AN OBJECT-ORIENTED REPRESENTATION FOR EFFICIENT REINFORCEMENT LEARNING

BY CARLOS GREGORIO DIUK WASSER

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

An Object-oriented Representation for Efficient Reinforcement Learning

by Carlos Gregorio Diuk Wasser

Dissertation Director: Michael L. Littman

Agents (humans, mice, computers) need to constantly make decisions to survive and thrive in their environment. In the reinforcement-learning problem, an agent needs to learn to maximize its long-term expected reward through direct interaction with the world. To achieve this goal, the agent needs to build some sort of internal representation of the relationship between its actions, the state of the world and the reward it expects to obtain. In this work, I show how the way in which the agent represents state and models the world plays a key role in its ability to learn effectively. I will introduce a new representation, based on objects and their interactions, and show how it enables several orders of magnitude faster learning on a large class of problems. I claim that this representation is a natural way of modeling state and that it bridges a gap between generality and tractability in a broad and interesting class of domains, namely those of relational nature. I will present a set of learning algorithms that make use of this representation in both deterministic and stochastic environments, and present polynomial bounds that prove their efficiency in terms of learning complexity.

Acknowledgements

I came to Rutgers in the Fall semester of 2003, planning to work at the intersection of machine learning, artificial intelligence and cognitive science. At that point in time, I did not know who I was going to work with. On my first year I took three different classes that felt relevant to my interests, and they all happened to be taught by Michael Littman. I realized then that he would be the perfect advisor, and I have not been disappointed. Working with Michael has been inspiring, he became an academic role-model and it never stopped being a lot of fun.

But working with Michael would have not been enough if it had not been for my labmates and colleagues at Rutgers Laboratory for Real-Life Reinforcement Learning, or RL^3. A special mention should be first made to the people I collaborated the most with: Thomas Walsh, Alexander Strehl and Lihong Li. But the lab would not have been the lab without the useful discussions, lunches and exchanges with Ali Nouri, Bethany Leffler, Istvan Szita, Ari Weinstein, John Asmuth, Chris Mansley, Sergiu Goschin and Monica Babes.

I want to give a special thank you to my thesis committee: Ken Shan, Alex Borgida and Andrew Barto.

The long lab afternoons were also made possible by my frequent corridor crossings into the office of my great friends Miguel Mosteiro and Rohan Fernandes. I refuse to call the hours spent in their office "procrastination", as they were incredibly educational, from formal probability theory and theorem proving, to the history and politics of India. Part of my intellectual development and life in the CS Department was also immensely facilitated by my friendship with Matias Cuenca Acuna, Christian Rodriguez, Marcelo Mydlarz, Smriti Bhagat, Nikita Lytkin and Andre Madeira.

One often misses the subtle nuances of a subject until one has to teach it. My understanding of many subjects has improved with my experiences as a teaching assistant. I owe a debt to the kind supervision of Martin Farach-Colton, S. Muthukrishnan and Leonid Khachiyan.

The years at Rutgers were also one of the most gratifying experiences I will always cherish thanks to many people. To the Rutgers crowd: Miguel Mosteiro, Silvia Romero, Pablo Mosteiro, Martin Mosteiro, Roberto Gerardi, Ignacia "Nachi" Perugorria, Alberto Garcia-Raboso, Rocio Naveiras, Jose Juknevich, Mercedes Morales, Matias Cuenca, Cecilia Martinez, Selma Cohen, Gustavo Crembil, Christian Rodriguez, Marcelo Mydlarz, Cecilia Della Valle, Macarena Urzua, Felipe and Claudia Menanteau, Felipe Troncoso, Claudia Cabello, Juan Jose Adriasola, Viviana Cobos, Molly Palmer, Silvia Font, Cristobal Cardemil, Soledad Chacon, Dudo and Guiomar Vergara.

Another paragraph is deserved by my friends at the Graduate Student Association and our union, the Rutgers AAUP-AFT: Scott Bruton, Ryan Fowler, Naomi Fleming, Brian Graf, Elric Kline and Charlotte Whalen.

My great friends in Argentina, who always visited, skyped and were there: Nico Barea, Juli Herrera, Hernan Schikler, Sebastian Ludmer, Fer Sciarrotta, Maria Elinger, Martin Mazzini, Sebas Pechersky, Ana Abramowski, Lu Litichever, Laura Gomez, Sergi Daicz, Sergi "Techas" Zlotnik, Dan Zlotnik, Charly Lopez Holtmann, Diego Fernandez-Slezak, Vic Robledo, Gaston Giribet, Julian Faivovich, and the list goes on and on! To Fiorella Kotsias and Alejo Flah, Fran D'Alessio and Pau Lamamie.

A source of constant encouragement and support has been my family: my parents Juan Diuk and Lilian Wasser, my siblings Pablo, Beatriz and Maria Ana Diuk, and my fantastic nephews Nico, Ari, Santi, Diego and Tamara.

Last, but by no means the least, Vale, to whom this dissertation is dedicated, and Manu, who arrived in the last few months of my graduate work and helped me finish. I will never forget his long naps on my lap while I was writing this dissertation.

Dedication

A Vale y Manu, por todo.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

> El ejecutor de una empresa atroz debe imaginar que ya la ha cumplido, debe imponerse un porvenir que sea irrevocable como el pasado.
>
> Jorge Luis Borges (1899-1986)
> El jardin de los senderos que se bifurcan.[a]

[a] Whosoever would undertake some atrocious enterprise should act as if it were already accomplished, should impose upon himself a future as irrevocable as the past.

The goal of this dissertation is to introduce a new representation for reinforcement-learning problems based on objects and their interactions. I will show how this representation, which I call Object-Oriented Markov Decision Processes (OO-MDPs), is a natural way of representing state in a large class of problems, enabling orders of magnitude faster learning.

In this chapter, I will introduce reinforcement learning as the problem of sequential decision-making and highlight some of its challenges. The proposed representation tackles some of these challenges by enabling a compact description of environment dynamics for a broad class of problems, while allowing smart and efficient exploration.

1.1 Learning by Direct Interaction

A computer scientist arrives at a new city for a conference and needs to get to her hotel. She can take a taxi, an action that will be expensive but has a high probability of reaching the destination, although the time it takes has high variance. She can also take a train, which costs less and has low variance in the duration of the trip, but she risks not getting off at the right stop or not finding her way from the station to the

hotel. If taking the train, she could also spend some extra time turning on her computer and looking for directions online, which incurs an extra cost but increases the chances of success.

An Internet protocol needs to route packets. It has to decide where to forward each packet, at what rate, which ones to queue and which ones to drop. The number of packets and response times involved keep changing. Its goal is to maximize the amount of packets successfully transmitted, and minimize the number of hops those packets make toward their destination.

A mouse is trapped in a psychologist's maze and needs to find food. All those sounds, lights and shocks are stressing him out, and as time goes by he gets hungrier. He can wait and count on the experimenter's pity, try running around randomly, or start pressing levers.

All of the above are examples of *sequential decision-making* problems. In each case, there's an agent (a human, an animal or a computer program) that needs to make decisions in an unknown environment, and those decisions have both immediate and long-term consequences, incur costs and can produce rewards.

Between the late 1980's and the mid-1990's, ideas from psychology (like Pavlovian and instrumental conditioning), optimal control and economics gave birth to the field of Reinforcement Learning (Sutton, 1988; Barto et al., 1989; Bertsekas & Tsitsiklis, 1996; Kaelbling et al., 1996; Sutton & Barto, 1998), a sub-area of Artificial Intelligence. In the words of Rich Sutton, the *reinforcement-learning (or reward) hypothesis* states that "all of what we mean by goals and purposes can be well thought of as maximization of the expected value of the cumulative sum of a received scalar signal (reward)." (Sutton, 1998). This idea gave place to a formal definition of intelligent behavior as the result of the actions of an agent that is trying to maximize its long-term expected reward (usually in an unknown environment), without the supervision of an expert that tells it what actions to take but just being led by its own goals and purposes.

The reinforcement-learning formalism has proven incredibly productive in terms of the algorithms and applications it has enabled. In robotics, where it has had particular

appeal, RL applications include flying an RC helicopter (Ng et al., 2004), making it do aerobatics (Abbeel et al., 2007), making a quadruped robot avoid complicated obstacles (Lee et al., 2006) or enabling an AIBO robot to learn the fastest way to walk (Kohl & Stone, 2004). RL has also been successfully applied to game playing, starting with the now famous TD-Gammon (Tesauro, 1994), which played backgammon at the level of a human world champion, and nowadays being successful in the game of Go, where RLGo (Silver et al., 2007) is playing on a $9x9$ Go board at the level of the best human masters. Lately, RL is also making headways into commercial video games, where human players confront computer agents that use RL to adapt to their opponents (Graepel et al., 2004; Merrick & Maher, 2006). Other successful applications include autonomic computing (Tesauro, 2005), elevator control (Crites & Barto, 1996), pricing of options (Tsitsiklis & Roy, 2000), the treatment of epilepsy (Guez et al., 2008), and optimization of memory controllers in the field of computer architectures (Ipek et al., 2008), among many others.

1.2 The Reinforcement Learning Problem Setting

The standard way of modeling reinforcement-learning problems is depicted in Figure 1.1 [1]. In the figure you can see an *agent* (the one-armed robot) interacting with an *environment* (the gray cloud). The interaction proceeds as follows: at time t, the environment is in state s_t, and the agent is provided with an observation i_t, which is a function of s_t. The agent's behavior function B then selects an action a and executes it. Executing this action leads the environment to transition to a new state s_{t+1}, at time $t+1$, according to its *state transition function* T, and produce a reward signal r (also a function of the state).

As stated before, the goal of the agent is to come up with a behavior B such that, for any given state, it picks the actions that maximize the long-term reward it receives from the environment. The learning problem deals with the situation in which the agent does not know the transition function T that induces state changes in the environment,

[1] The inclusion of the original Kaelbling et al. figure was suggested by Michael Littman as a way of starting a tradition. I hearby commit to encourage any future students to include this figure in their dissertations too.

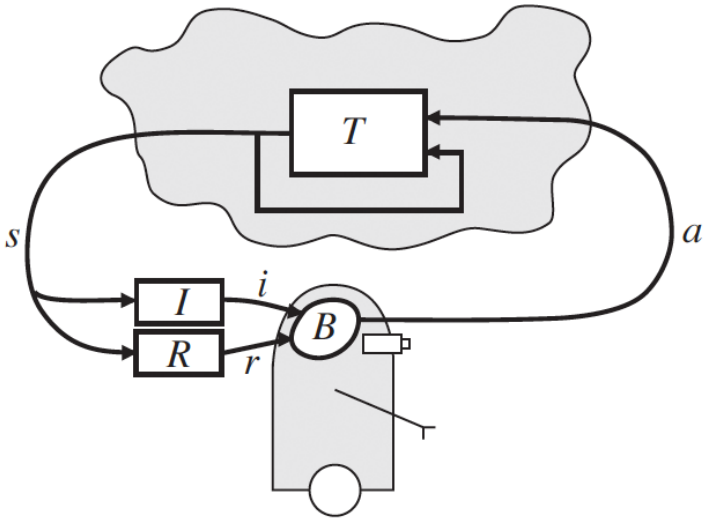


Figure 1.1: The standard RL model, as depicted in Kaelbling et al., 1996.

and/or it does not know the expected rewards to be attained from a given state.

Reinforcement-learning problems are commonly formalized as Markov Decision Processes (MDPs), which I will introduce in the next chapter. It is now worth noting that throughout this dissertation I will make some assumptions about the types of problems being considered:

- I only consider problems consisting of discrete state and action spaces, and assume that time advances in discrete steps. For an example of work that considers continuous time, states and actions, see Doya (2000).

- I assume there is a single agent interacting with the environment. Extensive work exists on multiagent reinforcement learning, for a survey see Shoham et al. (2003).

- I assume the state is fully observable. That is, the observation i perceived by the agent is a one-to-one correspondence with the state of the environment (in practical terms, $i = s$). For situations in which the state is only partially observable, and the agent needs to make inferences about it, see Kaelbling et al. (1998).

- I assume the environment is stationary, which means that state transitions and rewards do not depend on time. In contrast, in a non-stationary world the probability of transitioning from state s_i to state s_j given some action a could be

different depending on *when* the action a was performed. It is possible to render non-stationary worlds into stationary problems by considering time as part of the state space, making learning and generalization much more challenging.

- Finally, I assume environments follow the Markov property, meaning that the state observed by the agent is a sufficient statistic for the whole history of the environment. The agent's behavior can thus only depend on the currently observed state, and not on any past states.

These assumptions are common in the reinforcement-learning literature and, although not always realistic, they usually provide a reasonable approximation to many real-life problems. Although they certainly simplify the problem in a number of ways, many challenges still persist. Consider for example the games of Go or Chess, which follow all of the above assumptions and remain extremely challenging, unsolved problems in AI. In these games, states and actions are discrete and finite, and time progresses in discrete *moves*. The environment is fully-observable, stationary and Markovian: at any given time, the board is observed by the player and contains all the information needed to make a move, the result of which is independent of the point in time in which it is made.

1.3 Challenges in Reinforcement Learning

In reinforcement-learning problems an agent tries to maximize its long-term reward in an unknown environment (that is, in an environment whose transition and/or reward functions are unknown). In this section, I describe some commonly recognized challenges in reinforcement learning.

1.3.1 Exploration

Imagine you have two unbalanced coins and you are offered the opportunity to make 20 tosses. For each toss you can use either one of the coins, and you get a monetary reward when the coin comes up heads. The problem is that you do not know the probability

of heads of each coin, so you need to experiment with them. After a few throws of each coin, you might start getting a feel for which one is better (has a higher probability of coming up heads). At what point do you stick with what you think is the best coin and just play that one? How much do you keep trying out what you think is the worse one, just to make sure? The example of the two-coins is commonly generalized to the case where there are k options, into what is known as the k-armed bandit problem, a very well-studied problem in RL and optimal decision-making (see Berry and Fristedt (1985) for one of the first formal treatments of the problem, and Auer et al. (2002) for modern approaches).

As in the coin example, an autonomous agent confronted with an unknown environment needs to *explore* to gain the necessary knowledge that will enable it to maximize its long-term reward. Note that in the general case of sequential decision-making, an agent's actions affect future states and decisions can have long-term consequences that are potentially much more complicated than in the simple k-armed bandit case. Exploring requires the agent to take potentially suboptimal actions, with the hope of gaining new information that will help it obtain higher rewards in the future. A purely exploring agent will spend its life traveling into unknown parts of its environment, and never enjoying the fruits of its accumulated experience. A purely exploiting agent will jump to conclusions, assume there's nothing else out there to see and greedily gather whatever reward it can based on its limited experience. A smart agent will have a strategy that balances the two needs, facing what is known in RL as the *exploration/exploitation dilemma* (Sutton & Barto, 1998).

Bayesian RL

One approach to smart exploration is to start with a Bayesian prior over the parameters of an environment that appropriately model the agents' beliefs about the world. In the two-coin example, an agent could initially assume both coins have head/tail probabilities that were sampled from a uniform Beta(1,1) distribution. With each experience garnered (each toss of one of the coins), the agent updates the posterior distribution over

these parameters. An action can now be chosen that takes into account this posterior distribution, which accounts for both the agent's knowledge and its uncertainty.

Bayesian approaches to RL provide a natural way of tackling the exploration/exploitation dilemma by taking into account the posterior, and can be shown to resolve it optimally (Duff, 2003). An initial problem with these approaches is that they require specifying a prior. In the general reinforcement-learning problem, this could mean setting a prior distribution over transition and reward functions, which might be unfeasible in many real-life applications. Another problem is that computing the posterior distribution can be very expensive computationally (intractable in the general case), and completely impractical in environments with a large number of states. For a more complete treatment of Bayesian RL and some recent advances see Strens (2000) and Poupart et al. (2006).

Almost Solving the Problem, Most of the Time

Bayesian approaches to the exploration/exploitation dilemma can be optimal but intractable. An alternative framework provides a new definition of optimality, essentially what we can call *near-optimality with high probability*. This framework is called *Probably Approximately Correct (PAC)* and was introduced by Valiant (1984) in the context of supervised learning. Informally, PAC requires learning a hypothesis with approximate precision, and with high probability. That is, there is a small probability of not learning it, and even when learned, it is only approximated to some degree of accuracy.

The PAC framework has been adapted to the reinforcement-learning context, into a framework called PAC-MDP (Kakade, 2003). This framework has enabled the design of a host of algorithms that can now be proved to be efficient in the PAC-MDP sense. From now on, I will talk about provably efficient algorithms meaning that the algorithm is efficient in the PAC-MDP framework. I will provide formal definitions of PAC and PAC-MDP in Chapter 2.

Optimism in the Face of Uncertainty

One of the most productive approaches to the exploration/exploitation dilemma was introduced by Kearns and Singh (2002), and further studied by Brafman and Tennenholtz (2002). Both approaches are efficient in the PAC framework described above. The idea behind these approaches and the formalization they provided gave birth to a whole family of algorithms known as the $R_{\max}$ family, based on the concept of *optimism in the face of uncertainty*. The main idea is that any unknown parts of an environment are optimistically regarded as maximally rewarding. The agent will thus be naturally drawn toward these unknown states, unless they are so hard to reach that it is not worth it (that is, the cost of reaching unknown states compared to the reward they can yield makes them unworthy). Many algorithms have been proposed that follow this model of exploration (Kearns & Koller, 1999; Guestrin et al., 2002; Strehl, 2007), and a recently introduced learning framework, KWIK, theoretically unifies all these approaches (Li et al., 2008; Li, 2009). In Section 2.6 I will introduce KWIK in more detail, and it will be the basic framework underlying most of the algorithms presented in this dissertation.

1.3.2 The Curse of Dimensionality

I now introduce another major challenge in reinforcement learning. Imagine you want to implement a reinforcement-learning algorithm to get a robot to learn how to navigate a room. To make this problem feasible, you decide to break down the room into small square sections, and implement a mechanism for the robot to identify in which section it is standing. The robot can now be in a finite and relatively small number of different *states*: one for each section of the room it is navigating. After a while, you realize the robot also needs to consider the lighting conditions of the room, so you add a new variable, which has a value of 0 if the room is dark, 1 if there is light. The size of the state space just doubled! The robot now needs to decide what to do at each of the sections of the room under two different circumstances: when the light is on, and when the light is off.

The exponential explosion in the number of states as a function of the number of state variables used to represent a problem is known as the *curse of dimensionality*, a term originally coined by Bellman (1957). Reinforcement learning researchers have been combatting this curse since the field was created, and it represents probably the major challenge when trying to design algorithms that scale up to large problems. In all of the success stories in RL mentioned before, some form of approximation or generalization is used to represent the problem, which would otherwise be simply too big to be represented exactly. Approximation methods achieve computational efficiency at the expense of optimality. Generalization methods seek to reduce the size of the problem by projecting knowledge about certain already-explored states into unexplored ones.

State aggregation

One form of generalization is state aggregation. In our previous example, let us imagine we now equip our robot with a night vision system that renders the state of the lights irrelevant for the task at hand. We could now treat the states in which the robot is in a certain region with or without the lights on as equivalent, ignoring the lighting variable. At the same time, we had already made an assumption for our robot: when discretizing its possible locations into a small number of sections, we assumed that different physical states within each section could be treated as the same state.

State aggregation is one way of dealing with large state spaces: irrelevant features are ignored, and different ground states are aggregated into a smaller number of abstract states, therefore making the problem smaller (Singh et al., 1995; Jong & Stone, 2005). State abstraction necessarily loses information, but preferentially retains what is needed for making near-optimal decisions. Dean and Givan (1997) define a notion of redundancy, where two states are considered equivalent if the effect of available actions are the same from both of them, and develop methods for automatic minimization of domain models. This idea has been further extended to model minimization based on exploiting symmetries in domain models, with or without information loss (Ravindran & Barto, 2001; Ravindran, 2004; Taylor et al., 2008).

Some formal definitions of types of state abstraction and their qualitative properties can be found in Li et al. (2006). A major challenge that remains is applying the right state aggregation to a given problem, perhaps by properly encoding expert knowledge or by designing algorithms that learn to generalize correctly from experience.

1.4 A Change of Representation

In this section I argue that, when modeling many real-life problems, existing reinforcement-learning representations of state simply make problems harder in an unnatural way. Take the example of the robot navigating a room introduced to illustrate the curse of dimensionality problem. Instead of discretizing the robot's room into sections and trying to learn transition dynamics from each section, we could think of the problem as consisting of variables indicating the robot's location and orientation. A learner under such representation would only need to acquire knowledge about how actions alter these variables. For example, it could learn that a *forward* action alters the robot's location relative to its orientation, and that the effect of this action is independent from the absolute value of the location parameters.

Leffler et al. (2007) introduced a representation, called Relocatable Action Models (RAM), where states are clustered based on the outcomes of actions when executed from them, and action dynamics are learned by cluster. In the robot navigation task, imagine the room has two different types of terrain: slippery and carpeted. RAM models would cluster each state according to its terrain type, and would learn relative action outcomes for each terrain.

Consider, however, a further complication: the outcome of actions in the robot domain depends not only on the terrain, but also on whether or not there are walls the robot bumps against, or other obstacles on the robot's way. RAM models could cluster the state space further, into four clusters resulting from the combination of slippery/carpeted states that are near/far away from a wall. A more natural representation would instead define action dynamics in terms of relations between the robot and walls, or the robot and different kinds of obstacles.

1.4.1 Objects and the Physical World

Recent work in developmental and cognitive psychology (for a review see Baillargeon et al. (2008)) suggests that infants even a few months old already have a notion of *objects* and possess a rich model of their physical behavior. Experiments show that infants can recognize and distinguish whether an object is inert or self-propelled, and expectations about their behavior and interaction are affected by this distinction (Luo et al., 2009). It has further been suggested that objects might constitute the early units of attention in the visual system (Scholl, 2001).

The exact nature of the concept of object in infants, the exact ages at which different aspects of their physical properties are acquired, and what the innate components are are areas of contention and debate. However, regardless of their different positions in these matters, researchers share the underlying assumption that objects play a key role in our understanding of the dynamics of the physical world. This assumption is at the core of the present dissertation.

The initial insight for this work is that existing state representations in RL fail to model objects and their interactions appropriately. In fact, these representations usually obscure the fact that a given represented environment is composed of objects in a real or simulated physical domain, and so hinder the agents' ability to learn efficiently. In Chapter 3 I present an example of how different representation in RL dramatically affect learning efficiency.

The introduction of an object-oriented representation for RL problems, which I call OO-MDP, and a set of learning algorithms that takes advantage of it constitutes an attempt to bridge the gap between what cognitive psychologists have observed about a very basic bias of human cognition, and the kind of biases that as RL practitioners we have been willing to build into our agents.

1.4.2 Relational Reinforcement Learning

A sub-field of Reinforcement Learning, called Relational Reinforcement Learning (RRL), deals with a generalization of the MDP framework to first-order domains. Such

domains are those specified through the use of logic predicates that include quantifiers over sets of variables (for example, set of states that follow certain properties). Representations or statements that express properties or reason about groups of states are called *lifted*, and when a lifted statement is applied to a set of concrete objects it is said to be *grounded*.

Representations in RRL try to bring the power and expressiveness of first-order logic (FOL) to reinforcement learning. Multiple formalisms have been proposed that limit the kind of FOL expressions that are allowed in an attempt to make representations efficiently learnable (Džeroski et al., 2001; van Otterlo, 2005; van Otterlo, 2008). One common representation is the First-Order Markov Decision Process, or FOMDP, but to this day there is no provably efficient FOMDP learner that does not rely on significant prior knowledge provided as input.

One representation that bears a number of resemblances with OO-MDPs is that of Relational MDPs (RMDPs), introduced by Guestrin et al. (2003). The focus of RMDPs is on planning, and their main goal is to generalize behavior from a simple domain to new ones where more (or fewer) objects of known types appear, potentially establishing new relations among them. The example that appears in the RMDP paper is on the game Freecraft, a real-time strategy game where a player creates and controls a set of peasants, soldiers and other types of *units*, each with different skills (for example, resource gathering, structure building or fighting). The player faces different scenarios with different goals, like building certain structures or fighting enemy armies. RMDPs can represent relationships between units in a way that facilitates generalizing to different types and numbers of units in different scenarios. For example, an agent that learns an RMDP representation for a scenario where a fighter fights an enemy can easily re-plan for a new scenario where multiple fighters fight multiple enemies. Theoretical guarantees, in the form of a polynomial PAC bound, demonstrates that accurate planning is possible with a polynomial number of samples.

While the OO-MDP representations introduced in this dissertation have similarities with RMDPs, the focus of my work is on grounding the representation on knowledge

that can be directly extracted from propositional features of the environment. Features like objects' positions and other observable attributes constitute a representational bias that is natural and intuitive. Essentially, the OO-MDP representation is an attempt at bridging the gap between expressive power and efficient learning. In this dissertation, I will introduce the general representation and establish the assumptions that enable provably efficient learning and smart exploration.

1.5 Object-oriented Representations in Reinforcement Learning

As stated at the beginning of this chapter, this dissertation will support the statement that object-oriented representations are a natural way of representing state, in a large class of problems, to enable orders of magnitude faster learning. This representation tackles the challenges mentioned above by providing a generalization method that can drastically reduce the size of a problem, while enabling efficient exploration.

The main contributions of this work are:

- I introduce Object-Oriented Markov Decision Processes (OO-MDPs), an extension of the MDP formalism where the state is represented as a set of objects, each of which is composed of a set of features or attributes. Dynamics will be represented by *conditional effects*. Given a *condition* represented as a conjunction over object interactions and feature values, an *effect* occurs, represented as a change in one or more features of an object.

- I show that problems represented as specific subclasses of OO-MDPs can be learned efficiently by enabling smart exploration. I introduce learning algorithms and prove polynomial learning bounds.

- I present experimental results that show orders of magnitude faster learning compared with state-of-the-art algorithms that use standard representations not based on objects.

The rest of this dissertation is organized as follows:

- Chapter 2 presents background, formalizing some of the material just introduced and adding some necessary definitions.
- Chapter 3 demonstrates the role state representations play in learning. I will introduce many state-of-the-art learning algorithms on standard RL representations, and compare their performance on a well-know domain (the Taxi problem). I will present experimental results on how humans solve that same task.
- Chapter 4 presents the OO-MDP representation in detail. I will introduce the general representation, and then some sub-classes that are efficiently learnable.
- Chapter 5 shows how one particular sub-class of OO-MDPs, ones with deterministic action effects, can be learned efficiently under certain assumptions.
- Chapter 6 shows how a larger sub-class of OO-MDPs, with stochastic effects, can also be learned efficiently.
- Chapter 7 concludes and presents future challenges and potential extensions.

Chapter 2

Markov Decision Processes

> Sammy Jankis wrote himself endless notes. But he'd get mixed up. I've got a more graceful solution to the memory problem. I'm disciplined and organized. I use habit and routine to make my life possible.
>
> From the film *Memento* (2000)

Reinforcement Learning problems are often formalized as *Markov Decision Processes (MDPs)* (Puterman, 1994). In this chapter, I introduce the MDP formalism, the MDP learning and planning problems and the notion of sample and computational complexities. I then present the PAC, MB, PAC-MDP and KWIK learning frameworks.

2.1 Definitions

A Markov Decision Process is a five-tuple $\mathcal{M} = \langle \mathcal{S}, \mathsf{A}, T, \mathcal{R}, \gamma \rangle$, where:

- $\mathcal{S}$ is the set of states, which can be discrete or continuous. Unless otherwise noted, in this dissertation I will assume $\mathcal{S}$ is discrete and finite.
- $\mathcal{A}$ is the set of actions, which can also be discrete or continuous. Once again, I will assume a finite set of actions.
- $T : \mathcal{S} \times \mathcal{A} \to \mathcal{P}_{\mathcal{S}}$ is the *transition function*, with $\mathcal{P}_{\mathcal{S}}$ representing the set of probability distributions over $\mathcal{S}$. As notation, and assuming $\mathcal{S}$ and $\mathcal{A}$ to be discrete, I will use $T(s, a, s')$ to denote the probability of observing a transition to state $s' \in \mathcal{S}$ if action $a \in \mathcal{A}$ is executed from state $s \in \mathcal{S}$.
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathcal{P}_{\mathbb{R}}$ is the *reward function*, a distribution over the real numbers representing the amount of immediate reinforcement to be expected by executing

an action from a given state. I will use $R(s, a)$ to denote the expected value of the distribution $\mathcal{R}(s, a)$.

- $\gamma \in (0, 1]$ is the *discount factor*, and its meaning will become clear when I define value functions.

2.1.1 The Markov Property

Markov Decision Processes are used to model dynamical systems where the state $s_t \in \mathcal{S}$ at any given point in time t is a sufficient statistic for the global state of the system. That is, the history of the system before time t is irrelevant to the prediction of the next state $s_{t+1} \in \mathcal{S}$, given s_t and a_t. Notice that the transition function only depends on $\mathcal{S} \times \mathcal{A}$. A system where this property holds is called *Markovian*. This definition can be generalized to the case in which state s_{t+1} depends on the last k states, rather than just s_t, and we can say that the dynamical system is *order k Markovian*.

2.1.2 The Interaction Protocol

The interaction between an agent and an environment modeled as an MDP proceeds in discrete timesteps, for a possibly infinite amount of time, in the following way (see Figure 1.1):

1. The environment at time t is at state $s_t \in \mathcal{S}$. Under the MDP representation of the environment, we assume the agent is able to perceive this state directly, and we call the environment *fully observable*. Other representations assume the agent only perceives an observation derived from s_t, and we call these environments *partially observable*. This work assumes fully observable environments.

2. The agent picks an action $a_t \in \mathcal{A}$.

3. In response to this action, the environment randomly draws a new state s_{t+1} according to the distribution defined by $T(s_t, a_t)$. It also draws a reward r_t from the distribution $\mathcal{R}(s, a)$.

4. The clock (t) advances one timestep and the interaction is repeated.

2.2 Policies and Value Functions

The agent picks its actions according to a *policy*. Formally, a stationary policy is a function $\pi : \mathcal{S} \rightarrow \mathcal{A}$. That is, for any given state s, $\pi(s)$ returns an action a to be taken from s. Given a policy π and a state s, the return that the agent expects to obtain by following π from s can be defined by the following recursive function, known as the *value function*:

$$\begin{aligned} V^{\pi}(s) &= E_{\pi}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + ...|s_t = s] \\ &= E_{\pi}[\sum_{t=0}^{\infty} \gamma^t r_t | s_t = s] \\ &= E_{\pi}[r_t + \gamma V^{\pi}(s_{t+1})|s_t = s]. \end{aligned}$$

Using the transition and reward functions, it is possible to rewrite the previous equation in the form known as the Bellman equation (Bellman, 1957; Puterman, 1994):

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

The maximum value that can be attained by any policy is commonly written V^*, and can be defined as:

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

Similar to the value function V, it is common to define what is known as the state-action value function, or Q function, which represents the return expected from taking a given action a and then following policy π:

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

We can write the optimal state-action function as:

$$Q^*(s, a) = R(s, a) + \gamma \sum_{s'} [T(s, a, s') \max_{a'} Q^*(s', a')].$$

This definition lets us derive the *optimal policy*, π^*, as:

$$\pi^*(s) = \arg\max_a Q^*(s, a).$$

For formal derivations and proofs, see Puterman (1994).

2.3 Solving an MDP

A key insight, attributed to Bellman (1957), is that if we have access to an MDP model (that is, if we know the transition function T and reward function R), we can iteratively use the Bellman equations to compute the optimal value function and from it an optimal policy. This is an important sub-step in the reinforcement-learning problem, known as *planning*, or *solving an MDP*. Planning can be done exactly or approximately.

2.3.1 Exact Planning

The simplest method for exact planning in MDPs is called *value iteration* (Bellman, 1957), or VI for short. We will assume the transition and reward functions T and $\mathcal{R}$ of our MDP are given, and compute V^* iteratively starting from V_0, initialized arbitrarily (for example, $\forall s \in S, V_0(s) = 0$). The iterative step then updates V as follows:

$$V_{k+1}(s) \leftarrow \max_a [R(s, a) + \gamma \sum_{s'} T(s, a, s') V_k(s')].$$

It can be show that, in the limit, V_k converges to V^*. In practice, we can terminate the algorithm when the norm $\|V_{k+1} - V_k\|_\infty$ drops below some threshold. It can be shown that at such point V_{k+1} is arbitrarily close to V^* (Bellman, 1957).

For completeness and as a reference to the reader, another simple and common exact planning method is *policy iteration* and was first introduced by Howard (1960). Finally, it is also possible to express the optimal value function as the solution to a linear program (Hordijk & Kallenberg, 1979), which provides us with a proof that solving an MDP is a polynomial-time problem (Littman et al., 1995).

2.3.2 Approximate Planning and Sparse Sampling

The three exact planning methods mentioned above have their approximate counterparts, namely approximate value iteration (Munos, 2005), approximate policy iteration (Munos, 2003) and approximate linear programming (de Farias & Van Roy, 2003). All of these methods lead to the generation of policies that, albeit near-optimal, are *global*. That is, all these methods produce a value fuction that can be used to derive an action for every state.

A different approach to planning is to derive a policy only for a set of *local* states, the ones that are most likely reachable from a given current state. If an agent occupies state s_t, and a model of the MDP is available, a forward search algorithm (Russell & Norvig, 2003) can be used to look ahead a number of states and compute the best next action according to the samples taken. In the discounted reward setting it is possible to define a time-horizon T, after which the reward to be accrued approximates 0. To prove this, first let us define the *H-step expected discounted reward* as:

$$V_H^*(s) = E(\sum_{i=1}^{H} \gamma^{i-1} r_i | s, \pi^*),$$

where r_i is the reward obtained at the ith time step when executing optimal policy π^* starting from state s.

Let us also assume, without loss of generality, that rewards are bounded: $0 \leq r_i \leq 1$, and call R_∞ the infinite-sum of expected rewards.

Lemma 1. *Let $R_t(H)$ be an H-step discounted reward at time t:*

$$R_t(H) = \sum_{i=1}^{H} \gamma^{i-1} r_{t+i}.$$

Given any $\epsilon > 0$ and t, we can show that $0 \leq R_\infty - R_t(H) \leq \epsilon$ if

$$H \geq \frac{1}{1-\gamma} ln \frac{1}{\epsilon(1-\gamma)}.$$

Proof. Since all rewards were assumed to be greater than 0, it is clear that $R_\infty \geq R_t(H)$.

To show that $R_\infty - R_t(H) \leq \epsilon$, note that the inequality $\gamma \geq 1 + ln\ \gamma$ implies:

$$H \geq \frac{ln\frac{1}{\epsilon(1-\gamma)}}{ln\frac{1}{\gamma}} = log_\gamma \epsilon(1-\gamma).$$

Therefore, since rewards are also assumed to be ≤ 1, we can show that:

$$R_\infty - R_t(H) = \sum_{i=H}^{\infty} \gamma^i r_{t+i} \leq \sum_{i=H}^{\infty} \gamma^i = \frac{\gamma^H}{1-\gamma} \leq \frac{\epsilon(1-\gamma)}{1-\gamma} = \epsilon.$$

□

This means that it is sufficient to compute a plan by searching T steps ahead, instead of infinitely far into the future, as long as T is of order $O(1/(1-\gamma))$.

In deterministic MDPs, it is sometimes feasible to compute an exact local plan, simply by building a search tree of depth T and branching factor $|A|$ (using depth- or breadth- first search, for example). In stochastic MDPs, it becomes necessary to sample multiple trajectories and approximate the expected reward, the main idea behind the *sparse sampling* algorithm of Kearns et al. (2002).

Defining the 0-step value function as $V_0^*(s) = 0$, it is now possible to recursively re-write V_h^* as:

$$V_h^*(s) = R(s, \pi^*(s)) + \gamma \sum_{s' \in S} T(s, \pi^*(s), s') V_{h-1}^*(s').$$

What this equations tells us is that if it were possible to obtain an estimate of $V_{h-1}^*(s')$, we can inductively estimate $V_h^*(s)$. What sparse sampling does is approximate the expectation over all next-states s' in the previous equation by obtaining, for each action a, C samples from the generative model $T(s, a, s')$, resulting in a set of next-states S_a. Using these samples, the value function is estimated as follows:

$$\hat{V}_h^*(s) = \max_a [R(s,a) + \gamma \frac{1}{C} \sum_{s' \in S_a} \hat{V}_{h-1}^*(s')].$$

Note from this equation that the approximation error in sparse sampling is a function

of C and the lookahead depth h.

In all the experiments in this dissertation, planning will be either done exactly using value iteration, or by sparse sampling.

Instead of uniformly sampling from the model T, Kocsis and Szepesvári (2006) proposed using ideas from k-armed bandit algorithms to adaptively obtain more samples from the most promising actions. Their algorithm, called UCT, has achieved a significant success for planning in the game of Go (Gelly & Silver, 2007), and is pushing the limits of what is possible in RL planning in very large state spaces.

2.4 Learning

In planning, as in the previous section, it is assumed that the transition T and reward $\mathcal{R}$ functions are known. In a standard RL setting, this is not the case, and the agent needs to estimate T and $\mathcal{R}$ from experience. In such situations, two approaches are commonly taken: *model-free* and *model-based* learning (Atkeson & Santamaria, 1997). Model-free methods try to approximate the underlying value function directly (and are therefore sometimes called *direct* methods), and need no planning step (Sutton et al., 1992). Model-based or *indirect* methods try to first learn approximate models of T and $\mathcal{R}$ and use those models to estimate the value function (Sutton, 1991).

The proper functioning of model-based methods relies on the simulation lemma (Kearns & Singh, 2002), which establishes that if a sufficiently accurate estimate of T and $\mathcal{R}$ is available, then the (near-)optimal policy of the estimated MDP $\hat{M}$ is provably close to the (near-)optimal policy of the true MDP M. In the model-free case, theoretical results also guarantee the convergence of direct approximation methods to the optimal value function (Singh & Yee, 1994).

In Chapter 3, some canonical plus some state-of-the-art algorithms, both model-free and model-based, will be introduced in some detail.

2.5 Measuring performance

When analyzing reinforcement-learning algorithms, one important aspect is their convergence to an optimal policy. However, this alone is not enough: proofs of convergence *in the limit* do not tackle the issue of efficiency. We would like to know *when* our algorithms converge, and hope that this happens within a reasonable metric (for example, polynomial with respect to some relevant parameters). When measuring the performance of reinforcement-learning algorithms, three aspects need to be considered: computational costs, space and learning speed.

2.5.1 Computational complexity

The goal of a reinforcement-learning agent is to maximize long-term reward. That is, in the general case, we expect our algorithms to run forever! This means that by the usual definition of computational complexity, reinforcement-learning algorithms never halt and their total running time is infinite. For this reason, the computational complexity of reinforcement-learning algorithms is measured per step. That is, it is measured as the amount of computation performed between action choices.

When an agent observes a state and needs to choose an action, two types of computations are usually performed: learning from the just-acquired experience, and choosing the next action. The computation required for the learning part is highly dependent on the algorithm being used and the representation to be updated. The action-choice part commonly involves two operations that can be analyzed here, regardless of the algorithm being used: computing an estimate of the optimal value function, and choosing the best action given this estimate.

In model-based algorithms, the value function needs to be computed from the estimates for the reward and transition functions, using some of the planning methods described in Section 2.3. While the exact computational complexity of this step depends on the planning method, it is usually quite expensive. For example, the computational cost of Value Iteration is $O(\frac{|S|^2|A|^2}{1-\gamma})$ in the worst case.

In contrast, model-free algorithms, which are defined by their attempt to estimate

the value function directly and do not need a planning step, computational costs per step are very low. For example, if the state-action value function Q is represented as a table, most model-free algorithms that update a single value from each experience have an $O(1)$ computational cost.

The final step, choosing an action given a value function, only requires comparing among all available actions, an $O(|A|)$ operation (that could be improved depending on the representation of the value function).

2.5.2 Space complexity

Space complexity measures the amount of space required by an algorithm to maintain its data structures. An algorithm that stores the whole state-action value function as a table requires $O(|S||A|)$ space. Algorithms that use some sort of value-function approximation would require less space.

2.5.3 Learning complexity

Learning complexity measures the amount of experience needed by an algorithm before it can achieve a (near-)optimal policy. In order to learn about its environment, an agent needs to explore it, and in the process it will take a number of sub-optimal actions. This exploration cost has been formally defined by Kakade (2003), in what he calls the *sample complexity of exploration* (or just *sample complexity*). Two parameters are commonly defined when talking about sample complexity: a *precision* parameter $\epsilon \in (0, 1)$, and a *confidence* parameter $\delta \in (0, 1)$. The precision parameter controls how close to optimal we require the policy of our algorithm to be. The confidence parameter controls with what probability we want to achieve such performance.

Formally:

Definition 2. *(Sample Complexity) Given a reinforcement-learning algorithm A, A_t is the non-stationary policy executed by A at timestep t. For any given $\epsilon, \delta \in (0, 1)$, the sample complexity of A is the number of timesteps t such that $V^{A_t}(s_t) \leq V^*(s_t) - \epsilon$ with probability $1 - \delta$.*

PAC-MDP

Provided with a definition of sample complexity, the goal now is to define what we mean by an *efficient* learning algorithm. In Strehl et al. (2006), the definition of a Probably Approximately Correct (PAC) algorithm is extended to the MDP-learning case, as follows::

Definition 3. *(PAC-MDP) An algorithm A is PAC-MDP in an MDP M if, given $\epsilon, \delta \in (0,1)$, the sample complexity of A is bounded by a polynomial function in the quantities $1/\epsilon, 1/\delta, 1/(1-\gamma)$ and $|M|$ with probability $1-\delta$, where $|M|$ is some measure of the complexity of the MDP M. Typically, $|M|$ will be a polynomial in the number of parameters describing M, like the number of states $|\mathcal{S}|$ and actions $|\mathcal{A}|$ in the finite MDP case.*

We will say that a reinforcement-learning algorithm is sample efficient if it is PAC-MDP. A number of algorithms (Kearns & Singh, 2002; Kearns & Koller, 1999; Brafman & Tennenholtz, 2002; Strehl et al., 2007), some of which I will describe in Chapter 3, have PAC-MDP guarantees. Most of them base their exploration strategy on the notion of optimism in the face of uncertainty (c.f. 1.3.1). A recently introduced framework, KWIK, unifies these approaches (Li et al., 2008; Li, 2009), and I briefly introduce it in the next section.

2.6 The KWIK Framework

Sample efficient algorithms deal with the exploration/exploitation dilemma by keeping track of both their knowledge and their degrees of uncertainty. Smart exploration strategies use this information to guide the agent towards parts of the state space of high uncertainty, as long as these parts are easy-enough to reach. Otherwise, they will exploit the knowledge they have so far. Their PAC-MDP guarantees usually rely on the fact that, at all times, the agents are either following a near-optimal policy, or learning something new.

Li et al. (2008) introduced a supervised-learning framework called *Knows What It*

Knows, that unifies these approaches and provides explicit properties that are sufficient for an algorithm to be used in efficient exploration algorithms. The key idea is that if a class of MDPs can be KWIK-learned, then there exists a smart exploration approach that is PAC-MDP for that class (Li, 2009).

The problem setting for KWIK algorithms is as follows: there is an input set X and an output set Y. A hypothesis class H is a set of functions mapping inputs to outputs, $H \subset (X \to Y)$. The interaction then follows this protocol:

- The hypothesis class H, accuracy parameter ϵ, and confidence parameter δ are known to both the learner and the environment.
- The environment selects a target concept $h^* \in H$ *adversarially.*
- For *timestep* $t = 1, 2, 3, \ldots,$
  - The environment selects an input $x_t \in X$ in an *arbitrary* way and informs the learner. The target value $y_t = h^*(x_t)$ is unknown to the learner.
  - The learner predicts an output $\hat{y}_t \in Y \cup \{\perp\}$ where $\perp$ indicates that the learner is unable to make a good prediction of y_t. We call $\hat{y}_t$ *valid* if $\hat{y} \neq \perp$.
  - If $\hat{y} \neq \perp$, it should be accurate: $|\hat{y} - y| \leq \epsilon$, where $y = h^*(x)$.
  - If $\hat{y}_t = \perp$, the learner makes a stochastic observation $z_t \in Z = \{0, 1\}$ of the output y_t: $z_t = 1$ with probability y_t and 0 otherwise.

We say that H is *KWIK-learnable* if there exists an algorithm $\mathcal{A}$ with the following property: for any $0 < \epsilon, \delta < 1$, two requirements are satisfied with probability at least $1 - \delta$ in a whole run of $\mathcal{A}$ according to the KWIK protocol above:

1. (*Accuracy Requirement*) If $\hat{y}_t \neq \perp$, it must be ϵ-accurate: $|\hat{y}_t - y_t| < \epsilon$;
2. (*Sample Complexity Requirement*) The total number of $\perp$s predicted during the whole run, denoted $\zeta(\epsilon, \delta)$, is bounded by a function polynomial in $1/\epsilon$, $1/\delta$ and some measure of the size of the hypothesis class $\mathcal{H}$.

2.6.1 KWIK-RMax

As mentioned before, the key insight in the KWIK framework is that for any class of MDP whose structure can be KWIK-learned, it is easy to construct a PAC-MDP algorithm. Li (2009) defines a generic algorithm called KWIK-$R_{\max}$, that receives as input two KWIK learners (among other parameters): A_T and A_R, KWIK-learners for the transition and reward functions respectively. Given these sub-algorithms, KWIK-$R_{\max}$ guarantees PAC-MDP efficiency. Pseudo-code for KWIK-$R_{\max}$ is presented in Algorithm 1.

Algorithm 1: KWIK-$R_{\max}$

Input: $\mathcal{S}$, $\mathcal{A}$, γ, A_T, A_R, ϵ_T, ϵ_R, δ_T, δ_R, ϵ_P.
Initialize A_T with parameters ϵ_T and δ_T.
Initialize A_R with parameters ϵ_R and δ_R.
for all timesteps $t = 1, 2, 3, \ldots$ **do**
  Update empirical known state-action MDP $\hat{M} = \langle \mathcal{S}, \mathcal{A}, \hat{T}, \hat{R}, \gamma \rangle$:
  for all $(s, a) \in \mathcal{S} \times \mathcal{A}$ **do**
    if $A_T(s, a) = \perp$ or $A_R(s, a) = \perp$ **then**

$$\hat{T}(s, a, s') = \begin{cases} 1 & \text{if } s' = s \\ 0 & \text{otherwise} \end{cases}$$

      $\hat{R}(s, a) = r_{\max}$
    else
      $\hat{T}(s, a, s') = A_T(s, a)$
      $\hat{R}(s, a) = A_R(s, a)$
    end if
  end for
  Compute a near-optimal value function Q_t for $\hat{M}$.
  Observe current state s_t, greedily choose action $a_t = \text{argmax}_{a \in \mathcal{A}} Q_t(s, a)$, receive reward r_t and transition to s_{t+1}.
  if $A_T(s_t, a_t) = \perp$ **then**
    Provide A_T with the sample $\langle s_t, a_t, s_{t+1} \rangle$
  end if
  if $A_R(s_t, a_t) = \perp$ **then**
    Provide A_R with the sample $\langle s_t, a_t, r_t \rangle$
  end if
end for

2.6.2 Example: Learning a Finite MDP

Consider a finite MDP M, that is, one with discrete state and action sets. The transition function $T(s, a, s') = P(s'|s, a)$ is a multinomial distribution over next-states for each state and action pair. The problem of predicting a next-state in M can then broken down into predicting a next-state from each individual state-action pair (s, a) in M. That is, the hypothesis H over transition functions can be defined as a combination of hypotheses for each individual pair: $H = H_{s_1,a_1} \times \ldots \times H_{s_{|S|},a_{|A|}}$.

Li et al. (2008) first show that, if each hypothesis H_i is KWIK-learnable, then the combination is also KWIK-learnable. They call the algorithm for combining these learners *input-partition*. Then, they also show that multinomial distributions can be KWIK-learned, through an algorithm called *dice-learning*.

This shows that if we can KWIK-learn transition function with an algorithm A_T, we can provide it as input to KWIK-$R_{\max}$ and we obtain a PAC-MDP learner. In fact, what I just described is exactly the PAC-MDP algorithm $R_{\max}$ introduced by Brafman and Tennenholtz (2002), cast into the KWIK framework.

Other Supervised Learning Frameworks: PAC and Mistake-bound

I will mention two other supervised-learning frameworks, PAC and MB (mistake-bound), that are related to KWIK. After I introduce OO-MDPs, I will show how some aspects of the representation can be PAC- or MB- learned, which is not suitable for the reinforcement-learning setting, and it will be necessary to design suitable KWIK-learners.

The PAC framework (Valiant, 1984) for supervised learning assumes the learner is presented with independent and identically distributed (iid) samples of input-output pairs from the hypothesis to be learned. The learner has two parameters, ϵ and δ, and its goal is to infer the target function that maps inputs to outputs to ϵ-accuracy, with probability $(1 - \delta)$. Formally, if inputs follow a distribution D, the true hypothesis is

h^* and the learner's estimate is $\hat{h}$, PAC requires that, with probability $(1-\delta)$,

$$E_{x\sim D}[\mathcal{I}(\hat{h}(x) \neq h^*(x))] \leq \epsilon$$

The PAC learning framework is not directly suitable for reinforcement learning because of the iid assumption. An agent that is exploring observes samples of the type $\langle s_t, a_t, r_t, s_{t+1}\rangle$ in a non-iid fashion, as they depend on the exploration strategy being followed at time t.

The mistake bound framework (MB) (Littlestone, 1988), like KWIK, assumes inputs can be selected adversarially. The difference is that the learner is not allowed to refuse to make a prediction (by responding $\perp$), but rather is always required to predict something. In MB, only mistaken predictions are counted against the agent, and correct output labels are produced by the environment when a mistake has been made. A successful MB learner is expected to make only a small number of mistakes.

2.7 Summary

In this chapter, I introduced background that will be necessary for the rest of this dissertation. I presented the MDP formalism, how MDPs can be solved when the transition and reward functions are known, and how these functions can be estimated when unknown. The performance of reinforcement learning agents is measured in terms of computational, space and sample complexities. In the rest of this dissertation I will focus mainly on the problem of efficiently learning problems represented as OO-MDPs, so I introduced the KWIK and PAC-MDP frameworks, which will allow us to formally talk about efficiency.

Chapter 3

Learning Algorithms and the Role of Representations

> The computer is the first metamedium, and as such it has degrees of freedom for representation and expression never before encountered and as yet barely investigated.
>
> Alan Kay, computer scientist (1940-)

> Formal symbolic representation of qualitative entities is doomed to its rightful place of minor significance in a world where flowers and beautiful women abound.
>
> Albert Einstein (1879-1955)

In this chapter, I present a set of learning algorithms that illustrate the ways in which state representations, exploration and state aggregation impact learning. This presentation will provide the reader with an overview of the state-of-the-art and a motivation for the object-oriented representation that will be introduced in the next chapter. It will also establish some of the comparison metrics that justify the claim that object-oriented representations enable orders of magnitude faster learning. I start by introducing a common domain that will be used to compare the different algorithms, the Taxi problem of Dietterich (2000).

3.1 The Taxi Problem

Taxi is a grid-world domain (see Figure 3.1), where a *taxi* has the task of picking up a *passenger* in one of a pre-designated set of locations (identified in the figure by the letters *Y, G, R, B*) and dropping it off at a goal *destination*, also one of the pre-designed locations. The set of actions the agent can take are *North, South, East, West*,

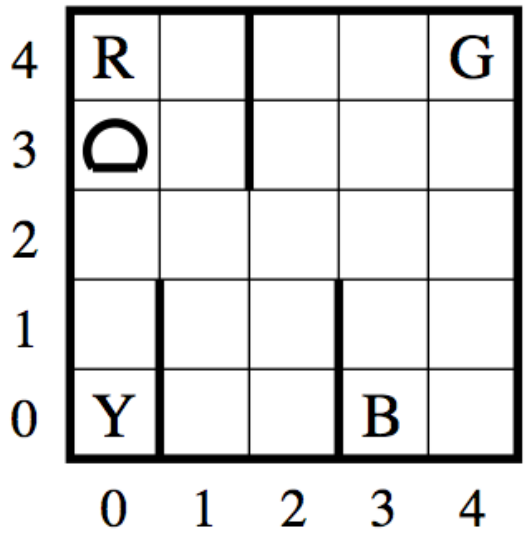


Figure 3.1: The original 5×5 Taxi problem.

Pickup and *Dropoff*. Walls in the grid limit the taxi's movements. In its standard form, the Taxi domain has 500 states: 5 x positions for the taxi, 5 y positions, 5 passenger locations (4 designated locations plus *in-taxi*) and 4 destinations.

The agent receives a reward of -1 for each navigation step it takes, -10 for trying to drop off or pick up the passenger at the wrong location, and 0 when it drops it off at the right location, at which moment the episode ends. The optimal policy in this domain is therefore to navigate the taxi to the passenger location using the shortest number of movement actions possible, execute a *Pickup* action, navigate to the destination and execute *Dropoff*.

Despite its simplicity (or because of it), the Taxi domain has been widely used in the RL literature since its introduction. Through the course of this research, I have come across more than 20 papers that contain experiments on the Taxi domain [1].

[1] A search for the query '`"reinforcement learning" taxi`' on Google Scholar on July 20th, 2009 yielded 304 results.

3.1.1 Taxi experimental setup

I will report on a number of different algorithms run on the Taxi problem. Each learning trial consists of running for 100 steps or until the agent reaches the goal of delivering the passenger to its desired location (whichever happens first). We call this trial an *episode*. At the end of each episode, the agent's learned policy thus far is evaluated on a set of six "probe" combinations of ⟨taxi (x,y) location, passenger location, passenger destination⟩. The probe states used were: $\langle(2,2),Y,R\rangle$, $\langle(2,2),Y,G\rangle$, $\langle(2,2),Y,B\rangle$, $\langle(2,2),R,B\rangle$, $\langle(0,4),Y,R\rangle$, $\langle(0,3),B,G\rangle$. If the policy of the agent is optimal for these 6 probes, we declare the problem solved and report the total number of steps taken to that point.

Experiments were repeated multiple times from random start locations and the results averaged. Unless otherwise noted, all experiments reported in this and subsequent chapters follow this experimental protocol, and I only report the values used for the particular parameters of each algorithm and the number of times experiments were repeated.

3.2 The Role of Models and Exploration

In this section I compare a model-free algorithm against a model-based one, both under a flat state-space representation. The example will illustrate a long-established empirical fact: that model-based approaches are more sample-efficient (Moore & Atkeson, 1993; Sutton, 1990). Plus, it will show how exploration impacts learning, by comparing a simple model-free exploration approach against KWIK-$R_{\max}$ style of smart exploration (cf 2.6.1).

3.2.1 Flat state representations

Under a *flat* representation, each state of the world is assigned a unique identifier with no particular meaning or structure. Indeed, the only purpose of this identifier is to serve as a hash code that lets an agent recognize whether it is visiting a state for the first time or it has visited it before. In the MDP formalism, flat simply means that

each element of $\mathcal{S}$ is identified by a unique number. In the Taxi domain, the flat states are $\{s_0 ... s_{499}\}$.

3.2.2 Model-free learning: Q-learning and ϵ-greedy exploration

Perhaps the main breakthrough in the early days of Reinforcement Learning was the development of *Q-learning* (Watkins, 1989; Watkins & Dayan, 1992), now the best known model-free algorithm for learning in flat state-space representations. Through a very simple update rule (Equation 3.1), Q-learning approximates Q^*, the optimal action-value function. The most important result at the time was that Q-learning converged to Q^* regardless of the policy being followed (Watkins, 1989). That is, an agent does not need to follow an optimal policy in order to learn the optimal action-value function. After an action a is taken from state s, and a new state s' and reward r are observed, the Q state-action value of (s, a) gets updated as follows:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a')]. \tag{3.1}$$

This simple update rule, the crux of the Q-learning algorithm, is a convex combination of the current value of $Q(s, a)$ and the estimated value due to the transition just observed. The degree to which Q-learning adapts to new experience or keeps its existing knowledge is expressed by the parameter α, known as the *learning rate*. The higher α is, the more importance is given to new experience. Therefore, at the beginning of the learning process, when Q values are still very inaccurate, it is desirable to have a large value of α. As the Q function converges, it is wise to have a low α and not give too much credit to new experience. A common adaptation of the standard Q-learning algorithm presented here is to cool down (gradually decay) α as time progresses. In stochastic domains, where the same action a from the same state s can lead to different outcomes, the role of α is to blend new outcomes in without completely overwriting prior experience. In deterministic domains, where the same outcome is always expected for any (s, a) pair, it is possible to simply set $\alpha = 1$.

Pseudo-code for Q-learning is presented in Algorithm 2.

Algorithm 2: Q-learning

Input: α, γ
Initialize $Q(s, a)$ according to initialization policy.
for all timestep $t = 1, 2, 3, \ldots$ **do**
  Observe current state s_t
  Choose action a_t according to exploration policy
  Execute action a_t, obtain reward r_t and observe new state s_{t+1}.
  Set α according to learning rate polict.
  Update Q: $Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a')]$
end for

Notice that in the pseudo-code presentation of Q-learning the *exploration policy* (that is, the policy for choosing the next action at each step) was left open. A very common exploration policy in Q-learning is called ϵ-greedy exploration: an extra parameter, $\epsilon \in (0, 1)$ is provided as input so that Q-learning takes a random action ϵ fraction of the time and a greedy action the remaining $1-\epsilon$ fraction. The greedy action is the one with the highest Q value from the current state s: $\text{argmax}_a Q(s, a)$.

A known improvement for guiding exploration in a smart way is to initialize the Q table to an optimistic value, usually the maximum possible value of any state action pair, $v_{\max}$. If the maximum possible immediate reward, $r_{\max}$, is known, it suffices to set all initial $Q_0(s, a)$ to $v_{\max} = r_{\max}/(1 - \gamma)$. This kind of initialization will lead Q-learning to consider unvisited state-action pairs as having high value, and it will consider as greedy actions those that will take the agent toward these states, a smarter type of exploration than just taking a random action ϵ percent of the times.

Q-learning results

I ran Q-learning on the Taxi problem using ϵ-greedy exploration, following the protocol described in Section 3.1.1. For each setting, the experiment was repeated 100 times from different random start states, and the number of steps until the optimal policy was reached averaged. Experiments were run both with and without optimistic initialization policy.

In the optimistic case, α was set to 1 and ϵ to 0. Since the maximum immediate reward in this domain is 0, $v_{\max} = 0/(1 - \gamma) = 0$, so all initial $Q(s, a)$ values are set to

zero. Under these settings, Q-learning reaches the termination criterion in an average of 29350 steps, with standard deviation 3930. The computation time per step, as expected in a model-free algorithm, was very low: less than 1ms on average.

Without optimistic initialization, a parameter search was conducted for the best empirical value of the exploration rate ϵ. Although Q-learning is theoretically guaranteed to converge, in practice it can take an extremely long time, especially if parameters are not set correctly.

After parameter search, the best value for ϵ was determined to be 0.6, and the learning rate α was set to 0.1. The problem was learned in an average of 106859 steps, with standard deviation 16474.

3.2.3 Model-based learning and KWIK-$R_{\max}$ exploration

As introduced in Chapter 2, another approach to solving an RL problem like Taxi is to estimate the transition and reward functions from experience, and use those estimates to plan a (near-)optimal solution using a planning algorithm like Value Iteration. This kind of approach is what we call model-based, or indirect learning (see Section 2.4). A state-of-the-art algorithm for model-based learning is $R_{\max}$, a provably efficient algorithm with an elegant approach to the exploration-exploitation dilemma (Brafman & Tennenholtz, 2002).

In a nutshell, $R_{\max}$ classifies state-action pairs into *known* and *unknown*. $R_{\max}$ keeps a count of how many times it has performed each action a from each state s, and builds an empirical distribution of transitions to next state s'. If it has observed enough transitions from a state s under action a, it uses the empirical model to plan. If it does not have enough observations, it assumes an optimistic transition to a fictitious state $s_{\max}$, with maximal value $v_{\max}$. The number of observations it has to experience to consider a transition to be known is given by parameter M, which has to be set appropriately. Note that in deterministic domains, where by definition there exists only one state s' for each state-action pair (s, a) such that $T(s, a, s') = 1$, setting $M = 1$ is enough to obtain a correct model.

The structure of the $R_{\max}$ algorithm follows that of KWIK-$R_{\max}$ (see Algorithm 1), and I only need to specify the learning algorithms for the transition and reward functions, $A_T = R^T_{\max}$ and $A_R = R^R_{\max}$, that KWIK-$R_{\max}$ expects as input. I break down these algorithms into two functions: Predict and AddExperience, corresponding to when they are queried in line 7 of KWIK-$R_{\max}$, and when they are provided with experience in lines 17 and 20. Predict is only presented for the transition function, as A_R is exactly the same structure, just returning $\hat{R}(s, a)$.

Algorithm 3: $R^T_{\max}$ AddExperience

Input: s, a, s'
Update transition count: $n(s, a, s') \leftarrow n(s, a, s') + 1$.
Increase count $n(s, a) \leftarrow n(s, a) + 1$.
Update empirical transition distribution: $\hat{T}(s, a, s') \leftarrow n(s, a, s')/n(s, a)$

Algorithm 4: $R^R_{\max}$ AddExperience

Input: s, a, r
Update empirical total reward $r(s, a) \leftarrow r(s, a) + r$.
Increase count $n(s, a) \leftarrow n(s, a) + 1$.
Update empirical reward distribution: $\hat{R}(s, a) \leftarrow \frac{r(s,a)}{n(s,a)}$

Algorithm 5: $R^T_{\max}$ Predict

Input: s, a
if $n(s, a) \geq M$ **then**
  return $\hat{T}(s, a, s')$
else
  return $\perp$
end if

The optimistic assumption that $R_{\max}$ makes about state-action pairs it has not yet experienced enough leads it to naturally want to reach those unknown states and take those actions, unless they are too hard to reach. That is, if it is the case that the cost of reaching those states is greater than the reward expected from the imaginary transition to $s_{\max}$, $R_{\max}$ will greedily exploit its current knowledge rather than try to reach them. This approach to the exploration-exploitation dilemma leads $R_{\max}$ to guarantee that, in all but a small number of steps, it is either taking a near-optimal action or it is learning something new. This property is at the center of the proof that $R_{\max}$ is guaranteed to

reach a near-optimal policy, with high probability, in a number of steps polynomial in $(1/\epsilon, 1/\gamma, |S|, |A|)$ (Brafman & Tennenholtz, 2002).

The $R_{\max}$ idea is related to optimistic initialization in Q-learning. However, in Q-learning optimistic initialization only allows the agent to choose an unknown part of the state that is immediately reachable from its current state (the greedy action), and plans to reach these unknown states only happen very slowly as optimistic values get propagated back to known states. The $R_{\max}$ scheme in a model-based approach lets the agent build much more complex plans to reach unknown parts of the state space. The experimental results in the next section show how significant this impact is.

3.2.4 Experimental results

$R_{\max}$ requires only one extra parameter, M, indicating how many times each action has to be taken from each state before the pair (s, a) is considered known. In the deterministic case, like Taxi, this parameter can be set to 1. Experiments were run on Taxi with this setting, repeated 20 times and averaged. $R_{\max}$ learns the task in 4151 steps.

Table 3.1 summarizes the experimental results under a flat state-space representation.

Domain knowledge	Algorithm	# of Steps	Time/step
$\|S\|$, $\|A\|$	Q-learning ($\epsilon = 0.6$)	**106859**	$<$ 1ms
$\|S\|$, $\|A\|$, $R_{\max}$	Q-learning - optimistic initialization ($\epsilon = 0$)	**29350**	$<$ 1ms
$\|S\|$, $\|A\|$, $R_{\max}$	$R_{\max}$	**4151**	74ms

Table 3.1: Summary of results for flat state-space representations.

Q-learning with no optimistic initialization serves as a baseline algorithm. By adding optimistic initialization, we are incorporating smarter exploration, which results in a factor of three improvement. $R_{\max}$ incorporates model-based learning, which makes better use of accumulated experience, and enables even smarter exploration, resulting in an additional factor of 7 improvement over the baseline algorithm.

3.3 The Role of State Representations

An environment's state can often be viewed as a combination of a set of state variables, or features. In the Taxi domain, the state can be represented as the following vector or 4-tuple: ⟨x location, y location, passenger location, passenger destination⟩. Such a representation is called a *factored-state* representation in the literature (Dean & Kanazawa, 1989; Boutilier & Dearden, 1994). Given a factored-state representation, it is possible to represent partial dependency relations between variables, opening the possibility to much more efficient learning. Consider navigation in the Taxi problem: the behavior of actions *North*, *South*, *East* and *West* depends on the taxi location, but is completely independent from the passenger's location or destination. Such dependencies are commonly represented as a *Dynamic Bayes Network* (DBN) (Boutilier et al., 1999), a particular type of graphical model with nodes representing state variables at time t and at time $t+1$, and edges representing dependencies. See Figure 3.2 for an example, within Taxi, of a DBN for action *North*, and Figure 3.3 for a DBN of action *East*.

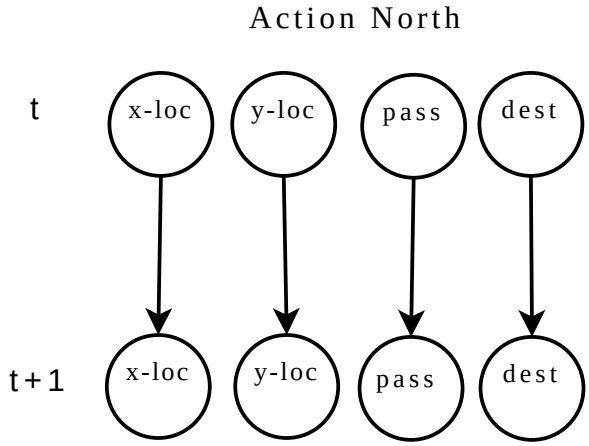


Figure 3.2: Dynamic Bayes Net for Action North.

In the case of action *North*, each variable at time $t+1$ simply depends on its value at time t. In particular, the variables *x-loc*, *passenger* and *destination* will not change under this action ($x\text{-}loc_t = x\text{-}loc_{t+1}$, etc). In the case of variable *y-loc*, the dynamics follow the rule shown in table 3.2.

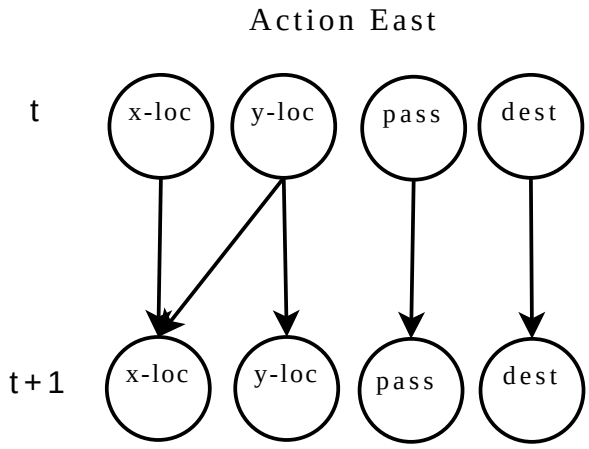


Figure 3.3: Dynamic Bayes Net for Action East.

Action *North*	
$y\text{-}loc_t$	$y\text{-}loc_{t+1}$
0	1
1	2
2	3
3	4
4	4

Table 3.2: Transition model under action North, for state variable *y-loc*.

On the other hand, observe that the variable *x-loc* under action *East* does depend on both *x-loc* and *y-loc*. This difference is explained by the existence of vertical walls in some locations of the grid (see Figure 3.1). Taking action *East* from location $(0, 1)$ will not move the taxi, whereas taking it from $(0, 2)$ will move it to $(1, 2)$. Table 3.3 shows the transition dynamics for action *East*, where the combinatorial explosion resulting from having to consider two state variables can be observed.

If the state s is composed of n state variables: $s = \langle v_1, v_2, \ldots, v_n \rangle$, the transition function can be factored into the product of transitions functions for each state variable:

$$T(s'|s, a) = \prod_{k=1}^{n} T_k(v'_k|s, a), \tag{3.2}$$

where transition functions T_k for each state variable depend on the variables that k depends upon, as indicated by the DBN. We will call this set of variables from which a state variable depends its *parents* $\mathcal{P}$. For example, for action *East*, the parents

Action *East*		
x-loc_t	y-loc_t	x-loc_{t+1}
0	0	0
0	1	0
0	2	1
0	3	1
0	4	1
1	0	2
1	1	2
1	2	2
1	3	1
1	4	1
2	0	2
2	1	2
2	2	3
2	3	3
2	4	3
3	0	4
3	1	4
3	2	4
3	3	4
3	4	4
4	0	4
4	1	4
4	2	4
4	3	4
4	4	4

Table 3.3: Transition model under action East, for state variable x-*loc*.

of variable x-*loc* are: $\mathcal{P}_{x\text{-}loc} = \{x\text{-}loc, y\text{-}loc\}$. The transition function can then be represented as (see Figure 3.3 for DBN representation):

$$T(s'|s,a) = \prod_{k=1}^{n} T_k(v'_k|\mathcal{P}_{v_k}, a), \tag{3.3}$$

which in the case of action *East* would be:

$$\begin{aligned} T(s'|s, East) = T_{x\text{-}loc}(x\text{-}loc|\langle x\text{-}loc, y\text{-}loc\rangle, East) \times \\ T_{y\text{-}loc}(y\text{-}loc|y\text{-}loc, East) \times \\ T_{pass}(pass|pass, East) \times \\ T_{dest}(dest|dest, East). \end{aligned}$$

Algorithms like $R_{\max}$ for provably experience-efficient exploration of MDPs have been generalized to factored-state MDPs with transition dynamics specified by DBNs. Two canonical examples are Factored E^3 (Kearns & Koller, 1999) and Factored $R_{\max}$ (Guestrin et al., 2002), both known to behave near optimally, with high probability, in all but a polynomial number of steps. These algorithms assume that a complete and correct DBN structure specification is provided as input.

Factored $R_{\max}$

Factored $R_{\max}$ follows the same general structure as $R_{\max}$, but instead of building a full transition or reward model, it builds for each action a and each state variable v_k, a small transition model T_k. Now instead of counting state-to-state transitions, Factored $R_{\max}$ counts transitions to state-variable values from the state variables they depend on.

Like $R_{\max}$, Factored $R_{\max}$ can be instantiated as KWIK-$R_{\max}$ by just specifying the prediction and learning components that KWIK-$R_{\max}$ expects as input. I will show these components in Algorithms 3.3 and 3.3, only for the transition function.

Algorithm 6: Factored $R^T_{\max}$ AddExperience

Input: s, a, s'
for all state variable v_k **do**
  Increase count $n(v_k, a) \leftarrow n(v_k, a) + 1$.
  Obtain next state-variable value from s': $v'_k \leftarrow s'[k]$
  Update transition count: $n(\mathcal{P}(v_k), a, v'_k) \leftarrow n(\mathcal{P}(v_k), a, v'_k) + 1$.
  Update empirical transition distribution:
  $\hat{T}_k(\mathcal{P}(v_k), a, v'_k) \leftarrow n(\mathcal{P}(v_k), a, v'_k)/n(v_k, a))$
end for

Algorithm 7: Factored $R^T_{\max}$ Predict

Input: s, a
if $\exists k : n(v_k, a) < M$ **then**
  return $\perp$
else
  $\hat{T}(s, a, s') \leftarrow \prod_{k=1}^{n} \hat{T}_k(\mathcal{P}_{v_k}, a, v'_k)$
end if
return $\hat{T}(s, a, s')$

One of the problems in Factored $R_{\max}$ is in the planning step (Line 15 in the KWIK-$R_{\max}$ algorithm). In order to plan, it must still compute a full state-action value function, with one value for each flat state s. This process is referred to as *blowing up* the state space, making Factored $R_{\max}$ computationally inefficient. An alternative to this blow up is to use a method of approximate planning (for example, Structured Value Iteration by Boutilier et al. (2000)) that loses the theoretical guarantees. In the experiments presented here, where the focus is on sample complexity, we opted for exact planning using regular value iteration and blowing up the state space (c.f. 2.5.1).

Another drawback of Factored $R_{\max}$ and other factored-state algorithms is the assumption that a DBN is provided as input. This assumption might be true in certain domains where the experimenter understands the dynamical dependencies among state variables, but it limits the generality of the approach. It is beyond the scope of this chapter, but it is worth noting that Chapter 6 introduces a learning setting called the k-Meteorologists, which can be used to relax this assumption. In Diuk et al. (2009), we introduce a new algorithm called Met-$R_{\max}$, which learns the structure of a DBN as well as its parameters. The only input required by Met-$R_{\max}$ is the maximum in-degree of the DBN, but not its full structure.

3.3.1 Experimental results

Table 3.4 extends the summary of Taxi results with experiments using Factored $R_{\max}$. The main tunable parameter in Factored $R_{\max}$ is M, the number of times a state-action pair needs to be observed before its transition model is considered known. Once again, since Taxi is a deterministic environment and therefore there exists only one next-state

s' for each (s, a) such that $T(s, a, s') = 1$, this parameter can be set to 1 and no tuning is necessary. The experiments were repeated 20 times, and average results are reported.

Domain knowledge	Algorithm	# of Steps	Time/step
$\|S\|$, $\|A\|$, $R_{\max}$	$R_{\max}$	**4151**	74ms
$R_{\max}$, DBN structure	Factored $R_{\max}$	**1676**	97.7ms

Table 3.4: Summary of results for flat state space representations.

Observe that an almost 2.5 times improvement in performance was achieved by changing the way state is represented and adding domain knowledge about state variable independences.

3.4 The Role of Task Decompositions

Many tasks that agents face are hierarchical in nature or, at the very least, can be intuitively broken down into a series of subtasks. In Taxi, it might be natural to think of the overall task as a composition of two subtasks: picking up the passenger and dropping it off. It would also be natural to expect these subtasks to share domain knowledge and skills, like navigating to any of the four specially designated locations. The notion of task decomposition has been explored by many RL researchers, even giving rise to a subarea of the discipline commonly referred to as Hierarchical RL. See Barto and Mahadevan (2003) or Diuk and Littman (2008) for reviews of this subarea.

MaxQ

The Taxi domain was introduced precisely as a test domain for a hierarchical task decomposition method and learning algorithm called MaxQ (Dietterich, 2000). In MaxQ, the global value function for the MDP is decomposed as an additive combination of smaller value functions, each for a different subtask. Figure 3.4 shows the tree hierarchy used to decompose Taxi in MaxQ.

An assumption in MaxQ is that MDPs contain final states, $\mathcal{F} \subset \mathcal{S}$, that when reached make the episode end. Given a task hierarchy, each subtask $1 \leq i \leq I$ can be viewed as a self-contained MDP with final states $\mathcal{F}_i$ and action set $\mathcal{A}_i$. Actions $j \in \mathcal{A}_i$

can be either the primitive actions of the MDP or any of the subtasks under i in the hierarchy. The root task $i = 1$ uses $\mathcal{F}_i = \mathcal{F}$, the final states of the actual MDP.

A *hierarchical policy* $\pi = \langle \pi_1, \ldots, \pi_I \rangle$ is a policy for each task i, $\pi_i : S \rightarrow A_i$. Policy π_i is considered locally optimal if it achieves maximum expected reward given subtask policies π_j for $j > i$. If local optimality holds for all tasks, the corresponding hierarchical policy is called *recursively optimal*.

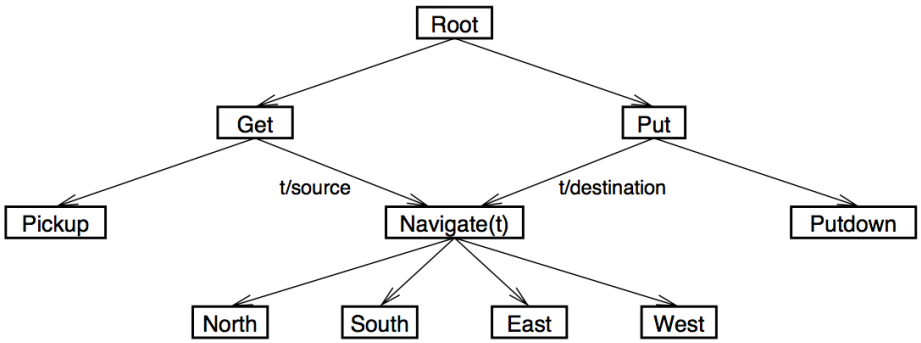


Figure 3.4: MaxQ Hierarchical Task Decomposition of the Taxi Problem.

Dietterich (2000) proposes an alternative construction of the value function V and state-action value function Q—the completion-function form. Intuitively, a completion function $C(i, s, a)$ returns the cost of completing subtask i after having completed task a from state s. In the Taxi problem, $C(\mathit{Dropoff}, s_0, \mathit{Pickup})$ indicates how much it would cost to dropoff the passenger after having picked him up from initial state s_0. Dietterich then proceeds to define a model-free learning algorithm, similar to Q-learning, for completion functions.

MaxQ is therefore a model-free approach, an extension of Q-learning. As shown in Table 3.5, MaxQ learns the Taxi task an order of magnitude faster than Q-learning with optimistic initialization. The main reasons for this are twofold. First, it takes advantage of the subtask abstractions to reduce the total number of state-action pairs it needs to learn. Second, it reduces the amount of exploration it has to do because the hierarchy limits the available actions at certain states. For example, during a navigation task, MaxQ will never try a *Pickup* or *Dropoff* actions that —by the policy space delimited by the hierarchy— will only be tried at any of the four special locations.

A Model-based Version of MaxQ

A logical question to ask at this point would be if it is possible to combine task decompositions with model-based learning and smart exploration. In Diuk et al. (2006), we introduced a model-based version of MaxQ, called DSHP, which does exactly that. It uses the same hierarchy and subtask abstractions as MaxQ but builds models for each sub-MDP. Building models allows for the use of $R_{\max}$-style exploration, dramatically improving performance and enabling a proof of its polynomial efficiency, both in learning and computational complexities. To this date and as far as I know, DSHP is the fastest learning algorithm on the Taxi problem (329 steps).

I ran MaxQ and DSHP on the Taxi problem. The input to both algorithms was the MaxQ hierarchy as shown in Figure 3.4 and the DBN representation for each subtask. Results for MaxQ and DSHP on Taxi are summarized below in Table 3.5.

Domain knowledge	Algorithm	# of Steps	Time/step
MaxQ hierarchy, DBN structure for each subtask, $R_{\max}$	MaxQ	**6298**	9.57ms
MaxQ hierarchy, DBN structure for each subtask, $R_{\max}$	DSHP	**329**	16.87

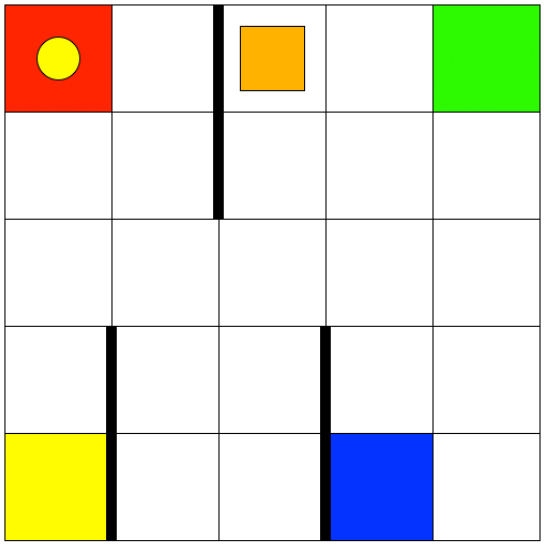
Table 3.5: Summary of results for flat state space representations.

Although DSHP used task decompositions to achieve the best result on the Taxi task, I claim that the type of prior knowledge introduced restricts its applicability. By providing the MaxQ task decomposition as input, the designer of the task is encoding a partial solution to the problem, imposing limitations on the number of policies to be considered. A desiderata that motivates this dissertation is to achieve comparable results with a more natural representation, one that more closely encodes the way people would approach a problem like Taxi. That is, the hope is to present state in a natural way and let the reinforcement-learning agent solve the problem, rather than restrict the solution space for it. In the next section, I look at what human participants do when tackling the Taxi problem.

3.5 What Would People Do?

To answer this question, I used an interactive version of the Taxi environment, where a person can actually play the role of the agent and submit actions through the computer keyboard. A laptop computer was setup at a shopping mall and participants were asked to play the game in exchange for a chocolate prize [2].

The screen that was presented to participants is shown in Figure 3.5. The four locations are indicated by color squares, and the location of the passenger is indicated by a circle in one of these locations. The taxi location is indicated by a yellow square. Note, however, that participants are not told any of this, they just observe the screen with the game. When the taxi successfully picks up a passenger, the circle shrinks, and from then on when the taxi moves, the circle moves along with it. The color of the circle indicates the desired destination.



Use the following keys to solve the game:
Up, Down, Left, Right, A and B

Figure 3.5: The Taxi Domain as Presented to Human Participants.

Participants play freely until they reach the end of an episode (when they drop the passenger off at the right location). They are then presented with a new episode, and they keep playing repeatedly until they play a full episode optimally, with no extra actions taken. This is considered the termination criterion for the task, and

[2]This experiment was run under the auspices of the Princeton University Department of Psychology, which once a semester set up a stand at Quakerbridge Mall and allows researchers to run experiments.

sample complexity is counted as the total number of steps taken in all the episodes. Instructions were minimal: participants were not told the goal of the game, just the keys they had to use and that completion of the task would be indicated once achieved.

A total of 34 people participated in the experiment. At the end, they were asked whether or not they considered themselves videogame players. Out of the 34 participants, 17 succeeded in solving the task, whereas the other 17 either quit, or had to be tipped off in order to solve it. Out of the 17 who did solve the task, 10 considered themselves videogamers and 7 did not. Out of the 17 who failed, only 4 were videogamers, and 13 were not.

In terms of performance, I measured the number of extra steps taken by participants (the non-optimal steps in all but the last, optimal episode). Videogamers learned the task in an average of 48.8 steps, whereas non-videogamers needed 101. The Boxplot on Figure 3.6 reflects the distribution of participants' performances, showing a clear advantage for self-defined videogamers. Using a single-tailed, 2-by-2 exact Fisher test, it is possible to reject the null hypothesis that videogamers are as good as non-videogamers ($p < 0.039$), in favor of the hypothesis that videogamers are better. It is expected that if we tested more subjects, the statistical power of this comparison would increase. This statistics seem to reveal a bias that humans bring to bear when confronted with this type of game, and which is more developed in video game players.

Further analysis of the exploration policies of the participants who solved the task are illuminating, albeit intuitive. First, all participants learned the rules of navigation after one step. They all started with an arrow press, observed that the yellow box (representing the taxi) moved according to the direction of the arrow, and explored no further. After this, all but two of the participants (who wandered around for a few extra steps), proceeded to move the taxi directly to one of the four distinct locations, saliently identified in bright colors, and only pressed the A or B keys (for Pickup and Dropoff) once the yellow box was in these locations. No participants seem to have tried to explicitly bump into a wall. Although it is impossible to tell with absolute certainty, there are only a handful of actions that move the taxi towards a wall that seem to be

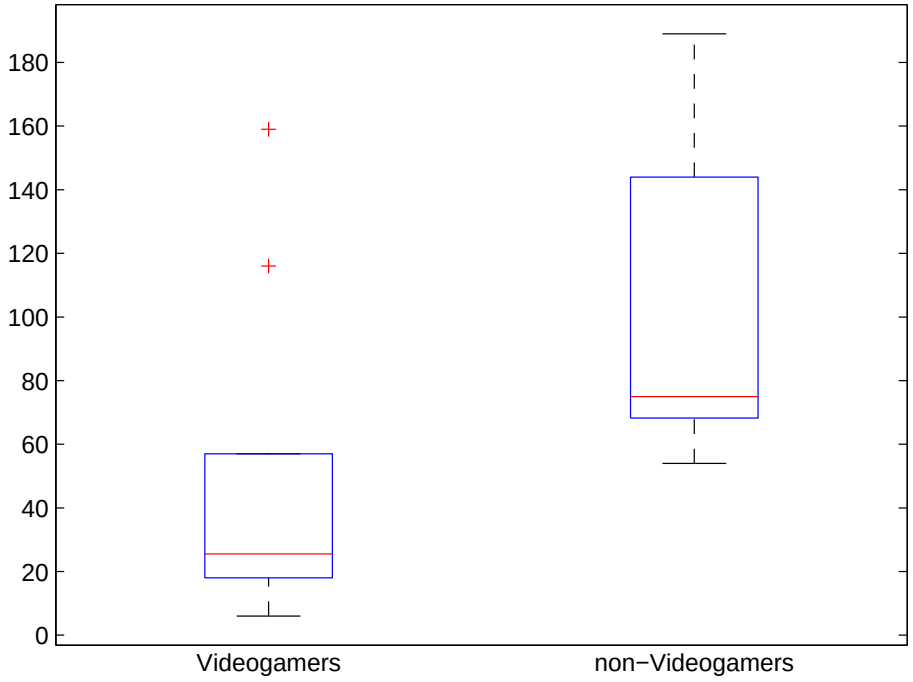


Figure 3.6: Number of extra steps taken before learning the task.

the effect of participants *overshooting*, just executing an extra action unintentionally, rather than actually trying to explore the outcome of such action.

From this experiment it is clear that humans come into the task with significant prior knowledge, although we can only speculate about its exact nature. However, analysis of their performance and exploration policies does indicate a clear representation of the fact that there is a taxi object that they control, the rules governing its movement, and a desire to explore the interactions between the taxi and other salient objects in the domain. Notice that humans who succeeded in completing the task did so 5 to 10 times faster than DSHP.

3.6 Object-oriented Representation

The examples presented in this chapter serve as a motivation for the object-oriented representation proposed in this dissertation, which will be formally and fully introduced

in the next chapter. We have seen how model-based learning improves sample efficiency over model-free methods. We have also seen the effect of smart exploration on learning speed, in particular using $R_{\max}$-style exploration. Factored representations and DBNs significantly reduce the number of states to be explored. Task decompositions have proved to be very useful too, but current methods require the designer to restrict the set of solutions considerably.

The question that motivated this dissertation is whether it is possible to find a representation that is natural for a significant number of problems, does not require unrealistic domain knowledge to be incorporated by the designer, and still provides all the advantages of model-based learning, smart exploration and state aggregation.

Consider the Taxi problem and how it was described: there is a taxi, a passenger, a set of distinguished locations, and walls that block the taxi's movements. The taxi needs to move to where the passenger is and pick him up, then move to the destination and drop him off. This description can be thought of as an enumeration of the objects in the domain, plus a set of interactions between those objects that need to occur, aligned with the kind of prior representation that people seem to bring into the task. This type of description is precisely the idea behind object-oriented representations.

Consider, for example, the model learned under factored-state representations. For action *North*, Factored-$R_{\max}$ had to learn all the elements of Table 3.2. The reader can look at that table and immediately observe a general rule: action *North* increases variable *y-loc* by 1, unless the taxi is hitting a wall. Learning what *North* does for each and every individual value of *y-loc* seems unnecessary if only we could learn the rules of taxi movements. In the case of action *East*, the rule is essentially the same: it increases the *x-loc* by 1 unless there is a wall. However, in the case of action *East*, a factored-state representation requires considering all combinations of values for *x-loc* and *y-loc*, a total of 25 independent values. Also note that an advantage of learning these general dynamical rules instead of each value location separately implies that if, for example, the size of the Taxi grid increases or more walls are added, nothing extra needs to be learned, the rules still apply.

Object-oriented representations allow domains to be described in terms of objects, and the dynamics of the system to be defined in terms of rules like the one suggested above. I will show in this dissertation that these rules can be learned efficiently, and present the appropriate algorithms. In order to complete the running example in this chapter, I ran one of those algorithms —DOO$R_{\max}$— on Taxi, and it learns in an average of 529 steps. A detailed description of the algorithm and how the Taxi experiments were run is presented in Chapter 4. DOO$R_{\max}$ still takes a little bit more than the best result so far, DSHP, which learns in 329 steps, but the solution space is not being restricted by the designer. The only prior knowledge that DOO$R_{\max}$ uses is a list of objects and a set of relationships between those objects that the agent should consider, which in the Taxi experiment included things like *'taxi is touching a wall from the North/South/East/West'*, or *'taxi is on top of passenger'*.

3.7 Summary and Discussion

To summarize, all the results for Taxi are presented again in Table 3.6.

Domain knowledge	**Algorithm**	**# Steps**	**Time/step**
$\|S\|$, $\|A\|$	Q-learning	**106859**	$<$ 1ms
$\|S\|$, $\|A\|$, $R_{\max}$	Q-learning - optimistic initialization	**29350**	$<$ 1ms
$\|S\|$, $\|A\|$, $R_{\max}$	$R_{\max}$	**4151**	74ms
$R_{\max}$, DBN structure	Factored $R_{\max}$	**1676**	97.7ms
MaxQ hierarchy, DBN structure for each subtask, $R_{\max}$	MaxQ	**6298**	9.57ms
MaxQ hierarchy, DBN structure for each subtask, $R_{\max}$	DSHP	**329**	16.87ms
Objects, relations to consider, $R_{\max}$	DOO$R_{\max}$	**529**	48.2ms
$\|A\|$, Figure 3.5	Humans (non-videogamers)	**101**	NA
$\|A\|$, Figure 3.5	Humans (videogamers)	**48.8**	NA

Table 3.6: Summary of all Taxi results presented in this chapter.

Chapter 4

Object-Oriented Markov Decision Processes

> It is only in the world of objects that we have time and space and selves.
>
> T.S. Eliot (1888-1965)

In this chapter, I introduce object-oriented representations, and extend the MDP formalism to what I call Object-Oriented Markov Decision Processes, or OO-MDPs. A few examples are introduced and formalized as OO-MDPs. I show how OO-MDPs differ from other representations, like Dynamic Bayes Networks, and relate this work to existing work in the sub-field of Relational Reinforcement Learning (RRL).

4.1 Pitfall: playing with objects

The introduction of the representation and formalism will be facilitated by including additional examples besides Taxi. I start with a videogame called *Pitfall*[1] for the Atari 2600 game console. The goal of the game is to have the main character (*Harry*) traverse a series of screens while collecting as many points as possible while avoiding obstacles (such as holes, water, logs, crocodiles and walls), which cause a loss of points or even death, under the time constraint of 20 minutes and 3 lives. All transitions in Pitfall, like in many video games, are essentially deterministic.

Figure 4.1 shows the initial screen of the game. Imagine you are first confronted with this screen: you have your Atari 2600 connected to your TV and a joystick in hand. What would you do? I hypothesize that you would first recognize that there are a series of objects on this screen: Harry, a Pit, a Log, a Ladder, a Wall. You would then move

[1] ©1982 Activision, Inc.

your joystick to see what happens, and would notice that Harry moves, and the way he moves depends on the joystick action you are taking. Probably your next step would be to make Harry interact with the other objects: make him jump the Pit, maybe climb the Ladder down, touch or jump the Log, etc. Of course, as humans, we bring so much prior information into these games that there are certain things we do not even try: the mere concept of "wallness" would likely lead us to the assumption that the Wall is impassable, and we would not even try making Harry bang himself against it. The idea behind object-oriented representations is not to build in all this knowledge, but to allow our agents to come into these situations with biases that I consider very elemental: the notion of object, spatial relations and object interactions. A fundamental assumption under this representation is that if, under two different states, the object relationships match, then the dynamics of the environment also match.

Figure 4.1: Initial screen of Pitfall.

The problem of object recognition is beyond the scope of this dissertation, so object-oriented representations will assume that detection of what the objects are in a domain is provided as input. I will now introduce OO-MDPs, and use both Pitfall and Taxi as examples.

4.2 The Propositional OO-MDP Formalism

For the reader familiar with the Relational RL literature (see 1.4.2), I would like to start by mentioning that some elements of this representation are similar to those of *relational MDPs*, or RMDPs (Guestrin et al., 2003), with significant differences in the way we represent transition dynamics. Similar to *RMDPs*, we define a set of *classes* $\mathcal{C} = \{C_1, \ldots, C_c\}$. Each class includes a set of *attributes* $Att(C) = \{C.a_1, \ldots, C.a_a\}$, and each attribute has a *domain* $Dom(C.a)$. A particular environment will consist of a set of *objects* $O = \{o_1, \ldots, o_o\}$, where each object is an instance of one class: $o \in C_i$. The state of an object *o.state* is a value assignment to all its attributes. The state of the underlying MDP is the union of the states of all its objects: $s = \bigcup_{i=1}^{o} o_i.state$. Notationally, when referring to a class, its name will be capitalized, and when speaking of an object, it will be in lower case. For example, *Passenger* refers to the class and *passenger* to an object of that class.

A possible OO-MDP representation of Taxi, the one I will use in experiments, has four classes: *Taxi*, *Passenger*, *Destination* and *Wall*. *Taxi*, *Passenger* and *Destination* have attributes x and y, which define their location in the grid. *Passenger* also has a Boolean attribute *in-taxi*, which specifies whether the passenger is inside the taxi. *Walls* have x and y attributes indicating a cell in the grid, plus an attribute that indicates their position with respect to that cell (above, below, left or right). The Taxi domain, in its standard 5×5 version, has one object of each class *Taxi*, *Passenger*, and *Destination*, and multiple (26) objects of class *Wall*. This list of objects points out a significant feature of the OO-MDP representation. Whereas, in the classical MDP model, the effect of encountering walls is felt as a property of specific locations in the grid, the OO-MDP view is that wall interactions are a property of object classes and therefore are the same regardless of their location. As such, agents' experience can transfer gracefully throughout the state space. Also, notice that while in the original Taxi domain there are four distinctive locations where the passenger can be or might want to go to, the OO-MDP representation is more general: passengers can be anywhere on the grid and choose any arbitrary destination. In experiments, in order to be able to

compare algorithms, I still use the pre-designated locations as passenger and destination coordinates.

In the Pitfall case, I will just mention the classes needed to describe the first screen: *Harry*, *Pit*, *Ladder*, *Wall*, *Log* and *Tree*. All objects have the attributes x_1, y_1, x_2 and y_2, which constitute the object's bounding box. In my experimental setup, I used a simple object-detection mechanism, designed by Andre Cohen for our joint paper on OO-MDPs (Diuk et al., 2008), that identifies the objects on the screen and builds a bounding box around them. The attributes mentioned above are the attributes that define those bounding boxes (see Figure 4.6) . *Harry* also has a Boolean attribute of *direction* that specifies which way he is facing.

Actions in OO-MDPs can be parameterized by a set of objects, or be global. If an action $a(o)$ is parameterized, it is considered to apply to the object o. Imagine a Taxi domain with two taxis, t_1 and t_2, where we would like to be able to move them independently. In such a domain, all actions would receive one or the other taxi as a parameter: $North(t_1)$, $Pickup(t_2)$, etc.

4.2.1 Relational transition rules

Before moving on to more definitions, I introduce Figure 4.2, a very quick summary of the flow of OO-MDP dynamics the reader might want to refer back to while reading this and upcoming sections.

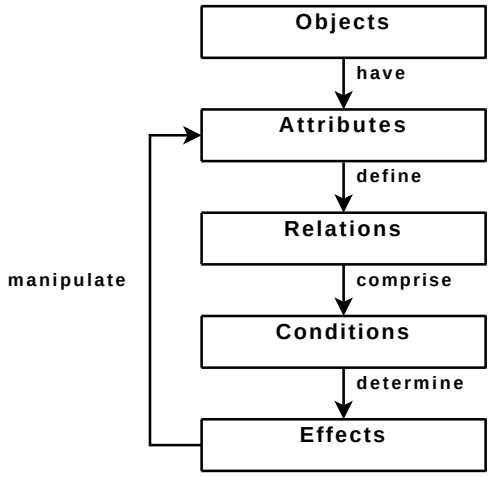


Figure 4.2: Summary of OO-MDP transition flow.

When two objects interact, they define a *relation* between them (they touch, one stands on top or inside the other, etc). As we shall see, relations are the basis for determining behavioral effects—a change in value of one or multiple attributes in either or both interacting objects. These behaviors are defined at the class level, meaning that different objects that are instances of the same class behave in the same way when interacting with other objects. Formally:

Definition 4. *A* relation $r : C_i \times C_j \rightarrow Boolean$ *is a function, defined at the class level, over the combined attributes of objects of classes* C_i *and* C_j*. Its value gets defined when instantiated by two objects* $o_1 \in C_i$ *and* $o_2 \in C_j$*.*

The focus and idea behind OO-MDP representations is on defining simple relations that can be easily derived directly from object attributes. For my Taxi representation, I define 5 types of relations between the Taxi and the other objects: $touch_N(Taxi, o)$, $touch_S(Taxi, o)$, $touch_E(Taxi, o)$, $touch_W(Taxi, o)$ and $on(Taxi, o)$, which define whether an object $o \in \{Passenger, Destination, Wall\}$ is exactly one cell North, South, East or West of a Taxi object, or if both objects are overlapping (same x, y coordinates).

In Pitfall, similar types of relations are used between Harry and all other objects ($touch_{N/S/E/W}(Harry, o)$), but I extended them to also describe diagonal relations, including: $touch_{NE}(Harry, o)$, $touch_{NW}(Harry, o)$, $touch_{SW}(Harry, o)$ and $touch_{SE}(Harry, o)$. These relations were needed to properly capture the effects of moving on and off of ladders.

Every state s in an OO-MDP is a value assignment to all attributes of all objects in the domain. An assumption of OO-MDP representations is that there is a function $pred(s)$ that, given a state, returns all relations being established in that state. For example, if provided with the state s depicted in Figure 4.3, $pred(s)$ returns the relations listed in Table 4.1.

The set of classes and objects that compose an environment, as well as the set of relations to be considered, are a form of background knowledge and, in this work, are assumed to be provided by the agent designer. Given a set of classes and relations,

it is also assumed that the agent designer provides a way of computing the output of $pred(s)$. As an example, in Pitfall an object recognizer was put in place between the actual game and the reinforcement-learning agent. It was the task of this recognizer to identify the relevant objects and compute the value of $pred(s)$.

$touch_N(taxi, wall)$	$\neg touch_S(taxi, wall)$
$\neg touch_E(taxi, wall)$	$touch_W(taxi, wall)$
$\neg touch_{N/S/E/W}(taxi, passenger)$	$\neg touch_{N/S/E/W}(taxi, destination)$
$\neg on(taxi, passenger)$	$\neg on(taxi, destination)$

Table 4.1: Relations induced by the state depicted in Figure 4.3

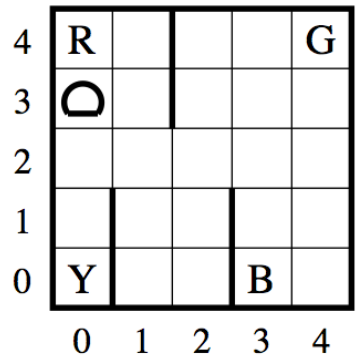


Figure 4.3: Example Taxi state, with passenger in location $(0, 3)$.

Transitions are determined by relations, which are established by interactions between objects. Every pair of objects $o_1 \in C_i$ and $o_2 \in C_j$, their internal states $o_1.state$ and $o_2.state$, an action a, and the set of relations $r(o_1, o_2)$ that are true—or false—at the current state, determine an *effect*—a change of value in some of the objects' attributes.

For example, when the object $taxi_i \in Taxi$ is on the northern edge of the grid and tries to perform a *North* action, it hits some object $wall_j \in Wall$ and the observed behavior is that it does not move. We say that a $touch_N(taxi_i, wall_j)$ relation has been established and the effect of an action *North* under that condition is *no-change*. On the other hand, if $\neg touch_N(taxi_i, wall_j)$ is true and the taxi performs the action *North*, the effect will be $taxi_i.y \leftarrow taxi_i.y + 1$. As stated before, these behaviors are defined at the class level, so we can refer in general to the relation $touch_N(Taxi, Wall)$ as producing the same kind of effects on any instance of $taxi_i \in Taxi$ and $wall_j \in Wall$.

Similarly, in Pitfall, we will define and learn rules like: if action is *StickRight* and condition is $\neg touch_E(Harry, Wall) \wedge \neg touch_E(Harry, Pit) \wedge \neg touch_E(Harry, Log) \wedge \neg on(Harry, Ladder)$, then the effect is $Harry.x \leftarrow Harry.x + 8$. We can formalize this notion in a definition:

Definition 5. *An* effect *is a single operation over a single attribute att in the OO-MDP. We will group effects into* types*, based on the kind of operation they perform. Examples of types are* arithmetic *(increment att by* 1*, subtract* 2 *from att), and* constant assignment *(set att to* 0*). Note that actions might affect multiple attributes at the same time, meaning that they can produce multiple effects in a single step.*

Definition 6. *A* term *t is any Boolean function. In our OO-MDP representation, we will consider terms representing either a relation between two objects, a certain possible value of an attribute of any of the objects or, more generally, any Boolean function defined over the state space that encodes prior knowledge. All transition dynamics in an OO-MDP are determined by the different possible settings of a set of terms T.*

Definition 7. *A* condition *is a conjunction T_c of terms and negations of terms from T that* must *be true in order to produce a particular effect e under a given action a.*

When determining if a condition holds at a given state, the environment has to search for a grounding of the relation's elements, which are classes, into objects of the corresponding class. A relation $r(C_i, C_j)$ can be interpreted as a logic statement over objects $\exists o_i \in C_i, o_j \in C_j : r(o_i, o_j)$. That is, the relation will be considered to hold if it is being established by *any* objects of classes C_i and C_j. In the case of conditions attached to parameterized actions, it is allowable to have relations that refer specifically to action parameters. For example, an action $a(o_p)$ can have a condition that includes relations like $r(o_p, C_i)$. This situation would translate into an expression $\exists o_i \in C_i : r(o_p, o_i)$.

As a simple example, consider the domain depicted in Figure 4.4, consisting of 3 objects of class *Boat* ($boat_1$, $boat_2$, $boat_3$) and 1 object of class *Island* (*island*). An action *WindBlowEast* could be specified at the class level by the following condition-effect (among others): $near_E(Boat, Island) \Rightarrow Boat.aground = True$. When the action

is executed, the relation *near* is translated into $\exists o_i \in Boat, o_j \in Island : near_E(o_i, o_j)$, which becomes true under the grounding $\langle o_i \equiv boat_1, o_j \equiv island \rangle$. While the action *WindBlowEast* is unparameterized, there could be another action *SailEast(o)*, $o \in Boat$ that only applies to the object o, and is used to move boats individually.

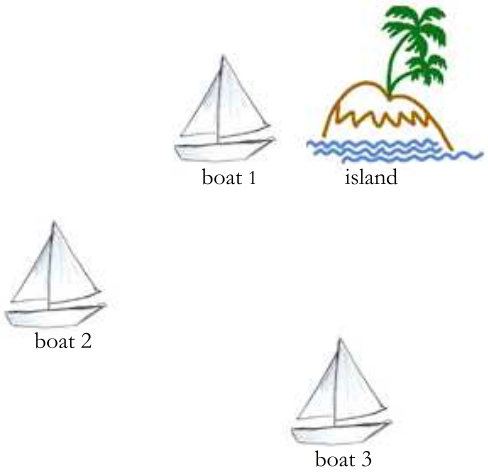


Figure 4.4: Simple Boats domain.

4.2.2 Formalism summary

To summarize the formalism, I define a Propositional OO-MDP as a tuple of the following elements:

- A finite set $\mathcal{C}$ of classes. A class $c \in \mathcal{C}$ has a name and a set of attributes $Att(C) = \{C.a_1, \ldots, C.a_a\}$. Attribute values belong to a domain $Dom(C.a)$.
- A finite set $\mathcal{O}$ of objects, where each object $o \in \mathcal{O}$ belongs to some class in $\mathcal{C}$. Notationally, $C(o)$ indicates the class to which object o belongs.
- A state s is the union of the attribute values of all objects: $s = \bigcup_{o_i \in \mathcal{O}} \bigcup_{j=1}^{|Att(C(o_i))|} o_i.att_j$.
- A finite set $\mathcal{A}$ of actions. An action may or may not be parameterized by objects. For notational simplicity, when actions are not parameterized, I will just refer to them by their name.

- A finite set $\mathcal{R}el$ of relations. A relation $r : C_1 \times \ldots \times C_n \rightarrow Boolean$ is a function over the combined attributes of n objects $o_i \in C_i$, and returns a Boolean value indicating if relation r is being established between the n objects or not.

- A finite set $\mathcal{F}$ of Boolean functions defined over any set of objects in the domain. These Boolean functions can be used to encode domain knowledge or state variables not defined by the attributes and relations.

- A set $\mathcal{T}$ of terms, which is a set of Boolean functions defined as the union of all relations and Boolean functions in the domain: $\mathcal{T} \equiv \mathcal{R}el \cup \mathcal{F}$.

- A function $pred : State \rightarrow 2^{|\mathcal{T}|}$, that given a state returns the value of all terms in $\mathcal{T}$.

- A set $\mathcal{D}$ of rules that specify the domain dynamics. A rule d is a tuple containing a condition, effect and probability: $\langle$ condition, effect, prob $\rangle$. A condition is a conjunction of terms. An effect is an operation (math operation, assignment) on a single attribute of a class. Effects are functions $f : Dom(C.a) \rightarrow Dom(C.a)$.

- A reward function $\mathcal{R}$, from a state s to a real number.

- A discount factor γ.

4.3 First-Order OO-MDPs: An Extension

The representation just introduced presents a few limitations, which can be overcome by defining a more general class of OO-MDPs, First-Order OO-MDPs. Let us first illustrate what those limitations are by considering a simple example, depicted in Figure 4.5

The Boxes environment consists of the classes *Man, Box* and *Wall*. The dynamics of this world allow a *man* to push rows of *boxes* around, unless they get stuck against a *wall*. The rows of boxes the man can push can be of arbitrary lenght, the only requirement is that there is free space at the side towards which the man is pushing.

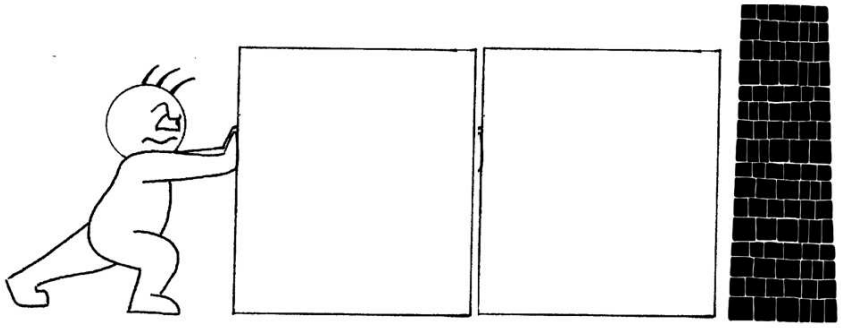


Figure 4.5: Boxes World.

Under the Propositional OO-MDP representation, a condition at the class level that says $touch_E(Man, Box) \wedge \neg touch_E(Box, Wall)$ would not work, because there is no way of tying the Box in the first term with the one on the second one. Such a condition would translate into $\exists b_i \in Box, b_j \in Box, w_k \in Wall : touch_N(man, b_i) \wedge \neg touch_N(b_j, w_k)$, which does not correctly establish that b_i and b_j must refer to the same box. A First-Order OO-MDP (FO-OOMDP) allows dynamics to be expressed directly as First-Order Logic (FOL) predicates. In order to correctly represent these dynamics, the rule under a FO-OOMDP would not be translated from a proposition, but rather be written directly as an existential with the parameters properly tied:

$$\exists b_i \in Box, w_k \in Wall : touch_N(man, b_i) \wedge \neg touch_N(b_i, w_k)$$

Likewise, conditions in Propositional OO-MDPs cannot represent recursive statements, unless these statements are included as simple Boolean functions and their values pre-computed by the environment. In the Boxes example, recursive statements are necessary to express conditions on arbitrarily long lines of boxes. The following recursive logic statement expresses this idea:

$$touch(man, box) \wedge movable(box) \Rightarrow box.x \leftarrow box.x + 1$$
$$movable(box_i) \equiv \neg touch(box_i, wall) \vee (touch(box_i, box_j) \wedge movable(box_j)).$$

The main reason for defining Propositional OO-MDPs as a specific sub-class of OO-MDPs is that, as we shall see in upcoming chapters, this subclass can be efficiently

learned, whereas general FO-OOMDPs cannot (some negative results about learning recursive rules are cited in Section 4.7). Propositional OO-MDPs can still be extended in certain ways and be learnable. For example, a small bounded number of tied parameters could be allowed, a similar assumption to the "constant depth" one commonly taken in the Inductive Logic Programming literature (Dzeroski et al., 1992; Cohen, 1995a). Also certain classes of recursive concepts can be PAC-learned, under some restrictive assumptions (Khardon, 1996). For the purpose of this dissertation, I will not make any such extensions and from now on focus on Propositional OO-MDPs.

4.4 Examples

This section presents a summarized description of how Taxi and Pitfall are represented as (Propositional) OO-MDPs, as well as all the rules that govern their dynamics. I also introduce a very simple domain, called Goldmine, that will serve as an example where multiple objects of the same class coexist. Finally, I show how a different classic videogame, Pac-Man, can also be modeled as an OO-MDP.

4.4.1 Taxi

Table 4.2 presents the classes defined in Taxi, and the attributes for each class. The attributes x and y define grid locations. The attribute *in-taxi* tells us if the passenger is inside the taxi or not. Walls have a *position* attribute, a value in the set {*below, above, left, right*} encoding each of the 4 possible wall positions around a grid cell.

Class	**Attributes**
Taxi	x,y
Passenger	x,y, *in-taxi*
Destination	x,y
Wall	x,y, *position*

Table 4.2: Classes in Taxi and their attributes

Table 4.3 presents the dynamics of Taxi, defined as a series of conditions and effects for each action.

Similar rules could be used to represent reward. However, in most experiments in

Action	Condition	Effects
North	$\neg touch_N(Taxi, Wall)$	$Taxi.y \leftarrow Taxi.y + 1$
South	$\neg touch_S(Taxi, Wall)$	$Taxi.y \leftarrow Taxi.y - 1$
East	$\neg touch_E(Taxi, Wall)$	$Taxi.x \leftarrow Taxi.x + 1$
West	$\neg touch_W(Taxi, Wall)$	$Taxi.x \leftarrow Taxi.x - 1$
Pickup	$on(Taxi, Passenger)$	$in\text{-}taxi \leftarrow True$
Dropoff	$in\text{-}taxi = True \wedge on(Taxi, Destination)$	$in\text{-}taxi \leftarrow False$

Table 4.3: Dynamics of Taxi

this dissertation, I will focus on learning dynamics and assume the reward function is available as a black box function. For the completeness of this example, I still do define the rewards in terms of conditions, as shown in Table 4.4.

Action	Condition	Effects
North, South, East, West	$\emptyset$	-1
Pickup	$on(taxi, passenger)$	-1
Pickup	$\neg on(taxi, passenger)$	-10
Dropoff	$in\text{-}taxi = True \wedge \neg on(taxi, destination)$	-10
Dropoff	$in\text{-}taxi = True \wedge on(taxi, destination)$	0

Table 4.4: Rewards in Taxi

Episodes in Taxi end when the following termination condition is reached: $in\text{-}taxi = False \wedge on(Taxi, Destination)$

4.4.2 Pitfall

In Pitfall, a simple object recognizer is run over each screen and bounding boxes defined over the relevant objects (see Figure 4.6). Classes are *Harry, Pit, Log, Ladder, Tree* and *Wall*, all of them with two pairs of x and y coordinates defining the bounding box, plus an attribute *direction* for Harry indicating which way he is facing. The function *pred* uses the bounding boxes to determine what Harry is touching. In the example in Figure 4.6, the only relation that is True is $touch_E(harry, wall)$, and all others are False.

The dynamics of Pitfall involve many more rules than Taxi, so I will not list them exhaustively. The reader should get a good idea of what the rules look like from the examples in Table 4.5.

Figure 4.6: Bounding boxes identifying objects in Pitfall.

Action	Condition	Effects
StickRight	$\neg touch_E(Harry, Wall) \wedge$ $\neg touch_E(Harry, Log) \wedge$ $\neg touch_E(Harry, Pit) \wedge \neg on(Harry, Ladder)$	$Harry.x \leftarrow Harry.x + 8$
StickRight	$touch_E(Harry, Pit)$	$Harry.y \leftarrow Harry.y - 75$
StickDown	$on(Harry, Ladder)$	$Harry.y \leftarrow Harry.y - 5$ $Harry.x \leftarrow Harry.x + 8$

Table 4.5: Some dynamics of Pitfall

4.4.3 Goldmine

Goldmine is a simple gridworld domain I created to make the presentation of the transition cycle easier to follow, and as an example where multiple objects of the same class coexist. It models a very simple resource-collection scenario, where multiple miners have to gather gold. Some cells of the grid contain gold pieces that disappear once collected. At each timestep, only one of the miners can be moved using North, South, East and West actions parameterized by the miner object that the decision maker wants to move. Another parameterized action, GetGold, picks up a piece of gold if it is present at the miner's location. Each action taken incurs a cost of -1, and each gold piece collected provides a reward of 5. The episode ends when all the available gold is collected. Walls limit the miners' movements. For simplicity, miners can be at the same location at the same time (that is, movement is not limited by bumping into another miner). See Figure 4.7 for an example with 2 miners and 3 pieces of gold.

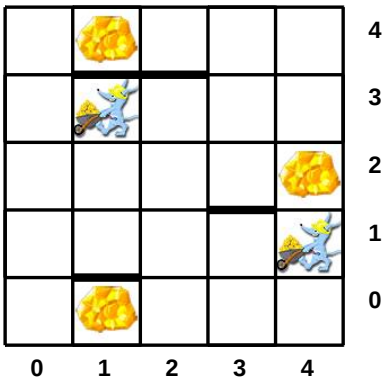


Figure 4.7: Goldmine domain example.

The classes in this domain are Miner, Gold and Wall. They all have attributes x and y. Walls also have a *position*, like in Taxi. When Gold is consumed, its attributes x and y become $\emptyset$. In the example of Figure 4.7, I will call the object of class Miner at location $(1, 3)$ "m_1" and the object at location $(4, 1)$ "m_2". Gold pieces will be identified, from top to bottom, as g_1, g_2 and g_3. I will not list here the specific identifiers of the 24 wall objects.

Dynamics are of the same kind as the ones in Taxi, summarized in Table 4.6. Notice that actions here are parameterized, since they can apply to any of the miners.

Action	**Condition**	**Effects**
North(m_i)	$\neg touch_N(m_i, Wall)$	$m_i.y \leftarrow m_i.y + 1$
South(m_i)	$\neg touch_S(m_i, Wall)$	$m_i.y \leftarrow m_i.y - 1$
East(m_i)	$\neg touch_E(m_i, Wall)$	$m_i.x \leftarrow m_i.x + 1$
West(m_i)	$\neg touch_W(m_i, Wall)$	$m_i.x \leftarrow m_i.x - 1$
GetGold(m_i)	$on(m_i, gold)$	$gold.x \leftarrow \emptyset$,$gold.y \leftarrow \emptyset$

Table 4.6: Dynamics of Goldmine

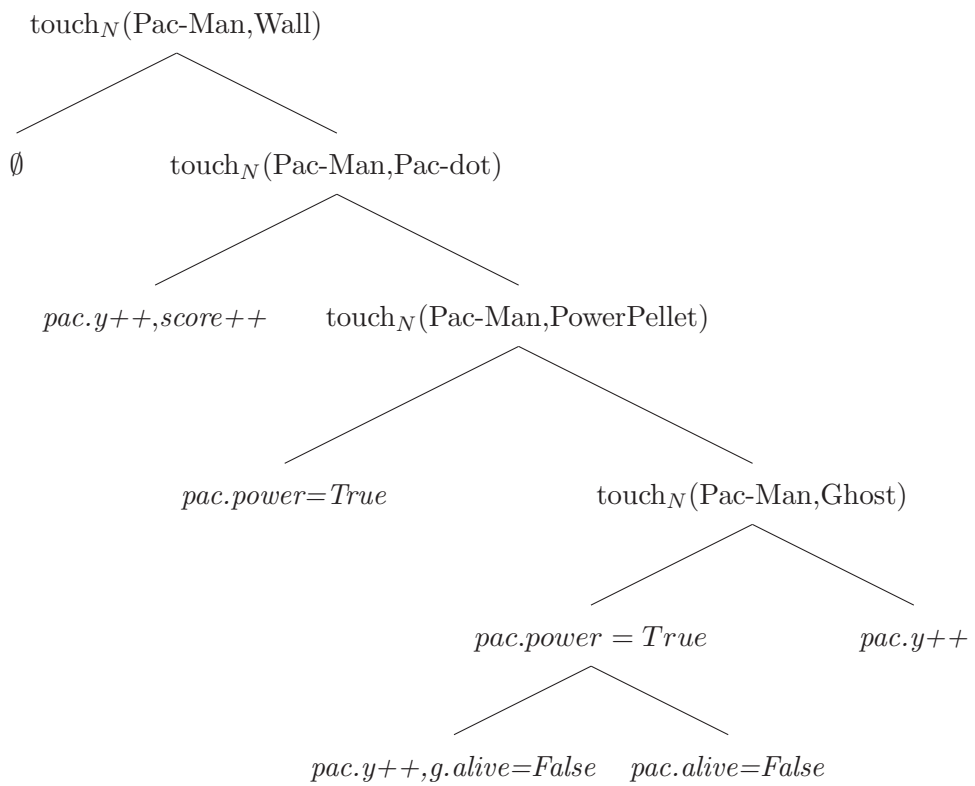
Notice that the effects of actions have a symmetric property (South vs North / East vs West). Currently, this fact is not encoded in the representation, and it would be up to a smart learner to exploit this fact (as humans do!).

4.4.4 Pac-Man

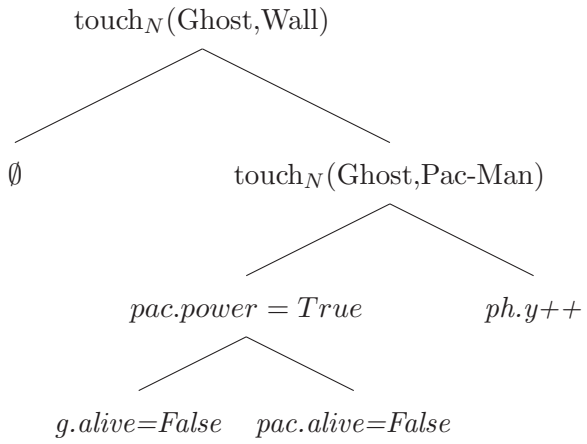
I will complete the set of examples with another video game, the famous Pac-Man (Figure 4.8).

Figure 4.8: Pac-Man.

The classes in this game are *Pac-Man, Ghosts, Pac-dot, PowerPellet* and *Wall*, and objects belonging to them will be denoted *pac*, *g*, *dot*, *pell* and *w*, respectively. The conditions ruling the dynamics of Pac-Man and the ghosts are quite complicated, and better represented in tree form, as shown below (note the convention that left branches represent the case when condition is true). I only represent one of the movement actions, *North*, the others being completely analogous.



In the case of Ghosts, the rules are represented as follows:



4.5 Transition Cycle

I will now present the transition cycle under an OO-MDP representation. That is, I will present step by step how an agent and an environment interact, moving the domain from state s_t to state s_{t+1}. This cycle repeats while the agent is acting (in the present case, until an end of episode is reached). Table 4.7 explains the cycle and exemplifies each step using Goldmine, with state s corresponding to the one depicted in Figure 4.7.

Step	Example
Agent observes current state s.	$m_1 \equiv< x = 1, y = 3 >$ $m_2 \equiv< x = 4, y = 1 >$ $g_1 \equiv< x = 1, y = 4 >$ $g_2 \equiv< x = 4, y = 2 >$ $g_3 \equiv< x = 1, y = 0 >$ $wall_{\{0..23\}} \equiv \ldots$
$pred(s)$ translates s into terms.	$touch_N(m_1, wall_{20})$ $touch_N(m_2, g_2)$ $touch_E(m_2, wall_8)$ *All other relations between Miners and Gold and Miners and Walls negated.*
Agent takes action a (parameterized or global)	$North(m_2)$.
Environment searches for condition a of action a that matches some grounding of the classes.	$North(m_2) \rightarrow \neg touch_N(m_2, Wall) \equiv$ $\neg \exists w_i \in Wall : touch_N(m_2, w_i) \equiv True$ (m_2 is not touching any wall)
For each fulfilled condition, an effect occurs. Effects are used to compute next state s'.	$m_2.y \leftarrow m_2.y + 1$
Environment chooses reward r from $\mathcal{R}(s, a)$ and tells it to the agent.	$r = -1$

Table 4.7: Transition cycle for OO-MDPs

4.6 OO-MDPs and Factored-state Representations Using DBNs

In this section, I will compare OO-MDPs to factored-state representations, and illustrate how they differ and why OO-MDPs can more succinctly represent task dynamics. As an example, let us consider what a factored-state representation of Goldmine would look like. One could represent this domain using a state variable for the x and y attributes of each miner and each gold piece, for a total of 10 variables. In a standard factored-state representation, actions would not be parameterized and there would be a different set for each miner, although having a parameterized version is a simple alteration. A DBN representation would indicate that, for North, South, East and West actions attached to a particular miner, x and y of that miner are enough to determine their next values x' and y'. In the case of the GetGold action, x' and y' of each gold piece depends on its previous values, plus the values of x and y of the miner in question.

Walls would not need to be represented as objects, because the dynamics of miners'

movements would depend on their absolute coordinates in the grid anyway. That is, a factored-state algorithm like Factored $R_{\max}$ would learn that when the x variables of miner m_1 is 0 and y is 4, action North does not change any variable, but when they are 0 and 3, x becomes 4. For action North, it would have to learn these transitions independently for all combinations of x and y. Notice that it cannot ignore x when learning about moving North, because while $(x, y) = (0, 3)$ moves the miner to $(0, 4)$, trying the same action from $(1, 3)$ does not.

Notice also that without parameterized actions and types, any knowledge about movement of one miner would have to be learned independently of any other one.

The comparison between an OO-MDP representation of Goldmine and its factored-state representation illustrates some of the reasons why OO-MDPs can be more succinct. In summary, the main differences between OO-MDPs and DBN representations are:

- *(In-class transfer)* Relational conditions allow learning about all objects of a given class at the same time. The dynamics of miners' movements and their interactions with gold generalize to all miners in the domain and all gold pieces. Adding more miners or more gold to an OO-MDP domain does not require any extra learning, while in the DBN case it requires adding new state variables and learning their transition dynamics anew.

- *(Attribute generalization)* Defining dynamics based on relations between objects extracted from attributes allows generalization across attribute values. The absolute (x, y) location of a miner does not define whether it can move North or not, only whether or not it is touching a wall does. Making the grid bigger, moving or adding walls or gold pieces does not require extra learning in the OO-MDP case, but it does in a DBN representation.

- *(Action generalizations)* Actions are generalized by their parameterization. Whatever is learned from executing an action North on one miner generalizes to all other miners. In a DBN representation, the dynamics of the state variables of one miner do not transfer to the state variables of the others.

- *(Relative effects)* Effects represented as a change in an attribute value also enable generalization. Once an OO-MDP learner understands, for example, that if there is no wall x or y change by 1, this knowledge applies to any particular grid location, regardless of its absolute value. In DBN representations, transition dynamics depend on absolute positions.

All of the above generalizations enabled by an OO-MDP representation are the crux of its representational advantages. We will see how they enable orders of magnitude faster learning when compared to factored-state representations.

4.7 OO-MDPs and Relational RL

In OO-MDPs without any extensions, the state is completely propositional. The set of relations extracted from a state by the function $pred(s)$, which returns a set of grounded terms, is also propositional. The only lifted aspect of OO-MDPs is the way in which transition rules and rewards are encoded in terms of classes that can be understood as object variables. This representation can be translated into existential statements of finite scope. De Raedt (2008) defines relational representations as those that that are able to represent relations over a variable number of entities, as opposed to logical representations, which can only refer to a fixed, grounded number of objects. In that sense, the OO-MDP representation can be called relational, as objects and relations are defined in terms of classes and are then instantiated in a variable number of objects.

OO-MDPs can be extended into FO-OOMDPs to cover the full expressive power of FOL. In their propositional form, they do not represent recursion, unbounded existentials, or many of the other constructs available in full FOL. While it would be desirable to include and be able to learn many of these constructs, there are both negative results as well as a number of open questions in terms of their learnability. For example, attempts have been made at learning recursive rules from examples, with mixed results. Essentially, with enough assumptions and constraints, some cases of recursion can be efficiently learned, but a slew of negative results exist about more general cases. The reader is encouraged to refer to two parallel papers, one with positive and one with

negative results about the learnability of recursion in the context of Inductive Logic Programming (Cohen, 1995b; Cohen, 1995a).

If we were willing to forgo learning guarantees, I hypothesize that it is a simple exercise to extend the expressive power of OO-MDPs by accepting limited forms of recursion like the cases described by Cohen (1995b), but this goal is beyond the scope of this dissertation.

4.8 OO-MDPs and Deictic Representations in Reinforcement Learning

Another type of representations introduced to RL, with similar goals as those of OO-MDPs, are deictic representations. The word deictic comes from the Greek *deiktikos*, meaning "able to show". Under these representations, expressions "point" to an object, and statements are made relative to the object in question. Examples of such expressions in linguistics are "the cup that I am holding" and "the person sitting next to me".

Deictic representations either have agency, or have an attentional or perceptual component. That is, a deictic statement must be relative to an agent ("the food *I* am eating") or be centered around a particular object or entity of reference ("the cup that is on the red table", "the chair *Mary* is sitting on").

In contrast, under standard propositional representations, objects have unique identifiers and statements are of the form $over(object_{27}, object_{14})$. Under a deictic representation in RL, expressions are usually relative to a pre-identified agent (*the-object-I-am-grasping*), and further extended through chains or relationships: *the-object-on-top-of(the-object-of-the-same-color-as(the-object-I-am-grasping))*.

Finney et al. (2002b) is one of the first attempts to bring deictic representations to RL, with mixed results, as indicated by the title of their paper: *The Thing that we Tried Didn't Work very Well: Deictic Representation in Reinforcement Learning.* The result was further extended in Finney et al. (2002a), showing that empirical performance in a Blocks World domain, under their deictic representation, was worse than under a

full propositional one. In their representation, the agent controls an attentional marker that can point to an object and obtain perceptual information about it, including which objects are in its immediate vicinity. More markers can be added and made to point to different objects at the same time. In the extreme case, if one marker per object is available, the representation becomes a standard propositional one. When there are fewer markers than objects, partial observability ensues. This partial observability may be beneficial (if it simplifies the problem by occluding irrelevant aspects of it) or it can hinder learning (if it occludes relevant ones).

Ponsen et al. (2006) apply a similar idea to a Real-Time Strategy game. They simulate one component of these kinds of games, which is the sub-problem of worker units needing to navigate to a location while avoiding enemy units. For this type of task, *local* information about nearby enemies is usually a sufficient statistic, and simplifies the problem compared to the case in which all other enemy units must be considered by the learner, no matter how far away they are. They refer to this representation as deictic: the player controls one unit at a time (therefore focusing attention on it), and only considers information relative to the vicinity of the unit being attended to.

A similar problem is tackled by Ravindran et al. (2007). They define a deictic representation involving what they call *relativized options*. Policies are constructed as schemas that can be applied relative to a given object in the world.

OO-MDP representations share a number of ideas with deictic representations, although formally they do not exactly match any of the ones mentioned or, as far as I know, any other existing one. Conceptually, however, relations in OO-MDPs can be thought of as deictic, as they usually refer to a given object (which can be considered the attentional focus) and its vicinity. As we shall see in upcoming chapters, one benefit of the current representation is that it enables efficient learning, which has not been demonstrated in previously existing deictic ones.

4.9 Summary

OO-MDP representations seek to strike a balance between generality, expressive power and efficient learning. It is already known that factored-state representations can be efficiently learned (Kearns & Koller, 1999; Guestrin et al., 2002), even when the full structure of DBNs is not provided as input (Strehl et al., 2007; Diuk et al., 2009), and are very general. It is also usually the case that representations with higher expressive power enable more succinct descriptions of domains and their transition dynamics. However, as the expressive power increases, such representations tend to become harder to learn. FOL representations have much higher expressive power than factored-state ones, but they cannot be learned efficiently without a large number of assumptions or the incorporation of prior knowledge.

OO-MDP representations enable a relational description of a domain that is fairly natural, relying on propositional descriptions of state and relations that do not go beyond simple computations performed directly over the attributes of the objects involved. However, OO-MDP representations present a number of advantages over DBNs, enabling, in many cases like Taxi or some of the examples presented in upcoming chapters, order-of-magnitude decreases in the description size of domain dynamics, corresponding to an order-of-magnitude speedup in learning time. What remains to be shown is that OO-MDP representations can be learned efficiently. In the upcoming chapters, I will demonstrate that OO-MDP representations provide many of the representational advantages of lifted representations without losing theoretical learning guarantees like the ones available for factored-state representations.

Chapter 5

Learning Deterministic OO-MDPs

In Chapter 3, I mentioned DOO$R_{\max}$(Deterministic Object-Oriented $R_{\max}$), a learning algorithm for deterministic OO-MDPs that is an instance of the KWIK-$R_{\max}$ family of algorithms. In this chapter, I fully describe it and analyze its sample complexity. DOO$R_{\max}$ is designed for deterministic Propositional OO-MDPs, that is, those in which for each action and a given condition there is only one effect that can occur, and it occurs with probability 1. I will build up the full presentation of the algorithm starting from simpler cases. First, I introduce a KWIK algorithm called *enumeration* that will be instantiated as a condition and an effect learner. Second, I present how a condition can be learned for a known effect. Third, I show how an effect can be learned. Then, I combine the condition and the effect learners to learn a single condition-effect pair. Finally, I present a more realistic model, which I call the *Tree Model*, in which each action can have up to k conditions, and show how it can be learned. Some of the material in this chapter has been introduced previously in Diuk et al. (2008).

5.1 The KWIK Enumeration Algorithm

Li et al. (2008) showed that if a hypothesis class is finite and its observations are deterministic, then it is KWIK-learnable, and an algorithm called *enumeration* can be used to learn it. I introduce enumeration in this section, and then show how deterministic OO-MDPs can be learned using instances of it.

If the hypothesis class to be learned is $\mathcal{H}$, enumeration keeps track of $\hat{\mathcal{H}} \subset \mathcal{H}$, the *version space* that contains the hypotheses that are consistent with the data observed so far. The algorithm initializes $\hat{\mathcal{H}}$ to $\mathcal{H}$. At each timestep t, the algorithm observes input x_t and is asked to make a prediction of the output $h(x_t)$ (which as in every KWIK

algorithm, can be $\perp$). Enumeration computes $\hat{L} = \{h(x_t)|h \in \hat{\mathcal{H}}\}$, the set of outputs computed by all the hypotheses in its current version space, and will progressively eliminate hypotheses that contradict the observations. An assumption in KWIK is that the correct hypothesis, h^*, is in $\mathcal{H}$, so enumeration will produce at least one output: $|\hat{L}| \geq 1$. If it has exactly one prediction, $|\hat{L}| = 1$, it means that all the hypothesis agreed on the output, and therefore the algorithm knows the answer. If $|\hat{L}| > 1$, there is a disagreement and the algorithm will have to respond $\perp$, and receive as observation the true output $h(x_t) = y_t$. This output will match some of the hypotheses in $\hat{\mathcal{H}}$ and contradict some others. The hypotheses that did not match y_t are eliminated from $\hat{\mathcal{H}}$. Note that there must be at least one agreeing hypothesis ($h^* \in \hat{\mathcal{H}}$) and at least one disagreeing (since $|\hat{L}| > 1$). Therefore, the size of the version space will be reduced by at least 1 with each $\perp$ prediction.

The resulting KWIK bound for enumeration is $\mathcal{H} - 1$, linear in the size of the hypothesis space. The complete description is shown in Algorithm 8.

Algorithm 8: Enumeration algorithm, for KWIK learning finite hypothesis classes with deterministic outputs.

Inputs: $\mathcal{X}, \mathcal{Y}, \mathcal{H}, \epsilon, \delta$.
Initialize $\hat{\mathcal{H}} \leftarrow \mathcal{H}$.
for $t = 1, 2, \ldots$ **do**
  Observe input $x_t \in \mathcal{X}$.
  Compute predictions $\hat{L} = \{h(x_t)|h \in \hat{\mathcal{H}}\}$.
  if $|\hat{L}| = 1$ **then**
    Predict $\hat{y}_t = y$, the only prediction from $\hat{L}$.
  else
    Predict $\hat{y}_t = \perp$.
    Observe y_t.
    Update version space: $\hat{\mathcal{H}} \leftarrow \hat{\mathcal{H}} \setminus \{h|h(x_t) \neq y_t\}$
  end if
end for

5.2 Learning a Condition

I will show how deterministic OO-MDPs can be KWIK-learned starting from simple cases. In this section, I will consider the case where there exists a single, known effect in the environment, and a single condition that produces it. The learning problem is

to find out what that condition is. Given a condition as input, the task of the KWIK learner is to produce a *True/False* prediction indicating whether the effect will occur or not.

As defined in Section 4.2.2, conditions are conjunctions of terms from the set $\mathcal{T}$. A condition can be a conjunction of any number of terms $t \in \mathcal{T}$ or their negations $\neg t$. That is, a condition c is a subset $c \subset \cup_{t \in \mathcal{T}} \{t, \neg t\}$ (I will represent conjunctions as sets of terms or negations of terms that need to be True). An initial approach to learning a condition is to simply use the enumeration algorithm for the hypothesis class that includes all possible conditions, yielding a set $\hat{\mathcal{H}}$ of size $|\hat{\mathcal{H}}| = 2^{2|\mathcal{T}|}$. However, given that each hypothesis represents a conjunction, some generalization is possible. Let us start with an example:

Consider a set of terms $\mathcal{T} = \{A, B, C\}$, and imagine that the effect occurs whenever A is true, regardless of the values of B and C. That is, the true hypothesis is $h^* = \{A\}$. The initial set of hypotheses $\hat{\mathcal{H}}$ would include the following list of conditions, for a total of $|\hat{\mathcal{H}}| = 16$:

$$
\begin{array}{rrrll@{\qquad}rrrl}
A & B & C & \rightarrow \textit{True} & & A & B & C & \rightarrow \textit{False} \\
A & B & \neg C & \rightarrow \textit{True} & & A & B & \neg C & \rightarrow \textit{False} \\
A & \neg B & C & \rightarrow \textit{True} & & A & \neg B & C & \rightarrow \textit{False} \\
A & \neg B & \neg C & \rightarrow \textit{True} & & A & \neg B & \neg C & \rightarrow \textit{False} \\
\neg A & B & C & \rightarrow \textit{True} & & \neg A & B & C & \rightarrow \textit{False} \\
\neg A & B & \neg C & \rightarrow \textit{True} & & \neg A & B & \neg C & \rightarrow \textit{False} \\
\neg A & \neg B & C & \rightarrow \textit{True} & & \neg A & \neg B & C & \rightarrow \textit{False} \\
\neg A & \neg B & \neg C & \rightarrow \textit{True} & & \neg A & \neg B & \neg C & \rightarrow \textit{False}
\end{array}
$$

Note that if $h^* = \{A\}$, the 4 hypotheses that include A (not negated) and predict *True* will always make a correct prediction. This means that a naïve approach might require up to 12 data points before the condition is learned. However, with the extra knowledge that hypotheses represent conjunctions, it is possible to learn with less experience (exponentially less, in fact!). In the previous example, imagine the learner is exposed to these two examples:

$$A \quad B \quad C \quad \rightarrow True$$
$$A \quad \neg B \quad \neg C \quad \rightarrow True$$

These examples are enough to discover that terms B and C do not matter, and that the fact that A is true is enough to predict that the condition is satisfied. Observe, however, that negative examples do not provide such an opportunity for generalization. Imagine the first observation had been:

$$\neg A \quad B \quad C \quad \rightarrow False$$

This observation only allows the enumeration learner to eliminate the hypothesis $\neg A \wedge B \wedge C \rightarrow True$, and nothing else. If we call $n = 2|\mathcal{T}|$, this asymmetry between positive and negative examples will yield a worse-case bound for learning deterministic OO-MDPs of $O(2^n)$ when lots of negative examples are observed, whereas in the best case, if provided with enough positive examples, it can learn in $O(n)$.

5.2.1 Condition Learner Algorithm

To specify the algorithm more formally, I need to specify the hypothesis class that is provided as input, how predictions are made, and how $\hat{\mathcal{H}}$ gets updated, corresponding to lines 2, 5 and 11 of Algorithm 8.

The Condition Learner will make use of the fact that hypotheses represent conjunctions in order to eliminate more than one hypothesis per observation, as exemplified before. In order to show how this generalization is achieved, I introduce some notation:

- For every state $s \in \mathcal{S}$, the function *pred*(s) returns the truth value of all terms in $\mathcal{T}$ induced by s. For example, if there is a term t indicating whether an object is touching a wall or not, *pred*(s) will return t or $\neg t$, depending on where the object is in relation to the wall in state s.

- I will assume the terms in $\mathcal{T}$ are ordered, resulting in a set $\{t_1, \ldots t_n\}$. A hypothesis h is represented by a string h_S of length n, where $h_S^i = 1$ if it makes its prediction based on t_i being True, $c_S^i = 0$ if it makes it based on $\neg t_i$ being True and $c_S^i = \star$ if it ignores t_i when making predictions. For example, in the Taxi problem

consider the four terms representing whether or not the taxi is touching a wall to the N/S/E/W: $touch_N(Taxi, Wall)$, $touch_S(Taxi, Wall)$, $touch_E(Taxi, Wall)$ and $touch_W(Taxi, Wall)$. The correct hypothesis h^* for action North requires only that $\neg touch_N(Taxi, Wall)$ is True. The resulting string used to represent this hypothesis is $h^*_S = 0\star\star\star$.

- The condition induced by state s, $pred(s)$, can also be represented in string form as $pred_S(s)$.

- Given two hypotheses represented as strings h_1 and h_2, I define the element-wise commutative operator $\oplus$ as follows:

h_1^i	h_2^i	$h_1^i \oplus h_2^i$
0	0	0
1	1	1
0	1	$\star$
0|1	$\star$	$\star$

- A hypothesis h_1 *matches* another hypothesis h_2, noted $h_1 \models h_2$, if $\forall 1 \leq i \leq n$: $h_1^i = \star \vee h_1^i = h_2^i$. Hypotheses that match the one for action *North* ($0\star\star\star$) include 0100, 0011 and 0101, but not 1000 or 1010.

First, I will assume that all the hypotheses in the enumeration algorithm are represented as strings. The condition learner will keep an extra variable h_T, initialized to $\emptyset$. This variable will represent all the hypotheses for which a True prediction should be made. Back to the Taxi example, we would like to learn that $h_T = 0\star\star\star$ and predict True for all $pred_S(s)$ such that $h_T \models pred_S(s)$.

The prediction step (line 5 of Algorithm 5) will look through all the hypotheses in $\hat{\mathcal{H}}$ searching for those such that $h(pred_S(s)) = False$. Plus, it will look at whether h_T matches $pred_S(s)$, or if it is $\emptyset$, and predict True if so. Formally, $\hat{L} = \{False | h \in \hat{\mathcal{H}} \wedge h(pred_S(s)) = False\} \cup \{True | h_T \models pred_S(s) \vee h_T = \emptyset\}$.

Given an observation y_t, the learning step (line 11 of Algorithm 5) will depend on whether this observation is True or False. If it is a negative example ($y_t = False$), it

will simply eliminate the hypothesis h that predicted True. If it is a positive example ($y_t = True$), first it will update h_T. If $h_T = \emptyset$, it will set $h_T \leftarrow pred_S(s)$. Otherwise, it will update it as follows: $\forall 1 \leq i \leq n : h_T^i \leftarrow h_T^i \oplus pred_S(s)$. Finally, it will eliminate all hypotheses h_i that predict False such that $h_T \models h_i$.

Analysis

The condition learner is initialized with $|\hat{\mathcal{H}}| = 2^{|\mathcal{T}|+1}$ hypotheses ($2^{|\mathcal{T}|}$ conditions times 2, once for predicting True and once for predicting False). In the worst case, where only negative examples are observed, it might take $|\hat{\mathcal{H}}| - 1$ examples before the right condition is learned, an exponential dependency on the number of terms that define the environment's dynamics. If the number of terms is large, and many negative examples are observed, condition learner becomes intractable. However, it takes at most two positive examples to learn at least one term in h_T. With a positive example, at least one term either becomes the correct value $0|1$, or it takes a second example to make it $\star$. For each term in h_T that is learned, an exponential number of hypotheses are eliminated. A best-case analysis, where sufficient positive examples are observed, yields a bound of $O(|\mathcal{T}|)$ to learn the condition, a linear dependency on the number of terms.

5.2.2 Example: Learning a Condition in Taxi

Let us assume a simplified Taxi domain, where there is a taxi surrounded by walls but no passenger to pick up and drop off. We simply want to learn under which conditions the action *North* moves the taxi by adding 1 to its y coordinate.

Let us assume that the terms are touch relations between the taxi and walls in all four directions: $touch_N(Taxi, Wall)$, $touch_S(Taxi, Wall)$, $touch_E(Taxi, Wall)$ and $touch_W(Taxi, Wall)$ (in the string notation, we will assume this ordering).

Imagine that we start in a state s_0 and execute two *North* actions, transitioning to states s_1 and s_2, as shown in Figure 5.1.

When the taxi transitions from state s_0 to s_1, a change in its attribute y is observed ($y = 0$ to $y = 1$), which means that this is a positive example. That is, the condition

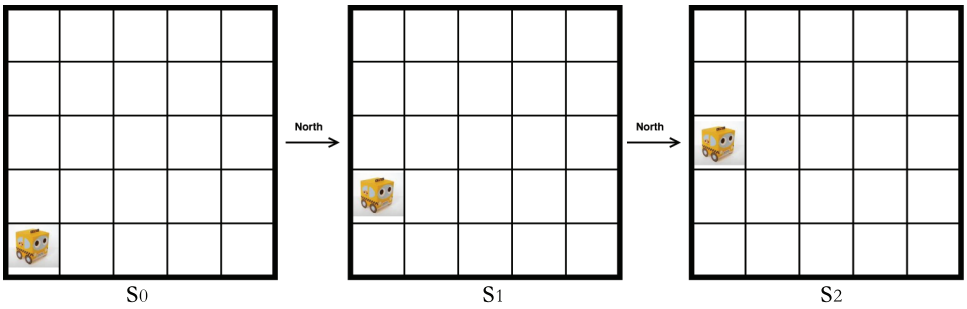


Figure 5.1: Simple Taxi with walls. An initial state s_0 transitions to s_1 and then s_2 through $North$ actions.

that was true in state s_0, $pred(s_0)$, was enough to produce an effect. Since initially h_T was $\emptyset$, it will be set to $pred(s_0)$ to become $h_T = 0101$, which corresponds to the conjunction:

$$\neg touch_N(Taxi, Wall) \wedge touch_S(Taxi, Wall) \wedge \neg touch_E(Taxi, Wall) \wedge touch_W(Taxi, Wall)$$

Another $North$ action is now taken, transitioning from state s_1 to s_2. The condition that enabled this effect to occur was $pred(s_1) \equiv \neg touch_N(Taxi, Wall) \wedge \neg touch_S(Taxi, Wall) \wedge \neg touch_E(Taxi, Wall) \wedge touch_W(Taxi, Wall)$, or 0001. Now that $h_T = 0101$, the $\oplus$ operator is applied: $h_T \oplus pred_S(s_1) = 0101 \oplus 0001 = 0 \star 01$. It just learned that the term $touch_S(Taxi, Wall)$ has no influence on action $North$. The current hypothesis h_T for predicting the occurrence of an effect corresponds to conjunction:

$$\neg touch_N(Taxi, Wall) \wedge \neg touch_E(Taxi, Wall) \wedge touch_W(Taxi, Wall).$$

A few steps later, we'll imagine the taxi is in state s_t and transitions to s_{t+1} after a $North$ action, as depicted in Figure 5.2. The condition that made this transition possible is $pred(s_t) \equiv \neg touch_N(Taxi, Wall) \wedge \neg touch_S(Taxi, Wall) \wedge \neg touch_E(Taxi, Wall) \wedge touch_W(Taxi, Wall)$, or 0010. As before, the operation $h_T \oplus pred_S(s_t)$ is applied, resulting in: $h_T = 0 \star 01 \oplus 0010 = 0 \star\star\star$. This hypothesis corresponds to the correct conjunction that determines the condition for a positive effect under action $North$:

$$\neg touch_N(Taxi, Wall)$$

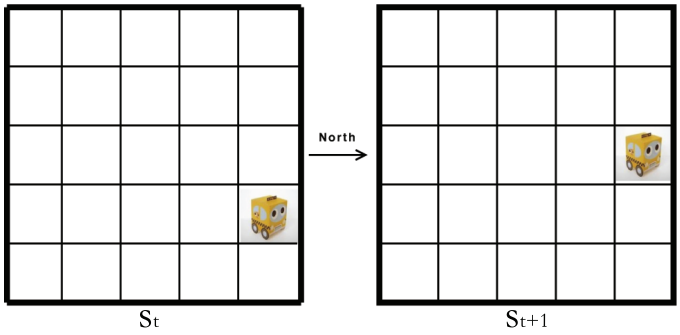


Figure 5.2: Two states s_t and s_{t+1} resulting from a $North$ action.

In the next section I will show how single effects are learned, and then combine the two learners to learn up to k condition-effect pairs.

5.3 Learning an Effect

In the previous section, it was assumed there was a single, known effect that could either happen or not happen depending on the condition. In this section, I will assume the effect itself also needs to be learned. Once again, I will use the enumeration algorithm to KWIK-learn the effect of an action.

An important assumption regarding effects is that there is a finite set of effect *types* that will be considered. For example, an effect of *arithmetic* type would be one that consists of applying a simple arithmetic operation to an attribute $(+, -, *, /)$. An effect of type *assignment* would be one that assigns a fixed constant value to an attribute. The designer of the algorithm needs to pre-define these effect types, and provide it with functions that compute, given two attribute values, the possible effects that can transform one value to the other. For example, if the x position of an object changes from the value $x = 2$ to the value $x = 4$, and the designer established as possible effect types *arithmetic* and *assignment*, the algorithm should be able to consider as possible effects the operations $x \leftarrow x + 2$ (arithmetic), $x \leftarrow x * 2$ (arithmetic) or $x \leftarrow 4$ (assignment). The set of operations that will be considered is up to the algorithm designer as long as, given two states s_i and s_j, the algorithm can access a function that returns all possible operations that would transform s_i into s_j:

- For any states s and s' and attribute att, the function $\mathit{eff}_{att}(s, s')$ returns a list of effects of each type that would transform attribute att in s into its value in s'.

Note that although the function $\mathit{eff}_{att}(s, s')$ is defined over an infinite domain (for example, arithmetic operations over Real numbers), it should always return a finite set of possible transformations. The enumeration algorithm for effects will then start with an empty hypothesis class, and only be initialized after the first observation (a transition from s_0 to s_1) with the results of $\mathit{eff}_{att}(s_0, s_1)$. The prediction step will receive as input an attribute value and it will apply all effects in its hypotheses $\hat{\mathcal{H}}$ to it. If the predictions for the resulting value of the attribute contradict, it responds $\perp$, waits for an observation, and deletes all hypotheses that do not match the observation.

Given that eff_{att} returns a finite set $\mathcal{E}_t$ for each effect type t, the effect learner can have a sample complexity of up to $(\sum_t |\mathcal{E}_t|) - 1$.

5.3.1 Example: Learning an Effect

Consider as an example an infinite gridworld with no walls or obstacles, and a taxi that can navigate freely around it, as shown in Figure 5.3.

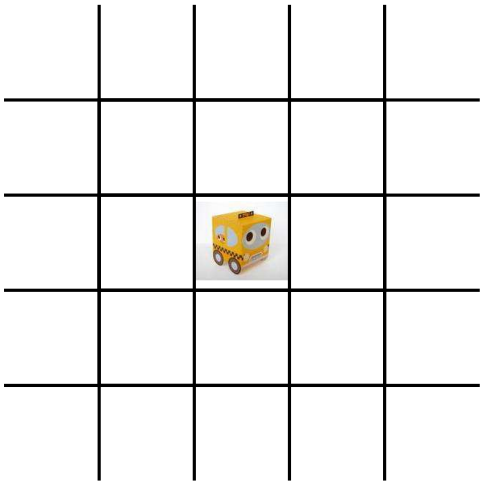


Figure 5.3: Infinite-grid Taxi.

Let us assume that the taxi has two attributes indicating its location, x and y, and it starts in position $< x, y >=< 0, 0 >$. At the beginning, the algorithm has no hypotheses of what action *North* does, so it executes it and observes that the new values

for the taxi attributes are $< x, y >=< 0, 1 >$. Let us also assume that the effect types considered by our algorithm are *addition* and *assignment*. In that case, after executing action *North*, the algorithm has the hypothesis that its effect is either $y \leftarrow y + 1$ or $y \leftarrow 1$, as returned by $eff_{att}(s, s')$. If asked what would happen to the taxi from any location in which $y = 0$, the algorithm can unambiguously predict, given its current hypotheses, that in the next step $y = 1$. However, now that $< x, y >=< 0, 1 >$, if asked to predict what another action *North*'s effect would be, the two hypotheses generate a contradictory outcome: either $y = 2$ (if $y \leftarrow y + 1$ is the true effect) or $y = 1$ (if $y \leftarrow 1$ is the correct one). The algorithm would thus not have an answer and needs to take another exploratory action, so it responds $\perp$. The result of another action *North* here is $< x, y >=< 0, 2 >$, which is coherent with the hypothesis that $y \leftarrow y + 1$, but contradicts the prediction of hypothesis $y \leftarrow 1$. The latter is eliminated and the only remaining hypothesis about the effect of action *North* is now $y \leftarrow y + 1$, which is the correct one.

5.4 Learning a Condition-Effect Pair

It is possible to combine the condition and effect learners just introduced into a single algorithm, which I call *Condition-Effect Learner* (CELearn). The combination is very simple: whenever the condition learner predicted *False*, it now predicts a null effect, symbolized by $\emptyset$. Whenever it predicted *True*, it now calls an effect learner for a prediction. If the effect learner responds $\perp$, then CELearn also responds $\perp$. Otherwise, CELearn returns the effect returned by the effect learner.

5.5 Learning Multiple Condition-Effect Pairs: the Tree model

In the previous sections I considered the case where only one condition was being learned, and then one effect. In this section, the model is extended to allow multiple conditions and effects for each action. To extend the current example, let us imagine that now the taxi is not only surrounded by walls but it also has an extra attribute indicating its fuel level. Plus, some locations in the grid might have puddles of mud

that force the taxi to make an extra effort. Each movement action consumes 1 unit of fuel in regular locations, and 2 if there is a puddle. The following condition-effect pairs now govern the behavior for action $North$:

$$\neg touch_N(Taxi, Wall) \wedge Taxi.fuel > 0 \Rightarrow Taxi.y \leftarrow Taxi.y + 1$$

$$\neg touch_N(Taxi, Wall) \wedge Taxi.fuel > 0 \wedge \neg on(Taxi,Puddle) \Rightarrow Taxi.fuel \leftarrow Taxi.fuel - 1$$

$$\neg touch_N(Taxi, Wall) \wedge Taxi.fuel > 0 \wedge on(Taxi,Puddle) \Rightarrow Taxi.fuel \leftarrow Taxi.fuel - 2$$

In this example, the action *North* has 3 different possible effects, based on 3 different conditions. To make learning of multiple conditions and effects feasible, a couple of assumptions must hold:

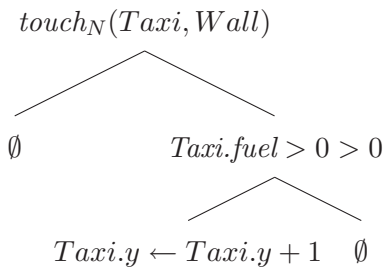
Assumption 8. *For each action and each attribute, only effects of one type can occur.*

This assumption just indicates that a given action cannot have effects of multiple types on the same attribute. There will be situations during learning in which, for example, a change in an attribute can either be attributed to an arithmetic operation or the assignment of a constant value. But, ultimately, only one of these types will be the true effect type. Since multiple condition-effect pairs are allowed, this assumption will be required to allow the learning algorithm to know when it still needs more examples to disambiguate between different effect types.

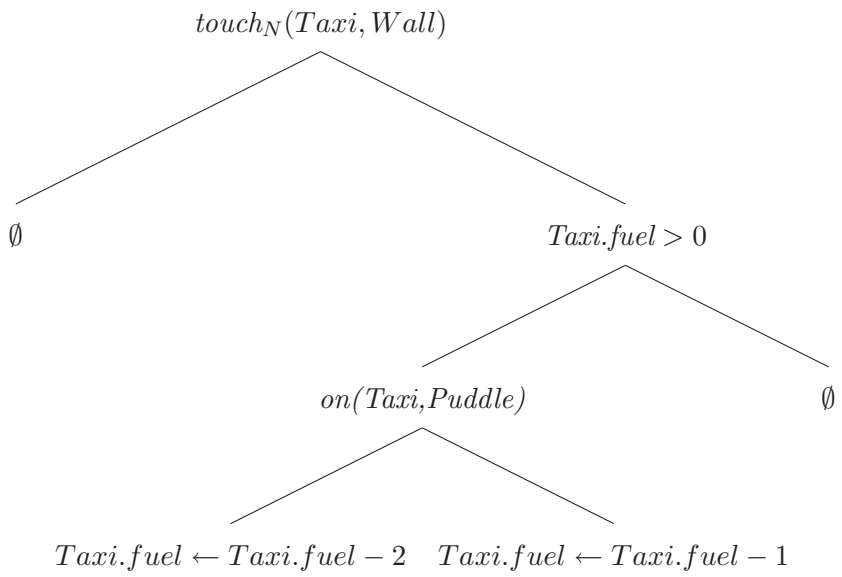
Assumption 9. *For every action a, attribute att and effect type t, there is a set $\mathcal{CE}_{t,a}^{att}$ of* condition–effect *pairs that determine changes to att given a. No effect can appear twice on this list, and there are at most k different pairs—$|\mathcal{CE}_{t,a}^{att}| \leq k$. Plus, no conditions T_i and T_j in the set $\mathcal{CE}_{t,a}^{att}$ contain each other: $\neg(T_i \subset T_j \vee T_j \subset T_i)$. The number of terms or negations of terms in any condition is bounded by a known constant D.*

This assumption is what gives the model the name of *Tree model*. We can think of this model as consisting of a tree with at most k leaves and depth D. Each leaf represents an effect and the internal nodes are the terms of the condition that enables the corresponding effect. Each leaf must be unique, and the tree structure prevents

conditions from overlapping. For example, the following tree represents the conditions and effects for action *North* and attribute *Taxi.y*.



In the case of attribute *Taxi.fuel*, the rules are:



5.5.1 Multiple Condition-Effects Learner Algorithm

I now present an algorithm, *MCELearner*, that learns multiple condition-effect pairs, assuming transition dynamics follow the Tree Model. The algorithm has two main components: prediction and learning. Both components share access to a set of lists of condition effects $\mathcal{CE}_{t,a}^{att}$ for each attribute *att*, effect type t and action a. Each element of these lists is a single condition-effect learner —a CELearner—, as presented in Section 5.4.

The learning sub-algorithm distributes its observations to the corresponding CE-Learners, and controls that the assumptions of the Tree Model hold. If for any effect type the number of CELearners is greater than k, or if their conditions overlap, it eliminates them. Details are shown in Algorithm 9.

Algorithm 9: MCELearner learning component.

Inputs: s, a, s'.
for all Attributes *att* **do**
  for all Effects $e \in \mathit{eff}_{att}(s, s')$ **do**
    $t \leftarrow e.type$
    Add experience to all condition-effect learners in set $\mathcal{CE}^{att}_{t,a}(pred(s), e)$.
    if $|\mathcal{CE}^{att}_{t,a}| > k$ **then**
      Delete $\mathcal{CE}^{att}_{t,a}$.
    end if
    if Two conditions overlap: $\exists c_i, c_j \in \mathcal{CE}^{att}_{t,a} s.t. c_i \models c_j \vee c_j \models c_i$ **then**
      Delete $\mathcal{CE}^{att}_{t,a}$.
    end if
  end for
end for

The prediction sub-algorithm will ask, for each attribute and effect type, for predictions from the corresponding CELearners. If any of them responds $\perp$, or if their predictions contradict, then it responds $\perp$. Details are shown in Algorithm 10.

Algorithm 10: MCELearner prediction component.

Inputs: s, a.
$att' \leftarrow att$
for all Attributes *att* **do**
  for all Effect types t **do**
    Get prediction p from $\mathcal{CE}^{att}_{t,a}(pred(s))$.
    if $p = \perp$ **then**
      Return $\perp$
    else
      Apply p to att'. If contradiction with current att', return $\perp$.
    end if
  end for
end for
Return s'.

5.5.2 Disjunctions and Effect Identifiability

The common assumption in this and upcoming chapters is that the conditions that determine domain dynamics are conjunctions. A disjunction of two conditions producing the same effect is not allowed, although it could be desirable. While the formalism does not disallow disjunctions per se, I have not been able to find an efficient algorithm to

learn them.

Based on the tree model, one might think that it could be possible to introduce disjunctions by means of a simple trick. Since effects are defined arbitrarily by the agent designer (as long as an adequate $\mathit{eff}_{att}(s, s')$ function is available), it is possible to create equivalent effects but give them different names. Imagine a designer deciding that there are two effects, called *add* and *superadd*, both producing the same arithmetic additions. There would be two branches in the tree, one ending in a leaf representing the effect *add*, and the other one ending in *superadd*. The branches would represent two conditions in a disjunction.

The problem with this idea is that an assumption of the algorithms proposed here is that effects can be identifiable. That is, all algorithms are able to tell when two actions produced equivalent effects. For example, a transition of an attribute's value from 2 to 3 and from 5 to 6 both produce an arithmetic-type effect +1, and all algorithms must recognize the different +1s as the same effect. This assumption is required so that the proper condition-effect pairs get updated under observations that produce the same effect, but may correspond to different states and different conditions. Faced with a +1 observation, the algorithm must be able to know whether to update the condition on the *add* or the *superadd* branches. It must not update both, or just one of them randomly, because then the condition learned will be nonsensical.

The only alternative is for the designer to introduce further prior knowledge, and implement the $\mathit{eff}_{att}(s, s')$ function in a way that distinguishes between equivalent effects with different names. If $\mathit{eff}_{att}(s, s')$ only and correctly produces *add* or *superadd* when they correspond (based on knowledge of which part of the disjunction should be updated given s and s'), then disjunctions can be learned and represented.

5.6 DOO$R_{\max}$

DOO$R_{\max}$ is an instance of KWIK-$R_{\max}$ that assumes that conditions and effects follow the Tree Model, and uses MCELearner for learning transitions. In this section, I present bounds and experiments.

5.6.1 Analysis

As mentioned before, there is a worst-case bound for learning conditions of $O(2^{|\mathcal{T}|})$, exponential in the number of terms involved in the conjunctions that determine transition dynamics. Consider the following example (sometimes referred to as the Combination Lock Problem): a domain contains a locked door, and can only be opened by setting the lock to its right combination. Assume the combination is an n-bit binary number, and the agent can perform the actions *Open* and *Flip-bit(i)*. The condition for action *Open* to produce the effect of opening the door is a conjunction of n terms, representing the correct combination. In the worst case, 2^n attempts (*Flip-bit* followed by *Open* actions) will have to be performed before this condition is learned.

The worst-case bound is reached when lots of negative examples (actions that have no effect) are observed, and conditions need to be eliminated one by one. However, in the face of positive examples (actions that do produce an effect), it is possible to eliminate an exponential number of hypotheses per observation. In the remainder of this section I analyze this best-case bound.

I split the proof in two parts. First, I show that learning the right condition-effect pair for a single action and attribute is KWIK-learnable, and then show that learning the right effect type for each action–attribute, given all the possible effect types, is also KWIK learnable.

Theorem 10. *The transition model for a given action a, attribute att and effect type $type$ in a deterministic OO-MDP is KWIK-learnable with a bound of $O(nk + k + 1)$, where n is the number of terms in a condition and k is the maximum number of effects per action–attribute.*

Proof. Given state s and action a, the predictor for effect type *type* will return $\perp$ if $pred(s)$ is not a known failure condition and there is no condition in $pred(a, att, type)$ that matches $pred(s)$. In that case, it gets to observe s' and updates its model with $pred(s)$ and the observed effect e. We show that the number of times the model can be updated until it always has a correct prediction is $O(nk + k + 1)$.

- If the effect e has never been observed before for this particular action, attribute and effect type, it gets added to $pred(a, att, type)$. This outcome happens at most k times, which is the maximum number of different effects allowed per action-attribute-type combination.

- If the effect e has never been observed, but $|pred(a, att, type)| = k$, the algorithm concludes that the current effect type is not the correct one for this action–attribute, and it removes all predictions of this type from its set $\mathcal{P}$. This event can only happen once for each type.

- if the effect e is such that there already exists a prediction for it, $\perp$ is only returned if the existing condition in the model does not match $pred(s)$. This case can only happen if a term in the model is a 0 or 1 and the observation is the opposite. Once it happens, that term becomes a $\star$, so there will never be another mismatch for that term, as $\star$ matches either 0 or 1. In the worst case, with every $\perp$ returned, one term at a time gets converted into $\star$. These updates can only happen n times for each effect in $pred(a, att, type)$, for a total of nk times.

Therefore, there can be at most $nk + k + 1$ updates to the model for a particular action a, attribute att and effect type $type$ before $pred(a, att, type)$ either has a correct prediction or gets eliminated. □

Corollary 11. *The transition model for a given action and attribute in a deterministic OO-MDPs is KWIK-learnable with a bound of $O(h(nk + k + 1) + (h - 1))$, where n is the number of terms in a condition, k is the max number of effects per action–attribute, and h is the number of effect types.*

Proof. Whenever DOO$R_{\max}$ needs to predict s' given state s and action a, it will consult its current predictions for each attribute and effect type. It will return $\perp$ if:

- For any of the h effect types $type_i$, $pred(a, att, type_i)$ returns $\perp$. As shown in Theorem 10, $pred(a, att, type_i)$ can only return $\perp$ up to $nk+k+1$ times. Therefore, this case can only happen $h(nk + k + 1)$ times.

- For some attribute *att*, there are two effect types $type_1$ and $type_2$ such that $pred(a, att, type_1) \neq pred(a, att, type_2)$. When this happens, we get to observe the actual effect e, which will necessarily mismatch one of the predictions. The model will therefore be updated by removing either $pred(a, att, type_1)$ or $pred(a, att, type_2)$ from its set of predictions. This case can only occur $h-1$ times for a given action and attribute.

We have shown that, in total and in the best case, DOO$R_{\max}$ will only predict $\perp$ $O(h(nk + k + 1) + (h - 1))$ times before having an accurate model of the transition dynamics for an action and attribute in the OO-MDP. □

5.7 Experiments

In this section, I present experimental results on two domains: the well-known Taxi domain, already presented in Chapter 3, and an Atari console videogame Pitfall.

5.7.1 Taxi

The Taxi domain was introduced in Section 3.1, and the experimental setup described there is the one I used to test DOO$R_{\max}$. Some of the results were also anticipated in that chapter, but are rehashed here.

We run experiments on two versions: the original 5×5-grid version presented by Dietterich (2000), which consists of 500 states, and an extended 10×10-grid version with 8 passenger locations and destinations, with 7200 states (see Figure 5.4). The purpose of the extended version is to demonstrate how DOO$R_{\max}$ scales by properly generalizing its knowledge about conditions and effects when more objects of the same known classes are introduced.

The set of terms T, which determines the transition dynamics of the OO-MDP, includes the four $touch_{N/S/E/W}$ relations between the taxi and the walls; the relevant relations between the taxi and the passenger and destination; the attribute value $passenger.in\text{-}taxi = T$; and all their negations:

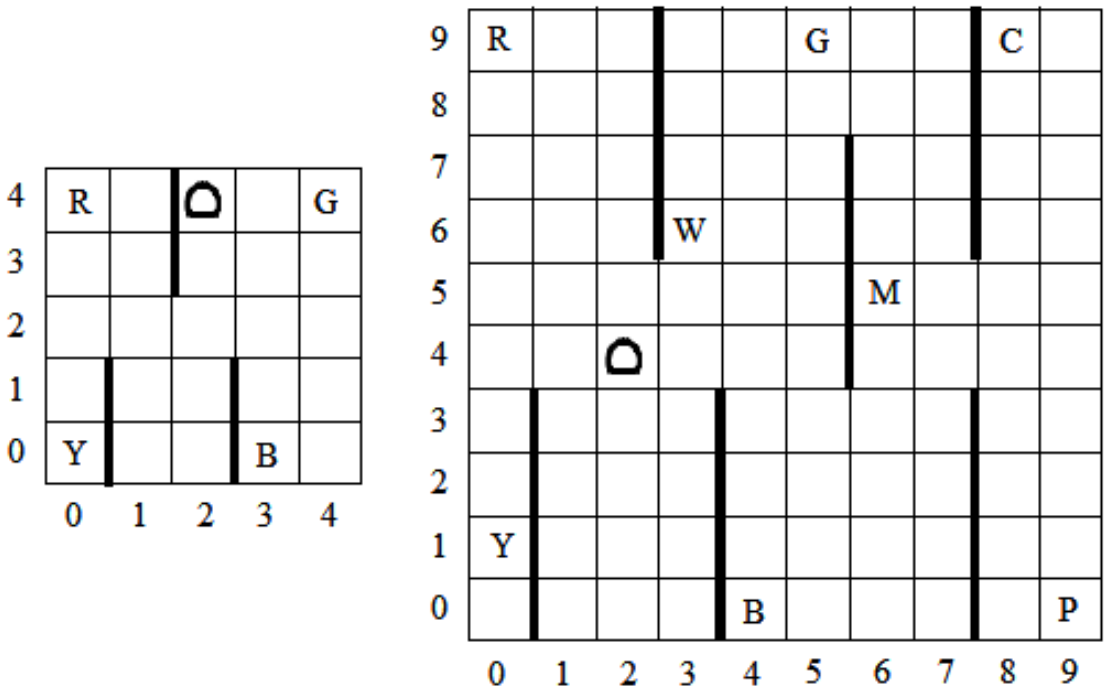


Figure 5.4: Standard 5×5 Taxi and extended 10×10 version.

$$\{ \quad touch_{N/S/E/W}(taxi, wall), \quad on(taxi, passenger),$$
$$\neg touch_{N/S/E/W}(taxi, wall), \quad \neg on(taxi, passenger),$$
$$on(taxi, destination), \quad \neg on(taxi, destination),$$
$$passenger.in\text{-}taxi = T, \quad passenger.in\text{-}taxi = F \ \}$$

The experiments for both versions of the Taxi problem were repeated 100 times, and the results averaged. For each experiment, I ran a series of episodes, each starting from a random start state. We evaluate the agent's learned policy after each episode on a set of six "probe" combinations of ⟨taxi (x,y) location, passenger location, passenger destination⟩. The probe states used were: $\{(2,2), Y, R\}$, $\{(2,2), Y, G\}$, $\{(2,2), Y, B\}$, $\{(2,2), R, B\}$, $\{(0,4), Y, R\}$, $\{(0,3), B, G\}$. We report the number of steps taken before learning an optimal policy for these six start states.

We also remind the reader here of how DOO$R_{\max}$ compares against *Factored-Rmax*, the state-of-the-art algorithm for factored-state representations. The results are shown in the following table, with the last column showing the ratio between the results for the 10×10 version and the 5×5 one:

	Taxi 5×5	Taxi 10×10	Ratio
Number of states	500	7200	14.40
Factored Rmax			
# steps	1676	19866	11.85
Time per step	43.59ms	306.71ms	7.03
DOO$R_{\max}$			
# steps	529	821	**1.55**
Time per step	13.88ms	293.72ms	21.16

We can see how DOO$R_{\max}$ not only learns with significantly smaller sample complexity, but also how well it scales to the larger problem. After increasing the number of states by more than 14 times, DOO$R_{\max}$ only requires 1.55 times the experience. This result can be explained by the fact that what DOO$R_{\max}$ needs to learn are the interactions between objects, which do not change from the smaller to the larger Taxi problem. The difference in sample complexity can be explained by the fact that in the 10×10 grid, more steps need to be taken in order to generate the necessary exploratory interactions, pick up the passenger, deliver it, etc.

The main difference between DOO$R_{\max}$ and *Factored-Rmax* is their internal representation, and the kind of generalization it enables. After just a few examples in which $\neg touch_N(taxi, wall)$ is true, DOO$R_{\max}$ learns that the action *North* has the effect of incrementing $taxi.y$ by 1, whereas under $touch_N(taxi, wall)$ it fails. This knowledge, as well as its equivalent for $touch_{S/E/W}$, is generalized to all 25 (or 100) different locations. *Factored-Rmax* only knows that variable $taxi.y'$ in state s' depends on its value in state s, but still needs to learn the transition dynamics for each possible value of $taxi.y$ (5 or 10 different values). In the case of actions *East* and *West*, the situation is even worse, as walls make $taxi.x'$ depend on both $taxi.x$ and $taxi.y$, which are 25 (or 100) different values.

As DOO$R_{\max}$ is based on interactions between objects, it assumes that the relation between taxi and wall is independent of the wall location. Each new wall is therefore the same as any known wall, rather than a new exception in the movement rules, the

kind *Factored-Rmax* needs to learn.

5.7.2 Pitfall

Pitfall was previously introduced in Section 4.1. The task is to have the main character (*Harry*) traverse a series of screens while collecting as many points as possible while avoiding obstacles (such as holes, pits, logs, crocodiles and walls) and under the time constraint of 20 minutes. All transitions in Pitfall are deterministic. My goal in this experiment is to have *Harry* cross the first screen from the left to the right with as few actions as possible. Figure 4.1 illustrates this first screen.

Experiments were run using a modified Atari 2600 emulator that ran the actual game and detected objects from the displayed image. The necessary extensions to the emulator and the object-detection scheme were designed by Andre Cohen (Diuk et al., 2008). He used a simple heuristic that identifies objects by color clusters and sends joystick commands to the emulator to influence the play. For each frame of the game, a list of object locations is sent to an external learning module that analyzes the state of the game and returns an action to be executed before the emulator continues on to the next frame. If we consider that we start from screen pixels, the flat state representation for Pitfall is enormous: $16^{640\times420}$. By breaking it down into basic objects, through an object recognition mechanism, the state space is in the order of the number of objects to the number of possible locations of each object: $(640 \times 420)^6$, a reduction of $O(10^5)$. To be fair, it is worth noting that only a few of these states are actually reachable during game play. OO-MDPs allow for a very succinct representation of the problem, that can be learned with only a few experience samples.

The first screen contains six object types: *Harry, Hole, Ladder, Log, Wall* and *Tree*. Objects have the attributes x, y, *width* and *height*, which define their location on the screen and dimension. The class *Harry*also has a Boolean attribute of *direction* that specifies which way he is facing. We extended the $touch_X$ relation from Taxi to describe diagonal relations between objects, including: $touch_{NE}(o_i, o_j)$, $touch_{NW}(o_i, o_j)$, $touch_{SW}(o_i, o_j)$ and $touch_{SE}(o_i, o_j)$. These relations were needed to properly capture

the effects of moving on and off of ladders.

In the implementation of DOO$R_{\max}$, seven actions are defined: *StickRight, StickLeft, JumpLeft, JumpRight, Up, Down* and *JumpUp*. For each of these actions, however, the emulator has to actually execute a set sequence of smaller frame-specific actions. For example, *StickLeft* requires four frames: one to tell Pitfall to move *Harry* to the left, and three frames where no action is taken to allow for the animation of *Harry* to complete. In the learner, effects are represented as arithmetic increments or decrements to the attributes x, y, *width*, *height*, plus a constant assignment of either R or L to the attribute *direction*.

The starting state of Pitfall is fixed, and given that all transitions are deterministic, only one run of DOO$R_{\max}$ was necessary to learn the dynamics of the environment. DOO$R_{\max}$ learns an optimal policy after 494 actions, or 4810 game frames, exploring the area beneath the ground as well as the objects en route to the goal. Once the transition dynamics are learned, restarting the game results in the *Harry* exiting the first screen through the right, after jumping the hole and the log, in 94 actions (905 real game frames). Faced with a similar screen, the agent would be able to escape without any extra actions.

Planning in Pitfall was done with a forward planner, akin to sparse sampling (see 2.3.2). A heuristic function was used to extend the search through branches in the tree that took *Harry* closer to the right of the screen (those that increased x).

A few examples of the (condition, effect) pairs learned by DOO$R_{\max}$ are shown below:

Action	Condition	Effects
StickRight	direction $= L$	{direction $= R$, $\Delta x = +8$}
StickRight	$touch_E(Harry, Wall)$	$\emptyset$
JumpRight	direction $= R$	{ $\Delta x = +214$ }
Up	$on(Harry, Ladder)$	$\{\Delta y = +8\}$

5.8 Discussion

In this chapter, I showed how deterministic OO-MDPs can be learned, under certain assumptions, very fast. Some of the assumptions, like the Tree Model, might seem restrictive. But, by means of the OO-MDP representation, and under these assumptions, a large and challenging real-life problem like Pitfall was tackled, serving as a proof-of-concept of the power of the proposed approach.

There is still a worst-case scenario in the analysis of DOO$R_{\max}$ that leads to an exponential bound. In the next chapter, I introduce a provable efficient algorithm for stochastic OO-MDPs, which can of course be used to learn deterministic ones as a special case. However, when the assumptions for DOO$R_{\max}$ hold and the worst-case conditions are not present in a given domain, it will be much faster than the general algorithm and should be preferred.

Chapter 6

Learning Stochastic OO-MDPs

> Improbable as it may be, no one had until then attempted to set up a general theory of games. A Babylonian is not highly speculative. He reveres the judgments of fate, he hands his life over to them, he places his hopes, his panic terror in them, but it never occurs to him to investigate their labyrinthian laws nor the giratory spheres which disclose them.
>
> Jorge Luis Borges (1899-1986), The Babylon Lottery.

In Chapter 5, I showed how, under a set of assumptions, deterministic OO-MDPs could be learned efficiently in the presence of positive examples. In this chapter, I will relax many of the prior assumptions (albeit adding some new ones) and show how efficient learning is still possible. The main relaxation, which enables the representation of a larger class of problems, is that effects need not be deterministic anymore. That is, in this chapter, I will consider a problem in which conditions induce a distribution over sets of effects. Unlike the Babylonians from this chapter's epigraph, our learner will try to understand and explain the "labyrinthian" laws governing its world's randomness.

The problem of learning stochastic OO-MDPs will be broken down into three components: learning conditions, learning what their effects are, and learning the probability with which each effect occurs. The chapter will start with the problem of learning conditions and show that it can be done efficiently. The main extra assumption will be that the maximum number of terms involved in any condition is known, an assumption not uncommon in a family of problems to which conjunction learning belongs. The following part of the problem will assume that effects are known, and we only need to learn their probability of occurring under each condition. This part, too, will be provably efficiently learnable. Finally, the problem of learning what the effects are will

be presented with a heuristic solution.

6.1 Learning conditions

The problem of learning conditions in Propositional OO-MDPs is the problem of learning conjunctions. That is, learning which terms out of all possible n ones need to be simultaneously true in order to enable an effect. But, learning conjunctions is hard! Consider a learner that receives as input a conjunction of n terms and needs to predict a binary label that depends on (part of) that conjunction. Assume also that labels are not arbitrarily attached to each individual conjunction of size n, but that generalization is possible. For example, the output label might only depend on $D << n$ of the terms involved. Ideally, we would like to have learners that do not need to experience examples of all possible 2^n conjunctions in order to make correct predictions with high probability.

Let's return to the example already introduced in Chapter 5: an environment includes a *combination lock* of n tumblers, and the agent needs to unlock it in order to reach some high reward area of the state space. Under the OO-MDP representation, an action *Unlock* only succeeds (induces the effect that the lock unlocks) if the n-term condition corresponding to the state of the tumblers matches the right combination. In the worst case, learning the correct conjunction might require trying out all possible 2^n combinations.

Kearns and Vazirani (1994) showed that learning conjunctions is possible in the PAC learning framework. The assumption in this case is that the learner is presented with examples that are independent and identically distributed (iid), generated from a fixed distribution. The algorithm, like DOO$R_{\max}$, starts by assuming all literals are necessary for the conjunction to hold. For each positive example, the learner eliminates literals in its hypothesis that contradict the positive example. For negative examples, the hypothesis will always be more specific than the example presented, and will thus guarantee that the learner does predict a correct negative outcome. Given that the PAC learner will never err on negative examples, and it will learn from positive ones, it

is possible to bound, using the iid distributional assumption, the maximum number of examples necessary until the conjunction is correctly learned with high probability. Unfortunately, this distributional assumption is not available in a reinforcement-learning setting. In the adversarial KWIK case, the environment could present all $2^n - 1$ negative examples first, forcing the agent to respond with an exponential number of $\perp$s. For a formalization and proof of the PAC-learnability of conjunctions, see Kearns and Vazirani (1994).

Conjunctions are also learnable in the Mistake Bound (MB) framework. Remember that in this framework learners are allowed to make guesses, and their performance is only measured in terms of the number of wrong predictions (mistakes) they make. A learner in this setting could always predict that a conjunction will fail. In the combination lock, it could always predict that the lock will *not* open. Each time this prediction is correct, no penalty is incurred. Whenever the learner makes its first mistake, it will be told so. That is, when faced with the true combination, it will predict it won't open and be told it's mistaken. It will therefore, with only 1 mistake counted against it, learn the proper combination. The MB framework is not suitable for our goals in reinforcement learning either: We want our agent to act in the world without supervision. Predicting that no combination will open the lock would result in an agent that doesn't try to open it, and a teacher is not present to tell it that it should. The only way to break the lock is to actually try out combinations (all 2^n of them) , and learn once the lock unlocks (for a setting in which a teacher is present in a reinforcement-learning setting, making conjunctions efficiently learnable, see (Walsh et al., 2010)).

The desiderata for this chapter is, then, to present the necessary assumptions to make conjunctions KWIK-learnable. The main assumption for the remainder of this chapter and the algorithm that will be presented is that conjunctions only depend on a known (and small) number of terms, D, where $D << n$. In the worst case, the agent should produce $O(2^D)$ "$\perp$" responses. I will now present a learning algorithm, called the Adaptive k-Meteorologists, and show it can be used to learn conjunctions of size D

efficiently in the KWIK framework.

6.1.1 The k-Meteorologists

The k-Meteorologist problem was introduced by Diuk et al. (2009), as a formalization and extension of ideas presented in the original KWIK paper (Li et al., 2008). Let us start through an intuitive presentation of what the k-Meteorologists problem is.

Imagine that you just moved to a new town that has multiple (k) radio and TV stations. Each morning, you tune in to one of the stations to find out what the weather will be like (to make it simpler, let's assume you only want to know the chance of rain). Which of the k different meteorologists making predictions every morning is trustworthy? Let us imagine that, to decide on the best meteorologist, each morning for the first M days you tune in to all k stations and write down the probability that each meteorologist assigns to the chances of rain. Then, every evening you write down a 1 if it rained, and a 0 if it didn't. How can this data be used to determine who the right meteorologist is?

In this section, I will present a solution to the problem, as developed by Diuk et al. (2009). In the next section, I will show how to construct a set of meteorologists where each of them represents a different OO-MDP condition, and the meteorologist that contemplates the correct condition for a given action is the one that will be determined to be the best one.

Probabilistic concepts

In the previous example, each meteorologist is allowed to provide a number representing the probability that it will rain, rather than a binary guess as to whether it will rain or not. Such predictions are termed *probabilistic concepts* (Kearns & Schapire, 1994; Yamanishi, 1992). They extend the notion of deterministic concepts (as most of the machine-learning literature considers) by allowing an instance or example to belong to a class with a certain probability.

Probabilistic concepts are a useful generalization of deterministic concepts and are

able to capture uncertainty in many real-life problems, such as the weather broadcasting example described in the previous section. Formally, a probabilistic concept h is a function that maps an input space X to the output space $Y = [0, 1]$; $h : X \mapsto Y$. In the meteorologist example, every $x \in X$ corresponds to the features that can be used to predict chances of rain, and $h(x)$ indicates the probability that x is in the concept. Namely, it predicts the chance that it will rain on that day. The hypothesis class H is a set of probabilistic concepts: $H \subseteq (X \to Y)$.

Using tools from statistical learning theory, Kearns and Schapire (1994) study how to learn probabilistic concepts in the PAC model, which assumes that learning is done through iid samples and there is therefore no active exploration involved. To incorporate efficient exploration, I present how the problem can be formulated and solved in the KWIK framework.

The (Adaptive) k-Meteorologists Problem

In the k-Meteorologist Problem (Diuk et al., 2009), the learner is given a finite set of k probabilistic concepts: $H = \{h_1, h_2, \ldots, h_k\}$, where $h_i : X \to Y$ for all $i = 1, \ldots, k$. The task of KWIK-learning a target concept $h^* \in H$ can be understood as one of identifying the true but unknown concept from a set of k candidates.

In some learning problems, the candidate concepts, h_i, are not provided as input. Instead, they have to be learned by the learner itself. This motivates a more general version of the k-Meteorologists Problem, which is termed as the Adaptive k-Meteorologists Problem. Here, the learner is given k classes of hypotheses, $H_1, \ldots, H_k$, and also provided with k sub-algorithms, $\mathcal{A}_1, \ldots, \mathcal{A}_k$, for KWIK-learning these classes. The goal of the learner is to make use of these sub-algorithms to KWIK-learn the *union* of these hypothesis classes: $H = H_1 \cup \cdots \cup H_k$.

Solution

The k-Meteorologists Problem is a special case of the Adaptive k-Meteorologists Problem where every hypothesis class H_i contains exactly one hypothesis: $H_i = \{h_i\}$. For

the sake of simplicity, let us start with the simpler k-Meteorologists Problem to explain the intuition behind the algorithm, and then provide detailed pseudo-code descriptions for the adaptive version.

The major challenge in the k-Meteorologists Problem is that the learner only observes stochastic binary labels while it is required to make predictions about the label probabilities. A natural idea is to get sufficient labels for the same input x and then estimate $\Pr(z = 1|x)$ by their relative frequency. But, since inputs may be drawn adversarially, this approach must have a sample complexity of $\Omega(|X|)$.

We can however expand an idea outlined by Li et al. (2008) to avoid the dependence on the size of X. Suppose $z_t \in \{0, 1\}$ is the label acquired in timestep t. Define the squared error of meteorologist h_i to be $e_t = (h_i(x_t) - z_t)^2$. We then maintain cumulative squared prediction errors for individual meteorologists. It can be shown that the target probabilistic concept, h^*, will have the smallest squared error *on average*. If any concept h_i has a much larger cumulative error than another concept h_j, it follows that $h_i \neq h^*$ with high probability. This trick now enables us to only need to observe enough data to have a good estimate of which meteorologist has the smallest squared error. Complete details are provided by Li (2009).

Algorithm 11 provides a solution to the Adaptive k-Meteorologists Problem, in which the additional parameter m will be specified in Theorem 12. Essentially, the algorithm runs all the k sub-algorithms simultaneously and does all $\binom{k}{2}$ pairwise comparisons among the k probabilistic concepts. If any probabilistic concept returns $\perp$, the algorithm outputs $\perp$ and obtains a stochastic observation z_t to allow the sub-algorithms to learn (Lines 7–9). Now, suppose no probabilistic concept returns $\perp$. If the set of predictions is consistent then an accurate prediction can be made (Line 12) although the algorithm does not know which concept is h^*. Otherwise, the algorithm outputs $\perp$ and then acquires a label, which contributes to distinguishing at least one pair of meteorologists (Lines 15–21). A candidate concept is removed if there is statistically significant evidence that it is worse than another concept (Line 19). This observation is in fact the intuition behind the proof of the algorithm's sample complexity.

Algorithm 11: The Adaptive k-Meteorologists Algorithm.

1: Input: ϵ, δ, m, $H_1, \ldots, H_k$, $\mathcal{A}_1, \ldots, \mathcal{A}_k$.
2: Run each subalgorithm $\mathcal{A}_i$ with parameters $\frac{\epsilon}{8}$ and $\frac{\delta}{k+1}$.
3: $R \leftarrow \{1, 2, \ldots, k\}$.
4: $c_{ij} \leftarrow 0$ and $\Delta_{ij} \leftarrow 0$ for all $1 \leq i < j \leq n$.
5: **for** $t = 1, 2, 3, \ldots$ **do**
6: Obtain x_t and run each $\mathcal{A}_i$ to get its prediction, $\hat{y}_{ti}$.
7: **if** $\hat{y}_{ti} = \perp$ for some $i \in R$ **then**
8: Let $\hat{y}_t = \perp$ and observe $z_t \in \mathbb{Z}$.
9: Send z_t to all subalgorithms $\mathcal{A}_i$ with $\hat{y}_{ti} = \perp$.
10: **else**
11: **if** $|\hat{y}_{ti} - \hat{y}_{tj}| \leq \epsilon$ for all $i, j \in R$ **then**
12: Let $\hat{y}_t = (\max_{i \in R} \hat{y}_{ti} + \min_{i \in R} \hat{y}_{ti})/2$.
13: **else**
14: Let $\hat{y}_t = \perp$ and observe z_t.
15: **for all** $i, j \in R$ such that $|\hat{y}_{ti} - \hat{y}_{tj}| \geq \frac{\epsilon}{2}$ **do**
16: $c_{ij} \leftarrow c_{ij} + 1$.
17: $\Delta_{ij} \leftarrow \Delta_{ij} + (\hat{y}_{ti} - z_t)^2 - (\hat{y}_{tj} - z_t)^2$.
18: **if** $c_{ij} \geq m$ **then**
19: $R \leftarrow R \setminus \{I\}$ where $I = i$ if $\Delta_{ij} > 0$ and $I = j$ otherwise.
20: **end if**
21: **end for**
22: **end if**
23: **end if**
24: **end for**

Analysis

I will now present *matching* upper and lower sample-complexity bounds for Algorithm 11. Complete details of the proofs can be found in the dissertation of Li (2009), so I will only sketch them here.

Observe that every $\perp$ output by Algorithm 11 is either from some sub-algorithm (Line 8) or from the main algorithm when it gets inconsistent predictions from different probabilistic concepts (Line 14). Thus, the sample complexity of Algorithm 11 is at least the sum of the sample complexities of those sub-algorithms plus the additional $\perp$s required to figure out the true h^* among the k candidates. The following theorem formalizes this observation:

Theorem 12. *Let $\zeta_i(\cdot, \cdot)$ be a sample complexity of sub-algorithm $\mathcal{A}_i$. By setting $m =$*

$O\left(\frac{1}{\epsilon^2}\ln\frac{k}{\delta}\right)$, *the sample complexity of Algorithm 11 is at most*

$$\zeta^*(\epsilon,\delta) = O\left(\frac{k}{\epsilon^2}\ln\frac{k}{\delta}\right) + \sum_{i=1}^{k}\zeta_i\left(\frac{\epsilon}{8},\frac{\delta}{k+1}\right).$$

Proof. (sketch) The proof has four steps. First, we show that the squared error of the target hypothesis must be the smallest *on average*. Second, if some hypothesis is $\frac{\epsilon}{8}$-accurate (as required by Line 2 in Algorithm 11), its average squared error is still very close to the average squared error of the predictions of h^*. Third, by setting m appropriately (as given in the theorem statement), we can guarantee that only sub-optimal hypotheses are eliminated in Line 19 with high probability, by Hoeffding's inequality. Finally, the condition in Line 15 guarantees that the total number of $\perp$s output in Line 14 is bounded by the first term in the desired bound of the theorem. □

Theorem 12 indicates that the additional sample complexity introduced by Algorithm 11, compared to the unavoidable term, $\sum_i \zeta_i$, is on the order of $\frac{k}{\epsilon^2}\ln\frac{k}{\delta}$. The following theorem gives a matching lower bound (modulo constants), implying the optimality of Algorithm 11 in this sense.

Theorem 13. *A sample-complexity lower bound for the k-Meteorologists Problem is*

$$\zeta_*(\epsilon,\delta) = \Omega\left(\frac{k}{\epsilon^2}\ln\frac{k}{\delta}\right).$$

Proof. (sketch) The proof is through a reduction from 2-armed bandits to the k-Meteorologists Problem. The idea is to construct input–observation pairs in the KWIK run so that the first $k-1$ hypotheses, $h_1,\ldots,h_{k-1}$, have to be eliminated one by one before the target hypothesis, $h^* = h_k$, is discovered. Each elimination of h_i (for $i<k$) can be turned into identifying a sub-optimal arm in a 2-armed bandit problem, which requires $\Omega(\frac{1}{\epsilon^2}\ln\frac{1}{\delta})$ sample complexity (Mannor & Tsitsiklis, 2004). Based on this lower bound, we may prove this theorem by requiring that the total failure probability in solving the k-Meteorologists Problem is δ. □

6.1.2 Learning Conditions using Meteorologists

In upcoming sections, I will present a complete learning algorithm for stochastic OO-MDPs, where conditions and effects are learned simultaneously. In this section, I will only focus on learning conditions, but not the effects. For simplicity, let us assume there is only one action to be taken, and the action has a certain probability of producing an effect e if a particular condition is met. The assumption is that this condition is a conjunction of at most D terms out of all possible n ones, and D is provided as input. Note that this assumption implies that two conditions c_i and c_j, each of size D, cannot be true simultaneously and have contradictory effects for any given action and attribute. If such were the case, then the dynamics are not truly dependent on just D terms, but possibly on the terms involved in the union of c_i and c_j.

The set of all possible conditions for action a and effect e defined in this way has $\binom{n}{D}$ elements. If each condition is now assigned to a different meteorologist, we can simply initialize Algorithm 11 with $k = \binom{n}{D}$ hypothesis, each considering a different set of D terms, and the meteorologist making the best prediction will be the one representing the correct condition for the corresponding action and effect.

Experiments

Assume that in the Taxi problem we want to learn the condition that enables action North to move the taxi one position to the north. Also, assume that Taxi is now stochastic, and under the condition $\neg touch_N(Taxi, Wall)$, action North only moves the taxi 80% of the times, and the other 20% it stays where it is.

If we know that only one term is involved in the condition, we can create 8 meteorologists representing conditions $touch_N(Taxi, Wall)$, $touch_S(Taxi, Wall)$, $touch_E(Taxi, Wall)$, $touch_W(Taxi, Wall)$, $\neg touch_N(Taxi, Wall)$, $\neg touch_S(Taxi, Wall)$, $\neg touch_E(Taxi, Wall)$ and $\neg touch_W(Taxi, Wall)$. The observation will be 1 if the taxi moved, 0 otherwise.

As an implementation detail, it is also possible to use the Adaptive k-Meteorologists the following way: just create 4 learners considering the conditions

$touch_{N/S/E/W}(Taxi, Wall)$, and the corresponding learning algorithms $\mathcal{A}_{N/S/E/W}$ will distinguish the case when the considered terms are true or false. This implementation is the one run for this experiment, and the following table shows what probabilities each of the 4 meteorologists assigned to each outcome and their mean squared error. The value of m was set to 20.

Condition	P(move N)	P(move S)	P(move E)	P(move W)	P(stay)	Error
$touch_N$	0.85	0	0	0	0.15	0.338
$touch_S$	0.45	0	0	0	0.55	0.559
$touch_E$	0.45	0	0	0	0.55	0.550
$touch_W$	0.45	0	0	0	0.55	0.650

As you can observe, the correct meteorologist, the one considering condition $touch_N$, is the one with the lowest squared error and the closest prediction to the true outcome, which is 0.80 for moving North and 0.2 for staying in place.

Note that in this particular example outcomes are observed unambiguously. That is, after each action North is taken, the learner is told which of the 5 possible outcomes (moved N/S/E/W or stay) occurred and can therefore estimate probabilities directly. In general, it might not be the case that observations are unambiguous: multiple outcomes could lead to the same next state, and the learner might not know to which outcome attribute the change. The next section addresses this issue.

6.2 Learning effect probabilities

As stated at the beginning of this chapter, the second aspect of the problem of learning transition dynamics in OO-MDPs is learning the probabilities with which different effects occur given a condition. That is, we assume that for each condition the set of possible effects is known, and the problem is to learn a multinomial distribution over that set. A particular difficulty in this case is that effects might be ambiguous. That is, given a certain state, two effects might lead to the same observed outcome, making it impossible to exactly attribute a given experience to a single effect. For example, consider an action that can have the effect of either adding 2 to an attribute (+2) or multiplying it by 2 (∗2), each with a certain probability. Now consider a state where

this attribute's value is originally 2, and after taking the action it becomes 4. Does that observation correspond to a $+2$ or a $*2$ effect?

This problem was considered by Walsh et al. (2009), where an efficient KWIK solution is provided. I will present the main result of this paper and expand on its application to OO-MDPs.

6.2.1 KWIK Linear Regression

In this section, I will present an online approach to linear regression, and in the next one show how it can be applied to the problem of learning the probabilities of ambiguous effects, as it was just introduced.

Linear regression has been a powerful tool in machine learning and statistics for decades (Bishop, 2006). In the reinforcement-learning setting, a few challenges arise. First, regression has to be performed online: We should not have to rely on a batch of data being available, but rather expect to update our regressor as each new datapoint is discovered. Second, we can not make iid assumptions about the observed data when exploration is involved. This constraint makes it hard to automatically port linear regression methods to reinforcement-learning problems, especially if theoretical sample and computational efficiency guarantees are sought.

Walsh et al. (2009) introduced an online regression method that does not have iid requirements and can be proven to be KWIK. I will simply introduce the algorithm here and refer readers to the original paper for proofs. As in that paper, I will refer to this method as KWIK-LR.

First, I introduce some notation. Let $X := \{\vec{x} \in \mathbb{R}^n \mid \|\vec{x}\| \leq 1\}$, and let $f : X \to \mathbb{R}$ be a linear function with slope $\theta^* \in \mathbb{R}^n$, $\|\theta^*\| \leq M$. That is, $f(\vec{x}) := \vec{x}^T\theta^*$. Assume a discrete timestep, with current value 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 $z_i := \vec{x}_i^T\theta^* + \eta_i$. Here, η_i is a random variable with some unknown distribution (for example, Gaussian noise). The only necessary assumption is that η_i must form a martingale, that is, $E(\eta_i|\eta_1, \ldots, \eta_{i-1}) = \eta_{i-1}$, and is bounded: $|\eta_i| \leq S$. Define

the matrix $D_t := [\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_t]^T \in \mathbb{R}^{t \times n}$ and vectors $\vec{y}_t := [y_1; \ldots; y_t] \in \mathbb{R}^t$ and $\vec{z}_t := [z_1; \ldots; z_t] \in \mathbb{R}^t$, and let I be an $n \times n$ identity matrix.

Whenever a new data point is observed (a query $\vec{x}_t$ is presented by the environment), it would be possible to predict output $\vec{y} = \vec{x}^T\theta$, where θ is the least-squares solution to the system, if we could solve $D_t\theta = \vec{z}_t$. However, it might not be advisable to solve this system directly: if D_t, the inputs observed so far, is rank deficient, the system may have more than one solution; and even if there is a unique solution we have no confidence guarantees.

The solution presented by Walsh et al. (2009) solves these problems through a very simple regularization trick. The system is augmented with $I\theta = \vec{v}$, where $\vec{v}$ is an arbitrary vector (that is, the input vector becomes $\begin{bmatrix} I \\ D \end{bmatrix}$). This regularization trick distorts the solution, but thanks to this distortion we get a measure of confidence. If the distortion is large (this will be defined properly when the equations are presented), the learner does not yet have enough experience and should respond $\perp$ to the query.

Let us now incorporate this regularization trick into the linear system and see how prediction should work. We will define a new matrix that results from concatenating the identity to the input vectors: $A_m := \begin{bmatrix} I \\ D \end{bmatrix} \in \mathbb{R}^{(m+n)\times n}$. 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 $\hat{\theta} = (A_t^T A_t)^{-1} A_t^T [\vec{0}^T; \vec{z}_t^T]^T$. Define $L_t := (A_t^T A_t)^{-1}$, $\vec{b} := \begin{bmatrix} \theta^* \\ \vec{y} \end{bmatrix} \in \mathbb{R}^{m+n}$, $\vec{c} := \begin{bmatrix} \theta^* \\ \vec{z} \end{bmatrix} \in \mathbb{R}^{m+n}$, $\vec{d} := \begin{bmatrix} \vec{0} \\ \vec{z} \end{bmatrix} \in \mathbb{R}^{m+n}$.

The prediction error on input $\vec{x}$ is

$$\begin{aligned} \hat{y} - y &= \vec{x}^T(\hat{\theta} - \theta^*) \\ &= \vec{x}^T L_t A_t^T \left(\begin{bmatrix} \vec{0} \\ \vec{z}_t \end{bmatrix} - \begin{bmatrix} \theta^* \\ \vec{y}_t \end{bmatrix} \right) = \vec{x}^T L_t A_t^T \left(\begin{bmatrix} \vec{0} \\ \boldsymbol{\eta}_t \end{bmatrix} - \begin{bmatrix} \theta^* \\ \vec{0} \end{bmatrix} \right). \end{aligned} \tag{6.1}$$

Algorithm 12 describes KWIK-LR, using the prediction error described above.

Algorithm 12: KWIK-LR

input: α_0
initialize: $t := 0$, $m := 0$, $L := I$, $\vec{w} := \vec{0}$
repeat
  observe $\vec{x}_t$
  if $\|L\vec{x}_t\| < \alpha_0$ **then**
    predict $\hat{y}_t = \vec{x}_t^T L\vec{w}$ *//known state*
  else
    predict $\hat{y}_t = \perp$ *//unknown state*
    observe z_t
  end if
  $L := L - \frac{(L\vec{x}_t)(L\vec{x}_t)^T}{1+\vec{x}_t^T L\vec{x}_t}$, $\vec{w} := \vec{w} + \vec{x}_t z_t$
  $t := t + 1$
until there are no more samples

6.2.2 Using KWIK-LR to learn effect probabilities

In this section, I will show how KWIK-LR can be used to solve the problem of learning a probability distribution over a set of effects for a given action and condition, even when observations are ambiguous in terms of which effect actually occurred. Let us start with an example of a stochastic OO-MDP on a simple 5×5 Maze domain, illustrated in Figure 6.1. The agent starts at location S and the goal is to arrive at G. Each step has a cost of -0.01, and arriving at the goal results in a reward of $+1$. The agent's actions are N, S, E and W. When executing an action, the agent will attempt to move in the desired direction with probability 0.8 and will slip to either side with probability 0.1. If it hits a wall, it stays where it is. This rule is what produces ambiguity in the effects. For example, imagine the agent has a wall to its North and East. If it attempts the N action, it could move to the West (with probability 0.1), or stay in place. If it stays in place, it might be because it attempted to move North (with probability 0.8) and hit the North wall, or it attempted to move East (with probability 0.1) and hit the East wall. For the sake of this example, I am assuming that the function $\mathit{eff}_{att}(s, s')$ does know what type of movement might have been attempted even when $s' = s$.

In the example above, we can think that each state-action pair actually induces a

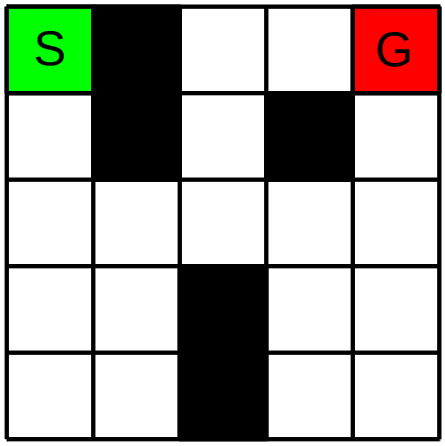


Figure 6.1: Stochastic Maze domain.

partition of effects 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. In the example of a state s_i that has a wall to its North and East, under action N the partition would be: $E(s_i, N) = \{\{move_N, move_E\}, \{move_W\}\}$. Notice that the probability of an equivalence class is equal to the sum of the probabilities of the effects it contains: the class $\{move_N, move_E\}$ has a probability 0.9 of being the effect that occurred, 0.8 for the probability of moving North plus 0.1 for the probability of slipping to the East. This is the crux of the link to linear regression. The efficient way to leverage KWIK-LR in this case is to construct inputs $\vec{x}$ where each of its elements x_j represents an effect, and $x_j = 1$ if the effect is in the equivalence class observed, or $x_j = 0$ otherwise. Back to the example, if $\vec{x} = \langle move_N, move_S, move_E, move_W \rangle$ and at timestep t on state s_i we performed action N and observed no movement, we construct input $\vec{x}_t = (1, 0, 1, 0)$. If, on the other hand, we observed that the agent slipped to the West, the input would be $\vec{x}_t = (0, 0, 0, 1)$.

An alternative to this approach is to keep counts for each and every possible equivalence class that can be generated by the states in a domain. For example, we could have kept a separate count of how many times we observed the ambiguous effect $\{move_N, move_E\}$, separate from the counts of the non-ambiguous $\{move_N\}$ or $\{move_E\}$. I will call this approach a Partition learner, and compare it against KWIK-LR for the Maze domain. The experiment assumes all conditions are known, and only

effect probabilities need to be determined. From each state, the learner predicts either an effect distribution when known, or $\perp$ when it doesn't know it. Figure 6.2 shows the results. We can see that KWIK-LR learns much faster than the Partition learner by sharing information between equivalence classes.

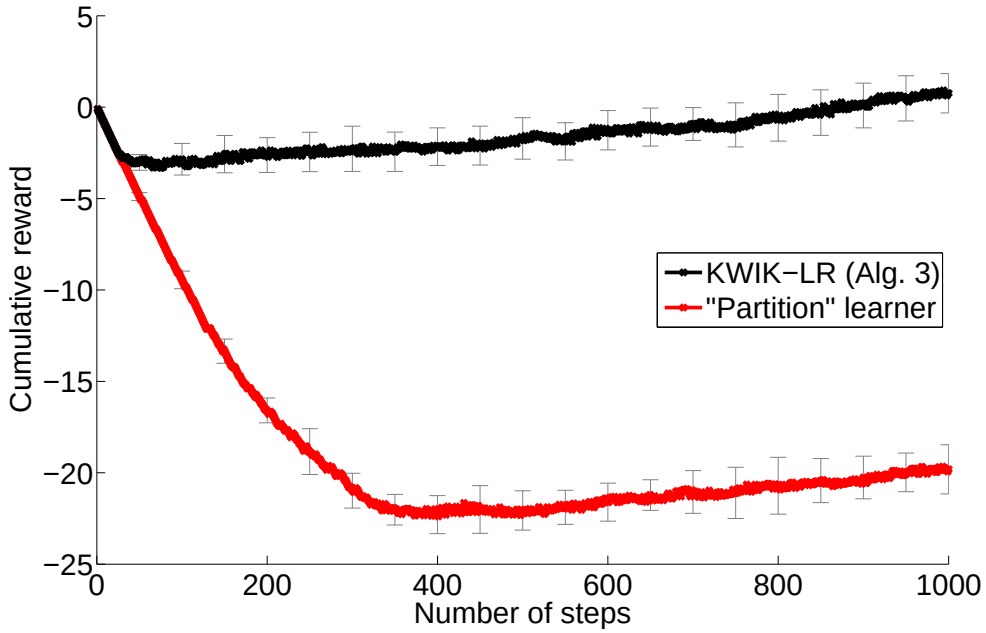


Figure 6.2: Results for KWIK-LR vs Partition learner in the Maze domain.

6.3 Learning effects

So far, I have assumed that the effect distribution is learned once the set of possible effects is provided as input. In the OO-MDP framework, it is assumed that given two states s and s', it is possible to compute a list of effects that might have produced the transition: $\mathit{eff}_{att}(s, s') = E_1 = \{\omega_1, ..., \omega_k\}$. It is possible then to imagine an online implementation of KWIK-LR which, after the first transition is observed, constructs an input vector of size k: $\vec{x}_1 =< \omega_1, ..., \omega_k >$. When a new transition is observed, new effects are computed. Some of them will already be part of $\vec{x}$, and some won't. The new effects $E_t = \{\omega_m, ..., \omega_{m+h}\}$ could then be added as an extension to the input vector: $\vec{x}_{i+1} = \vec{x}_i \bigcup E_t$. For all prior experience, a value of 0 is added to the columns representing the new effects.

I will introduce an example that illustrates this approach, and also highlights one of its problems. Let us go back to the infinite-grid Taxi example from chapter 5 (see Figure 5.3). In this example, the Taxi had coordinates x and y, which were increased by 1 given the appropriate action: the effect of taking action *North* is $y \leftarrow y + 1$. The environment has no walls or other constraints, so the Taxi can keep going North forever, with y increasing to infinity. Imagine that we start at state s_0, which as a notational abuse I will identify through the values of x and y: $s_0 =< 0, 0 >$. After taking action *North* once, we reach state $s_1 =< 0, 1 >$. Had we defined the universe of possible effect types as that of arithmetic addition and assignment, we obtain two possible effects: $\mathit{eff}_{att}(s_0, s_1) = \{+1, \textit{set-to}(1)\}$, and build the following input matrix for KWIK-LR:

$+1$	*set-to*(1)
1	1

We now take one more *North* action, and reach state $s_2 =< 0, 2 >$. The two possible effects in this case are $\mathit{eff}_{att}(s_0, s_1) = \{+1, \textit{set-to}(2)\}$. We thus expand the input matrix to:

$+1$	*set-to*(1)	*set-to*(2)
1	1	0
1	0	1

If we take another action *North*, the matrix becomes:

$+1$	*set-to*(1)	*set-to*(2)	*set-to*(3)
1	1	0	0
1	0	1	0
1	0	0	1

I believe it is now clear where this is going: what we wish KWIK-LR would learn is that there is only one *real* effect, $+1$, that occurs with probability 1, rather than assigning a small probability to each of the assignments, as we shall see is the case of KWIK-LR in the experiment of the next section.

6.3.1 Sparsity and KWIK-LR: SKWIK-LR

The sparsification mechanism introduced in this section is due to Istvan Szita, and to the best of my knowledge it is yet unpublished. The procedure involves two steps: a linear regression one (using KWIK-LR) followed by a sparsification of the weight vector obtained. Schematically:

1. Use KWIK-LR as presented in the previous section, obtaining after each step a vector of weights θ_i.

2. Sparsify θ_i to obtain θ_{i+1} using the following linear program:
   Minimize $\|\theta_{i+1}\|$ subject to $|\theta_i x - \theta_{i+1} x| < \epsilon/2$

The constraint implies that θ_{i+1} will be at most ϵ away from θ_i, and the minimization of its L1 norm will make it sparse (Shalev-Shwartz & Srebro, 2008).

Experiment

The following experiment illustrates the combination of sparsification and KWIK-LR in SKWIK-LR, as described above, and how it differs from KWIK-LR. The domain used is the infinite-grid Taxi, and all actions taken are *North* actions. If it has taken i steps, the matrix representing the accumulated experience will have $i+1$ columns. The first column represents effect +1 and it will be filled with 1s, whereas the rest of the matrix consists of a diagonal of 1s for each possible effect *set-to*(1) through *set-to*(i).

I run SKWIK-LR with accuracy parameter ϵ and 100 data points (100 *North* actions), and it results in a prediction that effect +1 occurs with probability $1 - \epsilon$, and all other effects are assigned a probability very close to 0 (in the order of 10^{-9}). Normalizing to obtain a proper multinomial distribution, effect +1 gets a probability very close to 1.0.

In contrast, I run KWIK-LR on the same data and it learns that the effect +1 has probability of 0.5, while each of the *set-to*(i) effects has probability 0.005.

The burden in terms of computational cost of SKWIK-LR is on the linear program used for sparsification. The code was implemented in Matlab and for 100 *North* actions

it ran in less than 1.7 seconds. For 1000 actions it took slightly less than 5 seconds.

6.4 KOOL: Putting it all together

In this section, I combine the three parts of the problem of learning a stochastic OO-MDP: learning the conditions, the effects and their probabilities. I will call the combined algorithm KWIK Object-Oriented Learner (KOOL), and present an experiment.

The inputs to KOOL are the inputs to the three different components: the total number of terms n, the maximum number of terms in any condition D, the tolerance parameter for KWIK-LR α_0, the sparsification parameter ϵ and the amount of experience needed before assuming something is known M. Note that different components of the algorithm could use different values of M, but for simplicity we will assume a unique one (which should be the largest one needed by any sub-component).

KOOL is an instance of KWIK-$R_{\max}$, where the transition learner is the Adaptive $k-$Meteorologists algorithm, and each meteorologist uses SKWIK-LR to predict effects. Algorithm 13 schematically presents KOOL.

Algorithm 13: KOOL

input: n, D, α_0, ϵ, M
initialize: Create $k = \binom{n}{D}$ Adaptive meteorologists. Initialize the Adaptive meteorologists with SKWIK-LR as learning algorithms $\mathcal{A}_1, \ldots, \mathcal{A}_k$.
repeat
  Observe state s
  Use planner to choose next action a. The planner will consult the best meteorologist for next-state predictions. If meteorologists don't yet have a prediction for a given state and action, assume optimistic transition to $s_{\max}$.
  Execute and observe s' and r.
  for all meteorologist i whose terms are true in $c(s)$ **do**
    Add experience (s, a, r, s') to SKWIK-LR learner $\mathcal{A}_i$
  end for
until termination criterion

6.5 Experiments

In this section, I present two experiments that use KOOL to learn an OO-MDP with stochastic effects and ambiguous observations. I introduce a simple domain that illustrates all the issues presented in this chapter, called the Mountain Climber Domain. Then, I present a version of the Taxi Problem where movement actions have stochastic effects.

6.5.1 The Mountain-Climber Problem

This domain simulates a climber that is trying to reach the top of a mountain. The climber starts at the bottom and tries to make her way up. The task is episodic, ending when the climber reaches the summit or after a maximum number of steps. At high altitudes, the mountain gets icy and the climber needs to wear crampons[1] to avoid falling. At lower altitudes, the deciding factor is the climber's stamina. The state of the climber is thus defined by her altitude, her level of stamina and whether or not she is wearing crampons.

The actions available are *ClimbUp*, *ClimbDown*, *EatPowerBar* and *ToggleCrampons*. If the climber has high stamina and is in a non-icy area, or is in an icy area but is wearing crampons, the action *ClimbUp* moves her one step higher with high probability p, and with probability $(1-p)$ she drops one step (staying in place is not an option). If she is low in stamina or on icy terrain without crampons, the probability of success is $q << p$. The action *EatPowerBar* always resets stamina to the highest level and *ToggleCrampons* gets her crampons on or off. Each step taken reduces stamina regardless of whether the action succeeds or not, and moving with the crampons on results in a penalty.

The optimal policy for our climber is to *ClimbUp* while stamina is high and there is no ice. When she reaches a state of low stamina, just *EatPowerBar*. When she reaches the ice, *ToggleCrampons* so that they are on.

The conditions induced by the state of the climber have four terms: whether the

[1] A crampon is an iron spike attached to the shoe to prevent slipping on ice when walking or climbing.

climber is at the bottom of the mountain (where action *ClimbDown* doesn't work), whether she has low stamina, she's in an icy area, and she has her crampons on or off.

An experiment was run where the altitude to be reached is 20, the maximum stamina (after *EatPowerBar*) is 15, a low stamina state is reached when stamina drops to 5, and there is ice at altitudes 14 and higher. In high-stamina and non-icy conditions, the *ClimbUp* action results in an increase of +1 in altitude with probability 0.8. With probability 0.2, the climber drops down one level. If the climber has low stamina or there is ice when she has no crampons, the probability of a successful climb up is 0.4, with 0.6 probability of falling one step down. Stamina is reduced by 1 at every step. Action *ClimbDown* succeeds with probability 0.8, and with probability 0.2 the climber stays where she is. *EatPowerBar* and *ToggleCrampons* always succeed. Ambiguity exists between effects $+1/-1$ and all possible *set-to*(i) for $i = 0 \ldots 20$. The climber receives a reward of -1 for each step taken without crampons, -1.5 if wearing them, and 0 when she reaches the top.

As a comparison, the experiment was ran telling the learner the correct number of terms to consider ($D = 2$), and inducing it to learn the incorrect model by telling it to consider only one term ($D = 1$). A coarse parameter search was performed for parameter M, and the best value was determined to be $M = 25$ for the correct model, and $M = 30$ for the incorrect one. In the $D = 2$ case, values of M lower than 20 resulted in the algorithm sometimes failing to learn the correct model. In the incorrect-model case ($D = 1$), other values of M yielded even worse performance. The other parameters were set to $\alpha_0 = 0.1$ (tolerance of SKWIK-LR) and $\epsilon = 0.1$ (precision of SKWIK-LR). The agent ran for 15 episodes per experiment, and the experiment was repeated 10 times. Figure 6.3 shows the averaged number of steps to goal, with error bars, for each of the 15 episodes. Note that in the case where the learner can only learn incorrect models given that it is considering a single term ($D = 1$), performance diverges after a few episodes of exploration.

This experiment shows that an OO-MDP with stochastic and ambiguous effects can be learned.

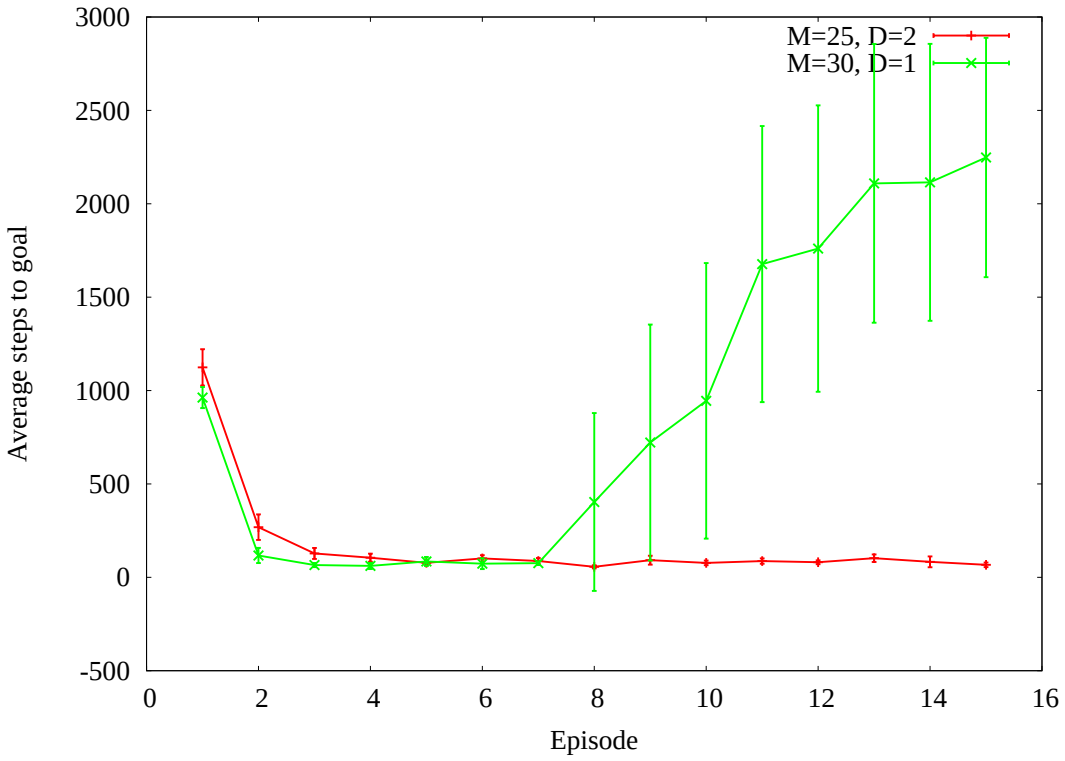


Figure 6.3: Mountain Climber results for $M = 25$ considering $D = 2$ terms, and an incorrect-model learner that only considers $D = 1$ terms. Results averaged over 10 runs.

6.5.2 Stochastic Taxi

This second experiment is on a variant of the Taxi Problem introduced before, in which the actions *North, South, East* and *West* have stochastic effects. When attempted, there is a 0.8 probability that the taxi moves in the desired direction (walls permitting), 0.1 probability that it will slip to the right, and 0.1 that it slips left. This style of stochastic domain was introduced by Russell and Norvig (2003).

The condition learner part of KOOL was initialized to consider a maximum of 3-term conjunctions, which is enough to describe the dynamics of the problem. For movement actions, the only thing that matters is whether there is a wall in the desired direction or any of its sides. The *Pickup* action only needs to consider whether the taxi is at the passenger location and *Dropoff* considers if the passenger is in the taxi and at the destination. If the appropriate conditions hold, both *Pickup* and *Dropoff* succeed with probability 1, so the only stochasticity in the domain is in the movements.

A parameter search was performed for M and the best value was determined to be $M = 20$. Figure 6.4 shows the average number of steps to goal per episode, for 10 episodes. Note that after episode 6, KOOL converges on a policy that solves the problem in an average of 67.1 steps, which is consistent with the expected number of steps of the optimal policy. At this point, variance is due to the stochasticity of the domain, and not to changes in policy.

6.6 Summary

In this chapter, I showed how stochastic OO-MDPs can be learned efficiently in the KWIK framework. The problem was broken down into three sub-problems: learning conditions, learning effects and learning effect probabilities. By connecting the learning problem to the Adaptive k-Meteorologists problem, it is possible to KWIK-learn conditions where the number of terms involved is bounded by a known constant. Using KWIK-LR, it is possible to learn effect probabilities even in the presence of ambiguous outcomes. Finally, a sparsification trick in KWIK-LR enables an extension called SKWIK-LR that heuristically learns an appropriate set of effects.

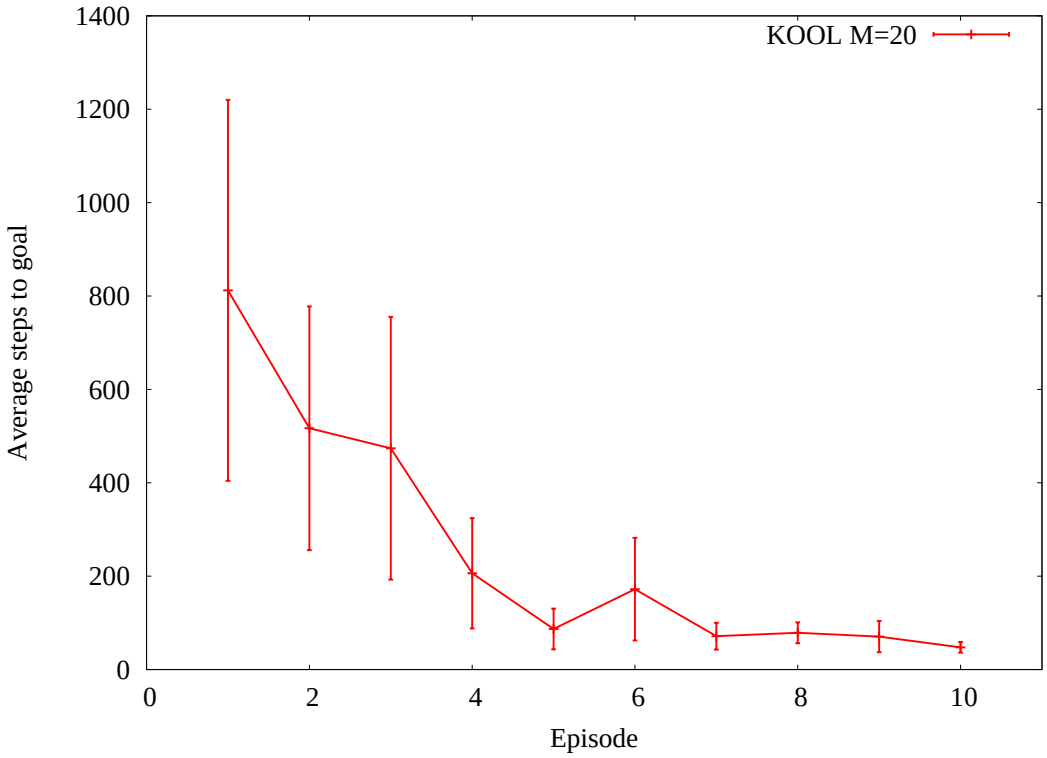


Figure 6.4: Stochastic Taxi results for $M = 20$, averaged over 10 runs.

The combination of the sub-algorithms for the three subproblems into an algorithm called KOOL enables learning a stochastic OO-MDP, and generalizes previous results for the deterministic case. KOOL is, to the best of my knowledge, the first algorithm that addresses all of these problems at the same time and has theoretical guarantees.

Chapter 7

Conclusions and Outlook

7.1 Conclusions

This dissertation introduces a novel representation of state for reinforcement-learning problems, based on objects, attributes of those objects, and transition dynamics based on the interactions between them. It was designed as an attempt to bridge two gaps: on one hand, the representational gap between how state is traditionally modeled in RL against the natural bias humans seem to apply when describing or tackling similar problems. On the other hand, the representation seeks to achieve a balance between the generality of first-order relational representations, and the goal of efficient learnability.

I first showed how state representations, generalization and exploration strategies interconnect to deeply impact learning. Through the simple Taxi domain, I showed how an agent looking at the world as a flat, combinatorial explosion of state variables, that goes out to naïvely explore it needs 200 times more experience than an agent that looks at the world as a set of objects, and smartly explores their behavior and interactions.

Second, I showed how OO-MDPs can naturally model a number of domains, from gridworlds to real-life videogames, in a natural way. In a recent paper (Walsh et al., 2010), we further showed how OO-MDPs can be used to model the dynamics of a real robot and be used to learn a simple navigation task in the real world. Transition dynamics of these domains are succinctly modeled, in an intuitive way that approximately matches the way a person would describe them. In the Taxi example, I showed how state-of-the-art factored representations would still require learning long tables of numbers to define how the taxi moves, whereas an OO-MDP description can be expressed through a simple, "human-readable" tree.

Given an OO-MDP, an important question is whether these representations are efficiently learnable. I presented two algorithms for the Propositional OO-MDP case, one for environments with deterministic dynamics (which happens to cover most videogames!) and one for domains where actions have stochastic outcomes. DOO$R_{\max}$, the algorithm for deterministic OO-MDPs, suffers from a bad worst-case bound (exponential in the number of terms describing dynamics), but can be shown to be very efficient in the best case. In practice, in any environment in which actions produce effects and only a few times they fail, DOO$R_{\max}$ will likely be the algorithm producing the best empirical results. These results were demonstrated in Pitfall, a real-life videogame for the Atari 2600 console, and on the Taxi problem. In stochastic environments, or when worst-case efficient theoretical guarantees are required, KOOL can be used with the only assumption that the maximum number of terms involved in any condition is known and —relatively— small. I hypothesize that in the case of deterministic domains it would not be hard to combine both algorithms to obtain the best of both: DOO$R_{\max}$ would provide fast learning (if possible), and KOOL would provide a fallback best-case (if not0.

To the best of my knowledge, KOOL is the first algorithm that tackles learning of these kinds of representations and provides theoretical guarantees. Work by Pasula et al. (2007) tackles similar kinds of problems for probabilistic STRIPS representations (Blum & Langford, 1999), but does it in a purely heuristic way. I hypothesize that simple modifications to KOOL could be used to learn these representations too.

7.2 Open Problems

Some open problems still persist, and I will briefly expand on two of them. First, there is still a gap between the performance of algorithms in the OO-MDP formalism and humans performing the same task, like Taxi. Second, while this dissertation focused on learning, efficient planning remains an open problem. At least, it is important to study the applicability of existing good planning heuristics.

7.2.1 Human Biases

In the Taxi experiments I showed how, while DOO$R_{\max}$ requires 529 actions to learn an optimal solution, (some) humans do it in an average of 101 steps, and those who self-identify as videogamers solve it in 48. Of course, what humans do is bring prior knowledge and biases to bear into the problem. Two clear examples are navigation and the concept of walls. While humans solved navigation in one step, by assuming what arrow keys would do, DOO$R_{\max}$ needs to actually try out a number of actions. This could be solved by encoding into DOO$R_{\max}$ some notion of symmetry, for example, or the fact that a *Down* action can be reversed with an *Up* one. At this point, I can only speculate as to whether work on MDP symmetries (Ravindran & Barto, 2001) could be brought to use here.

Another important bias is the concept of walls: no humans tried to bump into them in the Taxi experiments. Moreover, no human felt the need to explore conditions like how an *Up* action might be affected when there is a wall to the right or left of the navigating object. DOO$R_{\max}$, in many cases, does need to explore these conditions.

In the case of Pitfall, once again humans would use prior knowledge and make assumptions as to how objects that look like holes, staircases or walls influence the game dynamics, and most likely not even need to explore interactions with them. It would be interesting to devise an experiment in which these object semantics are blurred, and see how humans compare to DOO$R_{\max}$ or KOOL. It is noteworthy how, if we abstract away and ignore these assumptions about identifiable objects, we can see how DOO$R_{\max}$ explores Pitfall in a very human-like fashion.

7.2.2 Planning

Another open problem is efficient planning. In this dissertation, I used two strategies: either blowing up the state space into a flat representation and using exact-planning approaches like value iteration (for Taxi), or using approximate forward-search methods like sparse sampling (for Pitfall).

A whole research community is devoted to the planning problem, and every year a

competition is held in which the best planners face off against each other. Recently, this competition has even developed a track devoted to probabilistic planning (Younes et al., 2005), tackling planning problems akin to those that KOOL faces with stochastic OO-MDPs. Problems in these competitions are described in a well-established common language, PDDL[1] (Fox & Long, 2003). An extension to this work would be to find a way to translate OO-MDP representations into PPDDLs, and leverage existing planners.

Another extension, which I hypothesize is easier, would be to replace the sparse-sampling methods I have used by UCT (Kocsis & Szepesvári, 2006), a smarter forward-search planner that has already proven to be extremely fast.

7.3 Summary

In summary, this dissertation has shown that object-oriented representations are a natural way of representing state in a large class of problems, enabling orders of magnitude faster learning. The class of problems for which OO-MDPs are most suitable are relational in nature: they involve objects interacting with each other. Object-Oriented MDPs encode state in terms of objects and their interactions in a way that mimics how humans would describe many environments, and efficient algorithms have been introduced that demonstrate that OO-MDPs can be learned fast and with theoretical guarantees.

[1] PPDDL for the probabilistic case.

Bibliography

Abbeel, P., Coates, A., Quigley, M., & Ng, A. Y. (2007). An application of reinforcement learning to aerobatic helicopter flight. *In Advances in Neural Information Processing Systems 19* (p. 2007). MIT Press.

Atkeson, C. G., & Santamaria, J. C. (1997). A comparison of direct and model-based reinforcement learning. *IN INTERNATIONAL CONFERENCE ON ROBOTICS AND AUTOMATION* (pp. 3557–3564). IEEE Press.

Auer, P., Cesa-Bianchi, N., & Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, *47*, 235–256.

Baillargeon, R., Li, J., Ng, W., & Yuan, S. (2008). An account of infants' physical reasoning. In A. Woodward and A. Needham (Eds.), *Learning and the infant mind*. Oxford University Press.

Barto, A. G., & Mahadevan, S. (2003). Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, *13*, 341–379.

Barto, A. G., Sutton, R. S., & Watkins, C. J. C. H. (1989). Learning and sequential decision making. *LEARNING AND COMPUTATIONAL NEUROSCIENCE* (pp. 539–602). MIT Press.

Bellman, R. (1957). *Dynamic programming*. Princeton University Press.

Berry, D. A., & Fristedt, B. (1985). *Bandit problems: Sequential allocation of experiments*. London, UK: Chapman and Hall.

Bertsekas, D. P., & Tsitsiklis, J. N. (1996). *Neuro-dynamic programming*. Belmont, MA: Athena Scientific.

Bishop, C. M. (2006). *Pattern recognition and machine learning (information science and statistics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

Blum, A., & Langford, J. (1999). Probabilistic planning in the graphplan framework. *ECP* (pp. 319–332). Springer.

Boutilier, C., Dean, T., & Hanks, S. (1999). Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, *11*, 1–94.

Boutilier, C., & Dearden, R. (1994). Using abstractions for decision-theoretic planning with time constraints. *In Proceedings of the Twelfth National Conference on Artificial Intelligence* (pp. 1016–1022).

Boutilier, C., Dearden, R., & Goldszmidt, M. (2000). Stochastic dynamic programming with factored representations. *Artificial Intelligence*, *121*, 49–107.

Brafman, R. I., & Tennenholtz, M. (2002). R-MAX—a general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research*, *3*, 213–231.

Cohen, W. W. (1995a). PAC-learning recursive logic programs: Efficient algorithms. *J. Artif. Intell. Res. (JAIR)*, *2*, 501–539.

Cohen, W. W. (1995b). PAC-learning recursive logic programs: Negative results. *J. Artif. Intell. Res. (JAIR)*, *2*, 541–573.

Crites, R., & Barto, A. (1996). Improving elevator performance using reinforcement learning. *Advances in Neural Information Processing Systems 8* (pp. 1017–1023). MIT Press.

de Farias, D. P., & Van Roy, B. (2003). The linear programming approach to approximate dynamic programming. *Operations Research*, *51*, 850–865.

De Raedt, L. (2008). *Logical and relational learning: From ILP to MRDM (cognitive technologies)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

Dean, T., & Givan, R. (1997). Model minimization in markov decision processes. *In Proceedings of the Fourteenth National Conference on Artificial Intelligence* (pp. 106–111). AAAI.

Dean, T., & Kanazawa, K. (1989). A model for reasoning about persistence and causation. *Computational Intelligence*, *5*, 142–150.

Dietterich, T. G. (2000). Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, *13*, 227–303.

Diuk, C., Cohen, A., & Littman, M. L. (2008). An object-oriented representation for efficient reinforcement learning. *Proceedings of the Twenty-Fifth International Conference (ICML 2008), Helsinki, Finland, June 5-9, 2008* (pp. 240–247).

Diuk, C., Li, L., & Leffler, B. R. (2009). The adaptive k-meteorologists problem and its application to structure learning and feature selection in reinforcement learning. *Proceedings of the Twenty-Sixth International Conference on Machine Learning (ICML-09)*.

Diuk, C., & Littman, M. (2008). Hierarchical reinforcement learning. In J. R. R. Dopico, J. D. D. L. Calle and A. P. Sierra (Eds.), *Encyclopedia of artificial intelligence*. Information Science Reference - Imprint of: IGI Publishing.

Diuk, C., Littman, M., & Strehl, A. (2006). A hierarchical approach to efficient reinforcement learning in deterministic domains. *Fifth International Conference on Autonomous Agents and Multiagent Systems (AAMAS-06)*.

Doya, K. (2000). Reinforcement learning in continuous time and space. *Neural Comput.*, *12*, 219–245.

Duff, M. O. (2003). Design for an optimal probe. *ICML* (pp. 131–138). AAAI Press.

Džeroski, S., De Raedt, L., & Driessens, K. (2001). Relational reinforcement learning. *Machine Learning*, *43*, 7–52.

Dzeroski, S., Muggleton, S., & Russell, S. J. (1992). PAC-learnability of determinate logic programs. *COLT* (pp. 128–135).

Finney, S., Gardiol, N. H., Kaelbling, L. P., & Oates, T. (2002a). *Learning with deictic representations* (Technical Report). Massachusetts Institute of Technology, Artificial Intelligence Laboratory.

Finney, S., Wakker, P. P., Kaelbling, L. P., & Oates, T. (2002b). The thing that we tried didn't work very well: Deictic representation in reinforcement learning. *UAI* (pp. 154–161). Morgan Kaufmann.

Fox, M., & Long, D. (2003). PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, *20*, 2003.

Gelly, S., & Silver, D. (2007). Combining online and offline knowledge in UCT. *Proceedings of the Twenty-Fourth International Conference on Machine Learning (ICML-07)* (pp. 273–280).

Graepel, T., Herbrich, R., & Gold, J. (2004). Learning to fight. *Proceedings of the International Conference on Computer Games: Artificial Intelligence, Design and Education*.

Guestrin, C., Koller, D., Gearhart, C., & Kanodia, N. (2003). Generalizing plans to new environments in relational MDPs. *IJCAI* (pp. 1003–1010).

Guestrin, C., Patrascu, R., & Schuurmans, D. (2002). Algorithm-directed exploration for model-based reinforcement learning in factored MDPs. *Proceedings of the International Conference on Machine Learning* (pp. 235–242).

Guez, A., Vincent, R. D., Avoli, M., & Pineau, J. (2008). Adaptive treatment of epilepsy via batch-mode reinforcement learning. *AAAI* (pp. 1671–1678). AAAI Press.

Hordijk, A., & Kallenberg, L. (1979). Linear programming and markov decision chains. *Management Science*, *25*, 352–362.

Howard, R. A. (1960). *Dynamic programming and Markov processes*. Cambridge, MA: MIT Press.

Ipek, E., Mutlu, O., Martínez, J. F., & Caruana, R. (2008). Self-optimizing memory controllers: A reinforcement learning approach. *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture* (pp. 39–50). Washington, DC, USA: IEEE Computer Society.

Jong, N. K., & Stone, P. (2005). State abstraction discovery from irrelevant state variables. *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence* (pp. 752–757).

Kaelbling, L. P., Littman, M. L., & Cassandra, A. R. (1998). Planning and acting in partially observable stochastic domains. *Artif. Intell.*, *101*, 99–134.

Kaelbling, L. P., Littman, M. L., & Moore, A. P. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, *4*, 237–285.

Kakade, S. M. (2003). *On the sample complexity of reinforcement learning*. Doctoral dissertation, Gatsby Computational Neuroscience Unit, University College London.

Kearns, M. J., & Koller, D. (1999). Efficient reinforcement learning in factored MDPs. *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI)* (pp. 740–747).

Kearns, M. J., Mansour, Y., & Ng, A. Y. (2002). A sparse sampling algorithm for near-optimal planning in large Markov decision processes. *Machine Learning*, *49*, 193–208.

Kearns, M. J., & Schapire, R. E. (1994). Efficient distribution-free learning of probabilistic concepts. *Journal of Computer and System Sciences*, *48*, 464–497.

Kearns, M. J., & Singh, S. P. (2002). Near-optimal reinforcement learning in polynomial time. *Machine Learning*, *49*, 209–232.

Kearns, M. J., & Vazirani, U. V. (1994). *An introduction to computational learning theory.* Cambridge, MA, USA: MIT Press.

Khardon, R. (1996). Learning to take actions. *Machine Learning* (pp. 787–792). AAAI Press.

Kocsis, L., & Szepesvári, C. (2006). Bandit based monte-carlo planning. *ECML-06. Number 4212 in LNCS* (pp. 282–293). Springer Verlag.

Kohl, N., & Stone, P. (2004). Machine learning for fast quadrupedal locomotion. *The Nineteenth National Conference on Artificial Intelligence* (pp. 611–616).

Lee, H., Shen, Y., Yu, C.-H., Singh, G., & Ng, A. Y. (2006). Quadruped robot obstacle negotiation via reinforcement learning. *ICRA* (pp. 3003–3010). IEEE.

Leffler, B. R., Littman, M. L., & Edmunds, T. (2007). Efficient reinforcement learning with relocatable action models. *AAAI-07: Proceedings of the Twenty-Second Conference on Artificial Intelligence* (pp. 572–577). Menlo Park, CA, USA: The AAAI Press.

Li, L. (2009). *A unifying framework for computational reinforcement learning theory.* Doctoral dissertation, Department of Computer Science, Rutgers University, New Brunswick, NJ.

Li, L., Littman, M. L., & Walsh, T. J. (2008). Knows what it knows: A framework for self-aware learning. *Proceedings of the Twenty-Fifth International Conference on Machine Learning (ICML-08)* (pp. 568–575).

Li, L., Walsh, T. J., & Littman, M. L. (2006). Towards a unified theory of state abstraction for MDPs. *Proceedings of the Ninth International Symposium on Artificial Intelligence and Mathematics (AMAI-06).*

Littlestone, N. (1988). Learning quickly when irrelevant attributes abound: A new linear-threshold algorithms. *Machine Learning*, *2*, 285–318.

Littman, M. L., Dean, T. L., & Kaelbling, L. P. (1995). On the complexity of solving Markov decision problems. *Proceedings of the Eleventh Annual Conference on Uncertainty in Artificial Intelligence (UAI–95)* (pp. 394–402). Montreal, Québec, Canada.

Luo, Y., Kaufman, L., & Baillargeon, R. (2009). Young infants' reasoning about physical events involving inert and self-propelled objects. *Cognitive Psychology*, *58*, 441–486.

Mannor, S., & Tsitsiklis, J. N. (2004). The sample complexity of exploration in the multi-armed bandit problem. *Journal of Machine Learning Research*, *5*, 623–648.

Merrick, K., & Maher, M. L. (2006). Motivated reinforcement learning for non-player characters in persistent computer game worlds. *ACE '06: Proceedings of the 2006 ACM SIGCHI International conference on Advances in Computer Entertainment Technology* (p. 3). New York, NY, USA: ACM.

Moore, A. W., & Atkeson, C. G. (1993). Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning*, *13*, 103–130.

Munos, R. (2003). Error bounds for approximate policy iteration. *Proceedings of the Twentieth International Conference on Machine Learning (ICML-03)* (pp. 560–567).

Munos, R. (2005). Error bounds for approximate value iteration. *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)* (pp. 1006–1011).

Ng, A. Y., Kim, H. J., Jordan, M. I., & Sastry, S. (2004). Autonomous helicopter flight via reinforcement learning. *Advances in Neural Information Processing Systems 16 (NIPS-03)*.

Pasula, H. M., Zettlemoyer, L. S., & Kaelbling, L. P. (2007). Learning symbolic models of stochastic domains. *J. Artif. Intell. Res. (JAIR)*, *29*, 309–352.

Ponsen, M., Spronck, P., & Tuyls, K. (2006). Hierarchical reinforcement learning

with deictic representation in a computer game. *Proceedings of the 18th Belgium-Netherlands Conference on Artificial Intelligence (BNAIC 2006)* (pp. 251–258). University of Namur, Belgium.

Poupart, P., Vlassis, N. A., Hoey, J., & Regan, K. (2006). An analytic solution to discrete bayesian reinforcement learning. *ICML* (pp. 697–704). ACM.

Puterman, M. L. (1994). *Markov Decision Processes—Discrete Stochastic Dynamic Programming*. New York, NY: John Wiley & Sons, Inc.

Ravindran, B. (2004). *An algebraic approach to abstraction in reinforcement learning*. Doctoral dissertation, University of Massachusetts Amherst. Director-Barto, Andrew G.

Ravindran, B., & Barto, A. G. (2001). *Symmetries and model minimization in Markov Decision Processes* (Technical Report). University of Massachusetts Amherst, Amherst, MA, USA.

Ravindran, B., Barto, A. G., & Mathew, V. (2007). Deictic option schemas. *IJCAI'07: Proceedings of the 20th international joint conference on Artifical intelligence* (pp. 1023–1028). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Russell, S., & Norvig, P. (2003). *Artificial intelligence: A modern approach*. Prentice-Hall, Englewood Cliffs, NJ. 2nd edition.

Scholl, B. J. (2001). Objects and attention: The state of the art. *Cognition*, *80*, 1–46.

Shalev-Shwartz, S., & Srebro, N. (2008). Low l1 norm and guarantees on sparsifiability. *Sparse Optimization and Variable Selection Workshop (ICML/COLT/UAI 2008)*.

Shoham, Y., Powers, R., & Grenager, T. (2003). *Multi-agent reinforcement learning: a critical survey* (Technical Report). Stanford University.

Silver, D., Sutton, R., & Müller, M. (2007). Reinforcement learning of local shape in the game of Go. *IJCAI*.

Singh, S. P., Jaakkola, T., & Jordan, M. I. (1995). Reinforcement learning with soft state aggregation. *Advances in Neural Information Processing Systems 7* (pp. 361–368). MIT Press.

Singh, S. P., & Yee, R. C. (1994). An upper bound on the loss from approximate optimal-value functions. *Machine Learning*, *16*, 227.

Strehl, A. L. (2007). Model-based reinforcement learning in factored-state MDPs. *ADPRL 2007: Proceedings of the 2007 IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning*.

Strehl, A. L., Diuk, C., & Littman, M. L. (2007). Efficient structure learning in factored-state MDPs. *AAAI* (pp. 645–650). AAAI Press.

Strehl, A. L., Li, L., Wiewiora, E., Langford, J., & Littman, M. L. (2006). Pac model-free reinforcement learning. *ICML-06: Proceedings of the 23rd international conference on Machine learning* (pp. 881–888).

Strens, M. J. A. (2000). A bayesian framework for reinforcement learning. *ICML '00: Proceedings of the Seventeenth International Conference on Machine Learning* (pp. 943–950). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Sutton, R. (1998). The reward hypothesis. World Wide Web electronic publication.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *MACHINE LEARNING* (pp. 9–44). Kluwer Academic Publishers.

Sutton, R. S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. *Proceedings of the Seventh International Conference on Machine Learning* (pp. 216–224). Austin, TX: Morgan Kaufmann.

Sutton, R. S. (1991). Dyna, an integrated architecture for learning, planning, and reacting.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. The MIT Press.

Sutton, R. S., Barto, A. G., & Williams, R. J. (1992). Reinforcement learning is direct adaptive optimal control. *In Proceedings of the American Control Conference* (pp. 2143–2146).

Taylor, J., Precup, D., & Panangaden, P. (2008). Bounding performance loss in approximate mdp homomorphisms. *NIPS* (pp. 1649–1656). MIT Press.

Tesauro, G. (1994). TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, *6*, 215–219.

Tesauro, G. (2005). Online resource allocation using decompositional reinforcement learning. *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA* (pp. 886–891). AAAI Press / The MIT Press.

Tsitsiklis, J. N., & Roy, B. V. (2000). Regression methods for pricing complex american-style options. *IEEE Transactions on Neural Networks*, *12*, 694–703.

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

van Otterlo, M. (2005). *A survey of reinforcement learning in relational domains* (Technical Report TR-CTIT-05-31). CTIT Technical Report Series, ISSN 1381-3625.

van Otterlo, M. (2008). *The logic of adaptive behavior: Knowledge representation and algorithms for the Markov decision process framework in first-order domains.* Doctoral dissertation, University of Twente, Enschede.

Walsh, T. J., Subramanian, K., Littman, M. L., & Diuk, C. (2010). Generalizing apprenticeship learning across hypothesis classes. *Proceedings of the Twenty-Seventh International Conference (ICML 2010).*

Walsh, T. J., Szita, I., Diuk, C., & Littman, M. L. (2009). Exploring compact reinforcement-learning representations with li near regression. *Proceedings of The 25th Conference on Uncertainty in Artific ial Intelligence (UAI-09).*

Watkins, C. J. (1989). *Learning from delayed rewards*. Doctoral dissertation, King's College, University of Cambridge, UK.

Watkins, C. J., & Dayan, P. (1992). Q-learning. *Machine Learning*, *8*, 279–292.

Yamanishi, K. (1992). A learning criterion for stochastic rules. *Machine Learning*, *9*, 165–203.

Younes, H. L. S., Littman, M. L., & Asmuth, J. (2005). The first probabilistic track of the international planning competition. *Journal of Artificial Intelligence Research*, *24*, 851–887.

Vita

Carlos Gregorio Diuk Wasser

2003-2010 Ph. D. in Computer Science, Rutgers University

1994-2003 Licenciatura in Computer Science, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, Argentina

2009-2010 Senior Research Assistant, Department of Psychology, Princeton University.

2007 Summer Intern - Software Engineering, Google.

2006 Summer Graduate Intern - Research, Intel Corporation.

2004-2009 Graduate Assistant, Department of Computer Science, Rutgers University.

2003-2004 Teaching Assistant, Department of Computer Science, Rutgers University.

2010 Generalizing Apprenticeship Learning across Hypothesis Classes. With Thomas J. Walsh, Kaushik Subramanian and Michael L. Littman. *Proceedings of the 27th International Conference on Machine Learning*, Haifa, Israel.

2009 The Adaptive k-Meteorologists Problem and Its Application to Structure Learning and Feature Selection in Reinforcement Learning. With Lihong Li, Bethany Leffler and Michael Littman. *Proceedings of the 26th International Conference in Machine Learning*, Montreal, Canada.

2009 Exploring Compact Reinforcement-Learning Representations with Linear Regression. With Thomas Walsh, Istvan Szita and Michael Littman. *Proceedings of the 25th Conference on Uncertainty in Artificial Intelligence*, Montreal, Canada.

2008 An Object-Oriented Representation for Efficient Reinforcement Learning. With Andre Cohen and Michael Littman. *Proceedings of the 25th International Conference on Machine Learning*, Helsinki, Finland.

2007 Efficient Structure Learning in Factored-state MDPs. With Alexander Strehl and Michael Littman. *Proceedings of the 22nd Conference on Artificial Intelligence (AAAI 07)*, Vancouver, Canada.

2007 An adaptive anomaly detector for worm detection. With John Mark Agosta, Jaideep Chandrashekar and Carl Livadas. *Second Workshop on Tackling Computer Systems Problems with Machine Learning Techniques (sysML-07)*, Cambridge, MA, USA.

2006 A Hierarchical Approach to Efficient Reinforcement Learning in Deterministic Domains. With Alexander Strehl and Michael Littman. *Proceedings of the 5th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS06)*, Hakodate, Japan.