing the agent's model based on the teacher's experience. The OBSERVER system (Wang, 1995) also used a STRIPS-style language to represent operators and was trained with a mixture of both raw experience and grounded expert traces. Another STRIPS-style schema learner, TRAIL (Benson, 1996), used Inductive Logic Programming (ILP) to distill schemas from raw experience and a teacher. Early systems in Relational Reinforcement Learning also recognized the benefits of traces, including one (Driessens & Dzeroski, 2002) that empirically demonstrated the benefit of traces in several benchmark problems and investigated the difference between giving an agent optimal traces versus traces of a suboptimal policy. All of the early systems mentioned above recognized and demonstrated the benefit of experience beyond simple interaction with the environment (bootstrapping, traces, teachers). Our work provides a theoretical groundwork for understanding the empirical success of these and other (Lin, 1991) experiments that provided "trace" data to model learners.

Several recent systems mentioned in the previous chapter, including NID operators (Pasula et al., 2007) used logged data to create their models, essentially relying on a static corpora of traces. The ARMS algorithm (Yang et al., 2007) and its extension for HTN operator learning (Zhuo et al., 2009), learn STRIPS-style action schemas when states are not observed at all (only initial states, goal states, and action logs are available), using powerful constraint satisfaction techniques. A similar problem was covered in recent work on Simultaneous Learning and Filtering (SLAF) (Shahaf, 2007), which learned schemas from traces where changes in fluent values may not be reported immediately (partial observability). The SLAF research produced an algorithm and derived *computational* bounds on the runtime of this algorithm based on the type of language used (including STRIPS). Our work differs from theirs in that we are considering learning in fully observable environments and are concerned with the *sample* complexity of learning, which is also of critical importance in practical systems.

Finally, we note that prior work (Khardon, 1999b,a) has been done on the sample complexity of directly learning to enact the same policy as used in traces in relational domains. The solution in that work produced decision-list style policies and was analyzed in a PAC-style framework, assuming a distribution over possible worlds (an assumption we do not make here). The result also differs from our own because it considers partially observable environments, and more importantly, our goal is to learn models that allow us to do as well or better than the teacher, not necessarily to infer what the teacher's policy actually is.

4.2 Separating Sample Efficiency in the Online and Apprenticeship Frameworks

In this section, we introduce a class of models where PAC-MDP-Trace behavior can be induced. We do this by defining a sufficient condition for efficient model learning in the apprenticeship framework analogous to the use of KWIK in autonomous RL (see Section 2.3). This framework, called Mistake Bound Predictor (MBP) generalizes the class of functions that are KWIK-learnable and Mistake Bound (MB) learnable. Because of this connection, we now review the differences between MB and KWIK, specifically on the learning of conjunctions with n terms, a task that felled KWIK learners in the previous chapter (the combination lock example).

4.2.1 Reviewing KWIK and MB

An extended summary of the KWIK protocol was presented in section 2.2.2 and so far we have used the fact that in autonomous reinforcement-learning, if T and R are efficiently KWIK-learnable, then there exists a PAC-MDP reinforcement learning algorithm for that domain. However, we have seen in the previous chapter that conjunctions of n terms are not KWIK-learnable (see the combination lock example in Theorem 5). But, in the apprenticeship setting, learning to open a such a (deterministic) lock is simple: the agent only needs the teacher to supply a single trace to learn the combination! Thus, there are clearly models that are learnable in the apprenticeship setting that are not autonomously learnable.

Prior work on learning theory in the apprenticeship setting (Angluin, 1988; Goldman & Kearns, 1992) has established a link between teachable hypothesis classes (specifically those used with equivalence queries) and the mistake bound (MB) framework (Littlestone, 1988). We form a similar connection below, so we offer the following review of the MB learning protocol (Section 2.2.1) for model learning. MB is essentially the same as the KWIK protocol except for three changes. (1) In MB, there is no $\perp$ prediction. The agent must always predict a $\widehat{y}_t \in \{0, 1\}$ and receives a true label when it is wrong. (2) MB is only defined for deterministic hypothesis classes, so instead of z_t, the agent will actually see the true label. (3) Efficiency is characterized by a polynomial bound on the number of mistakes made, equivalently the number of labels provided. It follows (Li et al., 2008) that any efficient KWIK-learning algorithm for a deterministic hypothesis class can become an efficient algorithm for MB by simply replacing all $\perp$ labels with an arbitrary element from Y.

4.2.2 The Mistake-Bound Predictor Framework

We now define a generalization of the KWIK and MB frameworks that we will use for model learning in the apprenticeship setting.

Definition 19. *A* **mistake-bounded predictor** *(MBP) is an online learner with accuracy parameters ϵ and δ that takes a stream of inputs from set X and maps them to outputs from a set Y. After predicting any $\widehat{y}_t$, the learner receives a (perhaps noisy) label z_t produced by an unknown function from a known hypothesis class. An MBP must make no more than a polynomial (in $\frac{1}{\epsilon}$, $\frac{1}{\delta}$, and some measure of the complexity of the hypothesis class) number of mistakes with probability $1-\delta$. Here, a mistake occurs if, for input x_t, the learner produces $\widehat{y}_t$ and $||h^*(x_t) - \widehat{y}_t|| > \epsilon$, where h^* is the unknown function to be learned.*

This framework has similarities to KWIK and MB. Like KWIK, MBP observations can be noisy, and like MB, the learner is allowed to make a certain number of mistaken predictions (though in the MBP case, the agent sees observations whether it is right or wrong). In fact, we can formalize the relationships with the following propositions.

Proposition 2. *Any hypothesis class that is KWIK or MB learnable is MBP learnable.*

Proof. A KWIK learner can be used in its standard form, except that whenever it predicts $\perp$, the MBP agent should pick a $\widehat{y}_t \in Y$ arbitrarily. Also, the underlying KWIK learner should not be shown new labels when it does not predict $\perp$, though the "outer" MBP agent does receive them.

MB learners are defined for deterministic hypothesis classes, so we can assume no noisy observations exist. In this setting, the MB and MBP protocols line up except that under MBP, labels are always given but these could be intercepted just as above. □

Notice that the connection to MB learners, along with the equivalence of MB and equivalence-query models (Angluin, 1988), formally links MBP to the traditional computational learning theory literature. We also note that there has been prior work on online learning (mistake-bound style) of continuous functions in the regret framework (Kivinen & Warmuth, 1997), and MB learning in the presence of label noise (Auer & Cesa-Bianchi, 1998). These learning classes are not designed to provide the general properties of MBP learning we will exploit below but may have important connections to the MBP class, the investigation of which is left to future work.

We can combine MBP learners in several ways and still preserve the MBP properties. Here, we introduce a number of simple combinations that have analogues in the KWIK framework that have been very helpful in constructing algorithms for a diverse set of RL models.

Proposition 3. *The **Input-Partition** of k MBP learnable classes, that is when "low-level" MBP-learnable classes $C_1 ... C_k$ with the disjoint input spaces X_1 through X_k, respectively and a "high-level" class C is constructed by mapping an input x_t to the proper sub-learner (based on whether $x_t \in X_j$), is MBP learnable.*

Proof. Suppose each individual component MBP learner has a bound of $B_i(\epsilon_i, \delta_i)$. Then by invoking each one with $\epsilon_i = \epsilon$ and $\delta_i = \frac{\delta}{k}$ and using a union bound to combine the failure probabilities of the individual learners, we get a total bound of $\sum_i B_i(\epsilon, \frac{\delta}{k})$. □

Proposition 4. *The **Union** of k MBP learnable classes where "low-level" MBP-learnable classes $C_0 ... C_k$ with the same input (and possibly output) spaces, but potentially different subsets of the hypothesis class H represented by each one, is MBP learnable.*

Proof. An intuitive algorithm in this case is to simply make the prediction of the learner that has made the fewest mistakes so far and then give each datapoint to all the learners as experience. Assuming the class to be learned is realizable by at least one of the components, by definition this learner can make only $B(\epsilon, \delta)$ mistaken predictions, but since inputs are chosen adversarially it is possible for this learner to have to wait for each of the other $k-1$ learners to make mistakes one at a time. So, in the worst case, this could be the slowest of the k learners, giving us an MBP bound of $O(k * max_i B_i(\epsilon, \delta))$ □

Proposition 5. *The **Cross-Product** of two MBP learnable classes, that is when "low-level" MBP-learnable classes $C_0 ... C_k$ with disjoint input spaces $X_0 ... X_k$, respectively and a "high-level" class C is constructed by breaking up the input x_t into parts $x_1, ... x_k$, giving each to the corresponding sublearner, and reporting the set of all the outputs as $\widehat{y}_t$ is MBP learnable.*

Proof. This case is similar to the input-partition case in that each individual component learner may be responsible for $B_i(\epsilon_i, \delta_i)$ bad predictions where all the other components are making correct predictions, so again we have to sum the individual mistake bounds and use $\frac{\delta}{k}$ as the probability of failure in each one to keep the total probability of failure below δ. However, in this case we also have to contain the accuracy error (ϵ_i) of each component in order to bound the total accuracy $||\widehat{y} - y||_1 = \sum_i ||\widehat{y}_i - y_i||$. We can do so by introducing a factor α that increases the accuracy of each component[2], giving us a final MBP bound of $\sum_i B_i(\alpha\epsilon, \frac{\delta}{k})$. □

Beyond these simple combinations, a number of more complex combinations involving specific configurations of MB and KWIK learners is possible. For instance, the following ***MB-partitioned*** class will be helpful in deriving results in the pre-conditional action schema learning.

Proposition 6. *Consider two "low-level" MBP-learnable classes C_0 and C_1 with the input space X and disjoint output sets Y_0 and Y_1, respectively (that is $Y_0 \cap Y_1 = \emptyset$). Consider a "high-level" MB-learnable class C mapping X to $\{0, 1\}$. The composition of these classes where the output of the class C learner is used to select which low-level MBP class to use (if the output of the high-level learner is i, use class C_i) is MBP-learnable.*

Proof. On input x, get a prediction i from the C learner. Then, query the C_i learner and report its response as the solution. Observe y. Let $i \in \{0, 1\}$ be such that $y \in Y_i$, then train the learner C with (x, i) and C_i with (x, y). By construction, all learners get the appropriate training data and will individually make a small number of mistakes. The total number of mistakes is then simply the sum of those made by C, C_1, and C_2 since each piece of training data always goes to the right C_i and each piece of data that causes a mistake at the C level is used to train C. When each component makes accurate predictions, the overall learner is accurate. □

As an example of an MB-partitionable class, consider a factored MDP with a reward function defined as follows. If a predetermined conjunction c_R over all n factors is false, then the agent receives reward $R_{\min} < 0$ and otherwise it gets a reward drawn from a distribution over $[0, R_{\max}]$. Given that information (but not c_R or the distribution), the reward function can be learned using an MB conjunction learner and a KWIK learner for the distribution when the conjunction is true, because the cases are always discernible. In contrast, a class where the distribution for a false conjunction is still $R_{\min}$, but a true conjunction induces a reward over $[R_{min}, R_{max}]$, is *not* covered under this case because the outputs sets of the learning problems overlap (making it unclear how to solve the top-level learner).

[2] Similar arguments can be made for other common norms.

4.2.3 Efficient Apprenticeship RL

In this section, we show that if an MDP model is MBP learnable, PAC-MDP-Trace behavior is guaranteed in the Apprenticeship RL setting. Specifically, we introduce a model-based RL algorithm (MBP-Agent, Algorithm 25) for the apprenticeship setting that uses an MBP learner as a module for learning the dynamics of the environment within the confines of the apprenticeship protocol. Notice that because MBP learners never acknowledge uncertainty, our algorithm for the apprenticeship setting believes whatever its model tells it (which could be mistaken). While autonomous learners run the risk of failing to explore under such conditions, the MBP-Agent can instead rely on its teacher to provide experience in more "helpful" parts of the state space, since its goal is simply to do at least as well as the teacher. Thus, even model learners that default to pessimistic predictions when little data is available (as we see in later sections), can be used successfully in the MBP-Agent algorithm. Algorithm 25 has the following property.

Algorithm 25 MBP-Agent

```
The agent knows $\epsilon$, $\delta$, and $A$ and has access to the environment $E$, teacher $\mathfrak{T}$, and a planner
  $P$ for the domain.
Initialize MBP learners $L_T(\epsilon_T, \delta)$ and $L_R(\epsilon_R, \delta)$ //$\epsilon$'s defined below
for each episode do
  $s_0 = E.startState$
  $t = 0$
  while episode not finished do
    $a_t = P.getPlan(s_t, L_T, L_R)$.
    $\langle r_t, s_{t+1} \rangle = E.executeAct(a_t)$
    $L_T.Update(s_t, a_t, s_{t+1}); L_R.Update(s_t, a_t, r_t)$
    $t = t + 1$
  end while
  if $\mathfrak{T}$ provides trace $\mathcal{T}$ starting from $s_0$ then
    $\forall \langle s, a, r, s' \rangle \in \mathcal{T}, L_T.Update(s, a, s'), L_R.Update(s, a, r)$
  end if
end for
```

Theorem 13. *The MBP-Agent is PAC-MDP-Trace for any domain where the transitions and rewards are polynomially MBP learnable. That is, with proper settings of ϵ_T and ϵ_R, the agent will (with probability $1 - \delta$) only receive a polynomial number of valid traces (where $V^{\pi_A} < V^{\pi_T} - \epsilon$).*

The heart of the argument is an extension of the standard Explore-Exploit lemma, we call the *Explore-Exploit-Explain Lemma.*

Lemma 8. *On each trial, we can define a set of known state,action ($\langle s, a \rangle$) pairs as the ones where the MBP currently predicts transitions accurately. One of these outcomes occurs: (1)*

The agent will encounter an unknown $\langle s, a\rangle$ (explore) with high probability. (2) The agent will execute a policy π_t whose value is better or not much worse than the teacher's policy π_T (exploit). (3) The teacher's trace will encounter an unknown $\langle s, a\rangle$ (explain) with high probability.

Lemma 8 proves Theorem 13 because MBP can only make a polynomial number of mistakes, meaning cases (1) and (3) can only happen a polynomial number of times. Below is a sketch of the lemma's proof.

Proof. The quantity $V^{\pi_T}(s_0)$ is the value, in the real environment, of the teacher's policy and $V^{\pi_A}(s_0)$ is the value, in the real environment, of the agent's current policy. Analogously, we can define $U^{\pi_T}(s_0)$ as the value, in the agent's learned MDP, of the teacher's policy and $U^{\pi_A}(s_0)$, the value, in the agent's learned MDP, of the agent's policy.

First, we note that if our learned MDP model has only small errors in the transition and reward function, specifically, $\epsilon_T = \frac{\epsilon(1-\gamma)}{2\gamma V_{max}}$ and $\epsilon_R = \frac{\epsilon(1-\gamma)}{2}$ with high probability, then we can be confident that $||V^\pi - U^\pi|| \leq \epsilon$ because the maximum difference between the value functions is $\frac{\epsilon_R + \gamma V_{max}\epsilon_T}{1-\gamma}$ by the Simulation lemma (Lemma 1). In the case below, we will demand that the model be learned within an even tighter accuracy of $\epsilon_T = \frac{\epsilon(1-\gamma)}{4\gamma V_{max}}$ and $\epsilon_R = \frac{\epsilon(1-\gamma)}{4}$, ensuring that $||V^\pi - U^\pi|| \leq \frac{\epsilon}{2}$. These are the accuracy parameters that will be used to instantiate the MBP learners.

With this guarantee in hand, by any of several simulation lemmata, such as Lemma 1, if $|U^{\pi_A}(s_0) - V^{\pi_A}(s_0)| > \frac{\epsilon}{2}$, then, with high probability, case (1), explore, will happen. That is because executing π_A in the real environment will produce a sample of $V^{\pi_A}(s_0)$ and the only way it can be different from the agent's conception of the policy's value, $U^{\pi_A}(s_0)$, is if an unknown $\langle s, a\rangle$ pair is reached with sufficiently high probability.

Next, we consider the case where $U^{\pi_A}(s_0)$ and $V^{\pi_A}(s_0)$ are within $\frac{\epsilon}{2}$ of one another. If $V^{\pi_A}(s_0) \geq V^{\pi_T}(s_0) - \epsilon$, that means π_A is nearly optimal relative to π_T, and case (2), exploit, happens.

Finally, we consider the case where $U^{\pi_A}(s_0)$ and $V^{\pi_A}(s_0)$ are within $\frac{\epsilon}{2}$ of one another *and* $V^{\pi_A}(s_0) < V^{\pi_T}(s_0) - \epsilon$. Note that $U^{\pi_A}(s_0) \geq U^{\pi_T}(s_0)$ (because π_A was chosen as optimal). Chaining inequalities, we have $U^{\pi_T}(s_0) \leq U^{\pi_A}(s_0) \leq V^{\pi_A}(s_0) + \frac{\epsilon}{2} < V^{\pi_T}(s_0) - \frac{\epsilon}{2}$. We're now in a position to use a simulation lemma again: since $|U^{\pi_T}(s_0) - V^{\pi_T}(s_0)| > \frac{\epsilon}{2}$, then, with high probability, case (3), explain, will very likely happen when the teacher generates a trace. □

In summary, KWIK-learnable models can be efficiently learned in the autonomous RL case, but MB learning is insufficient for exploration. MBP covers both of these classes, and is sufficient for efficient apprenticeship learning, so models that were autonomously learnable as well as many models that were formerly intractable (the MB class), are all efficiently learnable in the apprenticeship setting. As an example, the combination lock described earlier could require an exponential number of tries using a KWIK learner in the autonomous case, and MB is insufficient in the autonomous case because it does not keep track of what combinations have been tried. But in the apprenticeship setting, the MBP-Agent can get the positive examples it needs (see Section 4.3.1) and will succeed with at most n (essentially one for each irrelevant tumbler) valid traces.

4.2.4 Apprenticeship Learning of Propositional and Continuous MDPs

While we soon use the results above to prove the efficient learnability of parts of relational action schemas in the apprenticeship setting, we briefly discuss results here for some of the propositional MDP classes to show the generality of this MBP-Agent algorithm. In the autonomous setting, flat MDPs can be learned with a KWIK bound of $\tilde{O}(\frac{S^2A}{\epsilon^2})$ (Li et al., 2008). Following Theorem 13, this gives us a polynomial PAC-MDP-Trace bound, a result that is directly comparable to the apprenticeship-leaning result under the earlier protocol described by Abbeel & Ng (2005).

The same work considered apprenticeship learning of linear dynamics. We note that these domains are also covered by Theorem 13 as recent results on KWIK Linear Regression (Walsh et al., 2009b) have shown that such n-dimensional MDPs are again KWIK-learnable with a bound of $\tilde{O}(\frac{n^3}{\epsilon^4})$. Thus, our apprenticeship learning algorithm achieves sample complexity results on par with early apprenticeship learning work and work in the autonomous case.

4.3 Apprenticeship Learning of Action Schemas

In this section, we prove that the apprenticeship setting can be used to learn action schemas that were not efficiently learnable in the online setting. The crux of this result is the use of MB algorithms to learn the conjunctions associated with conditions that contain $O(n)$ terms. We begin by reporting that all of the results of the previous chapter carry forward to the apprenticeship learning case because MBP subsumes KWIK.

Corollary 3. *All of the positive KWIK-learning results from Chapter 3, including the CD-Learning problem for action schemas with $|C^a| = O(1)$, are MBP learnable by their respective KWIK algorithms, and therefore, when combined with Algorithm 25, induce PAC-MDP-Trace behavior.*

While these results show that the theoretical bounds are not hampered by the move to the apprenticeship setting, we do often see a large empirical speedup from using traces, as they can often guide the agent to important areas of the state space, along with the need to only do as well as the teacher. Empirical evidence of this speedup appears in Figure 4.1 if one compares the autonomous KWIK learners and those using apprenticeship learning.

Having established that nothing is lost when moving from the autonomous setting to the apprenticeship setting, we now show that larger schemas can be learned in the apprenticeship learning case. As these results hinge on MB algorithms, we now review two classical MB algorithms for conjunction learning and k-term DNF that will be instrumental in our algorithms.

4.3.1 Learning Conjunctions in the Apprenticeship Setting

As mentioned earlier, KWIK and MB are separable when learning monotone conjunctions[3] over n literals when the number of literals relevant to the conjunction (L_R) can be as many as n. While we saw that the KWIK-Enumeration (Algorithm 12) could be used to learn conjunctions of size $k = O(1)$, conjunctions of size $O(n)$ result in an $O(2^n)$ hypothesis space, so there can be an exponential number of $\perp$ predictions. This situation arises because negative examples are highly uninformative. In the combination lock in Theorem 5, the agent has no idea which of the 2^n settings will allow the lock to be unlocked, so it must predict $\perp$ at every new combination. Note though that if it does see this one positive example it will have learned the correct combination. This asymmetry is not unique to the monotone case and is in fact a staple of learning general boolean functions.

In contrast, learners in the MB setting can efficiently learn conjunctions of arbitrary size by exploiting this asymmetry. Specifically, an MB agent for conjunction learning (Kearns & Vazirani, 1994) is presented in Algorithm 26. The algorithm (*MB-Con*) essentially maintains a set of literals $l_j \in L_R$ where $l_j = 1$ for every positive example it has seen before. If every $l_j \in L_R$ has a value of 1 in x_t, the agent correctly predicts *true* , otherwise it defaults to *false* . By using such defaults, which KWIK agents cannot, and by only counting the highly informative positive

[3] The results extend to the non-monotone setting using the standard method of including all negated literals.

samples, each of which subtracts at least one literal from L_R, polynomial sample efficiency is achieved.

Algorithm 26 Conjunction Learning (*MB-CON*)

```
$L_R = \emptyset$
while no mistakes made do
    Predict false for $x_t$
end while
$L_R = \{l_j \in x_t | l_j = 1\}$
for Input $x_t$ do
    if $l_j = 1$ in $x_t$ for all $l_j \in L_R$ then
        Predict true
    else
        Predict false
        if mistake, $x_t$ = true then
            $L_R = L_R \bigcap \{l_j \in x_t | l_j = 1\}$
        end if
    end if
end for
```

Another class that is MB learnable is the class of k-term-DNF (disjunctive normal form of $k = O(1)$ terms). k-term-DNF are of the form $(l_i \wedge l_j \wedge ...)_1 \vee ... \vee (... \wedge ...)_k$, that is, a disjunction of k conjunctive terms, each of at most size L (the number of literals). This class of functions is known to be MB learnable (Kearns & Vazirani, 1994) by creating a conjunction of literals with each literal representing the disjunction of k of the original literals. For instance, with $k = 3$ we would have $l_{ijm} = l_i \vee l_j \vee l_m$ for all possible i, j, m. From there, the *MB-Con* algorithm can simply be used to learn the formula. We note that recreating the actual k-term DNF formula from such a model is NP-Hard (Pitt & Valiant, 1988) computationally, but for our purposes, making predictions with this different internal model will suffice.

4.3.2 Positive CD-Learning Results with Apprenticeship Learning

Here, we show that the stochastic pre-conditional CD-Learning problem is solvable in the apprenticeship setting for conditions that are conjunctions of L or fewer terms. We later show how this problem becomes intractable when the full conditional case, but in some special cases, this problem too is solvable with the aid of a teacher.

CD-Learning in the Pre-conditional Setting

Our algorithm for the pre-conditional stochastic CD-Learning problem in the apprenticeship setting is an instantiation of Algorithm 25 using the MB-partitioning combination from Proposition 6 with an MB conjunction learner *MB-CON* from above for the pre-conditions and a KWIK-LR learner (Algorithm 9) for the effect probabilities. Algorithm 27 presents this procedure completely filled out with the mechanics of the partitioning and the interaction with the environment.

Algorithm 27 Pre-conditional Stochastic CD-TraceLearn

```
Given: Action set $A$ of max arity $m$, Predicate set $P$ of max-arity $n$, Effect set $\Omega^a$ for each
  $a \in A$, horizon $H$, Environment $E$, and teacher $\mathfrak{T}$
Initialize a copy of Algorithm 9 ($KWIK - LR_a(\Omega^a)$) for each $a \in A$
Initialize a copy of Algorithm 26 ($MB\text{-}CON_a$), a pessimistic pre-condition for each action
  ($Pre^a$)
for each episode ($s_0$, $\mathcal{G}$) do
    $t = 0$
    for each step $t$ until $s_t \in \mathcal{G}$ or $t = H$ do
        $\forall a$ Create Action Schema $\mathcal{A} = \{MB\text{-}CON_a, \Omega^a, KWIK\text{-}LR_a\}$
        $a_t$ = Suggestion from Known-Edge VI (Algorithm 10) with $\mathcal{A}$ and $s_t$
        $\langle s_{t+1}, r_t, failure \rangle = Env.execute(a_t)$
        if $failure = false$ then
            Update $KWIK\text{-}LR_a$ with $\langle s_t, s_{t+1} \rangle$
            Update $MB\text{-}CON_a$ with $\langle s_t, 1 \rangle$ //positive example
        end if
        $t = t + 1$
    end for
    if trace $\mathcal{T}$ is given by $\mathfrak{T}$ then
        for each $\langle s, a, r, s', failure \rangle \in \mathcal{T}$ do
            if $failure = false$ then
                Update $KWIK\text{-}LR_a$ with $\langle s_t, s_{t+1} \rangle$
                Update $MB\text{-}CON_a$ with $\langle s_t, 1 \rangle$ //positive example
            end if
        end for
    end if
end for
```

Unlike the autonomous learners in Chapter 3, this algorithm works by keeping a pessimistic version of the pre-conditions using the MB conjunction learner and an optimistic version of the probabilities of effects using the KWIK-LR learner. This combination of pessimism and optimism is admissible under the MBP framework and guarantees that we get the right kind of examples to do efficient learning. Specifically, the pre-condition learner is now only trained with positive examples, which are highly informative, as we saw in the combination lock example. This is because it defaults to predicting "false" for any state that does not satisfy its current hypothesis, which is the most specific hypothesis covering the previously seen examples. Thus, the agent will execute the best policy it can with actions it knows will execute in different states, or in the worst case (initially no actions are thought to work anywhere), just act randomly for H steps. For instance, consider a combination lock-style example with $n = 3$ tumblers, but where only the first two tumblers need to be set to 1 for the lock to open. Let's assume the initial start state in the first episode has the lock at $[0, 0, 1]$. The agent will think no combination will open the lock, and will try H random actions, probably without success. Assuming the teacher is using an optimal strategy, it would then show the agent something like "set the tumblers to all 1's and open it". Now let's say the agent is then placed in initial state $[0, 0, 0]$. The agent's hypothesis (based on the previous example) on the pre-conditions will be that every tumbler needs to be set to 1, so it will do so and open the lock. But the optimal teacher now shows it a trace that ends in $\langle[1, 1, 0], \mathit{open}, \mathit{Goal}\rangle$. The agent will now eliminate $\mathit{One}(\mathrm{X3})$ from its pre-condition list for the *open* action, and has now learned the correct pre-condition list. In general, the algorithm above solves the combination lock problem with at most $n - n_r$ traces, where n_r is the number of tumblers that actually need to be set to a specific number. Thus, we have achieved PAC-MDP-Trace behavior in an environment that was not PAC-MDP learnable in the autonomous case. Formally, we can now state the following theoretical property for CD-Learning in the stochastic pre-conditional setting.

Theorem 14. *Algorithm 27 solves the CD-Learning problem for stochastic pre-conditional action schemas when the pre-conditions are conjunctions of a polynomial size in the domain description. Therefore, this algorithm is PAC-MDP-Trace learnable if the agent is given the set of possible effects (Ω^a, but not Π^a) beforehand.*

Proof. Each transition sample is either a "failure" (c^a is *false*) or a transition that returns a next state s' (without a failure signal), so the output spaces are disjoint as required by *MB-partition.* We use the conjunction learner for*MB-CON^a* (Algorithm 26) to predict whether the pre-conditions of a grounding of that action hold. Each trace $\mathcal{T}$ received because of a suboptimal policy (with respect to the teacher) will provide positive examples of the pre-conditions, updating each *MB-CON^a* so no more than $|A|n$ traces will be needed, where n is the number of literals within the action's scope.

The other part of the partition is learning each Π^a, which is done separately from the conjunction learning with a mixture of real experience and trace tuples, using KWIK-LR (Algorithm 9), which, over all the actions, gives us a KWIK bound of $\tilde{O}(\frac{|A||\Omega|^3}{\epsilon^4})$ for this portion of the learning. Thus, given Ω^a, the dynamics are MBP learnable, and the domain can be learned

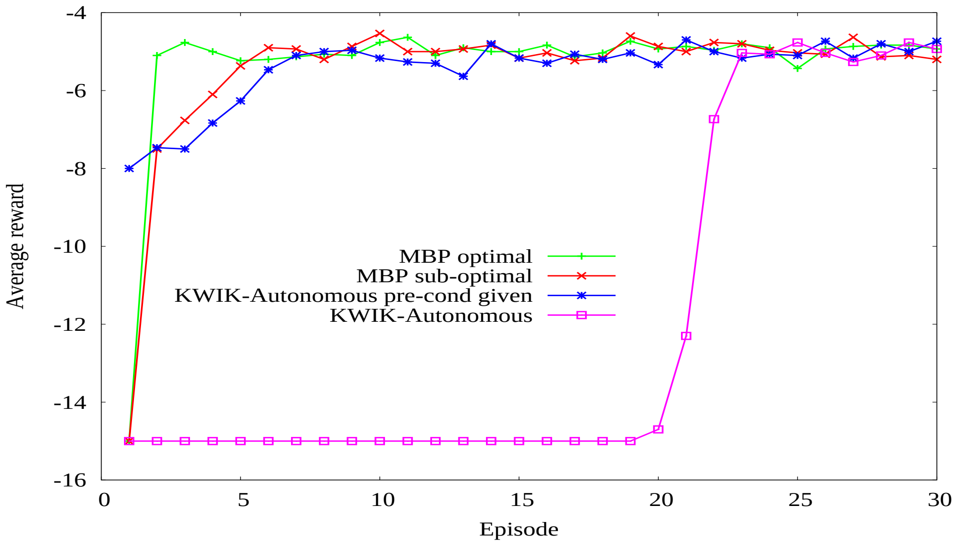


Figure 4.1: KWIK autonomous learners and an apprenticeship learners in a noisy 3-blocks world. The data is averaged over 30 runs and traces are provided after every episode.

by a PAC-MDP-Trace agent. □

An Experiment for Pre-conditional CD-Learning

We consider a Stochastic STRIPS noisy-blocks world with “dummy” actions: with probability 0.2, the normal pickup and putdown actions simply have no effect, but there are also two “extra” pickup/putdown actions that do nothing with probability 0.8. Figure 4.1 shows 4 agents in a 3-blocks version of this domain. The MBP-Agent with an optimal teacher learns the pre-conditions and to avoid the extra actions from a single trace. The MBP-Agent with a suboptimal teacher who mixes in the extra actions with probability 0.5 eventually learns the probabilities and performs optimally in spite of its teacher. Both MBP agents efficiently learn the pre-conditions. In contrast, a KWIK learner *given* the pre-conditions achieves similar results to the suboptimal-trace learner (though for different reasons), and a KWIK learner for both the pre-conditions and probabilities requires an inordinate amount of exploration to execute the actions correctly. We recorded similar results with 4-blocks and exploding blocks, although 2 or 3 optimal traces are needed in those domains. We now show how to extend these results to the full deterministic pre-conditional CED-Learning problem.

Deterministic Pre-conditional CED-Learning

In the deterministic pre-conditional CED-Learning problem, we do not need to use the KWIK-LR portion of Algorithm 27 because there are no probabilities to learn. However, we still need to combine two different MBP learners, because unlike the CD-Learning problem, we now need to learn the effects (Ω^a), not just the pre-conditions (C^a). To do so, we will combine the *MB-CON* learner for the pre-conditions with the KWIK-DetEffect learner (Algorithm 19). Again, *MB-Partition* will be used to combine the two outputs, as an effect of "failure" with no change in the current state literals can be predicted when *MB-CON* predicts false. The complete algorithm for the pre-conditional deterministic CED-Learning problem is exactly the same as Algorithm 27, but with a Det-EffectLearn KWIK Learner (Algorithm 19) in the place of KWIK-LR.

Intuitively, the algorithm again keeps pessimistic versions of the pre-conditions (the most specific conjunction covering all positive examples) and an interpretation of the possible effects (from DetEffectLearn) that considers any $\perp$ predictions to either cause no change, or be optimistically (as in the autonomous setting), or even pessimistically filled in. Essentially, with the teacher in the loop, any interpretation of $\perp$ for the effects can be used. In the analysis below, we consider the first of these interpretations (no change if $\perp$ is predicted, which we denote as *Det-EffectLearn*$^-$). Formally, we can state the following.

Theorem 15. *An MB-Partition combination of* MB-CON *and* Det-EffectLearn$^-$ *solves the CED-Learning problem the deterministic pre-conditional STRIPS when the pre-conditions are conjunctions of a polynomial size in the domain description. Therefore, this algorithm us PAC-MDP-Trace learnable.*

Proof. The use of *MB-CON* producing a limited number of mistakes was covered in the CD-Learning result. All that needs to be considered then is the number of mistakes introduced in predicting Ω^a, but this is simply the KWIK bound from Theorem 19, because each trace that shows a strictly better policy once the pre-conditions are known must show either an action adding a literal in a case where the learner's model predicted no change, or the agent will have encountered a deleted literal it did not anticipate during the episode or after viewing a trace. Hence, we have an MBP bound for this portion, which combined with the pre-condition learning portion, gives us an MBP bound of $O(APm^n)$, and thus through Theorem 13, the algorithm is PAC-MDP-Trace. □

As an example of this algorithm at work, consider learning the Blocks World schema for

After first trace

move(B, From, To):
PRE: *On*(B, From) , *Clear*(B) , *Clear*(From) , *Clear*(To) , *Block*(B) , *Block*(To) , *Table*(From)
ADD: *On*(B, To)
DEL: *On*(B, From), *Clear*(To)

After second trace

move(B, From, To):
PRE: *On*(B, From) , *Clear*(B) , *Clear*(To) , *Block*(B) , *Block*(To)
ADD: *On*(B, To) , *Clear*(From)
DEL: *On*(B, From), *Clear*(To)

Table 4.1: Learned STRIPS action schemas from the blocks world traces in Section 4.3.2. Variable names are inserted for readability.

the action move(B, From, To) which moves a block from one place (either a block or the table) to another block. This blocks world is composed of 4 blocks (**a**,**b**,**c**,**d**), a table (**t**), four predicates (*On*(X, Y), *Block*(X), *Clear*(X), *Table*(X)), and two actions (*move*(B, From, To) and *moveToTable*(B, From, T)). Table 4.1 shows the agent's model as learned from traces in this setting, a process we now describe in more detail. In the first episode, s_0 has all the blocks on the table and the goal is to stack three blocks on one another. The pessimistic agent, which initially thinks every possible pre-condition must hold to actually execute an action, either reports "no plan" or performs H unhelpful actions, but the teacher responds with the plan: [*move*(**a**, **t**, **b**), *move*(**c**, **t**, **a**)] The agent updates its *move* schema to reflect the new trace, as seen in the first schema in Table 4.1.

Next, the agent is presented with the same goal but an initial state where **a** is on **b** and **c** is on **d**. Because it has not yet correctly learned the pre-conditions for *move* (it believes blocks can only be moved from the table), the agent reports "no plan" or thrashes for H steps, and receives a trace [*move*(**c**, **d**, **a**)], which induces the second schema in Table 4.1.

Now, the agent receives the same initial state as the previous trace, but with a goal of ∃ X, Y *On*(X, Y), *On*(Y, **d**). Because the agent has learned the *move* action, it produces the plan [*move*(**a**,**b**,**c**)]. Further experience could refine the *moveToTable* schema.

Extending these results to the stochastic case requires replacing the deterministic effect learning component with the proper stochastic effect learning component (generally KWIK-CorEffect, Algorithm 23). Indeed, all the KWIK-learning algorithms for learning effects or distributions port to the apprenticeship setting in the pre-conditional case and link with *MB-Con*, allowing us to drop the assumptions on the size of conditions. The results above for the pre-conditional case used a conjunction learner that relied on a unique failure signal when the pre-conditions of an action fail. We now investigate conditional outcomes that do not provide this signal.

CD-Learning in the Conditional Setting

We now turn to the conditional case, where the apprenticeship results are not as straightforward. We will show that in general, the CD-Learning problem is likely not MBP learnable, but then present algorithms that solve the problem in the deterministic setting when the number of $\langle c_i, \omega_i \rangle$ pairs is $O(1)$ and discuss complications for such an algorithm in the stochastic case under similar assumptions.

Deterministic Conditional CD-Learning and Decision Trees

In the conditional case, the positive KWIK results in the autonomous setting assumed the possible conditions were enumerable, either by bounding the size ($|c|$) or number ($|C|$) of conditions by a constant. But above, we were able to eliminate the constraint on the size of pre-conditions, so it is natural to wonder if we can do the same relaxation in the conditional case. Unfortunately, the following theorem shows this is not likely. The proof is by reduction from decision tree learning, where the best known sample complexity bounds are still super-polynomial in the number of attributes and labels (Hellerstein & Servedio, 2007) and while many special cases have been shown to be learnable under different complexity measures (for example, Auer et al. (1995)), the lower bound on the sample complexity remains a significant open question (see Hellerstein & Servedio (2007) for more discussion).

Theorem 16. *If there is no polynomial mistake bound algorithm for decision-tree learning, the deterministic conditional CD-Learning problem is not MBP learnable without a restriction on the number of conditions in the operator.*

Proof. To perform the reduction, we simply create a unique "effect" for every possible label in the decision tree and translate all the $\langle \mathit{instance}, \mathit{label} \rangle$ pairs into $\langle s_t, \omega_t \rangle$ pairs to train the CD-Learner. A solution for the CD-Learning problem is directly translatable into the decision tree learner since the labels and effects have a one to one mapping. □

This reduction is somewhat intuitive because each root-to-leaf path in a decision tree represents a conjunction (c_i), and when we don't bound the length of the branches (as we did in the autonomous case) or the number of branches (as we do below), the hypothesis space is (naïvely at least) exponentially large. The reason the relaxation that worked in the pre-conditional case does not hold in the conditional case is that the problem is no longer negative examples being uninformative—since there is no "Else" clause in the conditional case, every example is a positive one. The problem is, we cannot be certain from the state transitions which effect actually occurred, so we don't know what condition learner to update. However, we now return to the case where number of conditions in a schema ($|C|$) is limited to be $k = O(1)$. This case had KWIK solutions in Section 3.8.3, but hidden in the bounds were large constants and enumerations of tree structures that we now attempt to eliminate under the MBP protocol.

Deterministic Conditional CD-Learning with Few Conditions

While the general CD-Learning problem is intractable in the conditional case, we show here that in the deterministic case with the number of conditions constrained to be $k = O(1)$, PAC-MDP-Trace behavior can be guaranteed. In general, the *MB-CON* algorithm cannot be used to learn the conditions in this case because of ambiguity as to which effect set (i for Ω_i) is responsible for a transition. As an example, consider invoking the *paint* action in deterministic Metal/Wooden Paint-Polish. While the agent is learning the conditions and suppose it paints an object that is already scratched. The outcome observed by the agent (the object is scratched and painted) does not tell it which of (ω_1, ω_2) actually occurred, so it is not immediately clear what c_i^a, should be updated.

However, we now show that the problem is solvable using an MB k-term-DNF learner to learn what conditions do *not* produce each effect. Specifically, instead of representing the condition that causes an effect to occur, we learn the conditions that do not cause effect ω_i, which is $c_1 ... \vee c_{i-1} \vee c_{i+1} \vee .. c_k$. Since each c_j is an arbitrarily sized condition, we are learning a k-term-DNF for each condition *not* occurring. This algorithm is presented in Algorithm 28.

Intuitively, the algorithm keeps track, for each action, of a k-term DNF formula for the conditions under which the associated effect does *not* happen. At each step (or each step in a trace), we update the condition learner for the effects that absolutely *did not* occur on that step. This refines the disjunctive formula representing the cases where that effect does *not* happen, which has the effect of making the conditions where the effect *does* happen more specific (so the effect will be predicted now for fewer states). The operators mined from such overly general conditions may predict multiple effects for a given state. For instance, in the beginning of learning, all the possible effects for an action will be predicted because the condition that says

Algorithm 28 Conditional Deterministic CD-TraceLearn

```
*Given*: Action set $A$ of max arity $m$, Predicate set $P$ of max-arity $n$, Effect set $\Omega^a$, with $|\Omega^a| = O(1)$ effects for each $a \in A$, horizon $H$, Environment $E$, and teacher $\mathfrak{T}$
Initialize a copy of $\mathit{MB\text{-}k\text{-}DNF}_{ai}$ for each action $a$ and effect $\omega_i \in \Omega^a$.
**for** each episode ($s_0$, $\mathcal{G}$) **do**
    $t = 0$
    **for** each step $t$ until $s_t \in \mathcal{G}$ or $t = H$ **do**
        $\forall a$ Create Action Schema $\mathcal{A} = \{\mathit{MB\text{-}k\text{-}DNF}_a, \Omega^a, \mathit{KWIK\text{-}LR}_a\}$
        $a_t$ = planners suggestion using $\mathcal{A}$ and $s_t$ and choosing effects arbitrarily if multiple conditions match $s_t$.
        $\langle s_{t+1}, r_t\rangle = \mathit{Env.execute}(a_t)$
        Let $\Omega^a_t$ ($\bar{\Omega}^a_t$) be the set of effects that do (do not) explain $\langle s_t, s_{t+1}\rangle$
        **for** each $\bar{\omega} \in \bar{\Omega}^a_t$ **do**
            Update $\mathit{MB\text{-}k\text{-}DNF}_{a\omega}$ with $\langle s_t, 1\rangle$ //positive example
        **end for**
        $t = t + 1$
    **end for**
    **if** trace $\mathcal{T}$ is given by $\mathfrak{T}$ **then**
        **for** each $\langle s, a, r, s', \mathit{failure}\ \rangle \in \mathcal{T}$ **do**
            Let $\Omega^a$ ($\bar{\Omega}^a$) be the set of effects that do (do not) explain $\langle s, s'\rangle$
            **for** each $\bar{\omega} \in \bar{\Omega}^a$ **do**
                Update $\mathit{MB\text{-}k\text{-}DNF}_{a\omega}$ with $\langle s, 1\rangle$ //positive example
            **end for**
        **end for**
    **end if**
**end for**
```

they will *not* happen is initialized to always *false* . When this occurs, the planner just picks one of the effects arbitrarily as the teacher will correct this behavior with a trace of its own if this leads to a suboptimal plan. Formally, the algorithm has the following property.

Theorem 17. *The deterministic conditional CD-Learning problem is PAC-MDP-Trace learnable if the number of conditions associated with each action is bounded by $k = O(1)$ using Algorithm 28.*

Proof. Formally, we show that the set of k-term-DNF learners makes a bounded number of mistakes in predicting next states, which gets us the desired result through Theorem 13. On every step, a state s is given to each of the k-term DNF learners associated with the current action a, which are queried as to whether condition c_i does not hold. Let L_1 be the set of such learners that reported their effect would *not* hold (the DNF is true) and L_0 that said their effects would happen. Let the corresponding effect sets be Ω_1 and Ω_0. One of the effects from Ω_0 is the prediction (chosen by the planner arbitrarily) of the transition. Now the observed transition $\langle s, a, s'\rangle$ comes in. One of the following happens: either $\eta(s, \omega_0) = s'$ (correct prediction), in which case we don't really need to do an update, or the prediction was wrong. In that case, we update every effect learner in $l_i \in L_0$ where $\eta(s, \omega_i) = s'$ (note there has to be at least one, namely l_0). Since each of these has a polynomial mistake bound, these updates can only happen a polynomial number of times (specifically kL^k), for each action, where L is the number of literals in the domain. Notice that because the condition learners keep the most specific hypothesis for *not* matching a state, the learners in L_1 that predicted their effects would not happen will never be wrong—these learners always err on the side of "false" predictions, which is translated to the L_0 set because we are learning a negation. □

Notice that this algorithm manages to solve the "small number of conditions" CD-Learning problem without incurring the large constants seen in Section 3.8.3. It does so by utilizing a more powerful MB algorithm where before only KWIK-Learners were available. Finally, we note that this mistake-bound efficient algorithm may require super-polynomial computation time to actually output the corresponding action schemas, although the *predictions* (which is all the MBP-Agent algorithm needs) do not require this computation. Specifically, while k-term-DNF are MB learnable with a polynomial number of mistakes, turning the internal representation (using the new clausal literals) into an actual k-term-DNF is NP-Hard (Pitt & Valiant, 1988). The same holds for producing the actual action schemas (condition/effect pairs) in our case, though again, producing this exact representation of the dynamics is not necessary in our algorithms.

Stochastic Conditional CD-Learning

The stochastic CD-Learning problem's general intractability follows from the negative sample complexity result in the deterministic case described in Theorem 16. However, there are some special cases worth considering. First, there is the case where the number of condition/effect-distributions is bounded by $k = O(1)$. Notice this is different from the problem we just solved with k-term DNF learners because now when a condition matches a state, an effect is chosen from the distribution $\langle \Omega_i^a, \Pi_i^a \rangle$, so the deterministic strategy cannot be exactly used because a single observed effect is no longer definitive as to what conditions did not occur. Thus, it is not clear if there is a workable extension of the k-term-DNF approach above. However, the KWIK approach for this setting (enumerated decision trees, AKMS, and KWIK-LR), is still usable, though not very practical.

A potentially more useful special case is one where there are again $k = O(1)$ conditions, but now each condition has one effect in its associated effect list $\omega_i^* \in \Omega_i^a$ such that $Pr(\omega_i^*) > \delta_0$

for c_i, but for all $c_j \neq c_i$, the probability of this effect is 0. In that case a two-stage algorithm is possible where, in stage 1, the algorithm only uses samples that show one of the indicator effects (since effects are given for each condition in CD-Learning these are trivial to identify), and uses *MB-Con* to learn the associated conditions. If it receives a state that matches only one condition (stage 2), it uses that to update a KWIK-LR learner to learn the distribution over the effects.

Conditional CED-Learning

Above, we have derived some positive (and a few negative) learning results in the apprenticeship learning paradigm for various subproblems from Chapter 3 without the previously necessary restriction on the size of the conditions. We reiterate that if those size restrictions also held in the apprenticeship learning setting, the original KWIK algorithms from Chapter 3 could simply be ported to this new setting since KWIK is a subset of MBP. We complete the picture here by noting that the hardness of the Conditional CED-Learning problem in the general setting, even if the schemas are deterministic, follows from Theorem 16, which showed the hardness of the easier CD-Learning problem.

Special cases do exist (including all the pre-conditional cases) where CED-Learning is possible with traces and without the condition-size constraints. It is left to future work as to whether CED-Learning when the number of condition/effect pairs is $k = O(1)$ (even in the deterministic setting) is one of these. To see the difficulty in this setting, consider the k-Term-DNF learners used in the deterministic conditional CD-Learning problem. We were able to use learners to represent which effect didn't happen because each learner had a known associated effect, so we could definitively tell which learners should get positive examples of their condition not happening. But if the effects are being learned as well, it becomes unclear whether the condition or effect learning portions are to blame.

This chapter has shown methods for relaxing assumptions on the size of pre-conditions (and sometimes conditions) in the apprenticeship setting. Table 4.2 summarizes some of these results.

Setting	D-Learning	ED-Learning	CD-Learning	CED-Learning
Deterministic Unconditional	Given	DetEffectLearn	Given	DetEffectLearn
Stochastic Unconditional	KWIK-LR	KWIK-CorEffect$\star$	KWIK-LR	KWIK-CorEffect$\star$
Deterministic Pre-conditional	Given	DetEffectLearn	*MB-Con*	*MB-Con* and DetEffectLearn (in parallel)
Deterministic Conditional	Given	DetEffectLearn	MB-k-term DNF learner$\star$	Union* and DetEffectlearn (multiple copies) †
Stochastic Pre-Conditional	KWIK-LR	KWIK-CorEffect$\star$	*MB-Con* and KWIK-LR (in parallel)	*MB-Con* and KWIK-CorEffect$\star$ (in parallel)
Stochastic Conditional	KWIK-LR	KWIK-CorEffect	Meteorologist* and KWIK-LR (multiple copies)	Meteorologist* and KWIK-CorEffect$\star$ (multiple copies)

Table 4.2: The 6 learning settings and their MBP solutions in the Apprenticeship Learning (trace) setting. A * denotes necessary assumptions about the size of conjunctive (pre)-conditions (or other formulas that are only KWIK-learnable when they are small). A $\star$ indicates the number of effects must be bounded by a constant k or some other special structure (like signaled effects) must be known. In all the pre-conditional cases, assumptions on the size of the conditions are dropped with apprenticeship learning. Notice that in deterministic conditional-CED-Learning setting, this assumption is replaced by a different assumption (†) on the *number* of conditions, and in the stochastic conditional CD and CED-Learning cases, we are not able to drop the assumption.

Chapter 5

Web-Service Task Learning

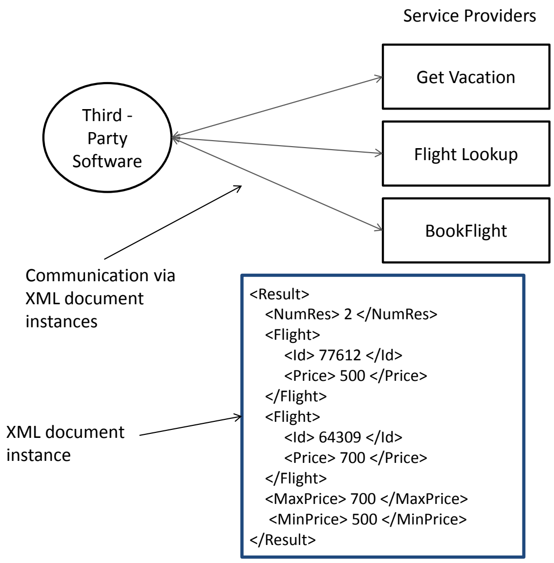
This chapter is concerned with a particular real-world application of relational models for learning in a sequential decision making setting.[1] Specifically, we consider the *web-service task learning* problem. We provide technical definitions of this problem later, but intuitively, the agent must learn to execute a given sequence of web services, such as looking up flights from two cities and then calling another service to book the cheapest one. Each service needs to be invoked with a number of parameters (such as the destination city and trip dates) and certain relations must hold between the objects encountered in the task. Our investigation centers around the sample complexity of learning relational models to support this decision making in the apprenticeship paradigm.

5.1 Web Service Task Learning

"Web Services" (Alonso et al., 2004) is a general term for small pieces of software that can be accessed programatically (or through a user interface) on the world wide web. The availability of services has expanded tremendously in the last 10 years both because content providers (like Amazon.com) have benefited from making their inventory and data available to third parties, and because of the blossoming of small software programs (like those used on mobile devices), that interact with the "cloud" of services. The communication process between providers, third-party software, and services, is illustrated in Figure 5.1. The medium for this communication is (usually) an XML instance document, a formatted and tagged textual description of some data, as seen in Figure 5.1.

The learning problem we consider in this chapter, is whether an agent can observe traces of these XML document instances being passed to and from service providers as part of a larger *task* (such as the flight booking mentioned above) and build relational models that will allow the agent to perform the same task. Since this learning problem is built around real-world

[1] Most of this chapter appeared in joint work with Michael Littman and Alex Borgida (in submission).



```
<Result>
   <NumRes> 2 </NumRes>
   <Flight>
      <Id> 77612 </Id>
      <Price> 500 </Price>
   </Flight>
   <Flight>
      <Id> 64309 </Id>
      <Price> 700 </Price>
   </Flight>
   <MaxPrice> 700 </MaxPrice>
    <MinPrice> 500 </MinPrice>
</Result>
```

Figure 5.1: The interaction between third-party software and web-service providers and an example XML document used as the communication medium.

data, we encounter a number of challenges that were not considered in earlier chapters. These include:

- **New Objects-** Unlike STRIPS environments, where the objects in a domain instance remain static throughout the agent's lifetime, in a web service task, new objects are usually being introduced on every step. For instance, looking up flights to a destination may create a list of new "flight" objects.

- **Non-unique Names-** Real world objects may not have unique names, breaking our earlier assumption that, for instance, two blocks could not both be called **b**. Indeed, non-unique names introduce a key learning problem in web-service task learning. For instance, if we are learning how to use a service for looking up music albums that takes a single input, and we see it invoked with the name of an artist with a self-titled album (the name of the band and the album are the same), this doesn't tell us which of these attributes to use should we want to look up an album with a title that is not the same as the band's name. In such a case, eliminating the ambiguity is essential for learning how to correctly execute the service.

- **States that are not observed as grounded predicates-** In this real application, states will not be described neatly in terms of ground predicates. Rather, we will infer these predicates (encoding structural and semantic relations) based simply on grounded XML documents used for communication to and from the services and the actual relationships of objects in these instances. These inferred relations will often be highly task dependent. For instance, learning that the input to a flight-booking service is always the cheapest flight is the correct behavior for a single task, but may not generalize to all tasks.

- **Missing and Listed Objects-** Web service outputs (and inputs) may often contain missing or optional elements and certain objects (such as flights returned from a search) should be treated as lists, or at least a collection, of similarly typed objects.

These challenges are non-trivial, and encountered specifically because we are working with real-world data. Because of this, we will be using a much different style of algorithms and representation in this chapter, though we now provide a key for mapping from this to the perspective used in the rest of the thesis.

At its core, the web-service task learning problem has all the components of the other problems posed in this thesis, just in somewhat unfamiliar forms. The new problem still calls for a *relational representation* to represent the task and the connections between the objects within the task. For instance, a relations like *PartOf*(Trip, StartCity) and *Min*(Price, FlightPrices) may encode the start city of a trip and the fact that a minimum-price flight is desired. Indeed, a relational representation is *necessary* in this domain class because the task could not be represented with any kind of usable generality (other than performing the same ground task over and over) without variables and a relational representation. The new problem is also still a *sequential decision making* problem, but since the task actions are given to the learner, one should think of the decision making portion of the problem as choosing the *parameters* to the actions (What city do I look up? Which flight do I book?). This decision has a sequential component because the choices have to be considered so that they match up with relations that will hold several actions in the future (we need to pick the flight to the same city where we will soon book a hotel). Viewed from the traditional RL perspective, each set of objects and relations that are known at a step is the state and the parameter choices define an action. Finally, because of the complicated nature of the problem and the large set of possible relations, we use the *apprenticeship-learning* protocol to do the learning, since having the agent autonomously try (for example) every possible string to pass to *lookupFlight* is not practical, but having a teacher showing it an example of invoking this service is quite feasible. As in Chapter 4, the *mistake bound predictor* framework is used to analyze the theoretical properties of the algorithms, limiting how many times a teacher may have to step in. In fact, we provide bounds in the smaller Mistake Bound (MB) framework, since the class of functions we are learning is deterministic. This sample complexity measure is of paramount concern in this real-world setting because collecting traces of users performing the tasks, and especially performing it under rare conditions, may be quite costly. In this light, the web-service task learning problem can be viewed as a novel instantiation of the sequential decision making problems considered in the previous chapters.

5.2 Terminology and Representation

Intuitively, the learning problem we consider is one where an agent must make a sequence of web-service calls to successfully complete a task, such as using the three services represented in Figure 5.2 to look up a person's vacation destination, look up the flights to this location, and buy the cheapest ticket. Whenever the agent's model leads to a mistake with respect to the current task, a teacher steps in and shows it the correct service behavior for that particular instance. This intuitively simple interaction is formalized in the following section, followed by a summary of some of the more vexing learning issues considered in this chapter.

5.2.1 Formal Problem Description

Since web services communicate via XML documents, we describe the semantics of services starting from XML DTDs (schematic descriptions of XML instances) describing their inputs and outputs, though we note that the eventual inputs to our learning system will be XML instances, not DTDs. To simplify the presentation, we eliminate element attributes by simply treating them as child elements. Such DTDs can be represented theoretically using regular expressions over an alphabet defined by their element names. For instance, the expression for "Res" in Figure 5.2 would be: (MaxPrice MinPrice Flight$^+$ NumFlights). Though learning such regular expressions could be intractable, studies have shown (Bex et al., 2006) that the expressions representing XML elements in 99% of XML DTDs on the web are from a restricted language, *chain regular expressions* (CHAREs), where element names can only repeat in a list and quantifiers are only applied to disjunctions of symbols. In this work, we further assume services do not have disjunctive elements (none of our real world experiments contained these), leading to the restricted non-disjunctive CHARE defined below.

Definition 20. *CHARE Definitions:*

- *A* non-disjunctive Chain Regular Expression *or non-disjunctive CHARE is a regular expression with no disjunction (no $|$, only the $^+$, *, and ? qualifiers), where any symbol may only appear once and where qualifiers can only act on a symbol, not a grouping. For instance a^*bc^+ is a CHARE but a^*ba and $(ab)^+$ are not.*

- *A* non-disjunctive CHARE rule *is a DTD element description where the right-hand side is a non-disjunctive CHARE.*

- *A* non-disjunctive CHARE DTD *is a DTD where all the element rules are non-disjunctive CHARE rules.* [2]

We represent such DTDs by trees in the following manner, though we note again that in our setting the tree is learned from document instances.

Definition 21. *A* non-disjunctive CHARE Structure Tree *(henceforth an "XML Structure Tree" for convenience) is constructed recursively by starting with a node n_0, labelled by the root element, and applying recursively the rule that for every node n with element label A, if the DTD has specification*

```
<!ELEMENT A (B_1, ..., B_k)>
```

then children nodes $n_1, \ldots, n_k$ are added to n, with labels $B_1, \ldots, B_k$ respectively. If any of the elements B_j is marked as $^+$, ?, or *, the corresponding node n_j is annotated *with $^+$, ?, or * respectively.*

As a step towards semantic relations (introduced below) we will represent an XML Structure Tree as a graph of the following form:

Definition 22. *An* XML Structure Graph $G_{str} = \{N, E, \Lambda_{str}\}$ *is a labeled directed graph with the same nodes (complete with annotations), as the corresponding XML Structure Tree, but with unlabelled edge (x, y) replaced by two edges: (x, y) labelled "part", and (y, x) labelled "whole". These are called* structural edges, *and $\Lambda_{str} = \{part, whole\}$.*

An example of such a graph is the subgraph of Figure 5.2 containing only the solid (structural) edges (only one of each part/whole combination is shown for clarity). This takes care of modeling the syntax of the document, but we are also concerned with modeling semantics based on mathematical relations as defined here.

[2] We also omit recursion in element definitions in this work.

Definition 23. *A mathematical relation $m \in \mathcal{M}$ is a binary relation between two objects o_1 and o_2 (where zero, one, or both objects can potentially be a grouping of other objects - see the definition of "object" below).*

For the purposes of this chapter, $\mathcal{M}$ can be any arbitrary collection of binary relations and our theoretical analysis provides bounds based on the cardinality of this set. However, to make our examples more concrete, we take a cue from database theory (Klug, 1982) and import five basic functions, which we turn into binary relations: *min*, *max*, *sum*, *average*, and *count*, along with an *identity* relation to check equality of objects. This set seems reasonable since many web services are simply wrappers around database operations. Throughout the chapter, we will extend this basic $\mathcal{M}$ to model increasingly complex tasks, including relations for modeling selection of objects in Section 5.4.3 and modeling relationships between dates in Section 5.6.1.

We now introduce semantic edges based on these relations.

Definition 24. *A* Semantic edge e_{λ_m} *is a labeled edge between two nodes in a graph n_1 and n_2 with label λ_m, where $m \in \mathcal{M}$.*[3]

Semantic edges represent task-specific relationships such as "the City2 input to ***FlightLookup*** should be filled with a DestCity instance from ***GetVacation***". Unlike structural relations, we cannot assume these semantic relations are provided to us directly through the observed instances (XML documents). However, each XML instance does give us information about what relations might consistently hold true. To learn these relations, we assume we have a set of common mathematical relations ($\mathcal{M}$ above), all of arity 2, whose semantics are known and can be easily checked[4].

Throughout this chapter, semantic edges are drawn with dashed lines to distinguish them from structural (solid) edges. A graph with structural and semantic edges is called an *SS-graph.*

Definition 25. *An* SS-Graph *is a labeled directed graph $G_{SS} = \langle N, E, \Lambda \rangle$ containing the nodes from an XML structure graph G_{str} and has edges $E = E_{str} \bigcup E_{sem}$ where E_{str} are the edges from G_{str} while E_{sem} is a set of semantic edges. Similarly, Λ is $\bigcup_{m \in \mathcal{M}} \lambda_m \cup \{part/whole\}$.*

We can now formally define a service in terms of its inputs and outputs, though we do not yet allow for semantic connections between these components.

[3] Intuitively, this represents that for object o_1 (o_2) instantiating node n_1 (n_2) , $m(n_1, n_2)$ is true.

[4] The theoretical efficiency results of this chapter generalize to relations of constant arity by building the corresponding hypergraph.

Definition 26. *A* service *s is a pair of SS-Graphs* [5] $\langle G_I, G_O \rangle$*, where* G_I *represents the inputs and* G_O *represents the outputs. Each node in these graphs represents a concept, which in turn represents either a grounded object or a set of grounded objects. Each node may have an* annotation *from the corresponding XML-Structure tree node (such as the "+" on the Flight node in Figure 5.2) that signifies syntactic properties of the node as described later. The edges in the graphs represent relations between these concepts.*

An example of a service is the **FlightLookup** box in Figure 5.2, which contains a 4-node input graph and 7-node output graph. The semantic edges crossing outside of the box are not part of the service, but are part of the task graph (as defined below). Such a "message generation" definition of a service follows the descriptions of services often seen in the Web Service Composition community (Liu et al., 2007). Since there are often semantic relationships between the inputs and outputs of a service, we define a similar structure that captures these relationships.

Definition 27. *A* service transformation *is a pair* $\langle s, \widehat{E} \rangle$ *where* $\widehat{E}$ *is a set of semantic edges that link the nodes of* G_I *to* G_O*. We assume that the service's behavior is deterministic given the inputs and that the semantic relations encoded in* $\widehat{E}$ *must always hold for any service invocation (though other relations can hold in any single instantiation). This means that while the full input/output mapping may not always be predictable, the constraints in* $\widehat{E}$ *will always hold.*

An example of such a structure is the **BookFlight** box in Figure 5.2 where the input graph (a single node) has a semantic link to the output graph.

The goal of our learning algorithm will be to model a sequence of service transformations, including relational links between transformations. This target hypothesis is called a *Task*, T^*, and is defined as follows.

Definition 28. *A* task $\langle S, R \rangle$ *is a sequence of service names S and a set of relations R between XML elements of the services in S . S defines a partial ordering* $\preceq_S$ *over the elements* e_i *(or collections of elements as defined earlier) of the corresponding service transformations. R is a set of triples* $\langle e_1, e_2, m \rangle$ *where* $m \in \mathcal{M}$*,* $e_1 \preceq_S e_2$*, and* $m(e_1, e_2)$ *is true. Notice that in a task, semantic relations include not only those from each service transformation, but also relations between elements from different services.*

[5] Equivalently, a service can be defined in terms of XML elements that form its inputs and outputs (E_i and E_O) along with a set of relations (R_s) that must hold within these graphs. However, the equivalent graph structure is easier to visualize and has a well defined construction from data (Definition 21) so we use this representation here.

Intuitively, S represents what is called the "control flow" in process mining (Yaman et al., 2009)—a sequence of service calls (ignoring the inputs and outputs) necessary to complete the task (see the top of Figure 5.2). R (representing the dataflow) encompasses relations between objects in service graphs G_i and G_j, where $G_i \preceq_S G_j$, including relations within (i) the same graph (a service), (ii) relations between the inputs and outputs of a service (a service translation), and (iii) those between graphs associated with different services (task specific relations).

Internally, we will represent a web-service task using a *task graph* as defined below. The edges in the task graph include all the structural edges from the individual services, but also include directed edges (direction in the figures flows downward) for the semantic relationships between objects. Intuitively, the semantic edges, including those associated with the *identity* relation (shown as unlabeled dashed edges), maintain a valid hypothesis over semantic relations between objects, including which outputs link to which inputs and the semantic relationships between objects in general.

Definition 29. *A* Task Graph *is a labeled directed SS-Graph representing a task. The annotated nodes in the graph are all of the SS-Graphs in the service transformations* $\bigcup_{\langle N,E\rangle \in s \in R} N$ *and the edges correspond to the union of all the edges in those graphs as well as inter-service (type iii above) semantic edges corresponding to the semantic relations* R *in the task* T^* *as defined above.*

We now consider the instantiations of these structures. The figures in this chapter showing task graphs all show *task graph instances* because we feel they are easier for readers to follow. Here we formally define each type of instance and, where necessary, show how the instances are populated from the original XML documents.

Definition 30. *An* object *is a grounded element (*#PCDATA*)* [6] *or a* collection *of grounded elements from an XML document entry A. A* collection *is either a* list *of entries* $[a_1, a_2...]$ *if A repeats in the document or a* sublist *if an element nested above A repeats (denoted* $\langle a_1, a_2...\rangle$*). Note that if both cases hold, one can have a sublist of lists (for example,* $\langle [a1, a2], [a3, a4, a5], [a6]\rangle$*).*

For example, the $Flight^+$ node in Figure 5.2 has a list associated with it (horizontally tiled objects in the diagram) while the Price node under it is associated with a sublist (vertically tiled objects in the diagram).

[6] For nodes with parts, we hash all the text (including tags) below the elements so that we can check for exact equivalence between structures. This results in the "Ref" objects in the figures in the chapter.

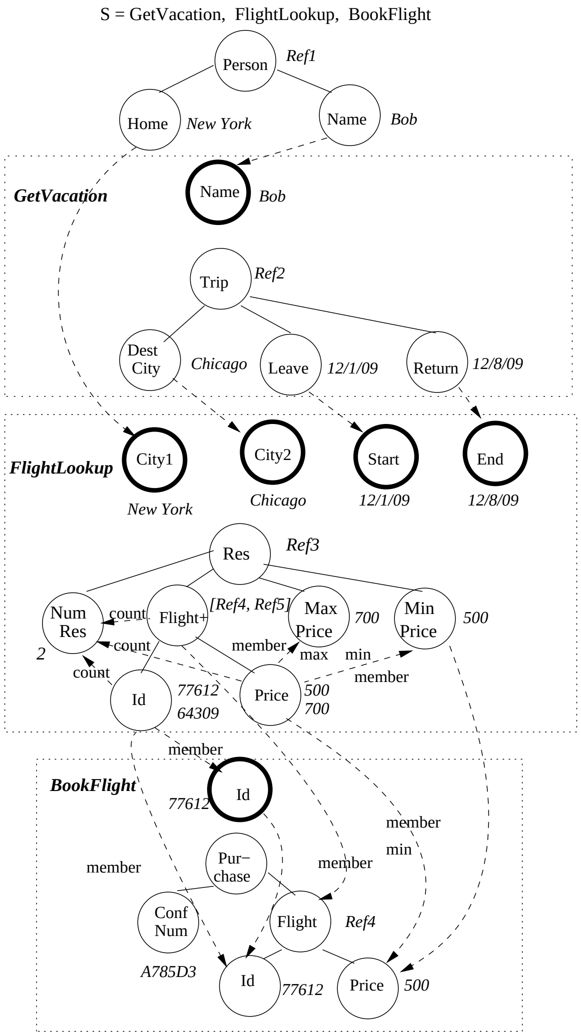


Figure 5.2: A task graph instance for an agent looking up a person's vacation information, searching for flights, and booking the cheapest one. An XML trace provides the structural (solid) edges, but not the semantic (dashed) edges, which must be learned. Inputs to services are shown with bold circles. Objects appear next to their corresponding nodes. When multiple objects map to the same node, they are grouped either as a list (the horizontal tiling for Flight$^+$) or as a Sublist (the vertical tiling for Price) based on Definition 30 .

Definition 31. *An* XML Structure Graph Instance *is a tuple $\langle G_{Str}, \mathcal{I} \rangle$ where G_{Str} is an XML Structure Graph and $\mathcal{I}$ maps each node n in $G_{Str} = \langle N, E_{Str} \rangle$ to an object o. In our diagrams, we place instances besides the corresponding nodes.* [7]

Since this definition associates objects with nodes, we outline the rules for performing this mapping given an XML document D and non-disjunctive CHARE Tree.

1. If element A is a child of an element which is a list, then the ground instances of A in the document are put into a "sublist" collection $\langle a_1, a_2 ... \rangle$. This sublist is associated with the graph node corresponding to A, but each a_i is also parsed based on the next two rules, as it may contain several (or no) instances of A in a list.

2. If an instance of element A in the document is optional and missing, or if any of A's parents through structural edges are optional and missing, an empty object [] is associated with the corresponding node (or the associated a_i if the previous rule was used).

3. If an instance of element A repeat as a list ($^+$ or * - detected as siblings in the document), these ground instances are put into a list collection $[a_1, a_2 ...]$. If the "sublist" rule above already partitioned the instances of A, the list is associated with a single a_i in the sublist, resulting in a "sublist" of lists.

Now that we have covered the instantiations of nodes, we turn our attention to edge instances and their validity.

Definition 32. *A* semantic edge instance *is a semantic edge connecting two instantiated nodes as defined above. A semantic edge instance is* semantically valid *if and only if $m(o_1, o_2)$ is true for $\mathcal{I}(n_1) = o_1$ and $\mathcal{I}(n_2) = o_2$.*

Instances of SS-Graphs, service graphs, and task graphs are all obtained by associating with them an instantiation function $\mathcal{I}$ coming from the underlying XML structure graph. That is, service graph instances still contain all the nodes and edges from the corresponding service graph, but each node is associated with a number of specific individuals, such as "*Chicago*" for the DestCity node in Figure 5.2. We say such graphs are *semantically valid* if all of their semantic edge instances are semantically valid.

We are concerned with learning task graphs from *traces* (task instances) of the task. Traces do not necessarily contain direct information about the semantic relations; instead they record

[7] Note that this differs from the standard tree representation of XML documents, because repeated elements resulting from A^+ are recursively collapsed in the structure graph.

a sequence of service instances that give clues as to these semantic relations. For most web services, traces can be obtained as sequences of XML documents exchanged by the client and the service. These can either be collected prior to the learning process or produced by a *teacher* when the learner makes a mistake. With the help of the above definitions, we can now define the task learning problem, which will be our main consideration throughout the rest of this chapter. Notice that this protocol is a real-world instantiation of the Apprenticeship learning protocol (Algorithm 24), but now with predictions explicitly built in (which MB and MBP are amenable to).

Definition 33. *Initially, the learner is provided with the names of the services to be called (S from the task definition), the set of relations to be considered $\mathcal{M}$, and an initial task instance (trace) $\mathcal{T}_0$. The* task graph learning problem *then occurs in a series of episodes. At the beginning of each episode, an initial SS-Graph G_0 is provided to the agent. The agent must then for each successive service in S :*

1. *Provide the instances of the input elements (a semantically valid SS-Graph G_I). The reasoning behind this* input-selection problem *once a model has been constructed is discussed in Section 5.5.*

2. *Make correct predictions about what semantic relations will hold with respect to the true (but unknown) Task T^* (predicting R). When possible, this may involve predicting the actual instantiations of nodes, as in the input-selection problem above, but in this case the agent may also make a more generic prediction, simply stating the relationship between two nodes (for example, Node $n1$ will contain the maximum value from the (not yet instantiated) list in node $n2$).*

3. *Predict the structure of each service instance, including annotations. Specifically, the agent must identify all possible nodes n and structural edges e_{struct} in the service instance as well as whether these nodes can be optional, lists, or both. However, the exact instantiations (the values assigned to each node) do not have to be predicted, except for the inputs as specified above.*

If during an episode, the agent errs in any of these, and if the current task instance refutes the agent's prediction, this is counted as a mistake *and the agent is provided with the task instance (trace $\mathcal{T}_t$) showing a full run of the episode and all the instances.*

For instance, in Figure 5.2, the agent is initially given the "Person" structure at the top. The agent must then make the correct sequence of service calls, also using the correct inputs for each service from the previous outputs (or as best it can in accordance with the true R from T^*) and making correct predictions about the data produced, as outlined above. For instance, an agent that has learned the task graph in Figure 5.2 can predict that a ***FlightLookup*** will produce a list of at least one Flight (from the annotation $^+$ on Flight) and that the MaxPrice node will contain the maximum value from the Price node. If the agent either makes any mistakes (as defined above), it receives a trace (task instance $\mathcal{T}$) as feedback. This feedback can either be a correct trace for the previous episode (as defined above and considered throughout this work), but more generally could be any example that will correct the misconceptions that led to the mistake (for example a stored trace from a similar task).

We consider the efficiency of task learning in the *Mistake Bound* paradigm.

Definition 34. Efficient Task Learning *occurs if the number of mistakes it makes is bounded by a polynomial function of the input parameters* $\{|G_I|, |G_O|, |S|, |\mathcal{M}|\}$.[8]

In practical terms, this constraint ensures we can train agents to perform complex tasks involving web services with a number of examples that scales polynomially with the size of the task schematic. This efficiency is crucial for any practical realization of this system as traces of specific tasks, while not necessarily scarce, certainly will be limited.

5.2.2 Key Learning Problems

While the presence of traces certainly makes the web-service task learning problem easier than a completely unsupervised approach, a number of non-trivial learning tasks remain. Here, we provide a sketch of the key challenges in the web-service task-learning problem.

- **Ambiguity** - A single (or even multiple) traces may not settle all the semantic relations between objects. For instance, if two lists are visible to a service (A=[1,2,3] and B=[3,4,5,6] and the service produces "3": was that max(A), min(B), or length(A)? Further traces are required to determine the correct pairing (if there is one).

- **List Structures** - Lists of objects are ubiquitous in web services. Figure 5.2 shows one such output with the list of possible flights. Detecting and modeling lists, including learning when

[8] $|G_I| = \max_j |G_{Ij}|$, and similarly $|G_O| = \max_j |G_{Oj}|$.

lists are potentially empty, is an important portion of the overall task-learning problem. While WSDL documents or other syntactic schemas often indicate which elements may repeat, we show in this chapter that under very common assumptions, the presence of lists and missing elements can be learned as well. This makes our learning algorithm more practically robust. It can construct valid models when WSDL are missing, incorrect, or in the presence of non-web, proprietary, or in-development services that may not have any such documentation.

- **Sublists and Selection** - When lists contain non-primitive structures, the parts of the elements in the list (for example, the Price of a Flight from the flight list in Figure 5.2), form a *sublist* as covered in Definition 30[9]. The elements of a sublist are not grouped together in the original XML document, but, we may need to consider them as a group to learn some semantic links. Additionally, the dataflow from some nodes may best be expressed in terms of the grouping induced by a "whole". For instance, in Figure 5.3, when a Flight is chosen, its "Stops" list is copied over, so the dataflow should capture the fact that these stops all belong to the same flight, and if possible, the reason for this selection.

5.3 Simple Task Learning

Before we handle the general form of the learning problem presented in Definition 33 , we consider a simplified web-service task-learning problem under a number of assumptions. We assume lists (for example, the Flight list in Figure 5.2) are not nested inside one another and can never have length 0 (no missing elements). We further assume that any time multiple parts of an object that came from a list appear in a later SS-Graph, the entire structure (not just a few parts) will appear in the later graph. For example, the entire Flight object appears in the output graph of ***BookFlight***, not just the Price and Id. In subsequent sections, we will relax all of these conditions, but we study this "simple task-learning problem" to convey the basics of our learning algorithm.

5.3.1 Mapping XML to Structure Graphs

Each input (trace $\mathcal{T}$) to our learning algorithm comes as a series of XML documents showing the inputs and outputs of each service instance. If the syntax of each service (what elements are lists and which ones are optional) is provided through correct WSDL or other documentation,

[9] In the figures, a horizontal tiling (Flight$^+$) is a list of instances and the vertical tiling (Price and Id) is a sublist (parts of listed elements).

translating these instance documents to XML Structure Graph instances can be done using the procedure outlined in Definitions 21, 22, and 30. However, we consider here the more general situation where this documentation is not provided, and hence the translation from XML documents to a task graph instance requires learning the syntax of each service's inputs and outputs (XML structure subgraphs of G_I and G_O). As noted earlier, this syntax (the XML-structure tree) can be represented using a regular expression for each element of the document. For instance, the expression for "Res" in Figure 5.2 would be: (MaxPrice MinPrice Flight$^+$ NumFlights). With only the traces to work from, these forms must be learned from multiple traces because each trace instance may only provide partial information about whether an element is a list (singletons and lists of length 1 are often indistinguishable) or optional (covered later).

In general, learning such regular expressions could be intractable, but studies have shown (Bex et al., 2006) that the expressions representing XML elements in 99% of XML DTDs on the web are from a restricted language, *chain regular expressions* (CHAREs), where element names cannot repeat unless they are in a list and quantifiers appear only on disjunctions of symbols. This constraint means when we see two "Flight" elements under a "Res" element, we can infer it is a list (Flight$^+$ or Flight*), not a sequence of two Flights. In this work, we further assume services do not have disjunctive elements (none of our real world experiments contained these), and therefore quantifiers are only applied to each element.

From that result, translating XML documents to graph nodes and *structural* edges in an XML Structure Tree forming the backbone of a Task Graph G_T is straightforward, even in the online-learning case. Each document maps to a specific service $\langle G_I, G_O \rangle$. Each XML element can be represented by a CHARE (where repeated elements will have a $^+$) and each symbol in the CHARE becomes a node in the graph. If there is a $^+$ on this symbol and the node in the graph does not yet reflect it, the annotation is added (other quantifiers are considered in the next section). Part-Whole relations and instances are then copied in from the XML. The instances of each node are populated from all the corresponding XML elements. Thus, under the assumptions above, learning the XML-structure trees within the true Task Graph G_T^* is tractable. The efficiency is discussed in the next section and modifications to the structure learning when these assumptions are relaxed are discussed in Sections 5.4.1 and 5.4.2. But now we turn our attention to the goal of learning the full Task Graph with semantic relations included, not just the syntactic XML-structure tree.

5.3.2 Learning Simple Task Graphs

Our goal is to construct a model of the syntactic and semantic relation in the true task T^* with at most a polynomial number of mistakes. The Task Graph Learning Algorithm (TGLA: Algorithm 29) does so using the Task Graph (G_T) representation by ruling out possible semantic relations between elements based on traces. TGLA can be viewed as a specific instantiation of classical machine learning techniques, specifically version-space learning (Mitchell, 1997), and its compact representation and attention only to the attributes of each successive instance have parallels in "learning from infinite attributes" (Blum, 1992).

Algorithm 29 Task Graph Learning Algorithm (TGLA) for Simple Tasks

```
The agent starts with $\mathcal{M}$ and an initial trace $\mathcal{T}_0$
Construct S and $\preceq_S$ exactly from $\mathcal{T}_0$
Extract structure of $G_I$ and $G_0$ for every service in $\mathcal{T}_0$
$G_T = \bigcup_j G_{Ij} \cup G_{Oj}$
for Every pair of nodes $(n_i, n_j) \in G_T$ where $n_i \preceq n_j$ and every $m \in \mathcal{M}$ do
  if $m(n_i, n_j)$ holds for the instances of those nodes then
    Construct the corresponding edge $\langle n_i, n_j, \lambda_m \rangle$
  end if
end for
for each episode do
  The agent receives an initial set of instances $G_{O0}$
  Execute each service in S by choosing inputs and making predictions from the semantic
  links throughout the task (in $G_T$)
  if Trace $\mathcal{T}$ received then
    Eliminate all semantic edges in $G_T$ refuted by $\mathcal{T}$
    Update annotations on list nodes ($^+$).
  end if
end for
```

Since we are temporarily assuming each XML element appears at least once, the first trace bootstraps all the nodes in the task graph, though $^+$ annotations may still need to be refined, and there are potentially incorrect semantic links. For instance, if the first trace in our flight-booking domain had a person whose name and home city were both "Austin", then both the "Home" and "Name" nodes in the graph would link (through the identity relation) to the "Name" node in ***GetVacation***. This will be problematic if the next episode starts with Bob from New York. Should the agent call ***GetVacation*** with "Bob" or "New York"? In the mistake-bound setting we have considered, when such ambiguity exists the agent can just pick one of the possible choices. If it is wrong, it will receive a trace and be able the eliminate the incorrect link. Even with more complicated semantics that require super-polynomial computation to make predictions (see our overview of reasoning with several different variant of $\mathcal{M}$ in Section 5.5), it only takes one trace with contrary information to remove a link.

Formally, we can exploit the intuition above to bound the number of mistakes. We begin with a lemma showing that the true task graph G_T^*, is sufficient for guaranteeing no mistakes will be made.

Lemma 9. *Using the true task graph G_T^* to make predictions and choose inputs in the task learning problem will not produce any mistakes.*

Proof. We consider all the possible sources of a mistake. First, an agent could provide the wrong inputs to a service. But using G_T^*, the agent knows all the valid semantic links between the input to the service and the currently known objects produced by other services (the inference is done globally over the links in the task, not just the current service), so the agent will be able to evaluate each potential input o' by checking each semantic edge $e = \langle n_1, n_{in}, \lambda_m \rangle$ leading into the input node n_{in} (by evaluating $m(\mathcal{I}(n_1), o')$. We note that for complex semantics the reasoning may require super-polynomial computation, as covered in our discussion of selecting inputs (Section 5.5). However, this does not affect the sample complexity.

Mistakes can also be made when the agent declares what semantic relations will hold between elements of the task, but since edges in G_T^* appear if and only if they correspond to relations in R , this cannot happen.

The only other way to make a mistake is to make an incorrect prediction about the structure of a service, (in this case predicting a $^+$ annotation) but the nodes of G_T^* have the true annotations of the true XML structure graph for all the services, so such a mistake cannot be made. □

Now we will show that Algorithm 1 converges to the true task graph G_T^* in the realizable setting with no more than a polynomial (in the relevant quantities) number of mistakes. The argument simply considers the number of changes made to the task graph as the learning algorithm generalizes it on every mistake (similar in nature to the conjunction learning algorithm from the previous chapter).

Proposition 7. *SimpleTaskLearn makes $O(((|G_O| + |G_I|)|S|)^2|\mathcal{M}|)$ mistakes in the simple web-service task-learning problem when the target semantics are representable.*

Proof. The initial Task Graph (T_{G0}) is constructed from the task instance $\mathcal{T}_0$ and then refined in subsequent episodes. T_{G0} is made up of the XML structure Tree and semantic edges that are valid with respect to $\mathcal{T}_0$.

There are at most $((|G_O| + |G_I|)|S|))$ nodes in the entire task graph which means there are $((|G_O|+|G_I|)|S|)^2$ possible unlabeled edges, which is then multiplied by the number of potential labels, $|\mathcal{M}|$ (since $|\mathcal{M} + 2| = |\Lambda|$) to give us $O((((|G_O| + |G_I|)|S|)^2|\mathcal{M}|)$ possible semantic edges represented in $\mathcal{T}_0$.

In each episode, the current G_T is used (as in Lemma 1) to choose the inputs and make predictions about what relations hold and what structure will be seen in each subsequent service instance. If multiple edges leading into a node suggest different semantics (for instance if a "min" and "max" edge both appear between two nodes), one is picked arbitrarily.

For each of the predictions made based on semantic edges $\langle n_1, n_2, \lambda_m \rangle$, one of three cases applies (where we use $m(n_1, n_2)$ as shorthand for $m(\mathcal{I}(n_1), \mathcal{I}(n_2)))$:

1. The prediction is correct, $m(n_1, n_2)$ is true, and no other edges $\langle n_1, n_2, \lambda_{m'} \rangle$ for $m' \neq m$ were proven invalid. No action needs to be taken.

2. The prediction is correct, $m(n_1, n_2)$ is true, but another edge $\langle n_1, n_2, \lambda_{m'} \rangle$ for $m' \neq m$ has been shown to be invalid with respect to the current task instance. The latter edge is removed from the graph.

3. The prediction is incorrect, $m(n_1, n_2)$ is false, resulting in a mistake. This edge, and all other edges $\langle n_1, n_2, \lambda_{m'} \rangle$ for $m' \neq m$ that are invalid with respect to the subsequent trace (task instance) are removed from the graph.

The worst case, in terms of the mistake bound, is that all the graph refinements after $\mathcal{T}_0$ occur because of case 3 and that only one refinement happens per task instance. However, because there are at most $O(((|G_O| + |G_I|)|S|)^2|\mathcal{M}|)$ edges in G_T to begin with, and because edges are never added back into the graph once they are removed, only $O(((|G_O|+|G_I|)|S|)^2|\mathcal{M}|)$ such mistakes can be made.

All that is left now is to bound the number of mistakes made when predicting the $^+$ annotation (other annotations later in the chapter are dealt with similarly). There are $O(|G_O| + |G_I|)|S|)$ nodes, and initially all of those that are not lists in $\mathcal{T}_0$ are considered singletons (no annotations). The algorithm can only make mistakes when predicting these cannot be lists when in fact they turn out to be lists, at which point their annotation is changed to $^+$ and never changes back. Since there is evidence that these elements can occur as lists, this must be the true annotation for the node in the service. This annotation learning introduces $O(|G_O| + |G_I|)|S|)$ mistakes.

In the realizable case, where G_T^* exists, the edges in T_{G0} but not G_T^* will be eliminated and all incorrect annotations will be made (a simple inductive argument shows that the same mistake is never made twice by the agent), so the algorithm converges to G_T^* and makes at most $O((((|G_O| + |G_I|)|S|)^2|\mathcal{M}| + (|G_O| + |G_I|)|S|)) = O((((|G_O| + |G_I|)|S|)^2|\mathcal{M}|)$ mistakes. □

5.4 Full Task Learning

We now relax the earlier restrictions, allowing nested lists, missing elements, and selection semantics. Each change, leads to increased complexity for TGLA, though the respective bounds for all of these extensions remain polynomial. To demonstrate these properties, we introduce a second "Flight Booking" example in Figure 5.3. This example contains several features that were formerly prohibited, including nested lists, optional elements, and portions of complex structures "selected" from earlier services. The complexity of learning such Task Graphs is considered in the following subsections.

5.4.1 Nested Lists

Allowing nested lists (as with "Stop*" and "Eatery$^+$" in Figure 5.3) requires a change in the definition of $\mathcal{M}$. As a matter of notation, we denote the maximum nesting depth, (which is 3 in Figure 5.3, but at most $\max(|G_O|, |G_I|)$), as d in subsequent bounds. Once $d > 1$, it is possible for sublists to also contain lists (as with the "Eatery$^+$" node). Hence, the semantic links may become ambiguous (does "member" mean the node contains a single list from the sublist, or is it a sublist of members from each list?). This is rectified by creating 2 versions of each relation $m \in \mathcal{M}$: *list-m* and *sublist-m*. We use these different forms of the relations (as defined below) to resolve the ambiguity discussed above, though we note that other changes to $\mathcal{M}$ are possible to achieve the same desired effect. *list-m*(n_1, n_2) has the following semantics:

- If n_1 has an associated list instance, and $m(\mathcal{I}(n_1), \mathcal{I}(n_2))$ is true, then the edge is valid.
- If n_1 has an associated list instance and n_2 has a sublist instance of elements $e_1 ... e_n$ then if $\forall_i m(\mathcal{I}(n_1), \mathcal{I}(e_i))$ is true, then the edge is valid. This is to indicate, for instance, that all the single elements in n_2, such as carriers for different flights, which are only grouped because of shared parent structure in the XML, are members of a list from n_1 (perhaps a

list of acceptable carriers given as input) [10].

- If n_1 has an associated sublist of lists $l_1 ... l_n$ (as in the "Stops*" node in Figure 5.3), n_2 must be instantiated with a corresponding sublist where for each element $e_1 ... e_n$, $m(l_i, e_i)$ is true for the edge to be valid. This produces a "mapped" version of m as with the "list-count" edge connected to Stops* in Figure 5.3.

If none of those cases hold, the edge is not valid. *sublist-m* has the same behavior in the first two cases above (but with sublists in n_1 instead of lists), but in the third case, when n_1 contains a sublist sl of lists as in the "Stops*" node in Figure 5.3, n_2 must contain an element e such that $m(sl, e)$ is true. For instance, "sublist-count" applied to the Stops* node in Figure 5.3 would link to a node containing "3", the number of lists in the sublist. As a second example from Figure 5.2, list-count on the Price node would only match another node with the sublist [1,1] (price looks like it has two lists, each of length 1), while sublist-count will only match Price to a node containing the single instance 2 (the length of the sublist). This is important for being able to maintain the list of valid relations since lists of size 1 and singletons are essentially indistinguishable in the XML documents. For simplicity, where ambiguity does not exist in our examples, we omit this distinction. This extension only increases $|\mathcal{M}|$ by a factor of 2 and therefore does not affect the bound stated earlier. However, the presence of nested lists gives rise to several difficulties regarding missing elements and instance selection. We now consider these problems in detail.

5.4.2 Learning with Missing Elements

Sometimes, the results from services can have missing elements. For instance, the non-stop flight in Figure 5.3 has an empty Stop list (and corresponding sublists). This structure is captured in XML Structure tree nodes using the * annotation for potentially empty lists and the ? annotation for potentially missing singletons. We now add these quantifiers to the possible annotations of the nodes in the task graph G_T the same way we utilized the $^+$ annotation on the Flight node in Figure 5.2. As with $^+$, it is possible that WSDL or other documentation provides this information before the task-learning problem begins, in which case no learning about these syntactic forms needs to be done. But we consider here the worst case situation

[10] Similar relations can be considered when n_2 contains a list, but since sublists are "artificially" grouped, the relation mentioned here seems to occur more frequently

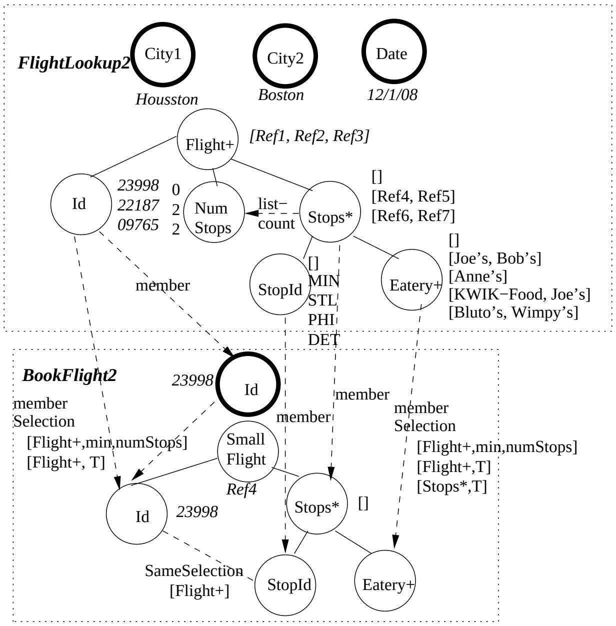


Figure 5.3: A partially learned task graph instance. Only some Selection edges are shown (for clarity). SameSelection links can exist between all the parts of SmallFlight. Further learning (on instances with no 0 or 1 stop flights) could eliminate the extra Selection label ([1,T]) shown for Eatery$^+$.

where all we have is traces. In that full learning setting, these new annotations are adjusted in the following ways by TGLA given a trace:

- If a node exists with no annotation or with $^+$, but an instance does not appear in the current trace, change the annotation to ? or *, respectively.
- If the trace contains a new node and it is not in G_T, create the new node with annotation ? or *, depending on whether the new node should be a list or not.
- If a node has no annotation or is annotated with ? and the current trace contains a list in this node, change the annotation to $^+$ or *, respectively.

Note that once a node has been determined to be optional or a list (or both), its annotation never goes back. We also need to deal with the semantic edges for these nodes, which might exist between optional nodes and required nodes (the "list-count" connection between NumStops and Stop). However, if the optional node has never been seen, we cannot test this relation. Therefore, we make the following adaptations to *SimpleTaskLearn*:

- When an optional node first appears in a trace, add in all edges to and from this node not refuted by current instances. This is the same initialization that formerly happened only with $\mathcal{T}_0$.

- When edge semantics are tested, we still test every edge, even if one end contains no instances. The underlying semantics of m_λ determine the validity (count([], 0) can be true, but max([], 7) will come back false).

Although nodes and edges may now be initialized in episodes other than the first, the number of potential edges in the task graph as a whole is still $O(|S|(|G_I| + |G_O|)^2)$, so the resulting mistake bound for TGLA is on the same order as before.

5.4.3 Selection Relations

We now relax assumption from Section 5.3.2 , which required that any time more than one part of a "whole" structure in the task graph repeated later in the task, the entire structure reappears. This assumption does not generally hold for common web service tasks. For instance, when buying a product on Amazon, after finding an offer, the buyer may need to enter the product and seller ids in a form, but shouldn't have to copy in the manufacturer name, country of origin, or all the other non-essential properties of the product in this purchasing form. To model such "selection" of parts of a compound object, we introduce a new set of labels for semantic edges based on templates defined below, but first we cover a more concrete example of this situation.

In Figure 5.2, one of the flights from the Flight$^+$ list in ***FlightLookup*** appears exactly copied in the ***BookFlight*** output. Because of this, semantic links emanate from the parts of the Flight$^+$ node *and* the Flight$^+$ node itself (note the "member" link between Flight$^+$ and Flight). From these links, a reasoner could determine that the minimum priced flight should be chosen and that Id should be fed as input to ***BookFlight***. However, in general (and as we have seen in our Amazon.com and Google experiments), web services do not repeat the same exact structure between services. More commonly, a few elements of the larger structure appear after a member is selected from a list, as seen in Figure 5.3, where the "NumStops" node is omitted in the ***BookFlight2*** output. This omission prevents not only reasoning about how this flight was selected, but also modeling the connection between all the parts in the output of ***BookFlight2***. They are not just a random collection of the members of sublists from ***FlightLookup2***, they have a semantic connection based on a shared "whole" instance. These are stops for a single (and specific) flight. For instance, if the third flight is chosen, then the ***BookFlight2*** output should certainly not have a StopId of *STL*, but just having the "member" link does not assert this relation.

To address this issue, we introduce *derived* semantic relations based on two templates defined below. Labeled edges with these semantics will give us a way of predicting user preferences in selection as well as maintaining groupings of instances based on shared ancestry in the XML document. Our experiments with tasks comprised of services from Amazon and Google (see Section 5.6) indicate that modeling such groupings is crucial for capturing the true semantics of tasks. Such preferences could be arbitrarily complex, so we focus here on a restricted subset of queries conforming to simple XPath[11] expressions of the form node[simplePredicate]/node/node/..., indicating the place (from the nodes) and simple reasoning (from the predicate, such as "minimum") of the selection. We do so as a proof of concept, to show how the sample complexity changes with these expanded semantics. Learning about more complex preferences is beyond the scope of this work.

The $Selection[n_{anc}, m', n_{pref}](n_1, n_2)$ template encodes a set of binary relations between two nodes n_1 and n_2, (like the two Eatery$^+$ nodes in Figure 5.3) where n_2 contains a subset of n_1's instances, and those instances were all parts of a larger "whole" structure above n_1 (for example, eatery lists for all the stops on the same flight). Each relation instantiates the following:

- n_{anc} the "ancestor" of n_1 where the "whole" instance was chosen. (from n_1 = Eatery$^+$ that's either Stop*, or Flight$^+$.)
- m', a semantic relation comparing two objects. For our purposes we assume that $m' \in \mathcal{M}' \bigcup \top$, where $\mathcal{M}' \subseteq \mathcal{M}$ and $\top$ is a wild-card relation explained below.
- n_{pref} is a child node in the same service graph (G_I or G_O) as n_1 and is reachable from n_{anc} by "part" edges without traversing another list node.

Intuitively, *Selection* edges say that n_2 contains a subset of the instances in n_1 all of which descend from a single instance of n_{anc}, picked over others in that node because the corresponding instance at n_{pref} satisfies $m'(n_{\text{pref}})$. For instance, the Eateries in the ***BookFlight2*** service (n_2) are a subset of those in the Eatery$^+$ node for ***FlightLookup2*** (n_1), and were chosen from a flight (n_{anc}) based on min (m') NumStops (n_{pref}).

For the purposes of our examples we consider $\mathcal{M}' = \{min, max\}$. Because these functions will not be able to model all preferences, the wild-card $\top$ is used to indicate that *some* selection

[11] http://www.w3.org/TR/xpath

of an instance from a node is being done, but we cannot qualify it with the relations in $\mathcal{M}'$. Overall, the number of possible instantiations of this template is $O(|\mathcal{M}'||S|d\max(|G_O|, |G_I|))$.

If two nodes in the same service graph (like Id and Eatery$^+$ in ***BookFlight2***) have Selection links with $m' \neq \top$, then their shared ancestry is easily checked from n_{anc} and n_{pref}. But, if $m' = \top$, one can't tell simply from the Selection links if, for instance, Id and StopId are chosen from the *same* flight instance. To combat this, we introduce a semantic relation that encodes such shared ancestry: $SameSeletion[n_{anc}](n_1, n_2)$. We only consider this relation between nodes n_1 and n_2 in the same service graph (G_I or G_O) where both have Selection links referencing the same node n_{anc}. Considering both templates, we have expanded Λ from its original $(|\mathcal{M}| + 2)$ to $O(|\mathcal{M}| + |\mathcal{M}'||S|d\max(|G_O|, |G_I|))$, leading to the following result.

Proposition 8. *Modifying TGLA (Algorithm* 1*) with the extended semantics described above makes* $O(((|G_O|+|G_I|)|S|)^2(|\mathcal{M}||S|d\max(|G_O|, |G_I|))$ *mistakes in the web-service task learning problem when the target semantics are representable with a Task Graph.*

Proof. The full proof of this Proposition is similar in form to that of Proposition 7, so here we simply outline the differences between the two. The full task graph has $O((|G_O| + |G_I|) * |S|)^2$ nodes. Selection and SameSelection introduce $O(\mathcal{M}'|S|d\max(|G_O|, |G_I|)) + O(|S|\max(|G_I|, |G_O|)) = O(\mathcal{M}|S|d\max(|G_O|, |G_I|))$ edge labels. Each of these edges can be checked by simply grouping the instances as per the edge parameters (a polynomial time operation). Thus, the edges can be introduced as each node appears and checked for validity against each trace just as before. Finally, we recall that $d = O(\max(|G_I|, |G_O|))$, so the sample complexity is polynomial in the parameters of the task-learning problem. □

5.5 Reasoning with Task Graphs

While the bulk of this chapter concerns the problem of *learning* a Task Graph model from traces of users performing a task, for an agent to automatically perform the task itself it must be able to select the correct inputs for each service based on the objects already encountered in the task. Note this problem of choosing the inputs for each service in the task is also a requirement of the online learning problem defined earlier (Definition 33), but here we turn our attention to the complexity of this reasoning process itself *given* a Task Graph.

We begin by formally defining the *Input-Selection* problem, where an agent has a known task $T = \langle S,\text{R}\rangle$ comprised of services $S = [s_1...s_m]$ (see Definitions 28 and 26). The agent also has a "maximal Task Graph" $G_T = \{N, E, \Lambda_{\mathcal{M}}\}$ corresponding to T (see Definition 29) (maximal in the sense that every true task relation in R is represented as an edge in G_T). The agent is also given an initial SS-Graph Instance corresponding to the initial known objects (for example, the "Person" structure and objects in Figure 5.2). The agent must now, for each service s_i, provide an SS-Graph Instance that is a valid instance of G_{Ii}, the input graph for s_i. Validity refers here not only to the structural and semantic edges within G_{Ii}, but since this is a subgraph in the larger G_T, the objects in the instance of G_{Ii} must be valid for some instantiation of G_T, given that the objects for nodes corresponding to prior services ($s_j \prec s_i$) are already set. For instance, when picking an input to the **BookFlight** service in Figure 5.2, the one-node G_I does not encode any constraints, but the larger G_T (even though part of it would not yet be instantiated) encodes that this input must be a member of the previous service's "id" list, and even more, must correspond to the id of the lowest cost flight. Note that the latter reasoning requires looking *forward* to a portion of G_T that will not be instantiated until after this service call, but nevertheless encodes important constraints. After each service call, the agent receives the output of the service (G_{Oi}), and can therefore instantiate that part of G_T (bind objects to nodes), so intuitively the problem is one of deciding the inputs to each service in a task, given a partially instantiated task-graph.

We note that unlike the learning problems considered earlier, this is purely a *computational* task, so the worst-case bounds will be in terms of computation time, not sample complexity. The complexity of the input-selection problem is highly dependent on the mathematical relations $\mathcal{M}$ behind the set of edge labels in the given task graph. Because of this, a full investigation of the complexity of reasoning given any combination of constructors is not practical. However, we provide below a snapshot of two $\mathcal{M}$'s that show how easily the complexity of input selection can jump from trivial to intractable depending on slight variations in the allowed semantics.

We begin with the simplest semantics, where $\mathcal{M} = \{=\}$, that is only equality relations between objects are considered. This corresponds to a task graph where all the semantic edges enforce equality between the instances corresponding to each node. This leads to the following simple result:

Proposition 9. *With $\mathcal{M} = \{=\}$, the input-selection problem for a given service s_i with corresponding input graph G_{Ii} comprised of n nodes with maximum degree δ can be solved in $O(n\delta)$ time.*

The result is straightforward—for every node n_j in G_{Ii}, one can simply check every semantic edge and see if the connected node in G_T is instantiated. If so, this value is object is copied into n_j. If not, then another link is checked. If no links lead to instantiated nodes, then an arbitrary object can be used because there are no constraints on previous services (since we assume G_T fully captures task T and we only need to make sure the relationships specified by T, not any arbitrary relationships that otherwise hold in the instance, are satisfied). Following longer paths of equality links is not necessary because of the transitivity of equality—any longer paths that lead to an instantiated node n_k means there will also be an edge between n_j and n_k because of our "maximal task graph" assumption [12]. However, as we have seen, task semantics usually require far stronger relations than equality, so we now consider a different version of $\mathcal{M}$ where the input-selection problem becomes intractable.

Proposition 10. *The input-selection problem with* $\mathcal{M} = \{member, sum, min, count, =\}$ *is NP-Hard.*

Proof. The reduction is from the well-known (and NP-Complete) Knapsack problem (Cormen et al., 2001), where, given a number of items, each with a value v_i and cost w_i, one must determine if there is a collection of these items with value greater than or equal to V^* that does not exceed a total cost W^*. A task graph that encodes this exact problem (and could be constructed from an arbitrary instance of the knapsack problem) is shown in Figure 5.4. The member links from the instantiated output nodes to the input of the service encode the selection of items while the min, count, and sum relations in the output of the Knapsack service encode the value and cost constraints. The chosen items are stored in a sublist to facilitate the mapping of the member relation (each element in the sublist must be a member of the original list). A dummy node is used as an extra part to create the sublist effect. □

We provide the preceding results only to give examples in the general landscape and show that the inference required to use a learned task graph to perform a task may require super-polynomial computation, depending on the complexity of the semantics. This is actually similar to the results in the earlier chapters, where action schemas could be *learned* efficiently but *planning* with them involved super-polynomial computation (even STRIPS planning is P-Space complete (Bylander, 1994)).

[12] In this case this assumption can actually be relaxed because even with a minimum number of equality edges the maximal graph can be reconstructed in (amortized) constant time using a unification-style algorithm.

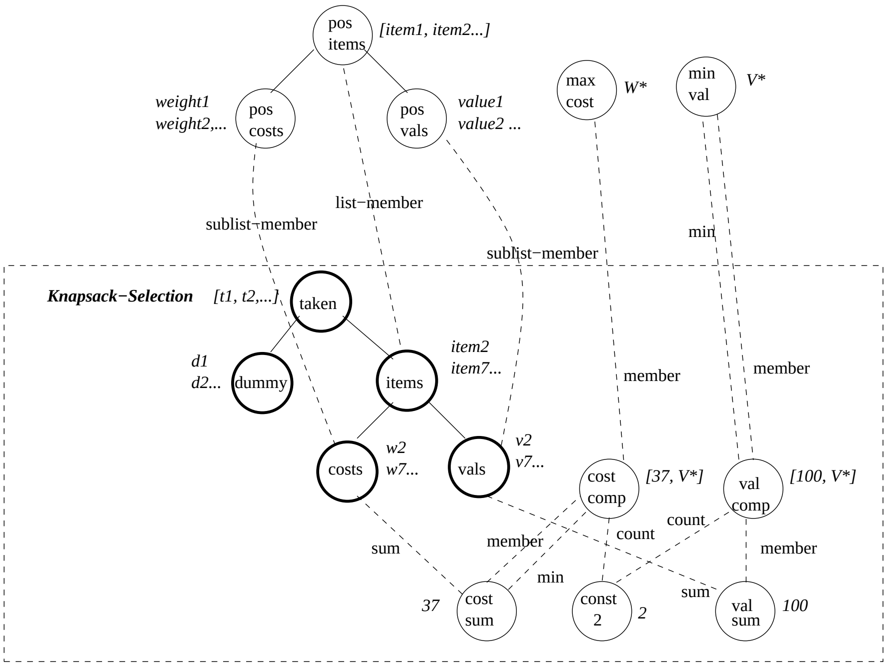


Figure 5.4: A task graph encoding the knapsack problem.

5.6 Examples with Real Services

We now discuss several experiments where TGLA was applied to tasks comprised of publicly available web services. We begin with tasks where the services are all from the same provider. We then discuss a task where services from Google and Amazon are combined and show the system is able to learn its own homogeneous task graph despite the heterogeneous origins of the services. Summary results from the experiments, including the maximum number of traces needed, are reported in Table 5.6. The accompanying graph charts the number of services where mistakes occurred for seven episodes in the two most complex tasks we studied. The results are averaged over 100 random orderings of the collected traces. While the total number of trace requests was generally the same in these runs, the quick descent of the curves indicates many instances of both tasks and many service calls within these instances can be completed correctly before all the nuances are learned. The full collection of traces are available at (http://www.research.rutgers.edu/t̃homaswa/traces.tar.gz).

5.6.1 Examples with Single Providers

Amazon.com offers an extensive web-service library (http://aws.amazon.com/aws), providing access to its inventory, customer wish-lists, and shopping carts. These services have been studied in the Web Service Composition (planning) community, where plans involving multiple Amazon services were dynamically constructed (Marconi et al., 2007). However, this automation relied on hand-crafted descriptions of each service. With an eye towards using our learned descriptions instead, we now present some results in the AWS testbed.

In the "AlbumBuy" experiment, an agent was given an artist and album title (tagged as "In1" and "In2", respectively) as well as a search index ("music") and a quantity to buy. The agent then had to find the corresponding ASIN (Item Id) and use it to create a shopping cart with the required number of copies of that item in it. The first trace provided to our agent involved a self-titled album (Title="Warren Zevon", Album="Warren Zevon"). This ambiguity resulted in the agent linking both In1 and In2 to both the "Title" and "Artist" inputs of ItemSearch. In the next episode (In1="Beatles", In2="Abbey Road"), the agent chose to send these parameters to ItemSearch backwards (for example, "Beatles" to Album), garnering no results. The agent then received a trace and deduced the correct identity links. Other semantic relations mined in this task included (1) When ItemSearch returned multiple items (it matches substrings on titles), the one with the title matching In2 (and Title) should be added to the cart and (2) the quantity passed to CreateCart should be linked to the requested quantity in the initial information—a non-trivial relationship since most of the traces requested only one album copy and many lists returned had a length of 1. [13] The former behavior was learned with the help of *SameSelection* relations, which linked the Title and ASIN of the item put in the cart. Since the Title was also linked back to the original input, the agent could infer which of the returned items to actually buy (by choosing the ASIN grouped with the matching title). Note that these behaviors are exactly right and the task would not be correctly executed without this knowledge. The full G_T for this task is illustrated in Figure 5.5.

We also experimented with tasks where agents learned about services for looking through wish-lists, searching for items (sometimes from earlier wish list searches), creating carts, and adding more items to a cart. Often the agents inferred rules for choosing items, such as buying minimum price items. Since exact structures are rarely copied, these rules were represented with the *Selection* and *SameSelection* relations. The mined graphs usually contain dozens of nodes, and typically fewer than 5 traces are needed before the learner can correctly execute tasks on arbitrary inputs.

[13] In other tasks, such links can be helpful. For instance, we ran another experiment where the quantity was not specified but traces showed it to always be one and the agent correctly learned this information was based on the "Count" of the ASINs passed to CreateCart.

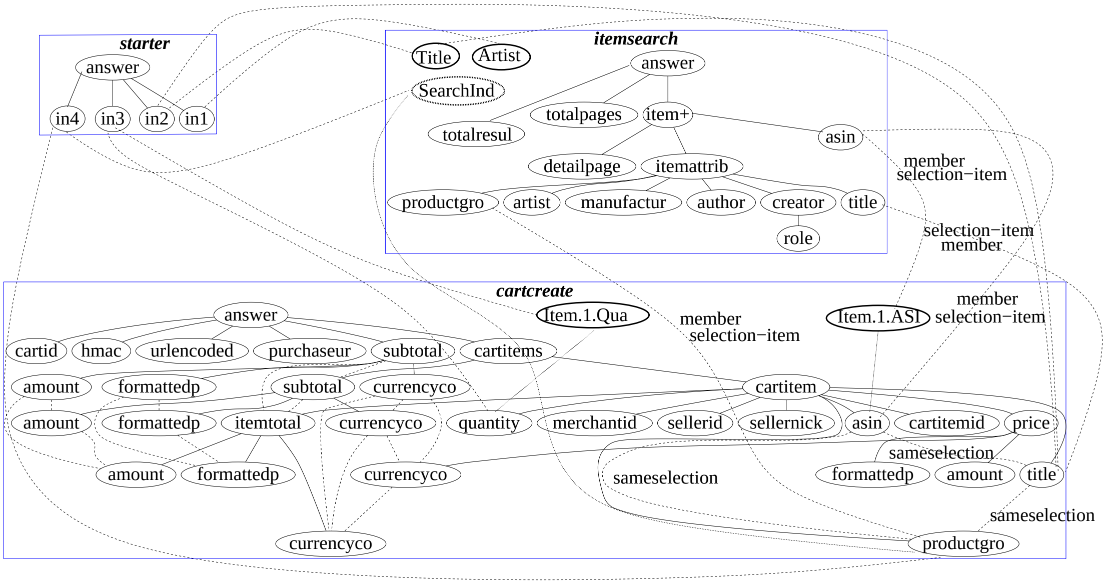


Figure 5.5: G_T for the Amazon AlbumBuy task. Some relations are suppressed for readability.

In addition to the Amazon experiments, we also trained the system on tasks involving services from the Google Data API (http://code.google.com/apis/gdata/). Services in this library allow users programmatic access and editing capabilities to their email, contacts, spreadsheets, calendars, and other content. In this setting, we constructed a task for users filling out a form to receive reimbursements for travel on a per diem basis. Specifically, the traces tracked users looking up a conference, stored either as a (potentially multi-day) appointment or a series of appointments, in their Google Calendar. When multiple appointments were used to encode a multi-day trip, the system identified the min/max dates (selected from a sublist) as the beginning/end of the conference. The "where" field for the appointment was then used to look up the per diem information from a Google Spreadsheet fashioned from a real US government per diem spreadsheet [14], and then a form was updated with the traveler's name, dates, and per diem information. This task involved several intricate relationships that needed to be learned. For instance, some of the per diem rates were listed seasonally (for example, searching for the rate in Las Vegas returns a list of rates and dates, so the system had to pick the correct rate corresponding to the trip dates). With our selection templates and adding *DateBefore* and *DateAfter* relations to $\mathcal{M}$, we were able to capture this behavior. Table 5.6 reports the maximum number of traces needed in our experiments and the number of nodes in the graph. The accompanying graph charts the number of services where prediction mistakes were made per

[14] http://www.gsa.gov/graphics/ogp/FY09DownloadablePerDiemRates.xls

Experiment	Nodes	Services	Max Traces Needed
Flight Booking	25	4	3
Amazon Album Buy	56	3	3
Google Per-Diem	147	4	5
Birthday Gift	231	6	7

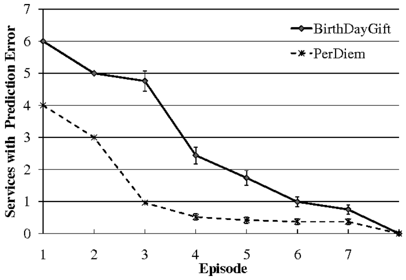


Figure 5.6: *Left*: Sample results for learning task graphs from several examples discussed in this chapter. *Right*: Number of services where mistakes were made (averaged over 100 orderings of task instances).

episode, averaged over 100 orderings of the task instances. Note many of the task instances can be executed without error even before all the nuances of the task are learned, and often only one additional trace is needed after the third episode.

We also trained the system on similar tasks where different information had to be entered in the final form. For instance, we experimented using traces where appointments were only encoded using multiple single-day appointments (rather than the mixed approach above) and the length of the trip had to be entered in the final form, which the system correctly determined to be the count of the number of calendar entries. These examples demonstrate the flexibility of our approach—by learning task-dependent semantics of the services, it can adapt to slightly different uses of the services.

5.6.2 Services with Different Providers

One of the goals of the service-oriented computing movement is to compose tasks with service calls from different providers. Unfortunately, non-uniformity in web-service descriptions has made this goal quite difficult from a semantic perspective. There has been work on learning unified service descriptions from heterogeneous providers in the ontology matching (Liang & Lam, 2008), semantic annotation (Hess & Kushmerick, 2003), and even the machine-learning (Klusch et al., 2009) communities. However, ontology matching requires semantic service descriptions (which are often not available, for instance Google and Amazon do not provide these), and the annotation techniques require an existing domain ontology (uncommon). Furthermore, the annotation techniques often try to mine universal descriptions from meta-data (web forms, WSDL files, etc.), rather than instances. In contrast, we have taken a more traditional machine-learning approach: because we are learning our own semantics based on traces, there is no need to reconcile mismatched or even missing descriptions.

To illustrate this point, we performed an experiment using services from both Google and Amazon. The task involved using the Google Calendar service to find all the birthdays of a user's colleagues within a given date range, then using the Google Contacts service to look up the email address of the person with the earliest birthday. This email address was then sent to the Amazon ListSearch service to find the user's Wish List. Then, the Amazon ItemLookup and CartCreate services were used to purchase the cheapest item on that list. Multiple traces were needed to learn certain finer points of this behavior, such as picking the earliest birthday, buying the cheapest gift, and eliminating erroneous date relations (such as relations based on publication dates). The system's success in inferring the semantic links in the task, even between multiple providers, shows that heterogeneity is not as vexing when learning from data, rather than analyzing sparse and potentially scarce description files. Also, because the Google Contacts service does not yet support full text indexing (instead returning a list of contacts), the selection templates were necessary for learning the correct behavior. Table 5.6 and the accompanying graph illustrate the number of services where mistakes are made on each episode. Notice that this task is more complex than the earlier *Per-diem* task, but only takes a few more traces to learn, with half the services usually learned sufficiently for the experimental task instances after 3 traces. Also, many runs required fewer than the maximum (7) number of traces as more informative traces were encountered earlier. Given the complexity of the task, the small number of traces needed is a strong justification of this apprenticeship learning framework and our learning algorithm.

5.7 Related Work

Several other works in AI, machine learning, and in the web-service community have considered problems related to work in this chapter. A previous application (Carman & Knoblock, 2007) of Inductive Logic Programming (ILP) showed promise in learning web-service descriptions from examples, based on known descriptions of other services. Unlike their approach, which relied on heuristic search and "sufficient" data, we have focussed on algorithms that can guarantee high performance with a limited amount of data. Our earlier work (Walsh & Littman, 2008), with goals more akin to ours, performed a sample complexity analysis of learning planning operator descriptions in a restricted language, but did not explicitly consider relations between operators, and the operators had extremely limited scope. The Task Graph representation itself bears resemblance to the structure from Simultaneous Learning and Filtering (SLAF) (Shahaf, 2007). However, unlike SLAFs our task graphs cannot represent arbitrary boolean formulas, but do have positive sample efficiency results.

A separate track of research focusses on creating web-service descriptions for heterogeneous sources based on a central ontology. This thread includes acquiring descriptions using text-mining algorithms on the service's documentation (Hess & Kushmerick, 2003), and the use of ontology merging techniques applied to full (but not directly compatible) semantic descriptions of services (Liang & Lam, 2008; Huang et al., 2006). However, both require an existing domain ontology and large amounts of service documentation, and even in more adaptive variations that use service instances (Lerman et al., 2006; Klusch et al., 2009), the focus on universal descriptions and semantic annotation differs from our goal of mining task specific relations between concepts directly from relatively (compared to full semantic descriptions) easy to find collections of XML documents used to communicate to and from the services.

Another area that future iterations of this work can draw upon is the field of workflow induction (also known as "process mining") (see van der Aalst & Weijters (2004)). This field is concerned with inducing models of tasks (but usually only the sequence of calls S , not the dataflow) that contain loops, conditions, and concurrency, usually representing the task with a Petri Net (Murata, 1989). Learning such powerful structures is inherently intractable (Petri Nets can represent Context Free Grammars), and even restricting the form of these nets usually leads to super polynomial learning times (van der Aalst et al., 2004). However, with certain restrictions systems from the field of workflow induction (van der Aalst & Weijters, 2004) could be used to learn these more complex versions of S .

More recent work on Workflow Induction from Traces (WIT) (Yaman et al., 2009) has moved from just learning the sequence of calls in a workflow (the goal of the previous 2 works) to also modeling the dataflow. These dual goals are more in line with our own, though in this chapter we have focused almost exclusively on the latter. The WIT algorithm considers a larger class of workflows (so-called "witty workflows") than our work in terms of the service calls (S) and employs powerful grammar induction algorithms to extract structures like loops and concurrency. However, when modeling the dataflow (R), WIT considers only equality (so $\mathcal{M} = \{=\}$), so in that sense they consider a more restricted language. WIT is both a sound and complete algorithm for learning the more complicated structure (as well as the more restricted dataflow). Our work has made harsher restrictions to the shape of the workflow to ensure that only a small number of traces (rather than just a countable amount) are needed to learn the task. However, we have considered a far more expressive language for modeling the dataflow R . Thus, there seems to be a promising future in combining these two algorithms using the grammar induction (and some heuristics) from WIT to learn complex service call patterns, with TGLA used to learn a semantically rich dataflow.

5.8 Linking Back to Action Schemas

We now briefly discuss ways to connect the Task Graph representation to the action schemas used in earlier chapters. The goal of such an effort would be to mine operator descriptions of individual services from the graph by treating the output nodes as literals added to the current *state* and the links to previous services as pre-conditions for executing the service. A number of planners have been developed in the web-service composition community (Hoffmann et al., 2007; Liu et al., 2007; Marconi et al., 2007) [15] that could be used with these mined operator descriptions since they generally cover richer languages than our own.

In the literature, there is no standard language for web-service operators, though many proposals exist (OWL-S[16], WSDL-S[17], etc.). Almost all share the idea of input and output parameters for the service, and pre/post-conditions involving them. These common structures can be derived from a task graph by treating the output nodes as literals added to the current state and the links to previous services as pre-conditions for executing the service. Several methods have been proposed for encoding the creation of new objects in action schemas, including the use of restricting an operator's effect-scope to only a set of new objects (Hoffmann et al., 2007), as well as using exemplars (Liu et al., 2007), or general functions of an action's parameters (Walsh & Littman, 2008) to represent new objects. This last approach seems to be the best fit for task graphs as many of the semantic relations can be represented by less generic functions (such as using the function "min" in Add: CheapestPrice(min(PriceList))). Also, because task graphs encode relations between service outputs and objects from previous services that may not be in the service's parameter list, deictic references (Pasula et al., 2007) may be required to expand the scope of operators.

Since G_T is built from a single task, schemas built using the guidelines above (such as the one in Table 5.1) may be heavily biased to the semantics of a single task, but they could

[15] Interestingly, the last planner listed considered planning with operators derived (by hand) from Amazon Web-Service descriptions, which served as our real world testbed, so this may be a natural vein for future work.

[16] http://www.w3.org/Submission/OWL-S/

[17] http://www.w3.org/Submission/WSDL-S/

FlightLookup(City1, City2, Start, End)
PRE: *Equal*(City1, Home), *Equal*(City2, DestCity)...
ADD: *Result*(X), *NumRes*(Y), *Flight*(Z), *Part*(Z, X), *Part*(Y, X), *ListCount*(Y, Z)...

Table 5.1: A partial action schema from the **FlightLookup** service from Figure 5.2

potentially be refined using other tasks. That is, several G_T's can be merged by dropping edges that do not appear in all the graphs. Links between nodes within a service (such as the "Count" relationship between NumRes and Flight$^+$) can easily be resolved in the unified version, but links to nodes outside of the service (for example, pre-conditions based on the links back, as in the **FlightLookup** operator in Table 5.1) are more complicated as they require either a disjunction or more intimate knowledge of the types of each node. This is where the connection between input/output nodes and ontologies would be helpful, and hence the combination of this work with the earlier-mentioned work on learning of service parameter classification, or semi-automatic schema merging between the XML tags in examples and ontologies, as considered by Patil et al. (2004) seems advisable. But notice that even without these inter-service links, service descriptions with limited semantic scope (similar to the scoping restrictions of STRIPS) can be achieved. Thus, while the representations used in this chapter are certainly different and more powerful than the basic STRIPS action schemas we have considered earlier, a lossy translation is possible. In the next chapter, we continue our investigation of more powerful relational languages, this time more recognizably within the action-schema framework used in Chapters 3 and 4.

Chapter 6
Language Extensions, Planning, and Concluding Remarks

This chapter covers a number of extensions and modifications of the representations and algorithms described in this work.[1] Specifically, we show that another relational language used in the RL community is covered by the results in this paper regarding action schema learning. We futher show how to represent more powerful action schemas composed with Description Logics, moving us slightly closer to the expressiveness of the Situation Calculus. Finally, we will discuss a modification to our core learning algorithm that replaces Value Iteration with a more computationally efficient *sample-based* planner.

6.1 OOMDPs as Action Schemas

We now discuss a language from related work on reinforcement learning with generalized models, Object-Oriented MDPs (OOMDPs) (Diuk et al., 2008). We show that this language is describable using the relational action schemas introduced in Chapter 3 and that the results in most of the learning cases from the online and apprenticeship settings apply to this different setting.

6.1.1 Object Oriented MDPs

Object Oriented MDPs (Diuk et al., 2008) are a special type of relational MDP that are used to describe environments where the dynamics are decomposable based on object attributes and certain relational predicates are defined over these objects and attributes. This formulation has proven successful in modeling large MDPs, including factored domains from the literature and the video game "Pitfall!" (both results can be seen in Diuk et al. (2008)). More formally, the following defines an OOMDP (using notation intentionally similar to our action schema

[1] The description-logic section of this chapter developed from discussions with Alex Borgida. Parts of the approximate-planning section appeared in joint work with Sergiu Goschin and Michael Littman (Walsh et al., 2010a).

MoveRight(Obj,Loc): *Reward* $= -1$
c_1: *ClearToRight*(Loc) $\wedge$ *GoodFooting*(Obj, Loc)
ω_{11} : Obj1.x = min(2 +Obj1.x, 5) (0.8)
ω_{11} : Obj1.x = min(1 +Obj1.x, 5) (0.1)
ω_{12} : Obj1.x = Obj1.x (0.1)
c_2: *ClearToRight*(Loc) $\wedge$ *WetFloor*(Obj, Loc) $\wedge$ *Freezing*(Loc)
ω_{21} : Obj1.x = min(1 +Obj1.x, 5) (0.7)
ω_{22} : Obj1.x = Obj1.x (0.3)
c_3: *WallToRight*(Loc)
ω_3 : Obj1.x = Obj1.x (1.0)

Table 6.1: An OOMDP operator for walking right with a limit of $x = 5$.

definitions):

Definition 35. *An* Object Oriented MDP *(OOMDP)* $\langle \mathcal{O}, F_{att}, P, \mathcal{A}, R, \gamma, S_T \rangle$ *is an MDP where the states are comprised of objects* $o \in \mathcal{O}$*, each with attribute-values* $f \in F_{att}$ *(for example, the location of the object as an integer). In every state, a set of predicates* $P_s \subseteq P$ *over the objects are true, but each of these is* ***defined*** *by the attribute values of the objects in that predicate (such as* On*(W, X)* $\leftarrow$ *W.y* $=$ *X.Y* $+ 1$*). The transition function is decomposed schematically (in* $\mathcal{A}$*) with conjunctive conditions based on the predicates and effects describing changes in the attribute values. The rest of the parameters follow the standard episodic MDP definition.*

An example of a stochastic OOMDP operator appears in Table 6.1. Notice that attribute changes are described with mathematical functions (like adding 1 or 2 to the x coordinate of the object in the example). These general changes to a numerical attribute are not easily captured in STRIPS without introducing logical functions or greatly expanding the number of conditions considered. However, like our earlier STRIPS domains, there can be ambiguity in OOMDPs when determining what effect caused a given transition. For instance, an agent with *GoodFooting* and without a wall to its right (condition c_1) in Table 6.1, could see its x coordinate changing from 4 to 5, but cannot tell which of the first two effects occurred, though it knows the third effect did not happen. Previous work (Diuk et al., 2008) presented an efficient algorithm for KWIK-learning deterministic effects. Below, we show that stochastic OOMDP parameters corresponding to the learning problems described earlier (CD-Learning, etc.) can be efficiently learned under the conditions described for each specific problem.

6.1.2 Connecting OOMDPs to Schema Terminology

Based on the correspondence between our action-schema definitions and the OOMDP language, we can state the following about OOMDPs.

Remark 1. *The theoretical results reported for general (not STRIPS-specific) action schema learning in the autonomous and apprenticeship cases are all applicable to domains encoded as OOMDPs as long as the actions have limited scope (for example, if only objects in an action's parameter list can have their attributes and predicates appear in the conditions and effects). Furthermore, some of the theorems stated for Effect learning (ED-Learning) for Stochastic STRIPS have analogues in the OOMDP case.*

Proof. First we show that any OOMDP fits in the action schema formalism. All OOMDP operators are by definition, describable by a set of objects $\mathcal{O}$ with attributes $o.a_i$ for each object, and a set of defined predicates P as above. The parameterized actions each have a set of conditions C^a (conjunctions over the predicates), and each of those induces a distribution over effects (Ω^a_i, Π^a_i) where each effect $\omega_j \in \Omega^a_i$ is a mathematical operation performed on the attribute values of some objects. Extensions of the learning algorithms for dealing with these changes to these numeric attributes are discussed below.

The translation from here to the stochastic conditional action-schema formalism is fairly straightforward. The set of objects maps directly for the domain instance, and the set of literals is made up of all the possible attribute/value pairings as well as all the predicates. C^a, Ω^a, and Π^a all map directly from there. Thus any OOMDP environment can be written as an action schema as defined in Chapter 3.

As for the theoretical results in Chapter 3 and 4, most of the theorems are stated as to hold for any action schema with enumerable conjunctive conditions (which OOMDPs in the autonomous case can be assumed to have) or just conjunctive conditions in the apprenticeship case, which again holds for OOMDPs. Likewise, the effect distribution learning for OOMDPs (D-Learning) can be done using KWIK-LR since the decomposed transition function is virtually identical (with only attribute mappings and defined predicates replacing the Add and Delete lists).

The only algorithms that need to change are those that learn the effects (ED-Learning and CED-Learning). In Stochastic STRIPS, these algorithms all used either the DetEffectLearn algorithm (Algorithm 19) or KWIK-CorEffect (Algorithm 23 to learn Add and Delete lists. OOMDPs do not have Add and Delete lists, instead they have mathematical functions for each attribute (such as $x+ = 2$ or $y = \sqrt(7) * z$). Learning in the space of all possible possible functions is intractable, but in the case where the number of possible effects in the hypothesis space is polynomial in the domain parameters (for instance if we know that all actions add 1, 2, or 0 to the x or y attributes), we can (case 1) enumerate the possible effects or (case 2) the possible effects for each literal, and use the meteorologist architecture or correlated effect learning, respectively. In the former case, this is done by pairing every possible condition and effect (or effect set) together, each as one of the "meteorologists". In the latter case, instead of intermediate nodes like X^+, we will have intermediate nodes for each possible effect on that literal (like $X+2$), and intermediate effects that do not happen will just end up with probability 0. □

6.2 Learning Description Logic Operators

We now consider a class of action schemas based on Description Logics (DLs) (Baader et al., 2003). These languages use *concepts* to describe conditions or pre-conditions, and their expressive power lies between the basic relational representations we have used in this thesis (STRIPS rules) and full first-order logic descriptions (like the Situation Calculus). Planning with operators in this family has been the focus of a PhD dissertation (Milicic, 2008), but thus far no work has been done on the *learning* of these operators from data. We study this expansion in schema description expressiveness for a number of reasons. First, by incorporating constructors like universal role restrictions[2] (∀ *hasPet.Dog*) and cardinality constraints (*atLeast* 3 *hasPet*), we can compactly represent concepts beyond those considered in our earlier studies (such as a pre-condition that says a person must have at least 3 pets, and all her pets need to be dogs). Using DLs also provides a natural way to incorporate background knowledge (such as primitive concept hierarchies) into action schema learning. Finally, using DL representations and operations, which are more "object centered", allows us to naturally expand the scope of the conditions of our operators beyond an action's parameter list.

Because we are concerned with learning complex pre-conditions, and we have seen the difficulty of learning even moderately sized conjunctions in the online case, we will investigate DL action schema learning in the apprenticeship paradigm. Thus, we will ultimately be concerned with the mistake bound learnability of these schemas. We will also consider the computational cost of the learning process, as considerably more computation will go into maintaining valid hypotheses than in the deterministic STRIPS case, where we just added or removed a few literals from a set of lists.

[2] Throughout this section we will use standard Description Logic notation with individuals in all capital letters (instead of boldface), and roles starting with lowercase letters and primitive concepts with uppercase. So were previously we might have said *On*(X,Y), *Block*(Y) we now have X: ∀ *on.Block*

Language	Constructors	Example
$\mathcal{AL}$	$\sqcap, \top, \forall$	$\mathit{Person} \sqcap (\forall\ \mathit{hasPet.Dog})$
$\mathcal{ALN}$	$\sqcap, \top, \forall, \geq, \leq$	$\mathit{Person} \sqcap (\forall\ \mathit{hasPet.Dog}) \sqcap (\geq 2\ \mathit{hasPet})$
$\mathcal{EL}$	$\sqcap, \top, \exists$	$\mathit{Person} \sqcap (\exists\ \mathit{hasPet.Dog})$
$\mathcal{ALC}$	$\sqcap, \top, \forall, \neg$	$\mathit{Person} \sqcap \neg\ (\forall\ \mathit{hasPet.Dog})$

Table 6.2: Some popular DL constructors

6.2.1 Description Logic Terminology

We begin by describing some basic DL terminology. Description logic *concepts* are built using a set of *constructors*. Table 6.2 lists a few common languages/constructors with example concepts. Different combinations of these constructors form different languages, which are referred to using acronyms formed by the symbols for each constructor (see Table 6.2). For instance, the language $\mathcal{ALN}$ supports conjunction, universal quantification, and cardinality restrictions (($\forall$ *hasPet.Dog*) $\sqcap$ (*atLeast* 2 *hasPet*)).

Description logic systems separate axiomatic background knowledge (*Dachshund* is subsumed by ($\sqsubseteq$) *Dog*) from facts that are actually true in the current state of the world (like FIDO:*Dachshund*). Background information is collected in a *T-Box* (terminology), which can be thought of as a set of axioms that we can check for consistency or merge with other T-Boxes. Facts about the current state of the domain, however, are collected in an A-Box, (assertions), which is normally interpreted using the open world assumption to reflect a set of possible worlds. For instance, if an Abox states $\mathit{BOB} : \forall \mathit{hasPet.Dog}$, BOB may have no dogs at all, or many different dogs. While it is tempting for us to interpret states as particular Aboxes, doing so will make learning in the apprenticeship (or just about any other) paradigm very difficult because an empty Abox can always be used to describe a particular state (since open world semantics allow an empty Abox to cover any possible state). Without restrictions on these Aboxes, valid traces could consist of essentially nothing but action invocations, leading to a POMDP learning problem outside of the scope of this work (but considered by others (Yang et al., 2007; Zhuo et al., 2009)).

To avoid this partial observability, we will instead make use of a more restricted description of states as a *D-Box* (database) $\mathcal{D}$ (Franconi et al., 2009). DBox facts are presented in terms of ground atoms such as *hasPet(BOB,FIDO)* and have a closed-world semantics—like STRIPS states. Therefore cardinality constraints and universal/existential restrictions that apply to any individual can be computed directly from the atomic facts in the DBox. By virtue of this, DBoxes have *closed world* semantics—like STRIPS states, any fact that is not stated or derivable from the DBox can be assumed to be false ($\neg(\mathcal{D} \vdash X) \rightarrow \neg X$).

6.2.2 Some Basic DL Operations

The fundamental operations/judgements in DLs are subsumption ($C \sqsubseteq D$), which compares two concepts in terms of generality, and concept membership ($b \in C$). The latter is determined with respect to an ABox/DBox which provides information about the individual b. (See Baader et al. (2003) for details.)

We now describe two additional DL operators that will be instrumental for pre-condition learning with DLs. The first operation we will need is *Most Specific Concept* (MSC), which maps an object in a DBox to a concept in the given DL. For instance, given a DBox containing objects BOB and FIDO, the facts *Man*(BOB),*Dog*(FIDO), *hasPet*(BOB, FIDO), BOB: ∀ *hasPet.Dog*, etc.), using the language $\mathcal{AL}$, MSC would produce the following concept for BOB: Man ⊓ ∀ *hasPet.Dog*. By definition, MSC(b,D1) for D-Box D1 is guaranteed to be subsumed by any concept C such that $b \in C$ holds in D1. By convention, MSC(b,D1) is abbreviated to MSC(b) whenever the DBox D1 is understood from the context. In some languages, a finite MSC may not actually exist. For instance, if the relation *obeys*(FIDO,BOB) was also true, an infinitely long chain $\forall textithasPet.$∀*obeys*.∀*hasPet*... would be needed to represent the most *specific* concept covering Bob. However, often a variant of MSC that only looks at relational links of length no more than k (MSC_k) is often computable (Baader & Küsters, 2006). $MSC_k(b, D1)$ is guaranteed, by definition, to be subsumed by any concept C of quantifier depth at most k such that $b \in C$ in D1. We will make use of this operation in subsequent sections to create concepts describing relevant parts of each ground state we see during learning.

While MSC will allow us to derive concepts from DBox "states", we will also need an operation to produce a generalized concept that covers two existing concepts. This is necessary, for instance, for realizing (in the presence of a background ontology) that the concept *Dog* might be the pre-condition for an action that executed correctly with a parameter that in one call was a *Dachshund*, and in another call was a *Terrier*. The DL operator for computing such generalized concepts is *Least Common Subsumer* (LCS). Specifically, $LCS(C, D) = L$ if for every concept F such that both $C \sqsubseteq F$ and $D \sqsubseteq F$ we are guaranteed $L \sqsubseteq F$. That is, LCS computes a concept that subsumes the two input concepts, but does not subsume any other concept that subsumes both—the most specific generalization. Again, this operation has been

buyCar(X1, X2): reward = +1
PRE: X1: $\mathit{Adult} \sqcap \mathit{atLeast}(2,\ \mathit{hasCreditCard}) \sqcap \forall\ \mathit{hasCreditCard}.\mathit{ValidCard}$
X2: *Car*
ADD: *OwnsCar*(X1, X2)
DEL:

Table 6.3: A very simple DL Action Schema in $\mathcal{ALN}$ for claiming an award for being a good parent and dog owner. A more complex example (where the pre-conditions require relations between the parameters that have to hold) appears in Table 6.2.3.

widely studied in many DL languages (Baader et al., 2003).

6.2.3 Learning Pre-conditions with Description Logics

We will now show how to use the operators above to construct an algorithm for learning pre-conditions in DL Action Schemas (DL Schemas hereafter) An example DL Schema in the language $\mathcal{ALN}$ appears in Table 6.3. While we deal with a richer example (involving relations specifically between the parameters) later, the schema in Table 6.3 has examples of each of the constructors in $\mathcal{ALN}$ in the pre-conditions. DL Schemas fit into the family of relational action schemas defined earlier. That is, whereas STRIPS schemas had conjunctions over simple relations for pre-conditions, now they are expressed ABox-style assertions about the action parameter individuals, using a DL language to describe concepts and roles relating individuals. However, the effects However, the effects are still just Add and Delete lists of primitive literals (essentially STRIPS effects), because reasoning with the open-world style effects that general DL-constructors introduce requires very delicate semantics to avoid undecidability when planning with these operators (Milicic, 2008)[3]. DL Schemas have been studied in prior work (Milicic, 2008) where it has been shown that increasing the complexity of the pre-condition language from STRIPS (P-Space Complete planning) towards First Order Logic (undecidable), has a very dramatic effect on the tractability. We will now see that the change in languages affects the learning process as well.

Figure 6.1 illustrates an algorithmic template (*DL-Pre*) for learning DL schema pre-conditions in the apprenticeship paradigm. The algorithm utilizes the MSC and LCS operations described above, so we assume these operators exist for the language used in the schema, or at least that LCS$_k$ and MSC$_k$ do. The algorithm takes each successive *DBox* instance s_t from a trace $\mathcal{T}$ and determines if its current hypothesis describes this instance. (It does so by replacing

[3] For example, what does it mean to Add (≥ 2 *hasPet*)? This statement may have an infinite number of possible groundings.

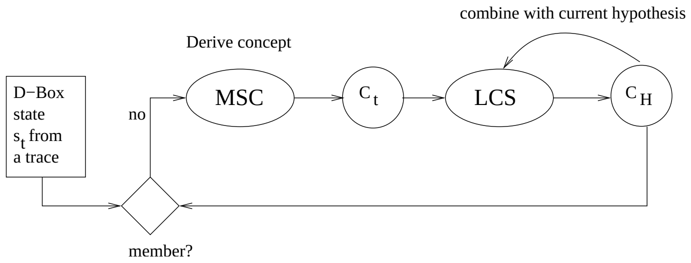


Figure 6.1: DL-Pre: A general DL learning template for learning pre-conditions from examples of action executions. This can be used as the subroutine for pre-conditional CD-Learning in the apprenticeship setting just as *MB-Con* was for traditional STRIPS.

the schema parameters with the actual parameters, and checking whether the pre-condition assertions hold in the Dbox s_t). If it does, the current hypothesis remains the same. If not, we characterize the actual parameter instances by computing their MSC and then find the LCS of this concept with the current hypothesis to compute a condition that covers all previously seen cases. Notice that while the input states are (grounded) DBox instances, the pre-condition hypotheses are modeled *conceptually*, that is they are ABoxes.

Put simply, the algorithm always uses the most specific hypothesis that covers the states where previous successful action invocations have taken place. This is essentially a DL-centered application of the specific-to-general learning or least general generalization that is popular in the ILP community where it has yielded several important sample complexity bounds (Cohen, 1995a; Horváth & Turán, 2001). Notice that background ontologies (T-Boxes) can be incorporated into this algorithm as part of the LCS and MCS computation and can also be used to help compute the MSC itself (recursive T-boxes add some leverage for certain languages). This allows us to, for instance, learn that if an action *walk*(H, D) has its pre-conditions satisfied when (H=BOB, D=FIDO) with *Man*(BOB) and *Dachsund*(FIDO), and also works for (NANCY, REX) with *Woman*(NANCY) and *Terrier*(REX), then the pre-conditions might be *Person*(H) and *Dog*(D). Of course, the complexity of the learning algorithm grows both in terms of samples needed and computation required due to these more expressive constructs.

DL-Pre can be used in place of the *MB-Con* subroutine for CD-Learning in Algorithm 27 in the apprenticeship case. Notice that there are many similarities between *MB-Con* as used to learn STRIPS pre-conditions and *DL-Pre* used to learn DL-Schema pre-conditions. For instance, both move from specific to general concepts, and both work exclusively from positive examples. When viewed in this manner, one can see the deletion of literals from the pre-condition list done by *MB-Con* as a simple version of LCS in a language that only allows primitive relations and conjunctions.

The use of MSC (or MSC$_k$) does not have a specific analogue in *MB-Con* because the relational descriptions are provided simply as a list in STRIPS, rather than as the more individual-centered DL concepts. However, special care needs to be taken to preserve relationships between action parameters if the given DL does not support nominals or the *same-as* constructor. For instance, in $\mathcal{AL}$, the STRIPS-style pre-condition relationship *On*(X, FROM) of *pickup*, from X's perspective would just be X:∀ *on.Block*, with no indication that it is specifically on the block referenced in the other parameter, $FROM$, of *pickup*.

We resolve this by relabeling in each trace instance all the parameter individuals with uniform special names, and introducing corresponding special primitive concept names *#X1...#Xm* to represent the property of "being parameter *i*". This representation can be seen in Table 6.2.3 where the constraint X1: ∀ *hasPet.#X2* has the intended meaning that the single pet of X1 is the same individual as the X2 parameter. This use of primitive concepts to represent nominals (which the variables in our case are analogous to) resembles the non-standard semantics nominals in practical DL systems like CLASSIC (Borgida et al., 1989) and while it does not provide the powerful reasoning associated with nominals or *same-as*, it also does not require their considerable computational (and in our case sample complexity) overhead. That is, any MSC operation for a language that can handle primitive concepts can use this approach to encode restrictions between parameters, such as the relationship between X2 (a dog) and X1 (its owner) in Table 6.2.3. Unfortunately, we note that in some DLs lacking existential restrictions (like $\mathcal{AL}$ or $\mathcal{ALN}$), this will only capture functional relations between parameters. For instance, if a block X1 would need to be on two different blocks, the $\forall On$(X2) concept would be dropped from X1's description.

General Theoretical Properties

The sample complexity of using *DL-Pre* in the CD-Learning setting, will be highly language dependent, as well as depending on the nature of the background information (anything from simple concept hierarchies to cyclic T-boxes). However, in languages without such background theories, an upper bound on the *mistake bound* of *DL-Pre* can be determined from the length of the longest path in the subsumption lattice, a connection that has been noted in rule learning as well (Khardon & Arias, 2006).

The other measure we are concerned with is computational complexity. This has two sources

$\mathit{claimAward}$(X1, X2): reward = +1
PRE: X1: $\mathit{Adult} \sqcap \#\mathit{X1} \sqcap \mathit{atLeast}(1,\ \mathit{hasChild}) \sqcap \forall\ \mathit{hasChildren}.\mathrm{NotAllergic} \sqcap \mathit{atMost}(1,\ \mathit{hasPet})$
$\sqcap\ \forall\ \mathit{hasPet}.\#\mathit{X2} \sqcap \ldots$
X2: $\mathit{Dog} \sqcap \forall\ \mathit{hasCollar}.\mathit{Fancy}\ \ldots$
ADD: $\mathit{AwardWinner}$(X1)
DEL:

Table 6.4: A partial (some pre-conditions omitted for readability) DL Action Schema in $\mathcal{ALN}$ for claiming an award for being a good parent and dog owner. A full tree representation of the pre-condit

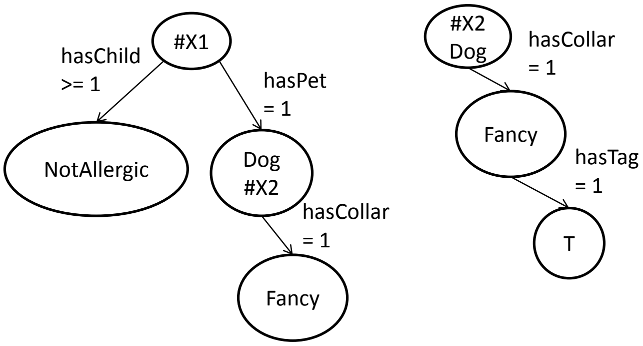


Figure 6.2: A concept tree in $\mathcal{ALN}$ for the pre-conditions of the actions in Table 6.2.3.

in the overall apprenticeship architecture: planning and updating the learner. As with STRIPS, the planning component for DL Schemas is worst case intractable (in fact even more so than STRIPS (Milicic, 2008)), and so we do not deal with that here. However, the computation required by the learner, which with pre-conditional STRIPS simply involved removing literals from a list, is now governed by the more costly LCS and MSC operations. We note that analyzing this computation for only a single iteration with arbitrary concepts and having a polynomial sample complexity bound is not sufficient to bound the computational complexity of the full algorithm, because the amount of time required to perform the LCS may actually grow with each successive iteration, as we see in our second case study ($\mathcal{EL}$) below.

6.2.4 Case Studies

While above we have laid out a generic algorithm for learning conditions described using description logics, the complexity of this learning operation is highly dependent on the specific DL that is used. Here we briefly present some results on the complexity of the above algorithm in two specific DLs that are widely studied in the literature.

$\mathcal{ALN}$

$\mathcal{ALN}$ allows for constructors representing conjunction (Dog $\sqcap$ Furry), universal restriction on roles (X: $\forall$ *hasPet.Dog*), and cardinality restrictions on the number of role fillers (*atLeast* 2. *hasPet*), (*atMost* 5. *hasChild*). Unfortunately, like many languages, a full MSC operation in $\mathcal{ALN}$ (or even just $\mathcal{AL}$) is not supported for an arbitrary DBox. To see why, consider the very simple DBox $\{r$(o), p(o, o)$\}$. The most specific concept covering o is ($\forall p.\forall p....r(o)$), where the chain of p repeats infinitely many times.

Algorithm 30 MSC$^{\mathcal{ALN}}$(k, o, $\mathcal{D}$)

```
1: Inputs: depth $k$ and object $o$
2: Output: The most specific concept (in $\mathcal{ALN}$ ) considering objects reachable in DBox $\mathcal{D}$ by
   traversing no more than $k$ roles. The concept is in normal-form (Cohen & Hirsh, 1994).
3: $P^o$ := roles that hold from $a$ in $\mathcal{D}$.
4: if $k = 0$ then
5:     Return ($\sqcap(\{A|A(o) \in \mathcal{D}\} \sqcap\{(\geq n.p_i|p_i \in P^o$, $n$ fillers for role $p_i$ on $o)$ $\} \sqcap\{(\leq n.p_i|p_i \in$
       $P^o$, $n$ fillers for role $p_i$ on $o)$ $\}$
6: end if
7: $C_k := MSC(0, o)$
8: for Each role $p \in P^o$ do
9:     $\{o_1...o_m\}$ := objects reachable from $o$ by role $p$
10:    $C_x = LCS(MSC(k-1, o_1), ...MSC(k-1, o_m))$
11:    $C_k = C_k \sqcap (\forall p.C_x)$
12: end for
13: Perform normalization on $C_k$ if necessary.
14: Return $C_k$
```

Algorithm 31 LCS$^{\mathcal{ALN}}$(c_1,c_2) (Adapted from the work of Cohen & Hirsh (1994))

```
Inputs: Concepts $c_1$ and $c_2$ in $\mathcal{ALN}$ , in normal form.
Output: The most specific concept that subsumes $c_1$ and $c_2$
Construct concept trees $T_{C1}$ and $T_{C2}$ by doing the following:
    Create a root node labeled by the set of all primitive concepts in $C$ //(e.g. $Dog \sqcap Brown...$)
    Create a branch $(root, p, n_p)$ for each top-level role $p$ appearing in $C$ with a restriction.
    Label this edge with $\forall p$, as well $\leq j$ and $gek$ for the corresponding cardinality restrictions.
    Recursively create tree rooted at $n_p$ from the concept $E$ such that there was restriction
    $\forall p.E$ in $C$
$T_C$ = $TreeMerge(T_{C1}.root$, $T_{C2}.root)$
Return concept $C$ corresponding to $T_C$
```

One way around this conundrum is to use a finite approximate MSC_k discussed for some languages in earlier work (Baader & Küsters, 2006), and defined for $\mathcal{ALN}$ in Algorithm 30. This algorithm traverses part of the D-Box starting at object o to a maximum depth of k, collecting all the concepts covering objects encountered at that depth. At each level, an LCS is performed (Line 10 and discussed more below) on all the concepts from one level below in order to reconcile all of the nested constraints (for example, *hasPet.*($\geq$ 2 *hasHouse*) and *hasPet.*($\geq$ 1

Algorithm 32 TreeMerge(n_1,n_2)

```
if $n_1$ or $n_2$ is empty then
    Return the non-empty node (if both are empty, return either).
end if
$n =$ empty concept node
$C_p = n_1.primitives \cap n_2.primitives$, or $\top$ if $n_1.primitive \cap n_2.primitive = \emptyset$
Add each $c_p \in C_p$ to $n$
for Each role $p$ from $n_1$ that is also in $n_2$ do
    For each $e_1 = (n_1, p, n_1')$ and $e_2 = (n_2, p, n_2')$ that are edges, let $n' = TreeMerge(n_1',n_2')$
    Create branch $(n, p, n')$ and label it $\forall.p$
    Add as labels $\geq j$, if $j = min(j_1, j_2)$ where $\geq j_1$ and $\geq j_2$ were the corresponding labels
      on $e_1$ and $e_2$ respectively. Similarly for upper bounds, but using $max$ instead of $min$
end for
Return $n$
```

hasHouse) reconcile to *hasPet*.(≥ 1 *hasHouse*)). The end result of this is a concept tree like the one in Figure 6.2. Notice that primitive concepts label nodes and role names and cardinality restrictions label edges and that the special primitive concepts for variables in the parameter list are used as described earlier. The use of a k-approximation of MSC also has the added benefit of constraining the size of the largest possible concept to be $O(PR^k)$ for a language with just P primitive concepts and R roles ($\mathcal{AL}$), because the largest concept is just a conjunction of (up to R) role-chains, each of length k.

The other central operation for *DL-Pre* is LCS, which has already been described for $\mathcal{ALN}$ in Algorithm 31 (and used in its MSC). The heart of the algorithm is the *MergeTree* operation described in Algorithm 32, which recursively goes through the two trees (our current hypothesis and the concept corresponding to the current instance), finding the least common subsumer of primitive concepts at each node (by intersection). For the roles, tree branches that match up from each node are recursively parsed, and if any role does not appear in both concepts from a given node, it is eliminated. Labels on the edges are refined simply based by updating the cardinality restrictions to more specific numbers (higher for *atMost*, lower for *atLeast*). Notice also that if there is a background ontology (such as primitive or role hierarchies), they could easily be incorporated into this procedure. This can be done efficiently in $\mathcal{ALN}$ because from each node, only a single branch is needed for each role (because it has to be a $\forall$ restriction, which satisfy the equality $(\forall p.C \sqcap \forall p.D) = \forall p.(C \sqcap D)$. We will see that the existential restrictions, which do not collapse as above, lead to difficulties in computing multiple LCS's later in our discussion of $\mathcal{EL}$.

On every step of *DL-Pre*, an MSC_k and an LCS operation are performed. The LCS$^{\mathcal{ALN}}$ operation will take only $O(PR^k)$ time (since it needs to traverse a tree of that size and do simple merging at each node) and $MSC_k^{\mathcal{ALN}}$ operation will take $O(PR^{2k})$ time (since it may have to build a tree of size R^k and perform LCS operations at each node). In terms of sample complexity, any mistaken predictions made while learning $\mathcal{ALN}$ pre-conditions will come when the LCS operator actually refines the hypothesis concept. How many such revisions are possible? Focusing just on the universal quantifier restrictions and primitive concept labels on each node, we see that every positive example will make one of the role restrictions more specific or eliminate one of the primitive concept labels. In the absence of the background ontology (thereby precluding generalizing *Dachshund* and *Terrier* to *Dog*), each sample will eliminate at least one role (edge) or primitive concept (node label) from the hypothesis concept. The number of such items is bounded by $O(PR^k)$. Turning our attention to the cardinality restrictions, it is easy to create a situation where the cardinality of the *atLeast* restrictions increases infinitely (on each step ($atMost\ i\ hasPet$) generalizes to ($atMost\ i+1\ hasPet$)). However, we note that the maximum size of any DBox in a trace ($|\mathcal{D}|$) bounds the maximum cardinality restriction, so this is a loose bound, though by specifying the maximum outdegree of any individual in any D-Box in a trace as N (thereby capping the number of *atLeast* or *atMost* refinements), we can tighten this simply to N.

In summary, learning pre-conditions in the language $\mathcal{ALN}$ from D-Boxes $\mathcal{D}_1, \mathcal{D}_2...$ can be accomplished with no more than $O(PR^k \max_i |\mathcal{D}_\rangle|$ mistakes or with a constraint, N, on the number of outgoing roles of the same name from any individual, this bound becomes $O(NPR^k|$. Only polynomial computation time per step, as long as we are willing to restrict the depth of concepts considered by some constant k (for the MSC_k operation). Finally, we note that this result is likely modifiable for extensions of $\mathcal{ALN}$ where PAC learnability results have been derived (Cohen & Hirsh, 1994) and may be extendible to other languages whose concept learnability has been studied in the ILP community (Lehmann & Hitzler, 2010).

$\mathcal{EL}$

While this result for $\mathcal{ALN}$ is encouraging, we will now see that the concept generalization step (LCS) is problematic in another simple language, $\mathcal{EL}$. $\mathcal{EL}$ allows constructors only for existential role restrictions ($\exists\ hasPet.Dog$) and conjunctions ($\exists\ hasPet.Dog \sqcap \exists\ hasPet.Cat$). As this example shows, unlike universal restrictions, a concept can have multiple existential restrictions for the same role p. The computation of MSC in $\mathcal{EL}$ faces the same problems as in $\mathcal{AL}$, and therefore it is usually approximated using MSC$_k$. Unfortunately, where LCS simply performed tree pruning and simple cardinality adjustments on $\mathcal{ALN}$ concepts, the LCS operator in $\mathcal{EL}$ requires one to consider all combinations of possible existential restrictions on role p. This was not the case with universal quantification, which essentially stated "all objects reachable by this role must have the following properties", so only one (generalized) branch needed to be maintained in the tree. As noted by Baader & Küsters (2006), while the computation of a single LCS operation is polynomial, there exist a sequence of n concepts $C_1, C_2, ..., C_n$ involving just 2 primitive roles, where the size of successive $\mathcal{EL}$ LCS concept (and hence the computation needed to perform each operation) grows exponentially in n. While their result showed that essentially the depth of the concept tree could expand exponentially, we can construct a similar example for the constant-depth trees produced by MSC_k whose width increases exponentially in the number of roles, even with a depth bound of two. This means that the computation required on each step of the learning process outlined above could grow exponentially over the entire learning process. Furthermore, this complicates the sample complexity analysis because generalization does not lead to a monotonically decreasing structure (the concept tree essentially maintains the same depth but may grow *wider* on each learning step). For this reason, previous theoretical analyses of learning $\mathcal{EL}$ concepts in the ILP community (Lehmann & Haase, 2009) have focused simply on convergence and completeness results, and we leave a complete study of this problem to future work.

6.3 Planning in Large State Spaces

Both the autonomous and apprenticeship learning agents in this work rely on planning algorithms to turn their learned models into actionable policies. Thus far, we have considered only the sample, and in some cases, the computational complexity of the model-learning components, but have said little about the planning problem. Unfortunately, the basic flat-MDP planning algorithms introduced in Section 2.1.3, such as Value Iteration or Linear Programming, all have computational dependence on the number of states ($|S|$), which is exponential in the domain parameters of most of the environments considered in this work.

Below, we consider some alternative planning algorithms from the literature, and one of our own design, which scale exponentially in the size of the horizon time (dependent on γ), but whose computation time does not increase with the size of the state space, only with the size of the compact parameters ($|M|$). Specifically, we consider a class of tree-based planners that approximate the value function by drawing samples of the transition and reward functions from the (partially) learned model. We show how to integrate such planners into the model-based reinforcement learning paradigm, specifically with KWIK-Rmax, and show that some of these planners are able to still guarantee PAC-MDP behavior. Finally, we describe a number of planners developed for specific relational domains and comment on their adequacy for integration with our learning framework.

6.3.1 Sample-Based Planners

Sample-based planners are a class of planning algorithms that approximate the value function for (unlike Value Iteration) only the current state by sampling transitions and rewards from a model along simulated trajectories. These planners usually only make guarantees about returning a (possibly stochastic) ϵ-optimal action selection for a given model, and are different from the original conception of planners for KWIK-Rmax in two ways. First, they require only a generative model of the environment, not access to the model's parameters. Nonetheless, they fit nicely with KWIK learners, which can be directly queried with state/action pairs to make generative predictions. Second, sample-based planners compute actions stochastically, so their policies may assign non-zero probability to sub-optimal action, though the policies they induce over time remain ϵ-optimal. The criteria we will use to test for this compliance is the following definition, which ensures that the planning algorithm is both accurate and computationally tractable in our exponentially-sized state spaces.

Definition 36. *An* **efficient state-independent planner** *is one that, given (possibly generative) access to an MDP model, returns an action a, such that the planning problem above is solved ϵ-optimally, and the algorithm's per-step runtime is independent of $|S|$ in that it is only polynomially dependent on $|M|$, and scales no worse than exponentially in the other relevant quantities ($\{\frac{1}{\epsilon}, \frac{1}{\delta}, |A|, (1-\gamma)\}$).*

To see how an algorithm satisfying this criteria might work, first note that there is a horizon length H, a function of γ, ϵ and $R_{\max}$, such that taking the action that is near-optimal over the next H steps is still an ϵ-optimal action when considering the infinite sum of rewards (Kearns et al., 2002). Next, note that the d-horizon value of taking action a from state s can be written $Q^d(s,a) = R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') \max_{a'} Q^{d-1}(s',a')$, where $Q^1(s,a) = R(s,a)$. This computation can be visualized as taking place on a depth H tree with branching factor $|A||S|$. Instead of computing a policy or a value function over the entire state space, sample-based planners estimate the value $Q^H(s,a)$ for all actions a whenever they need to take an action from state s and then choose the action with the largest estimate. Unfortunately, this insight alone does not provide sufficient help because the $|A||S|$ branching factor still depends linearly on $|S|$. But, the randomized Sparse Sampling algorithm (described below) eliminates this factor.

Sparse Sampling

Sparse Sampling or SS (Kearns et al., 2002) improves on the complexity of Value Iteration (in terms of $|S|$) by calculating $Q^H(s,a)$ based on only a *sample* of next states. The main result of the analysis of SS(Kearns et al., 2002) is that the size of this sample can depend on $R_{\max}, \gamma$, and ϵ instead of $|S|$. Specifically, the SS approximation of state-action value function can be written as:

$$Q^d_{SS}(s,a) = R(s,a) + \gamma \sum_{s' \in (s,a)^C} T(s,a,s') \max_{a'} Q^{d-1}_{SS}(s',a').$$

where C is the necessary size (based on the depth in the tree) and each s' is drawn according to the distribution $s' \sim T(s,a,\cdot)$. SS traverses the tree structure of state/horizon pairs in a bottom-up fashion. The estimate for a state at horizon t cannot be created until $t+1$ has been calculated for all reachable states. It does not use intermediate results to focus computation on more important or relevant parts of the tree. As a result, its running time, both best and worst case, is $\Theta((|A|C)^H)$. However, because of its ϵ-accuracy guarantee with high probability (Kearns et al., 2002), we can make the following statement regarding its use with KWIK R-max:

Proposition 11. *SS can be used as a planner in the KWIK-Rmax algorithm by (1) using the KWIK predictors themselves as the generative model for SS and (2) replacing all $\perp$ predictions with $R_{\max}$ transitions. Under this usage, all the results in this paper that used an exact planner hold for PAC-MDP agents in relational worlds.*

Proof. (sketch) The crux of the proof is showing the KWIK-Rmax algorithm remains PAC-MDP with a general planner that satisfies Definition 36. There are a number of technical issues in this proof because a general planner, and certainly SS, may induce a stochastic policy because every time SS is called it has a small (δ) probability of returning a suboptimal action. Several lemmas in the original KWIK-Rmax need to be adjusted in light of this fact, notably those dealing with the "optimism" and "accuracy" of the modified algorithm. Details of this modification are covered in Walsh et al. (2010a), but are of a technical nature and not particularly central to this thesis. □

Sparse Sampling has been used as a planner in at least two other systems that learned relational action descriptions, the work on NID rules (Pasula et al., 2007) and the MARLIE system for learning decision trees to represent relational MDP models (Croonenborghs et al., 2007b), but neither of these works investigated the integration of this planner with a provably sample-efficient learning algorithm, as we did above. Unfortunately in practice, SS is typically quite slow because its search of the tree is not focused, a problem addressed by its successor, UCT.

Upper Confidence for Tree Search

Conceptually, UCT (Kocsis & Szepesvári, 2006) takes a top-down approach (from root to leaf), guided by a non-stationary *search policy.* That is, planning proceeds in a series of *trials.* In each trial, a search policy is selected and followed from root to leaf and the state-action-reward sequence is used to update the Q estimates in the tree. Value updates are performed via simple averaging, so if the optimal policy were used in choosing each trajectory, values would converge quickly and a running time closer to C, rather than $(|A|C)^H$ can be achieved.

But since this optimal policy is not known at the beginning of planning, UCT samples actions at a state/depth node as determined by $v + \max_a \sqrt{2\log(n_{sd})/n_a}$, where v is the average value of action a from this state/depth pair based on previous trials, n_{sd} counts the number of times state s has been visited at depth d and n_a is the number of times a was tried there. This quantity represents the upper tail of the confidence interval for the node's value, and the strategy encourages an aggressive search policy that explores until it finds fairly good rewards, and then only periodically explores for better values. While UCT has performed remarkably in some very difficult domains including RTS games (Balla & Fern, 2009) and Go (Gelly & Silver, 2007), it can be shown that its worst case runtime is much worse than exponential in H (Coquelin & Munos, 2007). Thus, UCT does not satisfy Definition 36 and is therefore inadequate as a worst-case tractable planner for the KWIK-Rmax algorithm.

Forward Search Sparse Sampling

Forward Search Sparse Sampling (FSSS, Algorithm 33) employs a UCT-like search strategy without sacrificing the guarantees of SS. Recall that $Q^d(s, a)$ is an estimate of the depth d value for state s and action a. We introduce upper and lower bounds $U^d(s)$ and $L^d(s)$ for states and $U^d(s, a)$ and $L^d(s, a)$ for state-action pairs and control each top-down trial using the following algorithm.

Algorithm 33 FSSS(s, d)

```
if $d = 1$ (leaf) then
    $L^d(s,a) = U^d(s,a) = R(s,a), \forall a$
    $L^d(s) = U^d(s) = \max_a R(s,a)$
else if $n_{sd} = 0$ then
    for each $a \in A$ do
        $L^d(s,a) = V_{\min}$
        $U^d(s,a) = V_{\max}$
        for C times do
            $s' \sim T(s,a,\cdot)$
            $L^{d-1}(s') = V_{\min}$
            $U^{d-1}(s') = V_{\max}$
            $K^d(s,a) = K^d(s,a) \cup \{s'\}$
        end for
    end for
end if
$a^* = \operatorname{argmax}_a U^d(s,a)$
$s^* = \max_{s' \in K^d(s,a^*)}(U^{d-1}(s') - L^{d-1}(s'))$
FSSS($s^*$, $d-1$)
$n_{sd} = n_{sd} + 1$
$L^d(s,a^*) = R(s,a^*) + \gamma \sum_{s' \in K^d(s,a^*)} L^{d-1}(s')/C$
$U^d(s,a^*) = R(s,a^*) + \gamma \sum_{s' \in K^d(s,a^*)} U^{d-1}(s')/C$
$L^d(s) = \max_a L^d(s,a)$
$U^d(s) = \max_a U^d(s,a)$
```

Like UCT, FSSS proceeds in a series of top-down trials, each of which begins with the current state s and depth H and proceeds down the tree to improve the estimate of the actions at the root. Like SS, it limits its branching factor to C. Ultimately, it computes precisely the same value as SS, given the same samples. However, it benefits from a kind of pruning to reduce the amount of computation needed in many cases.

For instance, when $L^H(s,a^*) \geq \max_{a \neq a^*} U^H(s,a)$ for $a^* = \operatorname{argmax}_a U^H(s,a)$, no more trials are needed and a^* is the best action at the root. The following propositions show that, unlike UCT, FSSS solves the planning problem in accordance with Definition 36. The proofs are described in more detail in previous work (Walsh et al., 2010a).

Proposition 12. *On termination, the action chosen by FSSS is the same as that chosen by SS with analogous samples when drawing from a given* $\langle s, a, d \rangle$*.*

Proposition 13. *The total number of trials of FSSS before termination is bounded by the number of leaves in the tree.*

The proof of this second proposition comes from the intuition that every visited state-action-depth node becomes *closed* if its upper and lower bounds match, $L^d(s,a) = U^d(s,a)$. A leaf is

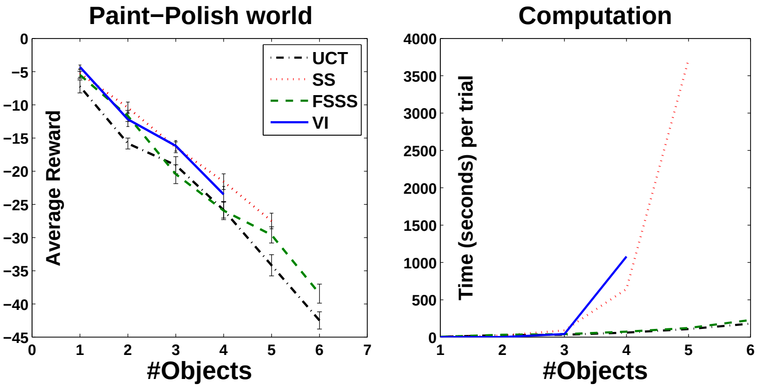


Figure 6.3: Planners in Paint-Polish world with increasing objects (40 runs). The average reward of an optimal policy (VI) decreases linearly with the number of objects. Note VI and SS become intractable.

closed the first time it is visited by a trial, and every trial closes a leaf, because if the search is not complete, the trial must end at *some* leaf which was previously open, thus upper bounding the number of trials before the search ends.

Figure 6.3 shows the three sample-based planners discussed above performing in the Paint-Polish Stochastic STRIPS domain. Like most relational domains $|S|$ here is exponential in $|O|$ and here $|A|$ grows linearly in $|O|$. With increasing $|O|$, Value Iteration quickly becomes intractable and SS falters soon after because of its exhaustive search. But, UCT and FSSS can still provide passable policies. FSSS's plans also remain more consistent than UCT, staying closer to the linearly decreasing expected reward of π^* for increasing O, even with the limited number of trials (2000) used for these two planners.

6.3.2 Possibilities for Existing Relational Planners

A number of planners have been proposed in the Relational RL community and more broadly in the planning community for relational action schemas and even for web-service description languages (Liu et al., 2007; Hoffmann et al., 2007). We provide a brief tour of these algorithms below, noting where complications arise in integrating them with the KWIK-Rmax framework.

We have already noted the similarity of the Stochastic STRIPS language we used as a testbed in this work and NID rules from Pasula et al. (2007), so it seems reasonable to investigate integrating an NID planner into our model-based RL architecture. While the original NID paper used SS to do planning, recently a more language-specific planner, PRADA (Lang & Toussaint, 2009) was shown to outperform sample-based planners in several NID benchmarks. It is tempting to then replace the sample based planners above with PRADA in the KWIK-Rmax algorithm, but this replacement is problematic. The reason is that PRADA uses a Bayesian inference strategy to plan, and inside this method each sequence of actions is considered to induce a belief state, a distribution over possible states. These belief states need to be induced from a set of static NID rules, which is *not* necessarily what our KWIK learners model during learning. Our KWIK learners instead solve learning problems in a predictive fashion, and are not beholden to any model type. PRADA currently does not support planning with such predictive “models”, unlike the generative sample based planners above. However, while it may not yet fit well into the KWIK-Rmax architecture, recent work on integrating PRADA with an E^3-inspired architecture and a factored (though heuristic) measure of uncertainty in the REX algorithm (Lang et al., 2010) has been empirically successful. Also, it may still be of use in the apprenticeship learning paradigm from Chapter 4 where operators without explicit measures of uncertainty can be used during learning (because mistakes will be corrected by the teacher). In summary, while PRADA does not have the formal theoretical guarantees of SS or FSSS, its empirical success warrants further investigation.

Perhaps a more promising class of planners for integration with the KWIK-Rmax architecture in relational domains are those that extend traditional MDP and fMDP planning algorithms to first-order MDPs. For instance, relational versions of Value Iteration (Boutilier et al., 2001) and Approximate Linear Programming (Sanner & Boutilier, 2005) have been developed based on first-order logic regression, and relational approximate policy iteration has also been proposed (Fern et al., 2006). These algorithms, which do not rely on explicit belief space representations and have more recognizable Bellman backups may prove more amenable to use in the KWIK-Rmax architecture with partially learned models. Another promising extension of traditional factored MDP planning has come in the form of First Order Decision Diagrams (FODDs) (Wang et al., 2008; Joshi et al., 2009). FODDs are first order generalization of arithmetic and binary decision diagrams (ADDs and BDDs), which have enjoyed success in factored MDP planning (Hoey et al., 1999). They work by creating directed graph structures whose internal nodes act like decision tree nodes and whose leaves contain values, but they are built from similar representations of the transition and reward function and then solved using a value-iteration like algorithm that includes First-Order regression. In such models, which explicitly make use of the transition and reward function, it may be possible to insert V_{max} transitions when constructing the model trees and thus make use of these planners in the KWIK-Rmax architecture.

6.4 Future Work

This thesis has endeavored to establish the first general sample complexity results for relational reinforcement learning domains in the online and apprenticeship settings. In Chapters 3 and 4, we have presented results to this effect in as general a form (language independent) manner as possible. However, in a number of situations, language dependent results were necessary because of the nature of the learned components. For instance, the ED-Learning results were presented essentially just for the Stochastic STRIPS language. With this in mind, the investigation of more languages in the action schema family could have two theoretical benefits. First, algorithms developed for these languages could generalize the more language specific results mentioned above. Second, further investigations may also be helpful in tightening the positive and negative KWIK and MBP learning results in Chapters 3 and 4. Finally, as we saw in Section 6.2, further research on expanding the constructors used in relational action schemas, between simple STRIPS and full FOMDP operators, will provide a better roadmap as to the tractability of these learning algorithms with respect to language expressiveness. One promising avenue for this is integrating results in the ILP community on learning expressive concepts (Lehmann & Hitzler, 2010) into the model learning component.

A number of future extensions are also possible in the newly emerging field of apprenticeship learning, as considered in Chapters 4 and 5. The protocol itself and the theoretical results pose a number of interesting questions, such as (1) Are there other language classes beyond MBP that are sufficient for efficient apprenticeship learning? (2) Are there other combinations of MBP and KWIK or other protocols (such as MB learning in the presence of label noise (Auer & Cesa-Bianchi, 1998)) that are sufficient for MBP learning? (3) Are changes to the protocol possible that will bound other important measures, such as the number of times an active optimal teacher might intervene?, (4) Are there ways to utilize active exploration to experiment with policies "near" the teacher without potentially investigating the whole (potentially dangerous) state space, and (5) What if the teacher can be considered to be less adversarial, as is the case in formal studies of *teaching dimension* (Goldman & Kearns, 1992).

While those questions remain for the larger apprenticeship learning paradigm, a number of questions remain regarding its use with relational action schemas. Specifically, we saw that in many cases, the restrictions from the online case regarding the size of conditions could be dropped in the presence of a teacher. It is a topic of future research to see whether other restrictive assumptions or otherwise impractical languages are learnable in the apprenticeship setting. Our results here with Task Graphs have shown that this is the case for at least a few models of interest. Also, given the connection between MB and equivalence queries (which are helpful in randomized algorithms for learning finite state automata from observations), it is an open question whether learning relational descriptions from partially observable traces (as done by Shahaf (2007) and also Zhuo et al. (2009)) can be done efficiently under certain assumptions.

Finally, there are a number of changes that could be made to the core RL investigation of our work. While this work has almost exclusively focussed on model-based RL algorithms, we have noted that most classic RRL (Dzeroski et al., 2001) has been done in the model-free paradigm. One of the benefits of this approach is it does not require an explicit planning step, saving an immense amount of computation time, but potentially giving that up in sample complexity. However, recent developments in mainstream RL have established sample efficiency results for certain model-free algorithms (Strehl et al., 2006), including some KWIK results for model-free algorithms (Li, 2009). This promising investigation of efficient model-free RRL, with an eye towards improving the early model-free RRL algorithms, is left to future work.

6.5 Concluding Remarks

In this work we have presented a number of results on the sample complexity of learning relational models for sequential decision making domains. The types of domains have varied from simple STRIPS descriptions to complex real world web-service task descriptions and we have considered two very different channels of experience: online and apprenticeship learning.

The common threads here are relational domains where the ramifications of an agent's actions need to be considered for future states, and where learning with limited experience is important or essential. It is this second property that differentiates our work from much of the prior work in RRL, action schema learning, and web-service description mining. Our algorithms are aware that experience is costly, in terms of time (and sometimes money) in most real world domains. They all either actively explore only parts of the state space that they need to, or need only a limited amount of guidance from a teacher. In addition, by using more complex languages and data structures like description logics or task graphs, and incorporating more complex algorithmic components, such as sample-based planners, we have pushed these algorithms to the point where they can function in more realistic domains.

It is our hope that these results and algorithms are more the fount of a literature stream, rather than its delta. That is, the combination of powerful relational representations and efficient algorithms makes practical agents possible for a host of new domains and we hope that this document has laid out the challenges facing practitioners in this new landscape as well as some vehicles for traversing it. But there's plenty to explore, and still the chance for high reward, so I'll see you out there.

Bibliography

Abbeel, Pieter and Ng, Andrew Y. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the Twenty-First international conference on Machine learning (ICML)*, 2004.

Abbeel, Pieter and Ng, Andrew Y. Exploration and apprenticeship learning in reinforcement learning. In *Proceedings of the Twenty-Second international conference on Machine learning (ICML)*, 2005.

Alonso, Gustavo, Casati, Fabio, Kuno, Harumi, and Machiraju, Vijay. *Web Services: Concepts, Architectures and Applications*. Springer, Berlin, 2004.

Angluin, Dana. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1988.

Auer, Peter and Cesa-Bianchi, Nicolò. On-line learning with malicious noise and the closure algorithm. *Annals of Mathematics and Artificial Intelligence*, 23(1-2):83–99, 1998.

Auer, Peter, Holte, Robert C., and Maass, Wolfgang. Theory and applications of agnostic pac-learning with small decision trees. In *Proceedings of the Twelfth International Conference on Machine Learning (ICML)*, 1995.

Auer, Peter, Jaksch, Thomas, and Ortner, Ronald. Near-optimal regret bounds for reinforcement learning. Technical report, University of Leoben, Leoben, Autria, 2007.

Baader, F. and Küsters, R. Nonstandard inferences in description logics: The story so far. In Gabbay, D.M., Goncharov, S.S., and Zakharyaschev, M. (eds.), *Mathematical Problems from Applied Logic I*, volume 4 of *International Mathematical Series*, pp. 1–75. Springer-Verlag, 2006.

Baader, Franz, Calvanese, Diego, McGuinness, Deborah L., Nardi, Daniele, and Patel-Schneider, Peter F. (eds.). *The description logic handbook: theory, implementation, and applications*. Cambridge University Press, New York, NY, USA, 2003.

Balla, Radha-Krishna and Fern, Alan. UCT for tactical assault planning in real-time strategy games. In *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI)*, 2009.

Bellman, Richard E. *Dynamic Programming.* Princeton University Press, Princeton, NJ, 1957.

Benson, Scott. *Learning Action Models for Reactive Autonomous Agents.* PhD thesis, Stanford University, Palo Alto, California, 1996.

Bertsimas, Dimitris and Tsitsiklis, John N. *Introduction to Linear Optimization.* Athena Scientific, Belmont, MA, 1997. ISBN 1-886529-19-1.

Bex, Geert Jan, Neven, Frank, Schwentick, Thomas, and Tuyls, Karl. Inference of concise DTDs from XML data. In *Proceedings of the Thirty-Second International Conference on Very Large Data Bases (VLDB)*, 2006.

Blum, Avrim. Learning boolean functions in an infinite attribute space. *Machine Learning*, 9 (4):373–386, 1992. ISSN 0885-6125.

Blum, Avrim. Separating distribution-free and mistake-bound learning models over the boolean domain. *SIAM Journal on Computing*, 23(5):990–1000, 1994.

Blum, Avrim L. and Furst, Merrick L. Fast planning through planning graph analysis. *Artificial Intelligence*, 90:1636–1642, 1995.

Borgida, Alexander, Brachman, Ronald J., McGuinness, Deborah L., and Resnick, Lori Alperin. Classic: A structural data model for objects. In *SIGMOD Conference*, 1989.

Boutilier, Craig, Reiter, Raymond, and Price, Bob. Symbolic dynamic programming for first-order MDPs. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI)*, 2001.

Brafman, Ronen I. and Tennenholtz, Moshe. R-max–a general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research*, 3:213–231, October 2002.

Brunskill, Emma, Leffler, Bethany R., Li, Lihong, Littman, Michael L., and Roy, Nicholas. CORL: A continuous-state offset-dynamics reinforcement learner. In *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence (UAI)*, 2008.

Bylander, Tom. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2):165–204, 1994.

Carman, Mark James and Knoblock, Craig A. Learning semantic definitions of online information sources. *Journal of Artificial Intelligence Research*, 30:1–50, 2007.

Cohen, William W. Pac-learning recursive logic programs: Efficient algorithms. *Journal of Artificial Intelligence Research*, 2:501–539, 1995a.

Cohen, William W. Pac-learning recursive logic programs: Negative results. *Journal of Artificial Intelligence Research*, 2:541–573, 1995b.

Cohen, William W. and Hirsh, Haym. Learning the classic description logic: Theoretical and experimental results. In *Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning*, 1994.

Coquelin, Pierre-Arnaud and Munos, Rémi. Bandit algorithms for tree search. In *Proceedings of the Twenty-Third Conference on Uncertainty in Artificial Intelligence (UAI)*, 2007.

Cormen, Thomas H., Stein, Clifford, Rivest, Ronald L., and Leiserson, Charles E. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 2001.

Croonenborghs, Tom, Driessens, Kurt, and Bruynooghe, Maurice. Learning relational options for inductive transfer in relational reinforcement learning. In *Proceedings of the Seventeenth Annual International Conference on Inductive Logic Programming (ILP)*, 2007a.

Croonenborghs, Tom, Ramon, Jan, Blockeel, Hendrik, and Bruynooghe, Maurice. Online learning and exploiting relational models in reinforcement learning. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI)*, 2007b.

Dean, Thomas and Kanazawa, Keiji. A model for reasoning about persistence and causation. *Computational intelligence*, 5:142–150, 1989.

Diuk, Carlos, Cohen, Andre, and Littman, Michael L. An object-oriented representation for efficient reinforcement learning. In *Proceedings of the Twenty-Fifth International Conference on Machine Learning (ICML)*, 2008.

Diuk, Carlos, Li, Lihong, and Leffler, Bethany. The adaptive k-meteorologists problem and its application to structure learning and feature selection in reinforcement learning. In *Proceedings of the Twenty-Sixth International Conference on Machine Learning (ICML)*, 2009.

Driessens, K. and Dzeroski, S. Integrating experimentation and guidance in relational reinforcement learning. In *Proceedings of the Nineteenth International Conference on Machine Learning (ICML)*, 2002.

Driessens, Kurt and Ramon, Jan. Relational instance based regression for relational reinforcement learning. In *Proceedings of the Twentieth International Conference on Machine Learning (ICML)*, 2003.

Driessens, Kurt, Ramon, Jan, and Blockeel, Hendrik. Speeding up relational reinforcement learning through the use of an incremental first order decision tree algorithm. In Raedt, L. De and Flach, P. (eds.), *Proceedings of the Twelfth European Conference on Machine Learning (ECML)*, volume 2167, September 2001.

Driessens, Kurt, Ramon, Jan, and Gärtner, Thomas. Graph kernels and gaussian processes for relational reinforcement learning. *Machine Learning*, 64(1-3):91–119, 2006.

Dzeroski, Saso. Relational reinforcement learning for agents in worlds with objects. In *Proceedings of the Convention on Artificial Intelligence and Simulation of Behavior*, 2002.

Dzeroski, Saso, Muggleton, Stephen, and Russell, Stuart J. PAC-learnability of determinate logic programs. In *Proceedings of the Fifth Annual ACM Workshop on Computational Learning Theory (COLT)*. ACM Press, 1992.

Dzeroski, Saso, De Raedt, Luc, and Driessens, Kurt. Relational reinforcement learning. *Machine Learning*, 43(1):7–52, 2001.

Fern, Alan, Yoon, Sung Wook, and Givan, Robert. Approximate policy iteration with a policy language bias: Solving relational markov decision processes. *Journal of Artificial Intelligence Research (JAIR)*, 25:75–118, 2006.

Fikes, Richard and Nilsson, Nils J. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 5:189–208, 1971.

Fong, Philip W. L. A quantitative study of hypothesis selection. In *Proceedings of the Twelfth International Conference on Machine Learning (ICML)*, 1995.

Franconi, Enrico, Seylan, Inan, and de Bruijn, Jos. Effective query rewriting with ontologies over dboxes. In *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI)*, 2009.

Garey, Michael R. and Johnson, David S. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990. ISBN 0716710455.

Gelly, Sylvain and Silver, David. Combining online and offline knowledge in UCT. In *Proceedings of the Twenty-Fourth International Conference on Machine Learning (ICML)*, 2007.

Gil, Yolanda. Learning by experimentation: Incremental refinement of incomplete planning domains. In *Proceedings of the Eleventh International Conference on Machine Learning (ICML)*, 1994.

Goldman, Sally A. and Kearns, Michael J. On the complexity of teaching. *Journal of Computer and System Sciences*, 50:303–314, 1992.

Großmann, Axel, Hölldobler, Steffen, and Skvortsova, Olga. Symbolic dynamic programming within the fluent calculus. In *IASTED International Conference Artificial and Computational Intelligence*, 2002.

Guestrin, C., Koller, D., Gearhart, C., and Kanodia, N. Generalizing plans to new environments in relational MDPs. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI)*, 2003a.

Guestrin, Carlos, Koller, Daphne, Parr, Ronald, and Venkataraman, Shobha. Efficient solution algorithms for factored MDPs. *Journal of Artificial Intelligence Research*, 19:399–468, 2003b.

Hellerstein, Lisa and Servedio, Rocco A. On pac learning algorithms for rich boolean function classes. *Theoretical Computer Science*, 384(1):66–76, 2007.

Helmbold, David P., Littlestone, Nick, and Long, Philip M. Apple tasting. *Information and Computation*, 161(2):85–139, 2000.

Hess, Andreas and Kushmerick, Nicholas. Learning to attach semantic metadata to web services. In *Proceedings of the Second International Semantic Web Conference (ISWC)*, 2003.

Hoeffding, Wassily. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.

Hoey, Jesse, St-Aubin, Robert, Hu, Alan J., and Boutilier, Craig. Spudd: Stochastic planning using decision diagrams. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence (UAI)*, 1999.

Hoffmann, Jörg, Bertoli, Piergiorgio, and Pistore, Marco. Web service composition as planning, revisited: In between background theories and initial state uncertainty. In *Proceedings of the Twenty-Second Conference on Artificial Intelligence (AAAI)*, 2007.

Holmes, Michael P. and Isbell, Charles Lee. Schema learning: Experience-based construction of predictive action models. In *In Advances in Neural Information Processing Systems (NIPS) 17*, 2005.

Horváth, Tamás and Turán, Gy&oouml;rgy. Learning logic programs with structured background knowledge. *Artificial Intelligence*, 128(1-2):31–97, 2001.

Huang, Jingshan, Dang, Jiangbo, and Huhns, Michael N. Ontology reconciliation for service-oriented computing. In *Proceedings of the IEEE International Conference on Services Computing (SCC)*, 2006.

Jong, Nicholas K. and Stone, Peter. State abstraction discovery from irrelevant state variables. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI)*, 2005.

Joshi, Saket, Kersting, Kristian, and Khardon, Roni. Generalized first order decision diagrams for first order markov decision processes. In *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI)*, 2009.

Kaelbling, Leslie Pack, Littman, Michael L., and Moore, Andrew W. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

Kakade, Sham. *On the Sample Complexity of Reinforcement Learning*. PhD thesis, University College London, UK, 2003.

Kearns, Michael and Singh, Satinder. Near-optimal reinforcement learning in polynomial time. *Machine Learning*, 49(2-3):209–232, 2002.

Kearns, Michael, Mansour, Yishay, and Ng, Andrew Y. A sparse sampling algorithm for near-optimal planning in large Markov decision processes. *Machine Learning*, 49:193–208, 2002.

Kearns, Michael J. and Koller, Daphne. Efficient reinforcement learning in factored MDPs. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI)*, 1999.

Kearns, Michael J. and Vazirani, Umesh V. *An Introduction to Computational Learning Theory*. MIT Press, Cambridge, MA, first edition, 1994.

Kersting, Kristian and Raedt, Luc De. Logical markov decision programs and the convergence of logical td(lambda). In *Proceedings of the Fourteenth International Conference on Inductive Logic Programming (ILP)*, 2004.

Khardon, Roni. Learning action strategies for planning domains. *Artificial Intelligence*, 113: 125–148, 1999a.

Khardon, Roni. Learning to take actions. *Machine Learning*, 35(1):57–90, 1999b.

Khardon, Roni and Arias, Marta. The subsumption lattice and query learning. *Journal of Computer and System Sciences*, 72(1):72–94, 2006.

Kivinen, Jyrki and Warmuth, Manfred K. Exponentiated gradient versus gradient descent for linear predictors. *Information and Computation*, 132(1):1–63, 1997.

Klug, Anthony. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of the ACM*, 29(3):699–717, 1982. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/322326.322332.

Klusch, Matthias, Kapahnke, Patrick, and Zinnikus, Ingo. Sawsdl-mx2: A machine-learning approach for integrating semantic web service matchmaking variants. In *Proceedings of the Seventh IEEE International Conference on Web Services (ICWS)*, 2009.

Kocsis, Levente and Szepesvári, Csaba. Bandit based Monte-Carlo planning. In *Proceedings of the Seventeenth European Conference on Machine Learning (ECML)*, 2006.

Kuter, Ugur, Nau, Dana, Reisner, Elnatan, and Goldman, Robert P. Using classical planners to solve nondeterministic planning problems. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS)*, 2008.

Lane, Terran and Wilson, Andrew. Toward a topological theory of relational reinforcement learning for navigation tasks. In *Proceedings of the Eighteenth International Florida Artificial Intelligence Research Society Conference (FLAIRS)*, 2005.

Lane, Terran, Ridens, Martin, and Stevens, Scott. Reinforcement learning in nonstationary environment navigation tasks. In *Proceedings of the Twentieth Conference of the Canadian*

Society for Computational Studies of Intelligence on Advances in Artificial Intelligence (CAI), 2007.

Lang, Tobias and Toussaint, Marc. Approximate inference for planning in stochastic relational worlds. In *Proceedings of the Twenty-Sixth International Conference on Machine Learning (ICML)*, June 2009.

Lang, Tobias, Toussaint, Marc, and Kersting, Kristian. Exploration in relational worlds. In *Proceedings of the European Conference on Machine Learning*, 2010.

Leffler, Bethany R., Littman, Michael L., and Edmunds, Timothy. Efficient reinforcement learning with relocatable action models. In *AAAI-07: Proceedings of the Twenty-Second Conference on Artificial Intelligence*, Menlo Park, CA, USA, 2007. The AAAI Press.

Lehmann, Jens and Haase, Christoph. Ideal downward refinement in the el description logic. Technical report, Universitt of Leipzig, Leipzig, Germany, 2009.

Lehmann, Jens and Hitzler, Pascal. Concept learning in description logics using refinement operators. *Machine Learning*, 78:203–250, 2010.

Lerman, Kristina, Plangrasopchok, Anon, and Knoblock, Craig A. Automatically labeling the inputs and outputs of web services. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI)*, 2006.

Levesque, Hector J., Reiter, Raymond, Lesprance, Yves, Lin, Fangzhen, and Scherl, Richard B. Golog: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31, 1997.

Li, Lihong. *A Unifying Framework for Computational Reinforcement Learning Theory*. PhD thesis, Rutgers University, NJ, USA, 2009.

Li, Lihong, Walsh, Thomas J., and Littman, Michael L. Towards a unified theory of state abstraction for MDPs. In *Proceedings of the Ninth International Symposium on Artificial Intelligence and Mathematics*, 2006.

Li, Lihong, Littman, Michael L., and Walsh, Thomas J. Knows what it knows: A framework for self-aware learning. In *Proceedings of the Twenty-Fifth International Conference on Machine Learning (ICML)*, 2008.

Liang, Qianhui Althea and Lam, Herman. Web service matching by ontology instance categorization. In *Proceedings of the IEEE International Conference on Services Computing (SCC)*, 2008. ISBN 978-0-7695-3283-7-01. doi: http://dx.doi.org/10.1109/SCC.2008.133.

Lin, Long-Ji. Programming robots using reinforcement learning and teaching. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI)*, 1991.

Littlestone, Nick. Learning quickly when irrelevant attributes abound. *Machine Learning*, 2: 285–318, 1988.

Littman, Michael Lederman. *Algorithms for Sequential Decision Making*. PhD thesis, Department of Computer Science, Brown University, Providence, RI, February 1996.

Liu, Zhen, Ranganathan, Anand, and Riabov, Anton V. A planning approach for message-oriented semantic web service composition. In *Proceedings of the Twenty-Second Conference on Artificial Intelligence (AAAI)*, 2007.

Madani, Omid. Polynomial value iteration algorithms for deterministic mdps. In *Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence (UAI)*, 2002.

Mahadevan, Sridhar. Average reward reinforcement learning: Foundations, algorithms, and empirical results. *Machine Learning*, 22(1):159–195, 1996.

Marconi, Annapaola, Pistore, Marco, Poccianti, Piero, and Traverso, Paolo. Automated web service composition at work: the Amazon/MPS case study. In *Proceedings of the IEEE International Conference on Web Services (ICWS)*, 2007.

Martino, Benedetto De, Kumaran, Dharshan, Seymour, Ben, and Dolan, Raymond J. Frames, biases, and rational decision-making in the human brain. *Science*, 313(5787):684–687, August 2006.

McCarthy, John. Situations, actions and causal laws. Technical report, Stanford University, Palo Alto, CA, USA, 1963.

Milicic, Maja. *Action, Time and Space in Description Logics*. PhD thesis, Dresden University of Technology, Dresden, Germany, 2008.

Minton, Steven. *Learning Search Control Knowledge: An Explanation-Based Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 1988.

Mitchell, Tom M. *Machine Learning*. McGraw-Hill, New York, 1997.

Murata, Tadao. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77 (4):541–580, April 1989.

Nienhuys-Cheng, Shan-Hwei and de Wolf, Ronald (eds.). *Foundations of Inductive Logic Programming*, volume 1228 of *Lecture Notes in Computer Science*. Springer, 1997.

Pasula, Hanna M., Zettlemoyer, Luke S., and Kaelbling, Leslie Pack. Learning symbolic models of stochastic domains. *Journal of Artificial Intelligence Research*, 29:309–352, 2007.

Patil, A., Oundhakar, S., Sheth, A., and Verma, K. Meteor-s web service annotation framework. In *Proceedings of the Thirteenth International World Wide Web Conference (WWW)*, 2004.

Pichuka, Chaithanya, Bapi, Raju S., Bhagvati, Chakravarthy, Pujari, Arun K., and Deekshatulu, B. L. A tighter error bound for decision tree learning using pac learnability. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI)*, 2007.

Pitt, Leonard and Valiant, Leslie G. Computational limitations on learning from examples. *Journal of the ACM*, 35(4):965–984, 1988.

Puterman, Martin L. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, New York, 1994.

Ratliff, Nathan D., Bradley, David M., Bagnell, J. Andrew, and Chestnutt, Joel E. Boosting structured prediction for imitation learning. In *Advances in Neural Information Processing Systems 19 (NIPS)*, 2006.

Reiter, Raymond. The frame problem in situation the calculus: a simple solution (sometimes) and a completeness result for goal regression. *Artificial intelligence and mathematical theory of computation: papers in honor of John McCarthy*, pp. 359–380, 1991.

Rota, Gian-Carlo. The number of partitions of a set. *American Mathematical Monthly*, 71(5): 495–504, 1964.

Russell, Stuart J. and Norvig, Peter. *Artificial Intelligence: A Modern Approach*. Prentice Hall, first edition, January 1995.

Sanner, Scott and Boutilier, Craig. Approximate linear programming for first-order mdps. In *Proceedings of the Twenty-First Conference on Uncertainty in Artificial Intelligence (UAI)*, 2005.

Sanner, Scott and Boutilier, Craig. Practical linear value-approximation techniques for first-order mdps. In *Proceedings of the Twenty-Second Conference on Uncertainty in Artificial Intelligence (UAI)*, 2006.

Sanner, Scott and Boutilier, Craig. Practical solution techniques for first-order mdps. *Artificial Intelligence*, 173(5-6):748–788, 2009.

Shafto, Patrick and Goodman, Noah. Teaching games: Statistical sampling assumptions for pedagogical situations. *Proceedings of the Thirtieth Annual Meeting of the Cognitive Science Society*, 2008.

Shahaf, Dafna. Logical filtering and learning in partially observable worlds. Master's thesis, University of Illinois at Urbana-Champaign, 2007.

Sharp, Henry. Cardinality of finite topologies. *Journal of Combinatorial Theory*, 5:82–86, 1968.

Sherstov, Alexander A. and Stone, Peter. Improving action selection in mdp's via knowledge transfer. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI)*, 2005.

Singh, Satinder P. and Yee, Richard C. An upper bound on the loss from approximate optimal-value functions. *Machine Learning*, 16:227, 1994.

Strehl, Alexander and Littman, Michael L. Online linear regression and its application to model-based reinforcement learning. In *Advances in Neural Information Processing Systems 20 (NIPS)*, 2007.

Strehl, Alexander, Li, Lihong, Wiewiora, Eric, Langford, John, and Littman, Michael L. PAC model-free reinforcement learning. In *Proceedings of the Twenty-Third International Conference on Machine Learning (ICML)*, 2006.

Strehl, Alexander L. and Littman, Michael L. A theoretical analysis of model-based interval estimation. In *Proceedings of the Twenty-Second international conference on Machine learning (ICML)*, 2005.

Strehl, Alexander L., Diuk, Carlos, and Littman, Michael L. Efficient structure learning in factored-state MDPs. In *Proceedings of the Twenty-Second Conference on Artificial Intelligence (AAAI)*, 2007.

Strehl, Alexander L., Li, Lihong, and Littman, Michael L. Reinforcement learning in finite MDPs: PAC analysis. *Journal of Machine Learning Research*, 10(2):413–444, 2009.

Sutton, Richard S. and Barto, Andrew G. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, March 1998. ISBN 0-262-19398-1.

Syed, Umar and Shapire, Robert E. A game-theoretic approach to apprenticeship learning. In *Advances in Neural Information Processing Systems 20 (NIPS)*, 2007.

Torrey, Lisa, Walker, Trevor, Shavlik, Jude W., and Maclin, Richard. Using advice to transfer knowledge acquired in one reinforcement learning task to another. In *Proceedings of the Sixteenth European Conference on Machine Learning (ECML)*, 2005.

Valiant, L. G. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, November 1984.

van der Aalst, W. and Weijters, A. Process mining: A research agenda. *Computers in Industry*, 53(3):231–244, 2004.

van der Aalst, Wil, Weijters, Ton, and Maruster, Laura. Workflow mining: Discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9): 1128–1142, 2004.

van Otterlo, Martijn. *The Logic of Adaptive Behavior - Knowledge Representation and Algorithms for Adaptive Sequential Decision Making under Uncertainty in First-Order and Relational Domains*, volume 192 of *Frontiers in Artificial Intelligence and Applications.* IOS Press, 2009.

Walker, Trevor, Torrey, Lisa, Shavlik, Jude, and Maclin, Richard. Building relational world models for reinforcement learning. In *Proceedings of the Seventeenth Conference on Inductive Logic Programming (ILP)*, 2007.

Walsh, Thomas J. and Littman, Michael L. Efficient learning of action schemas and web-service descriptions. In *Proceedings of the Twenty-Third Conference on Artificial Intelligence (AAAI)*, 2008.

Walsh, Thomas J., Nouri, Ali, Li, Lihong, and Littman, Michael L. Learning and planning in environments with delayed feedback. *Journal of Autonomous Agents and Multi-Agent Systems*, 18(1):83–101, February 2009a.

Walsh, Thomas J., Szita, Istvn, Diuk, Carlos, and Littman, Michael L. Exploring compact reinforcement-learning representations with linear regression. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence (UAI)*, 2009b.

Walsh, Thomas J., Goschin, Sergiu, and Littman, Michael L. Integrating sample-based planning and model-based reinforcement learning. In *Proceedings of the Twenty-Fourth Conference on Artificial Intelligence (AAAI)*, 2010a.

Walsh, Thomas J., Subramanian, Kaushik, Littman, Michael L., and Diuk, Carlos. Generalizing apprenticeship learning across hypothesis classes. In *Proceedings of the Twenty-Seventh International Conference on Machine Learning (ICML)*, 2010b.

Wang, Chenggang, Joshi, Saket, and Khardon, Roni. First order decision diagrams for relational mdps. *Journal of Artificial Intelligence Research*, 31(1):431–472, 2008.

Wang, Xuemei. Learning by observation and practice: An incremental approach for planning operator acquisition. In *Proceedings of the Twelfth International Conference on Machine Learning (ICML)*, 1995.

Watkins, Christopher J.C.H. *Learning from Delayed Rewards*. PhD thesis, King's College, University of Cambridge, UK, 1989.

Whitehead, Steven D. Complexity and cooperation in q-learning. In *Proceedings of the Eighth International Workshop on Machine Learning (ICML)*, 1991.

Yaman, Fusun, Oates, Tim, and Burstein, Mark. A context driven approach for workflow mining. In *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI)*, 2009.

Yang, Qiang, Wu, Kangheng, and Jiang, Yunfei. Learning action models from plan examples using weighted max-sat. *Artificial Intelligence*, 171(2-3):107–143, 2007.

Younes, Håkan L. S., Littman, Michael L., Weissman, David, and Asmuth, John. The First Probabilistic Track of the International Planning Competition. *Journal of Artificial Intelligence Research*, 24:851–887, 2005.

Zhuo, Hankz Hankui, Hu, Derek Hao, Hogg, Chad, and Munoz-Avila, Qiang Yangand Hector. Learning HTN method preconditions and action models from partial observations. In *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI)*, 2009.

Vita
Thomas J. Walsh

Education

Ph.D. in Computer Science, Rutgers University, 2010

Bachelor of Science in Computer Science, University of Maryland Baltimore County (UMBC), 2003

Experience

8/2004 – present Graduate Assistant, Rutgers University, New Brunswick, NJ

5/2008 – 8/2008 Intern, Siemens Corporate Research, Princeton, NJ

8/2003 – 5/2005 Teaching Assistant, Rutgers University, New Brunswick, NJ

5/2004 – 12/2004 Intern, Applied Signal Technology, Annapolis Junction, MD

5/2003 – 8/2003 Guest Researcher, National Institute of Standards and Technology (NIST), Gaithersburg, MD

Publications

1. Thomas J. Walsh, Sergiu Goshin, and Michael L. Littman “Integrating Sample-based Planning and Model-based Reinforcement Learning” *In Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI-10)*, Atlanta, GA, 2010.

2. Thomas J. Walsh, Kaushik Subramanian, Michael L. Littman, and Carlos Diuk “Generalizing Apprenticeship Learning across Hypothesis Classes” *In Proceedings of the Twenty-Seventh International Conference on Machine Learning (ICML-10)*, Haifa, Israel, 2010.

3. Thomas J. Walsh, Istvn Szita, Carlos Diuk, and Michael L. Littman “Exploring Compact Reinforcement-Learning Representations with Linear Regression” *In Proceedings of the*

25th Conference on Uncertainty in Artificial Intelligence (UAI-09), Montreal, Quebec, 2009.

4. Thomas J. Walsh, Ali Nouri, Lihong Li, and Michael L. Littman "Learning and Planning in Environments with Delayed Feedback" *In the Journal of Autonomous Agents and Multi-Agent Systems*, Volume 18, Issue1, 83-101, February, 2009.

5. Thomas J. Walsh and Michael L. Littman Efficient "Learning of Action Schemas and Web-Service Descriptions" *In Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI-08)*, Chicago, IL, 2008.

6. Lihong Li, Michael L. Littman, Thomas J. Walsh "Knows What It Knows: A Framework for Self-Aware Learning" *In Proceedings of the Twenty-Fifth International Conference on Machine Learning (ICML-08)*, Helsinki, Finland, 2008.

7. Fusun Yaman, Thomas J. Walsh, Michael L. Littman, Marie desJardins "Democratic Approximation of Lexicographic Preference Models" *In Proceedings of the Twenty-Fifth International Conference on Machine Learning (ICML-08)*, Helsinki, Finland, 2008.

8. Thomas J. Walsh and Michael L. Littman. "A Multiple representation approach to learning dynamical Systems" *AAAI Fall Symposium on Representation Change*, Washington D.C., 2007

9. Thomas J. Walsh, Ali Nouri, Lihong Li, and Michael L. Littman "Planning and Learning in Environments with Delayed Feedback" *In Proceedings of the 18th European Conference on Machine Learning (ECML-07)*, Warsaw, Poland, 2007.

10. Thomas J. Walsh and Michael L. Littman "Planning with Conceptual Models Mined from User Behavior" *In Proceedings of the AAAI-07 Workshop on Acquiring Planning Knowledge via Demonstration*, Vancouver, BC, 2007.

11. Thomas J. Walsh, Lihong Li, and Michael L. Littman "Transferring State Abstractions Between MDPs" *In Proceeding of the ICML-06 Workshop on Structural Knowledge Transfer for Machine Learning*, Pittsburgh, PA, 2006.

12. Lihong Li, Thomas J. Walsh, and Michael L. Littman "Towards a Unified Theory of State Abstraction for MDPs" *Proceedings of the Ninth International Symposium on Artificial Intelligence and Mathematics (AIMA06)*, Ft. Lauderdale, FL, 2006.

13. Alex Borgida, Thomas J. Walsh, and Haym Hirsh. “Towards Measuring Similarity in Description Logics” *In Proceedings of the 2005 International Workshop on Description Logics (DL2005)*, Edinburgh, Scotland, 2005.

14. Bethany R. Leffler, Michael L. Littman, Alexander L. Strehl, Thomas J. Walsh. “Efficient Exploration With Latent Structure” *In Proceedings of Robotics: Science and Systems.* Cambridge, Massachusetts, 2005.

15. Thomas J. Walsh and D. Richard Kuhn. “Challenges in Securing Voice over IP” *IEEE Security & Privacy* Vol 3(3) 2005 (May/June) : 44-49.

16. D. Richard Kuhn, Thomas J. Walsh, Steffen Fries. *Security Considerations for Voice Over IP Systems* Special Publication from the National Institute of Standards and Technology, 2005

17. Dennis D.Y. Kim, Thomas T.Y. Kim, Thomas Walsh, Yoshifumi Kobayashi, Tara C. Matise, Steven Buyske, and Abram Gabriel “Widespread RNA Editing of Embedded Alu Elements in the Human Transcriptome” *Genome Ressearch* 2004 14 (September): 1719-1725.