The Dissertation Committee for Nicholas Kenneth Jong
certifies that this is the approved version of the following dissertation:

Structured Exploration for Reinforcement Learning

Committee:

Peter Stone, Supervisor

Benjamin Kuipers

Risto Miikkulainen

Raymond Mooney

Satinder Singh

Structured Exploration for Reinforcement Learning

by

Nicholas Kenneth Jong, B.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2010

For Sarah

Acknowledgments

Several times during the course of writing this thesis, I did not think I would finish. Nevertheless, I resolved to make every effort to complete this work, not just for my sake but also for all those who supported me in so many ways. Without their help, I never would have had a chance, so to them I owe my sincere gratitude.

First and foremost, Peter Stone has been all I could have wanted in an advisor. He believed in me from the very beginning. In my first year, he brought me in as an additional author to augment an existing paper submission with new experiments. That summer he convinced our coauthors to allow me to give the oral presentation, at my very first conference. The lessons I learned then and in the years following were essential in my growth as a researcher. Throughout my career, Peter has been unfailingly positive and always ready to provide necessary guidance and constructive feedback.

Looking back, I owe much to that first conference paper, which introduced me to coauthors Satinder Singh and Michael Littman, and through them Rich Sutton. They welcomed me warmly into the community of Reinforcement Learning researchers, and their encouragement and constructive criticism were invaluable.

Along with Peter and Satinder, the other members of my committee, Ben Kuipers, Risto Miikkulainen, and Ray Mooney, have also been immensely influential in my graduate education. From Ben's groundbreaking work on agents that bootstrap hierarchical knowledge from raw sensory experience to Ray's exhortations to conduct rigorous experimental validations, their teachings shaped my development as a practitioner of Artificial Intelligence.

I would also like to thank all those friends I made along the way, whose camaraderie during my journey through graduate school made all the difference. Although I cannot thank everyone in this community individually, I want to call out the other inaugural members of the AIBO soccer team and the Learning Agents Research Group, including Kurt Dresner, Greg Kuhlmann, and Dan Stronger. I do wish I could thank everyone, from Misha Bilenko, who took my under his wing at that first conference in 2003, to Todd Hester, whose direct collaboration was instrumental for parts of my thesis research.

Finally, I would not be where I am today without the loving support of my family, from my parents, who provided me with every opportunity for success, to my wife, Sarah Paetsch, who inspires me always to grow as a person. Sarah in particular has borne the brunt of my obsession with my work, and for her patience and understanding I will be forever grateful.

Structured Exploration for Reinforcement Learning

Publication No. __________________

Nicholas Kenneth Jong, Ph.D.
The University of Texas at Austin, 2010

Supervisor: Peter Stone

Reinforcement Learning (RL) offers a promising approach towards achieving the dream of autonomous agents that can behave intelligently in the real world. Instead of requiring humans to determine the correct behaviors or sufficient knowledge in advance, RL algorithms allow an agent to acquire the necessary knowledge through direct experience with its environment. Early algorithms guaranteed convergence to optimal behaviors in limited domains, giving hope that simple, universal mechanisms would allow learning agents to succeed at solving a wide variety of complex problems. In practice, the field of RL has struggled to apply these techniques successfully to the full breadth and depth of real-world domains.

This thesis extends the reach of RL techniques by demonstrating the synergies among certain key developments in the literature. The first of these developments is model-based exploration, which facilitates theoretical convergence guarantees in finite problems by explicitly reasoning about an agent's certainty in its understanding of its environment. A second branch of research studies function approximation, which generalizes RL to infinite problems by artificially limiting the degrees of freedom in an agent's representation of its environment. The final major advance that this thesis incorporates is hierarchical decomposition, which seeks to improve the efficiency of learning by endowing an agent's knowledge and behavior with the gross structure of its environment.

Each of these ideas has intuitive appeal and sustains substantial independent research efforts, but this thesis defines the first RL agent that combines all their benefits in the general case. In showing how to combine these techniques effectively, this thesis investigates the twin issues of generalization and exploration, which lie at the heart of efficient learning. This thesis thus lays the groundwork for the next generation of RL algorithms, which will allow scientific agents to know when it suffices to estimate a plan from current data and when to accept the potential cost of running an experiment to gather new data.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

As technology grows in capability and complexity, society will increasingly rely on computers to make decisions and to take actions on our behalf. Next-generation computer systems, too intricate for humans to administer, will need to configure, optimize, and repair themselves. Smart cars will relieve humans from the monotony of driving during rush hour. Robots will accept tasks too dangerous for humans, from fighting fires to exploring disaster areas. In all these examples, *autonomous agents* choose actions without human intervention to achieve some task. A key research question is how to endow these agents' behaviors with the robustness that characterizes human decision-making.

Detemining fixed behaviors ahead of time yields agents that fail to adapt to the inevitable unforeseen situations and uncertainty that characterize real-world tasks. Attempts to endow agents directly with general knowledge and reasoning capabilities inevitably encounter the knowledge-engineering bottleneck: the difficulty of formally encoding the broad range of knowledge that humans apply in the real world. To replicate human expertise, agents must replicate the human ability to learn and to adapt. Such *learning agents* choose future behaviors in response to data from past behaviors.

Reinforcement learning (RL) provides an appealing framework for research into learning agents. It grew out of promising early algorithms that guarantee convergence to optimal behavior in arbitrary finite agent-environment systems. These algorithms achieve these theoretical guarantees by estimating the long-term value of every state in the system. However, convergence to the correct values requires unbounded amounts of data from each state, and in practice these methods are rarely feasible in realistic applications. RL research has focused on scaling these methods to harder problems, working towards the goal of a relatively simple learning algorithm that allows an agent to cope with the complexity of the real world.

This thesis continues in this tradition of scaling RL algorithms to harder tasks, but it argues that prior efforts largely focused on only half the problem. Over the past two decades, researchers have developed much more sophisticated and efficient algorithms for what this thesis will call *exploitation*: the estimation of optimal behavior (by way of optimal state values) given existing data. In contrast, most deployed RL algorithms still rely on the same mechanisms for *exploration*: the generation of behaviors intended to gather data that will improve future attempts at exploitation. In particular, most RL algorithms estimate optimal behavior from existing data while assuming that no new data will become available. They then modify this behavior after the fact to encourage the acquisition of new data, often just by adding random actions.

This thesis advocates the development of *scientific agents*, which explicitly choose future behaviors in order to improve the utility of future data. By gathering more data, the agent improves its future ability to exploit the environment. This approach draws inspiration from human learning, which is most effective in the context of active experimentation, not passive observation. In the ideal case, a scientific agent conducts experiments that directly help it to efficiently disambiguate among competing hypotheses, especially when those hypotheses imply different exploitation behaviors.

This focus on exploration will lead naturally to an investigation of the a priori assumptions that an agent makes about its environment, which govern how the agent can generalize from available data. From this perspective, the thesis illuminates synergies among some important threads of RL research, described in detail in the next section: model-based exploration, function approximation, and hierarchical decomposition. This examination culminates in the synthesis of all three of these ideas for the first time into a single general-purpose algorithm, extending the space of environments that RL agents can feasibly learn.

1.1 Motivation

One primary inspiration for this thesis is the observation that three of the most promising ideas in RL have been studied almost entirely in isolation from one another. Can they be productively combined, or are their benefits redundant?

The first of these ideas is *model-based RL*, an approach that led to the first polynomial bounds on the sample complexity of learning, established by the E^3 algorithm (Kearns & Singh, 1998), which can probabilistically find a near-optimal policy for any finite environment in a finite number of time steps. This algorithm achieves its guaranteed performance by reasoning explicitly about exploration and exploitation. It also differs from the typical RL algorithm by constructing an explicit model of the environment, which it uses to compute behaviors that reach unexplored regions. Despite these results, the majority of RL research continues to focus on algorithms that neither reason about exploration nor construct models. One notable exception is the R-MAX algorithm (Brafman & Tennenholtz, 2002), which simplifies E^3 and yields one of the currently best known bounds on the sample complexity of RL[1] (Kakade, 2003), but even R-MAX has not seen much use in practical applications. This thesis investigates the practical limitations of R-MAX and develops algorithms that address them. One obvious limitation is the incompatibility between the finite model representation of R-MAX and the continuous nature of most real-world environments.

The second idea, *function approximation*, tackles such environments by changing RL algorithms' representation of the estimated values of states. This technique employs a small number of parameters to approximate the enormous space of possible mappings from states to values. However, most research

[1]The Delayed Q-learning algorithm (Strehl et al., 2006) exhibits a tighter bound with respect to the size of the state space but a looser bound with respect to the degree of optimality and the discount factor.

into function approximation focuses on how to find the parameters that best approximate the optimal values given the available data (exploitation), almost always in a *model-free* context where these values comprise the only learned knowledge of the environment. Comparatively little work has addressed how to explore continuous environments effectively, an issue closely tied to the question of what assumptions about the environment a given approximation scheme encodes, as well as to the paucity in representations for continuous models.

The third idea, *hierarchical decomposition*, directly addresses the fact that in practice, environments have structure that agents should exploit. Hierarchical RL algorithms reason with high-level, abstract actions that comprise sequences of lower-level actions (Barto & Mahadevan, 2003), reflecting the human ability to plan at different levels of abstraction. Hierarchical RL frameworks such as options (Sutton et al., 1999) and MAXQ (Dietterich, 2000a) have demonstrated the benefit of this approach for model-free algorithms and largely in finite environments. However, despite the intuitive appeal of hierarchy, most work in this direction doesn't consider the precise mechanism behind this benefit.

Another primary inspiration is the key question of how to generate these hierarchical decompositions automatically. The discovery of hierarchies, like the discovery of function approximations, remains a critical challenge for the RL community. A solution to the discovery problem is also the ultimate goal of the line of work that this thesis begins. A fully developed scientific agent will conduct experiments exactly to differentiate among competing hypotheses about the structure of its environment, each of which suggest distinct hierarchies of behavior. This thesis presents the first step towards that goal: an understanding of how hierarchy contributes to efficient learning, as well as an agent that combines explicit reasoning about exploration with hierarchical reasoning, in infinite environments.

1.2 Objectives

This thesis focuses on the following question:

> **How can a reinforcement learning agent efficiently and pragmatically balance exploration and exploitation, to maximize its expected rewards in an unknown, infinite environment with realistic structure?**

This thesis adopts the assumptions of the standard RL framework. The agent knows a priori the set of possible states and the set of possible actions. Each action has unknown effects and may be stochastic, but it induces a stationary distribution over successor states and immediate rewards given the previous state and action. The objective of the agent is to maximize the expected sum of future rewards.

The weakness of these assumptions, and therefore the generality of the framework, is the appeal of RL. Almost any problem can be cast in these terms, and these assumptions suffice to guarantee optimal behavior in the limit and probably near-optimal behavior in polynomial time. Nevertheless, this thesis deviates from the bulk of RL research by explicitly reasoning about additional assumptions and biases that may be introduced to a learning agent, for two pragmatic reasons. First, the baseline assumptions so far have not sufficed to produce algorithms that learn sufficiently quickly in practice. Second, in infinite environments, the finite-time convergence guarantees do not apply. The existing sample complexity bounds for RL consider a worst-case analysis across environments. For any algorithm in this setting, it is possible to define an environment in which that algorithm's behavior is arbitrarily poor.

With this perspective, the research challenge becomes the identification of reasonable assumptions and biases that permit efficient learning in practice but that do not severely restrict the scope of RL. This thesis investigates two of the most appealing techniques for scaling typical model-free RL algorithms: function approximation and hierarchical decomposition, as described in Section 1.1. It determines precisely in what way these techniques modify the standard assumptions of RL and how these biases might be applied to R-MAX. The resulting understanding makes possible a unified algorithm that combines all three techniques into a single algorithm. The goal of this effort is to develop an algorithm that accepts some forms of intuitive knowledge from a human user, such as the gross hierarchical structure of the environment and which actions and state variables are independent of one another. By combining model-based reasoning, function approximation, and hierarchical decomposition, this thesis provides a *language of inductive bias* that allows users to instruct a scientific agent how to generalize from data.

1.3 Contributions

This thesis makes the following contributions to RL:

Model-Based Reinforcement Learning in Continuous Environments. The combination of the model-based R-MAX algorithm and the function approximation of the Fitted Value Iteration planning algorithm (Gordon, 1995) yields Fitted R-MAX, the first algorithm to apply model-based exploration methods to continuous-state environments in the general stochastic case. A key advantage of this algorithm is its ability to generalize in the relatively simple space of predicted action effects, not just in the space of estimated state values.

Investigation into the Utility of Hierarchical Decomposition. Although hierarchy has intuitive appeal and has improved the performance of popular model-free algorithms, the precise manner in which it helps RL was not well understood. Careful experimentation reveals confounding factors in prior demonstrations on the utility of hierarchy, as well as evidence revealing the connection between hierarchy and exploration. In particular, hierarchy's value lies in determining and constraining the states and actions that an agent must explore, not merely in encouraging agents to visit critical subgoal states.

Hierarchical Model-Based Reinforcement Learning. The combination of the model-based R-MAX algorithm and the MAXQ framework for hierarchical task decomposition yields R-MAXQ, the first algorithm to learn hierarchies of action models in the standard discounted-reward RL setting. Prior algorithms relied on model-free stochastic approximation techniques that required slower learning at high levels of the hierarchy so that lower levels would converge first. A fully model-based approach permits simultaneous learning across levels, with exploration needs propagating in a structured manner up the hierarchy.

Compositional Models for Reinforcement Learning. This framework casts function approximation, hierarchical decomposition, and the R-MAX approach to exploration in a simple, unified representation as model operators that may be easily composed in a general model-learning algorithm. This framework facilitates the definition of Fitted R-MAXQ, the first RL algorithm to combine these three techniques in the general case.

1.4 Overview

The remainder of the thesis is organized as follows:

Chapter 2 defines the reinforcement learning problem and describes the standard Markov decision process formalism. It also recaps the R-MAX algorithm, which serves as the basis for the novel algorithms presented in later chapters.

Chapter 3 considers the connection between generalization and exploration.

It combines the function approximation of fitted value iteration and the model-based exploration of the R-max algorithm to produce the fitted R-max algorithm.

Chapter 4 examines the role of hierarchy in RL, as a natural mechanism for instilling domain knowledge into an agent and constraining its exploration.

Chapter 5 draws upon the conclusions of the preceding chapter to motivate a learning algorithm that benefits from both aggressive exploration and hierarchical prior knowledge. It combines the hierarchical decomposition of maxq with the model-based exploration of the R-max algorithm to produce the R-maxq algorithm.

Chapter 6 proposes a compositional model framework that facilitates the development of a single algorithm that combines model-based exploration, function approximation, and hierarchical decomposition.

Chapter 7 describes the practical implementation of the Fitted R-maxq algorithm. The computationally intense nature of the algorithm requires careful software engineering to achieve reasonable running times.

Chapter 8 addresses the discovery problem and discusses its connection to the idea of active exploration. It includes some preliminary results towards the automatic discovery of abstractions.

Chapter 9 covers related work, both in the area of intelligent exploration and in the combination of model-based, hierarchical, and approximate learning techniques.

Chapter 10 considers future work and offers concluding remarks.

Chapter 2

Foundations

This chapter recapitulates the existing results in the RL literature that form the basis of this thesis. Section 2.1 defines the learning problem using the standard formalism of Markov decision processes, and it establishes the notation adopted throughout the thesis. Section 2.2 reviews the standard approach to the RL problem, in which an agent estimates the optimal value function for its environment. Section 2.3 discusses the exploration problem and reviews the R-MAX algorithm, which serves as the foundation for the algorithmic contributions of the thesis.

2.1 The Reinforcement Learning Problem

Robustness to uncertainty is one important motivation for learning, so the RL framework minimizes the assumptions it makes about the learning problem. Each problem comprises an agent and an environment. The learning algorithm fully determines the behavior of the agent, which executes a sequence of actions chosen from some action space A. In contrast, the algorithm may have no prior knowledge of the environment, which generates a sequence of states from some state space S, as well as a sequence of real-valued rewards.

The agent and the environment interact on a discrete time scale: at time step t, the agent observes the current state $s_t \in S$ and chooses an action $a_t \in A$. In response to a_t, the environment generates an immediate reward $r_{t+1} \in \mathbb{R}$ and a successor state $s_{t+1} \in S$. Figure 2.1 illustrates this interaction. For now, the state space S and action space A are assumed to be finite.

Roughly speaking, the goal of the agent at a given time step is to maximize the expected sum of future rewards, $\sum_{k=1}^{\infty} r_{t+k}$. In the general case, this sum may be unbounded, but the objective can be made well-defined either by specifying a finite horizon or using a discount factor γ to reduce the weight of future rewards. This thesis adopts the latter approach, which has the benefit that the agent's goal remains the same after each time step. Given a discount factor with $0 \leq \gamma \leq 1$, the agent attempts to maximize the expected *return*, $\sum_{k=1}^{\infty} \gamma^k r_{t+k}$.

States, actions, and rewards define the interface between agent and environment, and the return defines the objective of the agent. However, additional assumptions about the agent-environment system are necessary to give the agent any possibility of achieving its objective. Otherwise, the be-

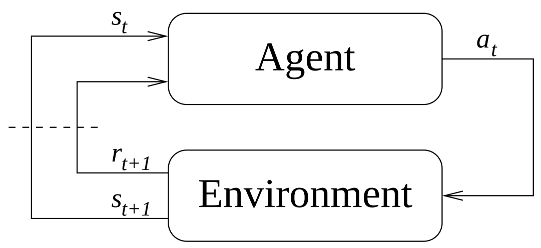


Figure 2.1: The agent-environment system.

havior of the environment at the current time step may be complete unrelated to its behavior at other time steps, foiling any attempt at learning. Modern RL algorithms rely on a critical assumption, known as the *Markov assumption*. It states that the output of the environment, the successor state s_{t+1} and immediate reward r_{t+1}, depends only on the preceding state s_t and action a_t. In particular, given the state and action at time t, the behavior of the environment at time $t + 1$ is conditionally independent of all other variables. The joint state-action space $S \times A$ therefore plays a vital role in reasoning about the agent-environment system.

For each environment, there exists a *state-action transition function* $P : S \times A \times S \to [0, 1]$ that gives the probability $P(s, a, s')$ of transitioning from the state-action sa to the successor state s'. It will prove useful to represent this function as a $|S||A| \times |S|$ matrix, so the rows are indexed by state-actions $sa \in S \times A$ and the columns are indexed by successor states $s' \in S$. The notation $P[sa, s']$ denotes the entry in row sa and column s', so that

$$P[sa, s'] = \Pr(s_{t+1} = s' \mid s_t = s, a_t = a). \tag{2.1}$$

Therefore, P is a *stochastic matrix*: each row sums to 1 and contains only non-negative numbers. Similarly, there exists a *reward function* $R : S \times A \to \mathbb{R}$, represented as a $|S||A|$-dimensional column vector, which gives the deterministic reward for the state-action sa, so that $r_{t+1} = R[s_t a_t]$. The combination $\langle S, A, R, P \rangle$ of a state space S, action space A, reward function R, and state transition function P completely defines the behavior of the environment and is known as a *Markov decision process* (MDP).

RL researchers almost universally accept the MDP formalism for environments, but the space of possible agents is less well defined. To facilitate comparisons among agents, this thesis introduces a formalism for agents. A *learning algorithm* $\langle \Sigma, \textsc{Choose}, \textsc{Update} \rangle$ comprises a set Σ of agent states, a subroutine Choose that determines the agent's action at each time step, and a subroutine Update that determines the agent's internal state after it perceives the result of each action, as shown in Figure 2.2. Formally, Choose may be considered a function $S \times \Sigma \times A \times \Sigma \to [0, 1]$ that gives the probability $\textsc{Choose}(s_t, \sigma_t, a_t, \sigma'_t)$ that given environment state s_t and agent state $\sigma_t \in \Sigma$, Choose will output action a_t and change the agent state to σ'_t. Note that Choose may modify the agent state because some algorithms require the agent to modify its internal state in the process of choosing an action, for example when setting goals that will inform action selection over multiple time steps. Similarly, Update may be considered a function $S \times A \times \Sigma \times \mathbb{R} \times S \times \Sigma \to [0, 1]$ that gives the probability $\textsc{Update}(s_t, a_t, \sigma'_t, r_{t+1}, s_{t+1}, \sigma_{t+1})$ that Update will

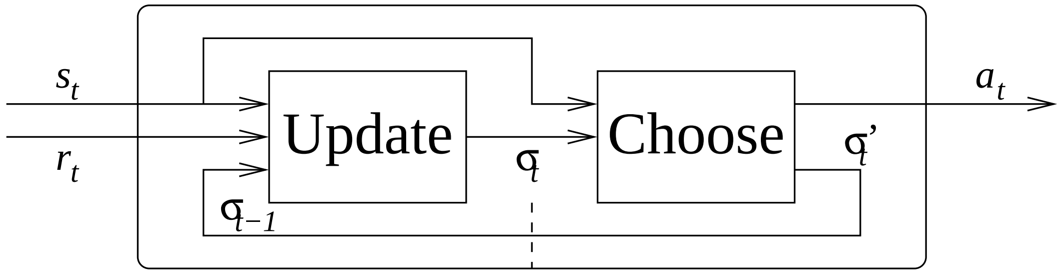


Figure 2.2: Decomposition of an agent into a learning algorithm comprising an Update subroutine and a Choose subroutine.

change the agent state to σ_{t+1} given previous environment state s_t, the agent action a_t and agent state σ'_t output by Choose, and the resulting reward r_{t+1} and environment state s_{t+1}.

Together, an MDP and a learning algorithm completely specify how the agent-environment system evolves. For a given environment state s_t and agent state σ_t, the Choose subroutine determines the agent's action a_t and updates the agent's state to σ'_t. The MDP reward function R determines the one-step reward r_{t+1}, and the MDP state transition function P determines the successor state s_{t+1}. Finally, the Update subroutine changes the agent state to σ_{t+1}. Specifying an initial environment state s_0 and agent state σ_0 (or a fixed distribution over initial states) therefore completely defines a stochastic process. Given a duration $t \in \mathbb{N}$, this stochastic process in turn determines the distribution over the *history* $h_t = s_0\sigma_0 a_0 \sigma'_0 r_1 s_1 \sigma_1 a_1 \sigma'_1 r_2 s_2 \sigma_2 \ldots s_t \sigma_t$. This joint agent-environment system also determines the distribution over possible returns, $\sum_{k=1}^{\infty} \gamma^k r_{t+k}$.

For clarity, this thesis will give individual values of Choose and Update in pseudocode. This pseudocode will priortize readability over efficiency, but Chapter 7 will discuss the software architecture of a practical implementation. In general, the agent state σ includes all of the data structures that the agent modifies during the course of its interactions with the environment. Within pseudocode, the variables that comprise the agent state will have the superscript $^\sigma$. Whenever possible, superscripts in this thesis will denote a component relationship, so that x^σ suggests that x is a component of σ. In fact,

σ may be thought of as a C-style data structure with a member x, with x^σ serving as a compact substitute for $\sigma.x$. The variables within pseudocode will fall into three categories:

1. Algorithm parameters, treated as global constants
2. Transient state, treated as local variables either passed as an argument or assigned within the scope of the pseudocode
3. Agent state, contained within σ, which is both an implicit argument and an implicit component of the return value for each block of pseudocode

2.2 Exploitation

Given a particular MDP, there exists an agent that maximizes the expected return at $t = 0$. Of course, what agent maximizes the return depends on the reward and state transition functions, which are unknown in the RL setting. While the goal of learning is to adapt to uncertainty about the environment, modern RL algorithms emerged from research into planning algorithms that assumed complete knowledge of the MDP. Therefore it is useful to begin by considering the case where the agent already knows the rewards and dynamics of the environment, and all that remains is to exploit this knowledge to choose the optimal behavior.

2.2.1 Planning Algorithms

To choose an action a_t an agent may use the entire history h_t, but the Markov assumption implies that knowledge of the current state s_t (along with knowledge of R and P) suffices to choose a_t optimally. A policy $\pi : S \times A \to [0,1]$ specifies the probability $\pi(s_t, a_t)$ of choosing action a_t given state s_t. Although a policy may thus be represented as a $|S| \times |A|$ matrix, it will prove convenient to define a *policy* π as a $|S| \times |S||A|$ stochastic matrix, so that the rows are indexed by states $s \in S$ and the columns are indexed by state-actions $sa \in S \times A$, which also index the rows of R and P. In a sense, a policy "transitions" from the state s_t to the joint state-action $s_t a_t$ at a given time step, subject to the usual constraints on stochastic matrices and the additional restriction that $\pi[s, xa] > 0 \implies s = x$ for all $s, x \in S$ and $a \in A$. This restriction formalizes the fact that an agent at time t cannot affect the value of s_t. This construction yields a concise notation for many of the important quantities to be defined later. As a simple example, the $|S|$-dimensional vector πR, obtained simply by multiplying a policy π with the reward function R, gives the expected value of r_{t+1} for each $s_t \in S$, assuming the agent chooses a_t according to π.

If an agent always chooses a_t according to a fixed policy π, then the *value function* $V^\pi : S \to \mathbb{R}$, represented as a $|S|$-dimensional vector, specifies the expected return $V^\pi[s]$ for each state $s \in S$. For a given MDP $\langle S, A, R, P \rangle$, policy π, and discount factor γ, V^π can be computed by solving what is known as the Bellman equation:

$$V^{\pi} = \pi(R + \gamma P V^{\pi}). \tag{2.2}$$

The value function therefore evaluates the quality of a given policy. In particular, there exists an *optimal policy* π^* that maximizes the value of every state simultaneously: for all $s \in S$, $V^{\pi^*}[s] = \max_{\pi} V^{\pi}[s]$.

Planning algorithms compute optimal policies for an agent given full knowledge of the MDP describing the environment. Standard algorithms take advantage of the relationship between a policy and its induced *state-action value function* Q^{π}, a $|S||A|$-dimensional vector defined as follows:

$$Q^{\pi} = R + \gamma P V^{\pi}. \tag{2.3}$$

From this definition follow some other useful equations:

$$V^{\pi} = \pi Q^{\pi} \tag{2.4}$$

$$Q^{\pi} = R + \gamma P \pi Q^{\pi}. \tag{2.5}$$

Intuitively, $Q^{\pi}[s_t a_t]$ estimates the return after the current time t assuming the system is in state s_t, the agent first executes a_t, and that the agent chooses future actions according to π. Given the state-action value function Q^{π} for a policy π, an improved policy π' can be obtained by choosing for each state s those actions that maximize Q^{π}. In particular, π' should satisfy

$$\pi'[s, sa] > 0 \implies Q^{\pi}[sa] \geq Q^{\pi}[sb] \tag{2.6}$$

for all $s \in S$ and $a, b \in A$. In this case, $V^{\pi'}$ will dominate V^{π}, with equality only if π was already optimal.

Policy iteration is one simple algorithm that applies this idea. Starting with an arbitrary intial policy π, it alternates between policy evaluation, which computes Q^{π} by solving Equation (2.5), and policy improvement, as described in the preceding paragraph. Regardless of the initial policy, this process converges to an optimal policy (Littman et al., 1995). Algorithm 2.1 presents this algorithm in pseudocode.

Algorithm 2.1 PolicyIteration(R, P, π_0)

```
$Q_0 \leftarrow$ solution to $Q_0 = R + \gamma P Q_0$        {Policy evaluation}
$i \leftarrow 0$
repeat
  $i \leftarrow i + 1$
  $\pi_i \leftarrow$ chosen so that $\pi_i[s, sa] > 0 \Rightarrow a \in \text{argmax}_a Q_{i-1}[sa]$        {Policy improvement}
  $Q_i \leftarrow$ solution to $Q_i = R + \gamma P Q_i$        {Policy evaluation}
  $Q_i \leftarrow$ PolicyEvaluation$(R, P, \pi_i)$        {solve (2.5)}
until $Q_i = Q_{i-1}$
return $\pi_i$
```

Note that the termination condition of the main loop in policy iteration tests the last two value functions for equality instead of the last two policies, since optimal value functions are unique while an agent-environment system may have several optimal policies that give rise to that value function. For this reason, it is not surprising that many methods attempt to compute the optimal value function directly. The value iteration algorithm, specified in Algorithm 2.2, iteratively improves an estimate of the optimal value function and only implicitly improves the current policy, by setting the state value function as the maximum over actions of the state-action value function. This algorithm only converges to the optimal value function asymptotically, but the iteration typically ends once no estimated value changes by more than some small quantity $\epsilon > 0$. Note that the notation $[s] \mapsto \max_a Q_i[sa]$ refers to a vector where the entry indexed by s has the value $\max_a Q_i[sa]$.

Algorithm 2.2 ValueIteration$(R, P, \epsilon, V_0,)$

```
i ← 0
repeat
  i ← i + 1
  $Q_i \leftarrow R + \gamma P V_{i-1}$                {Some policy evaluation}
  $V \leftarrow [s] \mapsto \max_a Q_i[sa]$          {implicit policy improvement}
until $\|V_i - V_{i-1}\|_\infty < \epsilon$
return $V_i$
```

Given the converged value function, the optimal policy may be recovered by computing the state-action value function using Equation (2.3) and applying policy improvement. Algorithms 2.3 and 2.4 make this point explicit by together defining a complete agent that simply employs value iteration to choose each action. To clarify the algorithm's inputs, it begins by reading the expected components of the agent state σ into local variables. Note that this agent requires the state transition matrix and reward vector as prior knowledge in the initial agent state σ. When choosing actions, it needs only to compute the optimal policy given this model, using value iteration. Since it assumes its prior model is correct, it does not perform any learning in its Update subroutine, Algorithm 2.4.

Algorithm 2.3 ValueIterationChoose(s)

```
$V^\sigma \leftarrow$ ValueIteration($R, P, \epsilon, V^\sigma$)
$Q \leftarrow R + \gamma P V^\sigma$
$\pi \leftarrow$ chosen so that $\pi[s, sa] > 0 \Rightarrow a \in \text{argmax}_a Q[sa]$    {Policy improvement}
$a \leftarrow$ chosen with probability $\pi[s, sa]$
**return** $a$
```

Algorithm 2.4 NoUpdate(s, a, r, s')

```
{No change to $\sigma$ is necessary}
```

2.2.2 The Q-Learning Algorithm

The planning algorithms just described require knowledge of the state transition and reward functions to compute optimal policies. Without this knowledge, an agent must rely on data from interaction with the environment. The *Q-learning* algorithm gave rise to the field of modern RL because it demonstrated that despite the noise in this data, an agent can still converge to the optimal value function. The basic algorithm incrementally improves the state-action value function Q once for each piece of data $\langle s_t, a_t, r_{t+1}, s_{t+1} \rangle$, known as an *instance*. Aside from revealing the reward for a state-action, each instance serves as a sample from the unknown state transition function. Q-learning uses stochastic approximation techniques to shift its value function towards Q^*, taking a weighted average of the current value and a value computed using the sampled reward and transition. The learning rate α specifies the relative weights of this average.

Algorithm 2.5 implements this idea, with the notation $x \overset{\alpha}{\leftarrow} y$ serving as a shorthand for $x \leftarrow (1-\alpha)x + \alpha(y - x)$ and describing the incremental stochastic approximation of x given a sample y. Algorithm 2.5 only specifies the UPDATE half of a complete learning agent: it outputs a value function and not an action. Since the output value function is an estimate of the optimal value function, it seems reasonable to apply the policy obtained via policy improvement given this value function. However, this value function has only been shown to converge to the optimal value function under certain conditions. The most important of these conditions is that the agent must visit every reachable state-action infinitely often.[1]

Algorithm 2.5 QLEARNINGUPDATE(s, a, r, s')

```
    $Q^\sigma[sa] \overset{\alpha}{\leftarrow} r + \gamma \max_{a'} Q[s'a']$
```

To achieve this condition, a common approach is to add random noise to the policy induced by the current value function. For example, the popular ϵ-greedy method only chooses an action from the estimated optimal policy with probability $1 - \epsilon$ (Algorithm 2.6). Otherwise, it chooses an action uniformly at random. This method maintains a nonzero probability of long sequences of random actions, which in turn imply a nonzero probability of visiting any given state-action. This condition compensates for the fact that the greedy policy might never visit certain state-action pairs, whose values may be under-estimated due to random noise or lack of data. In practice, this approach is highly inefficient. Many of the random actions do not lead to the acquisition

[1]The learning rate α must also decrease at an appropriate rate in QLEARNINGUPDATE, although most practical implementations don't satisfy this condition.

of useful data, since several consecutive non-greedy actions may be necessary to reach states with underestimated values.

Algorithm 2.6 EPSILONGREEDYCHOOSE(s)

```
$p \leftarrow$ drawn uniformly at random from $(0, 1)$
**if** $p < \epsilon$ **then**
    $a \leftarrow$ drawn uniformly at random from $A$
**else**
    $a \leftarrow$ chosen such that $a \in \text{argmax}_a Q[sa]$
**end if**
**return** $a$
```

2.3 Exploration

The elegance and simplicity of Q-learning comes at a significant cost: the algorithm only directly addresses the issue of exploitation. Given the available data, it estimates the fixed optimal policy, but nothing about its design acknowledges the availability of future data. In particular, one of the assumptions required to ensure convergence to optimal behavior is exactly a solution to the exploration problem: sufficient data from all reachable state-actions. The random methods used in practice to achieve this condition are too inefficient for most realistic applications.

Despite these limitations, the vast majority of RL algorithms still focus exclusively on the problem of how best to estimate the optimal value function given past data, instead of considering what future data would help improve future estimates of this value function. The problem of how to explore an environment properly may seem hopelessly ill-defined, but Section 2.3.1 discusses a theoretically optimal but intractable solution. Section 2.3.2 then describes the more feasible R-max algorithm and its appealing theoretical results, along with a discussion of the practical limitations that this thesis will address in seeking to apply this algorithm to problems with more realistic qualities.

2.3.1 Optimal Exploration

For any given environment, an optimal policy exists, and the optimal action or actions depend only on the current environment state. Of course, a learning agent does not know a priori what environment it faces, so it must explore its environment. During the course of exploration, the agent's policy may change as it learns: its behavior at a given state depends on its current knowledge. Although the agent's policy is in this sense nonstationary, the Update and Choose subroutines depend only on the current environment state and current agent state. Therefore, another way to formalize a learning algorithm is as a stationary policy in a "meta state space" comprising both environment states and agent states.

A reasonable question then is whether a planning algorithm could compute the optimal policy for this meta-problem, and thereby determine an optimal learning algorithm. This feat is theoretically possible, by adopting a Bayesian approach and assuming the existence of some prior belief about the possible values of the reward and state transition functions. In particular, let B be the belief space over possible MDPs: the set of joint probability distributions over reward functions R and state transition functions P. The optimal agent's state σ includes its belief state $b \in B$, which specifies the joint probabity density $\Pr(R, P \mid b)$.

The optimal agent must be able to update its belief state each time step, as it gathers data. It can compute the posterior distrubution $\Pr(R, P \mid b')$, given its prior distribution b and an instance $\langle s, a, r, s' \rangle$, using Bayesian conditioning. This process updates any joint distribution over models by increasing the density given to models consistent with the data:

$$\begin{aligned}
\Pr(R, P \mid b') &= \Pr(R, P \mid b, s, a, r, s') && (2.7)\\
&\propto \Pr(r, s' \mid R, P, b, s, a) \Pr(R, P \mid b, s, a) && (2.8)\\
&\propto \Pr(r \mid R, s, a) \Pr(s' \mid P, s, a) \Pr(R, P \mid b, s, a) && (2.9)\\
&\propto \delta(r, R[sa]) P[sa, s'] \Pr(R, P \mid b). && (2.10)
\end{aligned}$$

Algorithm 2.7 details the corresponding Update subroutine.

Algorithm 2.7 OptimalUpdate(σ, r, s')

```
  $b' \leftarrow$ chosen to satisfy (2.10)
  $b \leftarrow b'$
```

In the Choose subroutine, the optimal agent must execute the optimal policy for the *belief-state MDP* $\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P} \rangle$. This continuous-state MDP shares the same action space $\mathcal{A} = A$ as the original MDP, but its state space $\mathcal{S} = B \times S$ includes both the actual environment state and the agent's belief about the environment. The belief-state reward function $\mathcal{R}$ predicts immediate rewards using the expectation taken across possible reward functions:

$$\begin{aligned}
\mathcal{R}(bs, a) &= \int r \Pr(r \mid b, s, a) \, \mathrm{d}r && (2.11) \\
&= \int r \int \delta(r, R[sa]) \, \mathrm{d}R \, \mathrm{d}r && (2.12) \\
&= \iint r\delta(r, R[sa]) \, \mathrm{d}R \, \mathrm{d}r. && (2.13)
\end{aligned}$$

Finally, the state transition function $\mathcal{P}$ must predict both the transitions among the actual states $s \in S$ as well as the belief states $b \in B$. For any given joint state $bs \in B \times S$ and action $a \in A$, the belief-state MDP first transitions the environment state by again taking an expectation across possible models. Since the OptimalUpdate subroutine is deterministic, the successor (posterior) belief state depends only on the previous state and the predicted successor state (and reward). Therefore, $\mathcal{P}(bs, a, b's') = 0$ for all b' that does not satisfy Equation (2.10)[2]. For b' consistent with the predicted data, the belief-state MDP transition function is given by:

$$\begin{aligned}
\mathcal{P}(bs, a, b's') &= \Pr(r, s' \mid b, s, a) && (2.14) \\
&= \iint \delta(r, R[sa]) P[sa, s'] \Pr(R, P \mid b) \, \mathrm{d}R \, \mathrm{d}P. && (2.15)
\end{aligned}$$

With a well-defined state space, action space, reward function, and state transition function, the belief-state MDP has an optimal value function and therefore an optimal policy. This policy is optimal in the sense that the agent maximizes its expected future discounted rewards, subject to the

[2]This equation must further be averaged across predicted rewards.

uncertainty about the rewards and transitions encoded by its prior beliefs. The value of a given belief-state will depend on the actual state, the most likely rewards and transitions, and the amount of variance in the estimated dynamics. The policy will therefore balance exploiting the estimated dynamics with the value of gathering information about the dynamics. If this policy could be computed, it could be used to complete the definition of an optimal agent, as shown in Algorithm 2.8.

Algorithm 2.8 OPTIMALCHOOSE(s)

```
  $\mathcal{S} \leftarrow B \times S$
  $\mathcal{A} \leftarrow A$
  $\mathcal{R} \leftarrow$ chosen to satisfy (2.13)
  $\mathcal{P} \leftarrow$ chosen to satisfy (2.15)
  $\pi \leftarrow$ optimal policy for $\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P} \rangle$
  $a \leftarrow$ chosen with probability $\pi(bs)$
  return $a$
```

Unfortunately, this optimal value function and policy is infeasible to compute. Each action at a given state will reveal that state-action's deterministic reward and transition to one of finitely many successor environment states. For a fixed underlying MDP, each state of the belief-state MDP only has a finite number of immediate successors (since the belief-state transitions are deterministic given the environment transitions), but the number of reachable belief states is infinite: the variance of the belief state decreases as more data accumulates. Despite the infeasibility of this formulation, the existence of a theoretically optimal learning algorithm can inform practical approximations. Considering the belief-state MDP approach leads to the following ideas concerning the problem of efficient exploration and exploitation.

Beliefs as well as states have value. A given state-action may have value beyond the immediate reward it earns and the capacity of the successor state for future rewards. It can lead to data that refines the agent's beliefs and improves the future ability of the agent to exploit.

Prior beliefs determine optimal exploration behavior. The belief-state MDP formulation does not distinguish between initial agent states, intermediate agent states, and "converged" agent states. There is no sense in which the agent can start with a truly blank slate. Instead, it must begin with some assumptions about the environment, in the sense that even before interacting with the environment, the agent has a probability distribution weighting the possible state transition and reward functions. Furthermore, this prior belief plays a critical role in the agent's strategy.

Learning can be solved with planning. Coherent beliefs about the possible environments allow an agent to reason about the exploration-exploitation tradeoff as a planning problem.

2.3.2 The R-MAX Algorithm

Researchers first established the ability to converge to near-optimal behavior in any MDP in polynomial time by proposing an algorithm, E^3, that explicitly reasons about exploration (Kearns & Singh, 1998). The R-MAX algorithm improved upon these bounds by using a simpler algorithm that implicitly balances between exploration and exploitation (Brafman & Tennenholtz, 2002). R-max serves as the foundation of the algorithmic contributions of this thesis, so this section first defines the algorithm precisely and then discusses its capabilities and shortcomings.

The R-max algorithm, defined in Algorithms 2.9 and 2.10, applies a policy obtained by planning with a special MDP derived from the available data. Intuitively, this MDP is defined on a row-by-row basis, depending on the amount of data available for the state-action corresponding to each row, measured as the number of times $N[sa]$ that the history includes state s and action a at the same time step. If $N[sa]$ reaches a given threshold m, then the standard maximum-likelihood estimate is used for row sa of R and P. Otherise, R-max assumes an optimistic model of the state-action: an immediate reward equal to some given upper bound $r^{\max}$ and a deterministic transition back to s, which would allow the agent to obtain this optimistic reward every time step.

Algorithm 2.9 MaximumLikelihoodUpdate(s, a, r, s')

```
  $R^\sigma[sa] \leftarrow r$                                   {Update reward model}
  $D^\sigma_P[sa, s'] \leftarrow D^\sigma_P[sa, s'] + 1$        {Update transition data}
  $N^\sigma[sa] \leftarrow N^\sigma[sa] + 1$                    {Update sample sizes}
  for all $x' \in S$ do
    $P^\sigma[sa, x'] \leftarrow \frac{D^\sigma_P[sa,x']}{N^\sigma[sa]}$
  end for
```

The optimistic rows of R and P lead R-max to explore state-actions for which it has insufficient data. This effect applies not just when the current

Algorithm 2.10 RMaxChoose(s)

```
$R \leftarrow R^\sigma$                                   {MLE reward vector}
$P \leftarrow P^\sigma$                                   {MLE transition matrix}
for all $sa \in S \times A$ do {Modify MLE model}
  if $N[sa] < m$ then {Sample size smaller than threshold}
    $R[sa] \leftarrow r^{\max}$                           {Optimistic reward}
    for all $s' \in S$ do
      $P[sa, s'] \leftarrow 0$
    end for
    $P[sa, s] \leftarrow 1$                               {Optimistic transition}
  end if
end for
$V^\sigma \leftarrow$ ValueIteration($R, P, \epsilon, V^\sigma$)       {Planning}
$Q \leftarrow R + \gamma P V^\sigma$
$\pi \leftarrow$ chosen so that $\pi[s, sa] > 0 \Rightarrow a \in \text{argmax}_a Q[sa]$
$a \leftarrow$ chosen with probability $\pi[s, sa]$
return $a$
```

state s_t has an unexplored action but also whenever planning can find a trajectory from the current state to an unexplored state-action. Although a given plan might not guarantee that the agent will attain data from an unexplored state, planning will automatically balance the rewards from exploitation with the optimistic reward for exploration.

It is instructive to compare R-max with the optimal Bayesian approach to exploration described in Section 2.3.1. Both approaches reduce the problems of learning and exploration to a planning problem. The first major difference is representational: R-max maintains a single estimate of R and P instead of a probability distribution over possible MDPs. For the known state-actions, planning with the single maximum-likelihood model can be seen as an approximation of planning with a distribution of models that will become increasingly peaked around the maximum-likelihood model as the amount of data increases. For the unknown state-actions, the optimistic model can be seen ironically as a worst-case assumption about the space of MDPs consistent with the known state-actions. From the perspective of exploiting the known state-actions, the worst-case scenario is that the unknown state-actions are actually optimal. In this sense, the optimistic model chooses the one estimate for each unknown state-action that guarantees that the planned policy will not overlook a possibly optimal state-action. However, R-MAX does not distinguish between state-actions with no data and those with almost enough data to be known. In the latter case, if the state-action in question is suboptimal enough, the Bayesian approach can conclude that the state-action is suboptimal given just the available data. Some more recent RL algorithms adopt a more complex and computationally intensive approach using confidence intervals, instead of choosing the purely best-case model (Strehl & Littman, 2005).

The second major difference is that R-MAX generates a plan that does not explicitly acknowledge the fact that the agent will gather more data in the future. In contrast, the Bayesian approach outputs a policy for a belief-state MDP, where the state incorporates future data by way of its effects on the belief-state. R-MAX partially compensates for this myopic perspective on data by recomputing the policy after each acquisition of useful data.

A final major difference is that R-MAX does not explicitly incorporate a prior distribution over possible MDPs. In a sense, R-MAX again takes a worst-case perspective: that each state-action cannot be known until the agent collects some (large) amount of data for that state-action. This assumption corresponds to a prior distribution in which each state-action is statistically independent: even full knowledge of one state-action does not affect the marginal distribution of any other state-action.

Clearly, this class of prior distributions is not the only one compatible with the optimal Bayesian approach to learning, and its adoption by R-MAX leads to a common complaint about attempts to use R-MAX in practice. By seeking to explore every state-action several times, R-MAX incurs an enormous cost of exploration (Hester & Stone, 2009).

In concrete terms, this thesis introduces the possibility of stronger prior beliefs to the R-MAX algorithm, allowing it to apply in practice to larger and more complex environments than in the largely theoretical research that developed the original algorithm. In one sense, it seeks classes of assumptions that approximate more structured prior distributions over MDPs, without incurring the full cost of reasoning about arbitrary distributions over reward and state transition functions.

Chapter 3

Generalization

From the perspective of applying RL to realistic problems, the most glaring shortcoming of standard algorithms is their assumption of finite state spaces and tabular representations of knowledge. Q-learning maintains a single primary data structure, the $|S||A|$-dimensional vector Q, and its known convergence guarantees assume that the agent updates and therefore visits each state-action infinitely often. However, when S includes uncountably many states, such as is often the case in applications when state variables are real-valued, then not even an infinite history can hope to visit each state even once, let alone infinitely often. A more immediate problem is the data structure used to store Q. A table or array with one entry for each entry of Q is infeasible for even a very large finite state space, as well as any infinite state space.

The large number of states in real-world problems underscores the need for generalization: agents cannot afford to treat every nominal state as unique and independent. Section 3.1 describes standard approximation techniques that reduce the number of degrees of freedom in the learned value function to make learning feasible in infinite state spaces. Section 3.2 describes one of this thesis's primary contributions, an extension of this form of approximation to learned models of the environment. This extension permits the extension of model-based RL algorithms such as R-MAX to infinite state spaces, yielding the Fitted R-MAX algorithm (Jong & Stone, 2007b; Jong & Stone, 2007a) described in Section 3.2.2. Finally, Section 3.3 presents experimental results with Fitted R-MAX.

3.1 Approximating the Value Function

Feasible algorithms that compute value functions for infinite state spaces necessarily employ some form of *function approximation*, since no finite data structure can represent all possible functions over an infinite domain, even granted the finite floating-point representation of real numbers. It follows that RL algorithms adopt a hard bias, whether explicitly on implicitly. They assume that value functions lie in the space of functions spanned by their representation. The key challenges in research on function approximation are what the representation (and thus the space of value functions) should be, and how to compute the value function efficiently given a particular representation.

These challenges remain even when the reward and state transition functions are known, so function approximation remains an open problem in planning research, as well. This thesis builds off of the fitted value iteration (FVI) algorithm, which provides a particularly clean framework for reasoning about value approximation. FVI differs from standard value iteration, which was described in Section 2.2.1, in that it interleaves an additional fitting step with the standard value backup step. In particular, after computing an improved value function V_{i+1} by backing up the fitted value function $\hat{V}_i$, it approximates V_{i+1} with a fitted value function $\hat{V}_{i+1}$ from some restricted space of value functions. Algorithm 3.1 details this modification to the standard value iteration algorithm (Algorithm 2.2).

Algorithm 3.1 FittedValueIteration(R, P, ϵ, V_0)

```
$i \leftarrow 0$
$\hat{V}_i \leftarrow$ Approximate$(V_i)$
repeat
  $i \leftarrow i + 1$
  $Q_i \leftarrow R + \gamma P \hat{V}_{i-1}$                    {Some policy evaluation}
  $V_i \leftarrow [s] \mapsto \max_a Q_i[sa]$            {Implicit policy improvement}
  $\hat{V}_i \leftarrow$ Approximate$(V_i)$
until $\|\hat{V}_i - \hat{V}_{i-1}\|_\infty < \epsilon$
return $\hat{V}_i$
```

For some choices of approximation, even this straightforward planning algorithm may diverge, but prior work showed that the sequence of value functions $\{\hat{V}_i\}$ converges to a stable fixed point if Approximate is implemented using an averager (Gordon, 1995). An *averager* approximates the value of a state in S as a weighted average of the values of a finite subset $X \subset S$. In particular, let Φ be a $|S| \times |S|$ stochastic matrix such that $s' \notin X \implies \Phi[s, s'] = 0$ for all $s, s' \in S$, and $\Phi[x, x] = 1$ for all $x \in X$. The row of Φ indexed by s gives the weights over X that specify how to compute $\hat{V}_i[s] = \sum_{x \in X} \Phi[s, x] V_i[x]$ as a weighted average of the values of states in X. FVI using an averager Φ thus implements Approximate$(V_i) = \Phi V_i$.

To understand how averagers allow FVI to converge to a fixed point, first consider one of the chief benefits to the practical implementation of the algorithm. Since $\hat{V}_i$ only depends on the entries of V_i that correspond to X, it suffices only to compute the values of each $x \in X$. While iterating through the main loop, FVI never uses those rows of R, P, or V_i that correspond to states outside of the finite set X. FVI with an averager can therefore be thought of as conducting exact planning with a finite MDP, whose state space is X, derived from the combination of Φ and the original infinite MDP.

This finite derived MDP, specified by the state space X, the action space A, a $|X||A|$-dimensional reward vector $\tilde{R}$, and a $|X||A| \times |X|$ state transition matrix $\tilde{P}$, can be constructed as follows. Define

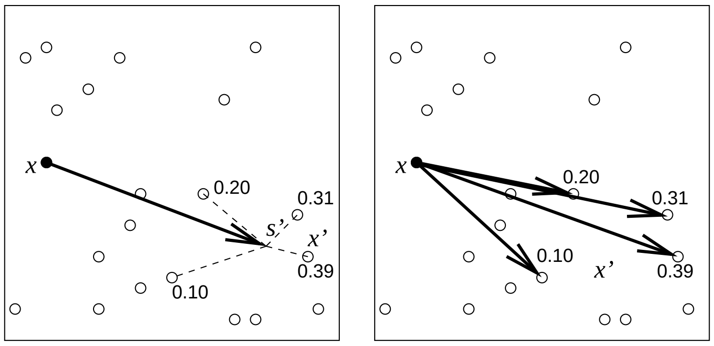
$$\tilde{R}[xa] = R[xa] \tag{3.1}$$

for all $x \in X$ and $a \in A$. Define

$$\tilde{P}[xa, x'] = \sum_{s' \in S} P[xa, s']\Phi[s', x'] \,, \tag{3.2}$$

so that the row of $\tilde{P}$ indexed by xa is equal to the row of $P\Phi$ indexed by xa. (Note that $\tilde{P}$ is therefore a submatrix of the $|S| \times |S|$ matrix $P\Phi$.) Figure 3.1 illustrates the construction of $\tilde{P}$ for one particular state-action sa. Let $\tilde{V}$ be the value function obtained by running value iteration on $\tilde{R}$ and $\tilde{P}$. Then $\tilde{V}$ assigns the same values to states in X as the final value function in Algorithm 3.1. Since value iteration converges to a fixed point for any MDP, it follows that fitted value iteration converges to a fixed point for any combination of an MDP and an averager.

The complete FVI agent given in Algorithm 3.2 (combined with NoUpdate, Algorithm 2.4) makes this construction explicit. Instead of invoking



(a) (b)

Figure 3.1: (a) The arrow in this diagram shows the deterministic transition from x to s' predicted by the state transition function P. The circles represent other elements of X. The dashed lines and accompanying probabilities represent the nonzero elements of $\Phi[s', x]$, indicating how to approximate s' as a weighted average of elements of X. (b) The possible transitions predicted by the resulting fitted model, $\hat{P}$, along with their transition probabilities. If the original state transitions function P were stochastic, $\hat{P}$ would be a summation or integration over possible values of s'.

FVI (Algorithm 3.1), it constructs the fitted MDP and then directly invokes standard value iteration (Algorithm 2.2). The value function computed on the fitted MDP is then used to compute the action values for the current state in the trajectory. By computing a converged $\tilde{V}$, FVI essentially reduces planning in the full state-space S to a one-step lookahead, assuming that after one step the system will transition to (and remain in) a state in X.

Note that Algorithm 3.2 still calls for enumeration of S in two spots. First, the computation of $\tilde{P}$ calls for a summation over the possible successors $s' \in S$ for each state-action xa. Similarly, to compute the action values in the current state s_t, it also sums over the possible successor states in S for each action. Practical implementations of FVI either rely on deterministic domains,

Algorithm 3.2 FittedValueIterationChoose(s)

```
                    {Construct model defined on $X$ from model defined on $S$}
for all $x \in X$ do
  for all $a \in A$ do
    $\tilde{R}[xa] \leftarrow R[xa]$
    for all $x' \in X$ do
      $\tilde{P}[xa, x'] \leftarrow \sum_{s' \in S} P[xa, s']\Phi[s', x']$
    end for
  end for
end for
$\tilde{V}^\sigma \leftarrow$ ValueIteration($\tilde{R}, \tilde{P}, \epsilon, \tilde{V}^\sigma$)        {Compute values over $X$}
for all $a \in A$ do {Compute value at $s$}
  $\hat{Q}[sa] \leftarrow R[sa] + \gamma \sum_{x \in X} \sum_{s' \in S} P[sa, s']\Phi[s', x]\tilde{V}[x]$
end for
$a \leftarrow$ chosen from $\text{argmax}_a \hat{Q}[sa]$
return $a$
```

where only one s' has non-zero probability, or on Monte Carlo integration, which essentially replaces the summation (or integral, in continuous domains) over S with a summation over an independent sample from S. In Section 3.2.2, this thesis will apply FVI to an approximate model with only a finite number of possible successors in S, sidestepping this problem.

FVI using averaging approximation provides a clear framework for estimating value functions for large MDPs, but it leaves many details open. These details include the precise choice of averager and the choice of the finite sample X. Section 3.2.2 will address these design considerations, but first Section 3.2 will address the primary obstacle to applying FVI in the RL setting: the absence of direct knowledge of the reward and state transition functions R and P.

3.2 Approximating Models

Almost all of the research into RL algorithms for large and infinite systems rely on model-free methods. These methods only require a compact representation of the value function for the system, and function approximation techniques provide such representations. This thesis seeks to extend the benefits of the model-based R-MAX algorithm to large state spaces as well. Just as the original R-MAX applies standard value iteration to a learned model of a finite MDP, this thesis will apply FVI to a learned model of a potentially infinite MDP. The key missing ingredient is a compact representation of this learned model.

Learning a model is properly a supervised learning problem: given a state and action, predict the immediate reward and next state. Several researchers have applied various regression algorithms to specific instances of this problem with success. However, in the general case, planning algorithms need the complete distribution over successor states, not just the expected successor state. In some applications, the average successor may be a reasonable approximation to the true probability distribution over successor states, but environments with a large degree of stochasticity may require more accurate modeling of the variance in an action's effects. Chapter 4 will also discuss contexts in which an action represents a long sequence of lower-level actions, further increasing the potential stochasticity of the high-level action's transition behavior.

In the general case, model learning is therefore a density estimation problem, with the added wrinkle that the probability density function being estimated (the transition function) is parameterized by the previous state and action. To avoid unwarranted assumptions about the transition dynamics, this thesis adopts a non-parametric, instance-based approach to learning models. This approach is inspired in part by the Kernel-Based Reinforcement Learning (KBRL) algorithm (Ormoneit & Sen, 2002), which converges to the optimal value function in the limit, when the state space is a bounded metric space, and given unbounded amounts of uniformly sampled transition data.

3.2.1 Kernel-Based Reinforcement Learning

Whereas FVI approximates the value of a given state as the weighted average of the values of a finite set of states X, KBRL approximates the dynamics of a given state-action as the weighted average of the observed dynamics in state-action data sampled from the environment. Although the original KBRL algorithm assumed data sampled uniformly from the state-action space to achieve its convergence results, this thesis presents a version of the algorithm that interacts with the environment, as defined in Section 2.1. For a given state-action, KBRL estimates the immediate reward as a weighted average of all the rewards in the data, weighted by a kernel function $K : (S \times A) \times (S \times A) \to [0, 1]$, which defines the similarity $K(sa, s_i a_i)$ between a given state-action sa and a historical state-action $s_i a_i$:

$$\hat{R}[sa] = \frac{\sum_i K(sa, s_i a_i) r_{i+1}}{\sum_i K(sa, s_i a_i)}. \tag{3.3}$$

Similarly, the predicted distribution of successor states is equal to the empirical distribution of successor states observed, weighted by the kernel function:

$$\hat{P}[sa, s'] = \frac{\sum_i K(sa, s_i a_i)\delta(s_{i+1}, s')}{\sum_i K(sa, s_i a_i)}. \tag{3.4}$$

Note that typically, almost all of the terms in the summation in the numerator of Equation (3.4) will be 0. If the agent never visits the same state twice, so that $i \neq j \implies s_i \neq s_j$, then at most one term will be nonzero, corresponding to the instance when the agent previously transitioned into s'. Equation (3.4) also implies that $\hat{P}$ assigns nonzero probability to a number of successors bounded by the amount of data. Figure 3.2 illustrates the approximation of $\hat{P}$ for a state-action xa given three instances that have nonzero kernel weight.

Although the approximate reward function $\hat{R}$ and transition function $\hat{P}$ are defined over the full state-action space $S \times A$, the optimal value function $\hat{V}^*$ for the approximate MDP $\left\langle S, A, \hat{R}, \hat{P} \right\rangle$ can be computed efficiently. Let $X = \{s \in S \mid \exists i(s = s_{i+1})\}$ be the set of observed successor states. Then $\hat{P}$ only predicts successor states in X, similar to the derived state transition matrix used by FVI. It thus suffices to compute the value function only over X, effectively solving the MDP $\left\langle X, A, \tilde{R}, \tilde{P} \right\rangle$, where $\tilde{R}$ and $\tilde{P}$ are simply $\hat{R}$ and $\hat{P}$ restricted to the domain $X \times A$.

Algorithm 3.3 illustrates the resulting algorithm in detail. Since the last visited state s_t is in X, since it is an observed successor state, the action value function $\tilde{Q}$ defined over $X \times A$ suffices to guide the selection of a_t. Therefore, Algorithm 3.3 may be combined with ValueIterationChoose

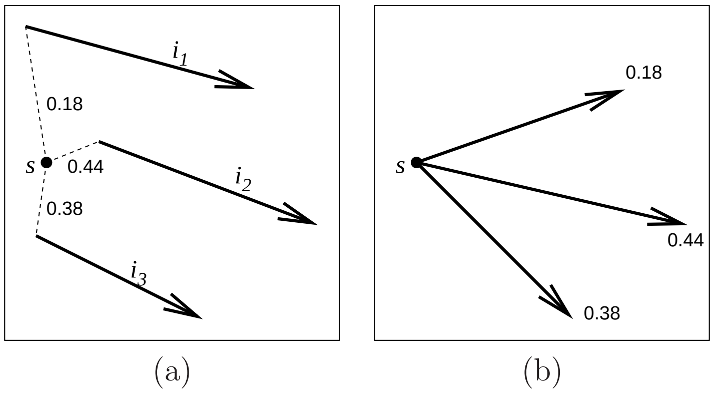


Figure 3.2: (a) Three instances i_1, i_2, and i_3 used to approximate some action at state s. The dashed lines indicate the kernel values $K(sa, s_i a_i)$. (The action a is not represented in the diagram.) (b) The possible transitions predicted by the resulting fitted model, $\hat{P}$, along with their transition probabilities.

(Algorithm 2.3) to obtain a complete agent.

Algorithm 3.3 KernelBasedReinforcementLearningUpdate(s, a, r, s')

```
$X \leftarrow X \cup \{s'\}$                    {Add $s'$ to state sample}
for all $xa' \in X \times A$ do {Define model over $X \times A$}
    $w \leftarrow K(sa, xa')$      {Weight of current instance for approximating $xa'$}
    $\alpha \leftarrow \frac{w}{N^\sigma[xa'] + w}$           {Fraction of cumulative weight}
    $N[xa'] \leftarrow N[xa'] + w$               {Update cumulative weight}
    $R[xa'] \leftarrow (1 - \alpha)R[xa'] + \alpha r$
    for all $x' \in X$ do
        $P[xa, x'] \leftarrow (1 - \alpha)P[xa, x']$
    end for
    $P[xa, s'] \leftarrow P[xa, s'] + \alpha$
end for
```

Although in this case, value iteration effectively only computes the value function $\tilde{Q}$ over $X \times A$, the approximate model defined by Equations (3.3) and (3.4) would permit the computation of the full value function $\hat{Q}$, defined over $S \times A$, as a one-step lookahead:

$$\hat{Q} = \hat{R} + \gamma \hat{P} \tilde{V}. \tag{3.5}$$

As the amount of data increases without bound, $\hat{Q}$ converges in the limit to the true optimal value function Q^* of the underlying MDP, if the data is sampled uniformly at random from $S \times A$ (instead of sampling a single trajectory) and if the kernel K satisfies the following conditions:

1. $K(sa, s'a') > 0$ if $a = a'$,
2. $K(sa, s'a') = 0$ if $a \neq a'$,
3. $K(sa, s'a)$ is a Lipschitz continuous function of $\frac{d(s,s')}{b_t}$,

where d is a metric over the state space and $\{b_t\}$ is a sequence of kernel bandwidths that control the degree of generalization, with $b^{\dim(S)+1}\sqrt{t} \to \infty$ and $b \to 0$ as $t \to \infty$.

KBRL carries many of the properties of Q-learning into the context of continuous state spaces. It guarantees convergence to the optimal value function in the limit, subject to an assumption of sufficient data for each state-action that sidesteps the problem of exploration. The key difference is that instead of requiring infinite data for every state-action, it assumes the reward function is Lipschitz continuous and requires infinite data for increasingly small neighborhoods around every state-action. The requirement that the size of this neighborhood decrease at an appropriate rate mirrors the requirement for the convergence of Q-learning that the learning rate decrease at an appropriate rate.

Although KBRL computes a finite model that approximates a potentially infinite environment, the only guarantees about the quality of this model rely on unbounded amounts of data. Furthermore, the data is assumed to be generated uniformly at random, which is not typically possible in practical applications. To adapt KBRL to the online setting, some exploration policy must be applied to the learned value function, just as in Q-learning. The experiments in Section 3.3.3 demonstrate that given limited amounts of data generated from actual trajectories through the state space, KBRL generates flawed policies.

3.2.2 The Fitted R-MAX Algorithm

Section 3.2.1 described the KBRL algorithm, which predicts the immediate reward and the distribution over successor states for any given state-action using data from similar state-actions. Despite its intuitive appeal, this algorithm remains impractical for two main reasons. First, the set of states X, for which KBRL computes values, grows with the amount of data. This growth increases the burden both on the approximation of the model (which computes a kernel function between every pair of states in X) and on planning (which iterates over every state in X). Second, the algorithm does not address the issue of efficient exploration. This section presents a main contribution of this thesis: an algorithm that combines the model generalization of KBRL, the efficient planning in large state spaces of FVI, and the directed exploration of R-MAX.

Adding the fitted planning of FVI improves the efficiency of KBRL. Fitting the approximate model to a fixed sample X, instead of the set of all experienced states, ameliorates the computationally intensive nature of computing the approximate model. This combination is relatively straightforward: it suffices to substitute $\hat{R}$ and $\hat{P}$, defined in Equations (3.3) and (3.4), for R and P when computing $\tilde{R}$ and $\tilde{P}$ in Equations (3.1) and (3.2). Figure 3.3 illustrates the composition of the approximate model $\hat{P}$ with a value averager Φ to obtain a fitted model $\tilde{P}$.

To incorporate the exploration mechanism of R-MAX, consider the form of the KBRL approximation for an "exact" kernel function K that does not generalize, so that $K(sa, s'a') = \delta(s, s')\delta(a, a')$. In this case, the approximations $\hat{R}$ and $\hat{P}$ reduce to the maximum-likelihood estimates of R and P:

$$\begin{aligned}
\hat{R}[sa] &= \frac{\sum_i K(sa, s_i a_i) r_{i+1}}{\sum_i K(sa, s_i a_i)} \\
&= \frac{\sum_i \delta(s_i, s)\delta(a_i, a) r_{i+1}}{\sum_i \delta(s_i, s)\delta(a_i, a)} \\
&= \frac{\sum_i N[sa]R[sa]}{N[sa]} \\
&= R[sa] \\
\hat{P}[sa, s'] &= \frac{\sum_i K(sa, s_i a_i)\delta(s_{i+1}, s')}{\sum_i K(sa, s_i a_i)} \\
&= \frac{\sum_i \delta(s_i, s)\delta(a_i, a)\delta(s_{i+1}, s')}{\sum_i \delta(s_i, s)\delta(a_i, a)} \\
&= \frac{D_P[sa, s']}{N[sa]},
\end{aligned}$$

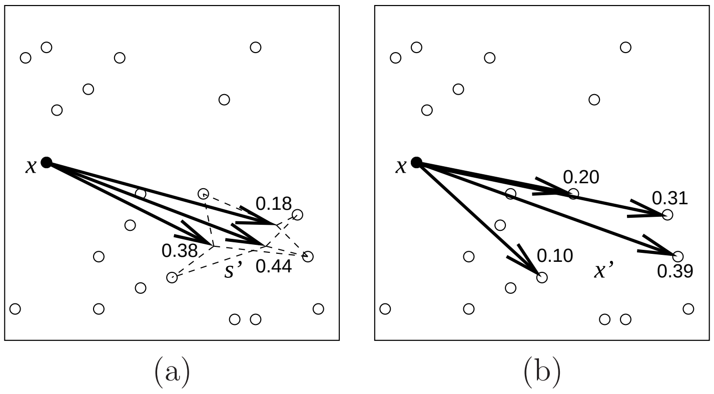


Figure 3.3: (a) The arrows indicate the three transitions predicted by the KBRL approximation $\hat{P}$ for a given state x. The open circles represent other states in X. The dashed lines indicate which states in X have nonzero weight in the value averager Φ for each predicted successor state s'. (b) The possible transitions predicted by the resulting fitted model, $\tilde{P}$, along with their transition probabilities. Each predicted successor x' receives a portion of the transition probabilities assigned to each s' by $\hat{P}$, weighted by Φ.

where $N[sa]$ counts how much data exists for $sa \in S \times A$ and $D_P[sa, s']$ counts the transitions from sa to s'. (Note that $\hat{R}[sa] = R[sa]$ due to the assumption of deterministic rewards.)

The kernel function K can therefore be interpreted as weighting the degree $K(s_i, s)$ to which data at some state s_i may be used to approximate data for some query state s. In Algorithm 3.3, the variable $N[xa']$ counts for the current state-action xa' how much data is available to estimate the dynamics of xa', including data generalized from nearby state-actions. The Fitted R-MAX algorithm compares this count to the threshold m to determine whether to use the KBRL approximation (fitted to a fixed state sample X, as described earlier in this section) or to an optimistic model.

Algorithm 3.4 specifies the FittedRMaxUpdate subroutine, which may be combined with RMaxChoose (Algorithm 2.10 to obtain Fitted R-max.

Algorithm 3.4 FittedRMaxUpdate(s, a, r, s')

```
for all $xa' \in S \times A$ do
  $w \leftarrow K(sa, xa')$        {Weight of current instance for approximating $xa'$}
  $\alpha \leftarrow \frac{w}{N^\sigma[xa']+w}$        {Fraction of cumulative weight}
  $N[xa'] \leftarrow N[xa'] + w$        {Update cumulative weight}
  $R[xa'] \leftarrow (1-\alpha)R[xa'] + \alpha r$
  $s'' \leftarrow E(s, s', x)$
  for all $x' \in X$ do {Define transitions into $X$ only}
    $P[xa, x'] \leftarrow (1-\alpha)P[xa, x'] + \alpha\Phi[s'', x']$
  end for
end for
```

Formally, Fitted R-max employs the following approximate, fitted model:

$$\hat{N}[sa] = \sum_i K(sa, s_i a_i) \tag{3.6}$$

$$\hat{R}[sa] = \begin{cases} r^{\max} & \text{if } \hat{N}[sa] < m \\ \frac{1}{\hat{N}[sa]} \sum_i K(sa, s_i a_i) r_{i+1} & \text{if } \hat{N}[sa] \geq m \end{cases} \tag{3.7}$$

$$\hat{P}[sa, x'] = \begin{cases} \delta(s, x') & \text{if } \hat{N}[sa] < m \\ \frac{1}{\hat{N}[sa]} \sum_{s'} \sum_i K(sa, s_i a_i) \delta(E(s_i, s_{i+1}, s), s') \Phi[s', x'] & \text{if } \hat{N}[sa] \geq m, \end{cases} \tag{3.8}$$

where $E : S \times S \times S \to S$ is an *action effect function* that predicts the successor state $E(s_i, s_{i+1}, s) \in S$ that results when translating to s the transition from s_i to s_{i+1}. The original KBRL algorithm predicts absolute transitions:

$$E^{\text{abs}}(s_i, s_{i+1}, s) = s_{i+1}, \tag{3.9}$$

which assume that the successor state $E^{\text{abs}}(s_i, s_{i+1}, s)$ is conditionally independent of the current state s given the instance i used to approximate the action's effect. The instance effect function generalizes the model approximation by allowing alternative predictions for the successor state.

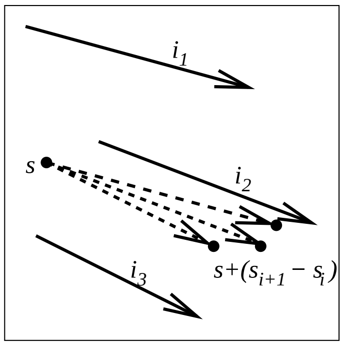
The inclusion of the action effect function is motivated by the poor performance of the KBRL approximation when data is limited. This scenario requires the model approximation to generalize broadly, using a large kernel bandwidth. In this case, the displacement between a given state s and the instance at s_i used to approximate s may be very large, dominating the predicted effect on the state transition. KBRL sidesteps this problem by waiting until the kernel bandwidth shrinks sufficiently, but to address this issue in a more pragmatic algorithm, this thesis proposes an alternative action effect function. For a given state s and instance $\langle s_i, a_i, r_{i+1}, s_{i+1} \rangle$, the *vector effect* function predicts the successor state

$$E^{\text{vec}}(s_i, s_{i+1}, s) = s + (s_{i+1} - s_i). \tag{3.10}$$

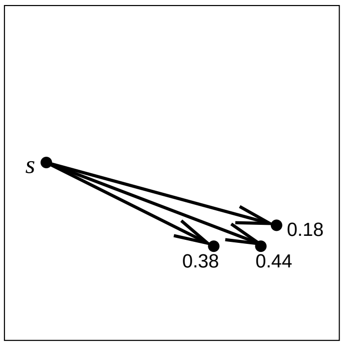
In other words, E^{vec} predicts that action a_i will cause the same vector displacement $s_{i+1} - s_i$ observed in instance i instead of the same absolute successor state s_{i+1}. The vector effect function assumes that the state space is a vector space, but the Euclidean spaces common to practical applications satisfy this assumption. Figure 3.4 illustrates the vector effect function.

3.3 Experimental Results

Fitted R-MAX learns with good data efficiency by using a combination of model-based exploration and stable function approximation. This sec-



(a)



(b)

Figure 3.4: (a) The approximation of some action at state s given three instances and the vector effect function E^{vec}. For each instance, the predicted successor is obtained by adding the vector displacement observed at the instance to s. (b) The possible transitions predicted by the resulting fitted model, $\hat{P}$, along with the transition probabilities revised from Figure 3.3.

tion describes experiments demonstrating that Fitted R-MAX converges more rapidly to near-optimal policies than several other recent RL algorithms evaluated on some benchmark problems with continuous state spaces. It then examines the importance of the vector effect function introduced in Equation (3.10) compared to the original KBRL approximation, which uses Equation (3.9). Finally, it investigates the importance of the kernel bandwidth parameter, b, which controls the breadth of generalization.

3.3.1 Implementation Details

A primary practical concern for any instance-based algorithm is computational complexity. The computationally intensive step of Fitted R-MAX is the construction of the derived finite model $\left\langle X, A, \tilde{R}, \tilde{P} \right\rangle$. In general, this construction requires $O(|X|^2|A||T|)$ for each invocation of Algorithm 3.4, which occurs at each time step. A naive implementation would repeat much of the same computation at each time step, and Chapter 7 discusses the architecture of an implementation that aggressively caches the intermediary results of this computation. Another concern is that the computational complexity of the model construction depends on the amount of data $|T|$ in the history. The longer the agent runs, the slower the model approximation will become, but judicious choices for the model and value approximations can drastically reduce the computational complexity of Fitted R-MAX.

Many function approximation schemes are possible for choosing X and defining the value averager Φ. In the experiments described in this section, X is a uniform grid spanning the state space, and $\Phi[s', x]$ gives the coefficients for multilinear interpolation of s' from the 2^d corners of the hypercube containing s', where d is the dimensionality of the state space. This approximation bounds the number of values of x' for a given s' (and therefore the number of iterations for the loop in Algorithm 3.4) by 2^d, instead of $|X| = r^d$, where r is the resolution of the grid, specifying the number of points used to approximate each dimension of the state space.

Preliminary experiments showed that this simple function approximation scheme performed better than a number of alternatives, including instance-based approaches that added either visited states s_t to X or predicted successors s' to X as necessary. Nevertheless, this grid-based approach may be less appropriate in high-dimensional state spaces, since the number of hypercube corners grows exponentially with the dimensionality. Additionally, some care must be taken for relatively large grid spacings. If one of the grid points enclosing the agent's current state is on the other side of a wall or other obstacle, an action with no effect can seem to bypass the obstacle.

The model approximation depends on the kernel K. Inspired by standard techniques in nonparamtric density estimation, a Gaussian kernel seems an obvious choice:

$$K(sa, s'a') = \delta(a, a')e^{-(d(s,s')/b)^2}, \tag{3.11}$$

where d is the Euclidean distance between s and s'. In practice, this kernel requires Fitted R-MAX to iterate over every instance i in h (with a matching action a_i) to approximate the dynamics of a given state-action xa, even though the instances near xa dominate the approximation. (If all the instances with nonzero weight are far from xa, then the total amount of data is unlikely to exceed the R-MAX threshold m, and xa will be considered unknown anyway.) The experimental implementation of Fitted R-MAX therefore uses a kernel in which the minimum nonzero value is 0.01; any smaller value is truncated to 0. It follows that the approximated dynamics for state x depends only on data within distance $b\sqrt{-\log 0.01} = 2.146b$ from x. For a given x, the applicable instances can be found efficiently by storing each s_i in a cover tree (Beygelzimer et al., 2006).

The pruning described above still allows the implementation to use an unbounded number of instances to approximate a given state-action, causing the model approximation and the planning given the model to become slower as the history grows. Therefore, for a given state-action xa, the implementation begins adding the instances closest to x first, but it stops adding instances to the approximation if adding the kernel weight would fail to increase the cumulative kernel weight by 1%. This additional pruning process bounds to 100 the number of instances used to approximate x. Note that this pruning does not bias the approximation, which essentially becomes k-nearest neighbors with $k = 100$ and Gaussian weighting whenever sufficient data exists to override optimism. The precise thresholds used to prune did not significantly affect the performance of the algorithm.

3.3.2 Benchmark Performance

This section compares the performance of Fitted R-max to algorithms submitted to the RL benchmarking workshop held at NIPS 2005 (Dutech et al., 2005). This event invited researchers to implement algorithms in a common interface for online RL. Participants computed their results locally, but direct comparisons are possible due to the standardized environment code, which presents the same sequence of initial states to each algorithm. This section examines two of the benchmark domains and gives the Fitted R-max parameters used to solve them. It then evaluates the performance of Fitted R-max against selected algorithms.

3.3.2.1 Mountain Car

In the Mountain Car simulation (Sutton & Barto, 1998), an underpowered car must escape a valley (Figure 3.5) by backing up the left slope to build sufficient energy to reach the top of the right slope. The agent has two state variables, horizontal position x and horizontal velocity v. The three available actions are `reverse`, `neutral`, and `forward`, which add -0.001, 0, and 0.001 to v, respectively. In addition, gravity adds $-0.0025\cos(3x)$ to v at each time step. The agent receives a reward of -1 for each time step before reaching the goal state. Episodes begin in a uniformly random initial position x and with $v = 0$, and they last for at most 300 time steps. The only domain knowledge available is the upper bound $r^{\max} = 0$ on the rewards and the minimum and maximum values of each state variable: -1.2 and 0.5 for x and -0.07 and 0.07 for v.

Fitted R-MAX scaled both state variables to $[0, 1]$. The generalization breadth b was 0.08. X consisted of uniform 64×64 grid overlaying the state space. Since Mountain Car is deterministic, the exploration threshold was $m = 1$. To compute the value function, Fitted R-MAX applied value iteration with $\epsilon = 0.01$.

3.3.2.2 Puddle World

The Puddle World (Sutton, 1996) is a continuous grid world with the goal in the upper-right corner and two oval puddles (Figure 3.6). The two state variables are the x and y coordinates, and the four actions correspond to

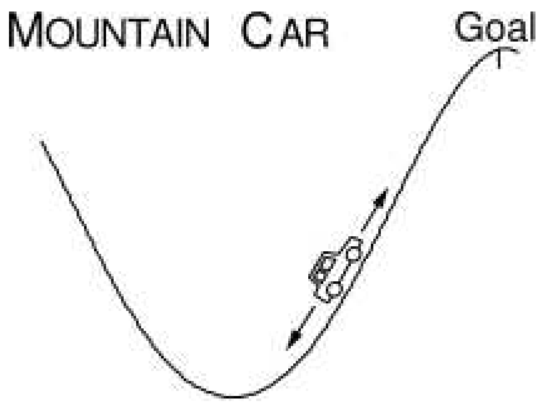


Figure 3.5: The Mountain Car environment.

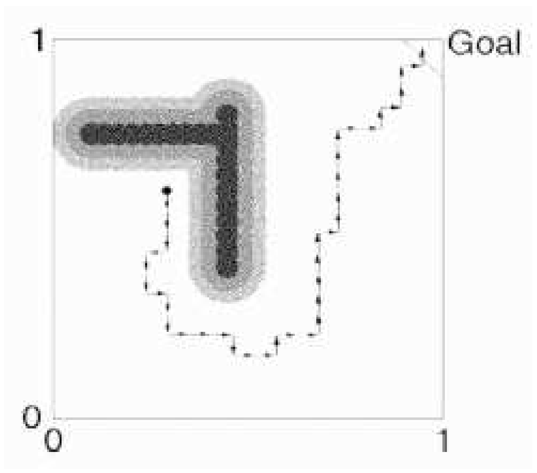


Figure 3.6: The Puddle World environment.

the four cardinal directions. Each action moves the agent 0.05 in the indicated direction, with Gaussian noise added to each dimension with $\sigma = 0.01$. The agent receives a -1 reward for each action outside of the two puddles, which have radius 0.1 from two line segments, one from $(0.1, 0.75)$ to $(0.45, 0.75)$ and the other from $(0.45, 0.4)$ to $(0.45, 0.8)$. Being in a puddle incurs a negative reward equal to 400 times the distance inside the puddle. The goal region satisfies $x + y \geq 0.95 + 0.95$.

For this domain, Fitted R-MAX used generalization breadth $b = 0.08$. A 64×64 grid was again used for X. Although Puddle World is stochastic, thresholds $m = 1$ continued to suffice. To compute the value function, Fitted R-MAX applied value iteration with $\epsilon = 0.01$.

3.3.2.3 Benchmark Results

Figures 3.7 and 3.8 compare the performance of Fitted R-MAX to three selected algorithms. (Each point is the average of fifty sequential episodes, as reported to the NIPS workshop.) These three algorithms, implemented and parameterized by other researchers, were among the most competitive submitted. One is the standard R-MAX algorithm applied to a fixed discretization of the state space. This algorithm employed the same exploration mechanism at Fitted R-MAX, but it lacked the model generalization of Fitted R-MAX. Least Squares Policy Iteration (Lagoudakis & Parr, 2003) is similar to Fitted R-MAX in that it uses a given sample of transitions to compute the parameters of a function approximator that best approximates the true value

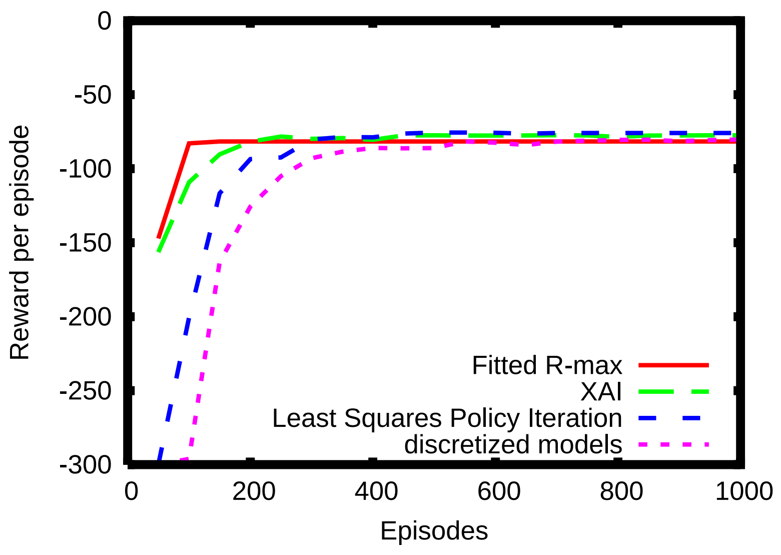


Figure 3.7: Benchmark results for the Mountain Car environment.

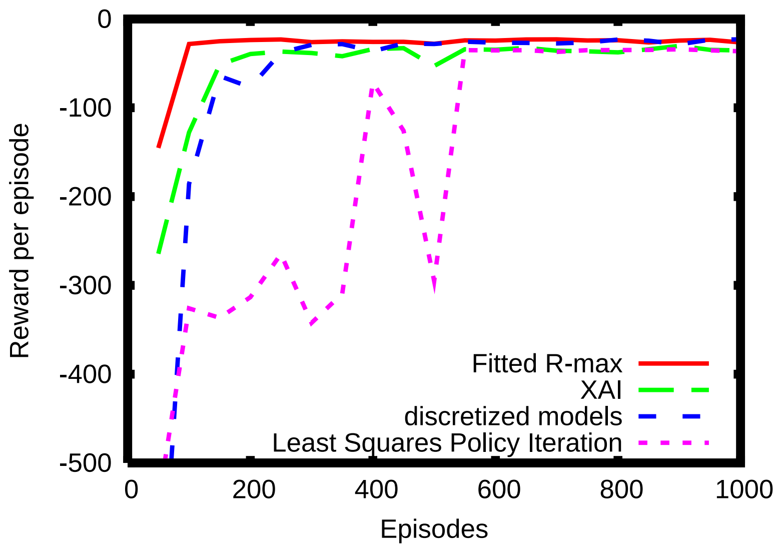


Figure 3.8: Benchmark results for the Puddle World environment.

function. However, LSPI relies on random exploration and a fixed set of kernels to represent the state space. XAI (eXplore and Allocate, Incrementally) is a method that represents the value function with a network of radial basis functions, allocated online as the agent reaches unexplored regions of the state space (Dutech et al., 2005). It thus resembles Fitted R-MAX in its instance-based use of Gaussian weighting for approximation, but XAI is a model-free method that uses gradient descent and Sarsa(λ) to update the value function. None of these algorithms achieves the same level of performance as Fitted R-MAX, which combines instance-based model approximation, stable function approximation, and model-based exploration.

3.3.3 Ablation Study

This section illustrates the benefit of Fitted R-MAX's approach to model-based RL in large environments. It compares three algorithms. The first is Fitted R-MAX, employing the vector effect function E^{vec} given in Equation (3.10). The second is a version of Fitted R-MAX that uses the absolute effect function E^{abs} given in Equation (3.9) and used by KBRL, to measure the importance of action effect component of the transition function. The third algorithm is the original discrete R-MAX algorithm (Brafman & Tennenholtz, 2002), to measure the importance of the novel decomposition of the transition function.

Figure 3.9 shows the performance of each algorithm, averaged over 50 independent trials in the Mountain Car domain. This implementation of R-

MAX uses the same parameters as the implementation submitted to the NIPS workshop: it discretizes each state dimension into 100 intervals and uses $m = 1$. Fitted R-MAX used the same parameters described in Section 3.3.2.1.

Fitted R-MAX with absolute effects converges much more quickly than discrete R-MAX, but at the expense of converging to suboptimal policies. Further experimentation has shown that decreasing b improves the average quality of the final policy but quickly decreases the learning speed of the algorithm. The standard version of Fitted R-MAX uses the more accurate vector effect generalization to preserve fast convergence while achieving near-optimal policies in this domain. For comparison, Figures 3.10 and 3.11 illustrate typical learned policies for both versions of Fitted R-MAX.

An optimal policy would execute `forward` roughly when the velocity is positive, in the upper half of the state-space diagram, and it would execute `reverse` roughly when the velocity is negative, in the lower half of the state-space diagram. This run of Fitted R-MAX with absolute effects incorrectly selects `reverse` in a large region with positive velocity. Inspection of the relevant states revealed that the local neighborhood of the sample $S^{\texttt{reverse}}$ happened to contain more high-value states. The absolute effect model incorrectly concluded that the `reverse` action would transition to this higher-value region; the vector effect model correctly concluded that this action decreases the value of any state in the neighborhood.

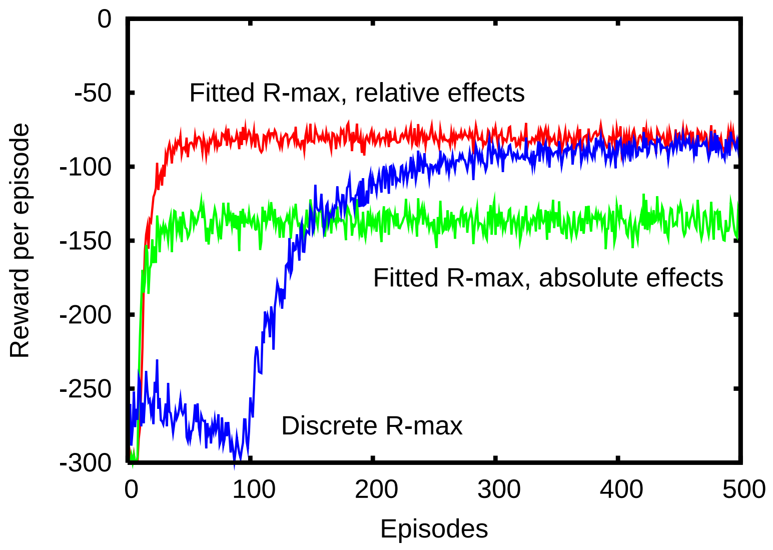


Figure 3.9: Learning curves for Mountain Car. Each curve is the average of 50 independent trials. The policy learned by Fitted R-MAX using relative effects is significantly better than that learned used absolute effects ($p < 0.001$).

3.4 Discussion

The primary contribution of this chapter is its integration of model-based exploration with stable function approximation. Fitted R-MAX extends the data efficiency of model-based methods to continuous systems, which previously presented a difficulty in representing continuous models. Locally weighted regression (Atkeson et al., 1997) addressed this problem in the deterministic case, also using local weighting from instances. Locally weighted regression estimates the average successor state for each state-action pair; Fitted R-MAX approximates the distribution over successor states and thus copes with forms of stochasticity beyond simple noise. Locally weighted regression also does not address the issue of exploration. Fitted R-MAX permits the

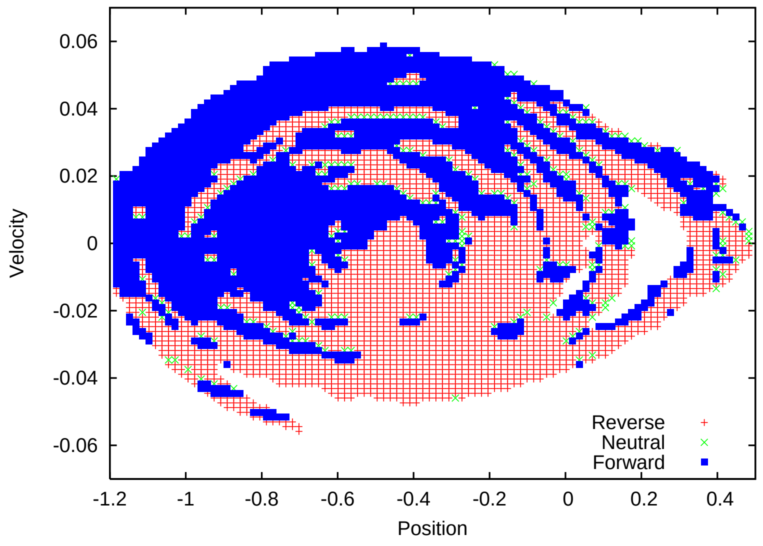


Figure 3.10: Policy for the Mountain Car environment learned using Fitted R-MAX with absolute action effects. The solid region of the state space indicates where the policy selects the `forward` action; the hatched region indicates where it selects the `reverse` action.

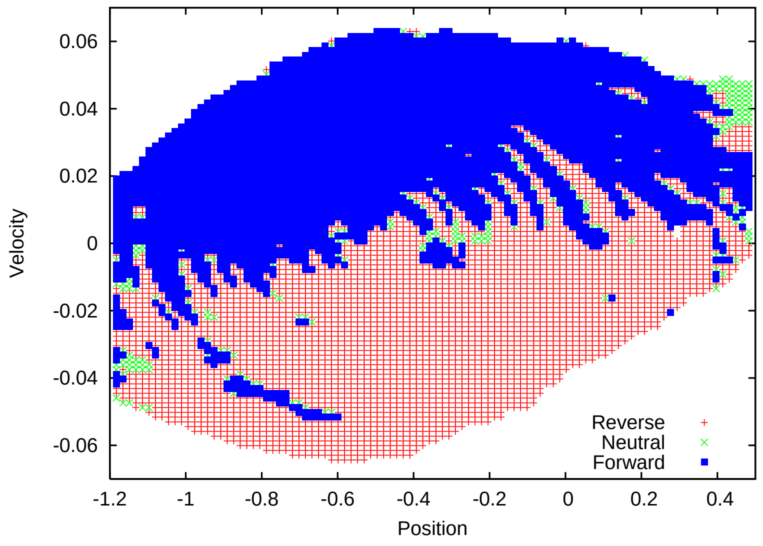


Figure 3.11: Policy for the Mountain Car environment learned using Fitted R-MAX with vector action effects. The solid region of the state space indicates where the policy selects the `forward` action; the hatched region indicates where it selects the `reverse` action.

application of intelligent exploration mechanisms originally designed for finite systems. It employs the same mechanism as Prioritized Sweeping (Moore & Atkeson, 1993) and R-MAX (Brafman & Tennenholtz, 2002), perhaps opening the door for generalizing the latter algorithm's polynomial-time PAC convergence guarantees to certain continuous systems.

Introducing model-based reasoning to function approximation also provides novel insight into the problem of generalizing from finite data to knowledge of an infinite system. Most approaches to function approximation rely on a static scheme for generalizing the value function directly, despite the difficulty in intuiting the structure of value functions. Fitted R-MAX explicitly generalizes first in a model of the system, where intuitions may be easier to represent. For example, a high degree of generalization is possible in the model for Mountain Car, since the effect of an action changes smoothly with the current state. In contrast, the optimal value function for this system includes large discontinuities in locations that are impossible to predict without first knowing the optimal policy: the discontinuity separates those regions of the state space where the agent has sufficient energy to escape the valley from those regions where it must first build energy. Approaches that only generalize the value function must use little enough generalization to represent this discontinuity accurately; Fitted R-MAX uses a learned model to generalize both broadly and accurately.

Chapter 4

The Role of Hierarchy

Chapter 3 demonstrated the ability of directed exploration to improve the robustness of RL. The R-max algorithm directs exploration by computing policies that guide the agent to states where insufficient data exists. Fitted R-max applies this idea to large state spaces by generalizing from nearby states to a given state. However, this generalization does not avoid the curse of dimensionality: a combinatorial explosion in the number of nominal states merely becomes a combinatorial explosion in the number of state neighborhoods.

In general, no algorithm can do better than exploring every reachable state or every neighborhood, since in the worst case an unexplored state or neighborhood might be essential to optimal behavior. Only additional prior knowledge can avoid exhaustive exploration. This knowledge constitutes an assumption about the range of possible environments. RL practitioners most commonly impart knowledge to learning agents in large state spaces by choosing function approximators that generalize beyond local neighborhoods: these approximations extrapolate instead of interpolate or average. In practice, these approximations are chosen largely for reasons of convenience, not due to any strict correspondence to any actual domain knowledge.

This chapter investigates the potential of hierarchical decomposition to serve as a more intuitive way for practitioners to convey domain knowledge to a learning agent.[1] The concept of hierarchy has strong intuitive appeal to artificial intelligence researchers. Humans cope with the extraordinary complexity of the real world in part by thinking hierarchically, so this faculty would be a valuable one for autonomous agents to possess. In the RL literature, this idea has taken shape in work on temporal abstraction, in which abstract actions represent sequences of lower-level actions (Barto & Mahadevan, 2003). Early work demonstrated the potential of handcrafted temporal abstractions for improving the performance of RL agents in particular problems, but it raised the question of how to discover this hierarchical structure automatically (Barto & Mahadevan, 2003). This open question remains at the forefront of research into hierarchical RL.

The discovery algorithms proposed so far attempt to capture intuitions about what constitutes a useful abstract action. Most of these algorithms look for subgoal states and then define abstract actions that attempt to reach the discovered subgoals (Mannor et al., 2004; McGovern & Barto, 2001; Şimşek & Barto, 2004). In principle, the creation of a subgoal decomposes the learning problem into at least two smaller problems: learning to attain the subgoal and learning optimal behavior from a subgoal state. In practice, prior work has

[1]This chapter describes research done in collaboration with Todd Hester (Jong et al., 2008).

not made clear how these approaches improve learning performance.

Despite the agreement about the importance of hierarchy and of the discovery problem, the two most widely studied frameworks for hierarchical RL exhibit remarkable differences, both in their motivations and their usage. For example, the idea behind the popular options framework (Sutton et al., 1999) is "to permit one to add temporally extended activities to the repertoire of choices available to an RL agent, while at the same time not precluding planning and learning at the finer grain of the core MDP. The emphasis is therefore on augmentation rather than simplification of the core MDP" (Barto & Mahadevan, 2003). In contrast, the MAXQ framework employs temporal abstraction to constrain the choices available to an RL agent: "There are many reasons to introduce hierarchical reinforcement learning, but perhaps the most important reason is to create opportunities for state abstraction," so that "individual MDPs within the hierarchy can ignore large parts of the state space" (Dietterich, 2000a). These differences underscore a general lack of insight into how precisely hierarchy improves RL performance.

Although the discovery of hierarchy served as a primary motivation for this thesis, research into discovery algorithms seems premature given the uncertainty surrounding the exact function of the hierarchies to be discovered. This chapter investigates precisely how hierarchical decomposition benefits RL (Jong et al., 2008). Section 4.1 describes the prevailing formalisms for hierarchical RL: the semi-Markov decision process and options. Section 4.2 conducts detailed experiments to determine the conditions in which options help or even hinder learning performance. Section 4.3 continues the investigation with experiments that focus on the interaction between options and exploration. Finally, Section 4.4 concludes with some discussion, which motivates the definition in the next chapter of a model-based, hierarchical learning algorithm.

4.1 Hierarchical Reinforcement Learning

The two dominant approaches to hierarchical RL share the same formal underpinnings, despite their philosophical differences. They allow an agent to apply *temporal abstraction*, so it can represent entire sequence of actions as a single abstract action. For example, consider an MDP formalizing the interaction between a mobile robot and its environment. The action space may include several actions that allow the agent to turn or to move forward. One abstract action might be the sequence of these movement actions that navigate the robot to a particular landmark or that follow a corridor until reaching an intersection. With an appropriate set of abstract navigational actions, the agent can reason at the level of landmarks and intersections instead of arbitrary poses in the environment. An abstract action decomposes the overall learning problem into the low-level problem of how to execute the abstract action efficiently and the high-level problem of planning when to execute each abstract action.

Note that this abstraction takes place within the agent. The state and action spaces that define the interface between agent and environment still depend only on the given learning problem. These actions are *primitive actions*, and they always take exactly one unit of time to execute. Any abstract actions used to reason at a higher level depend on the agent and the learning algorithm. These *composite actions* comprise one or more primitive actions executed sequentially, so they take some positive integer number of time steps to execute.

The states and the primitive actions of a learning problem define an MDP (which is not known to the agent), but a key advantage of temporal abstraction is the ability to reason at the level of the composite actions, not the primitive actions. However, these composite actions do not exactly induce an MDP, since they may have different durations, which impact the discounting of future rewards. The hierarchical RL literature instead draws upon the *semi-Markov Decision Process* (SMDP) formalism (Sutton et al., 1999).

Formally, an SMDP $\langle S, A, R, P\rangle$ is an MDP in which the action space A contains composite actions, which may take more than one time step to execute. The state transition function $P : S \times A \times S \times \mathbb{N} \to [0, 1]$ must account for the duration of actions by giving a joint distribution over successor states and durations for each state-action. For a given state s and action a, $P(s, a, s', k)$ gives the probability that a will transition from s to s' in exactly $k > 0$ time steps.

When the reward and state transition functions R and P are not available, RL methods designed for the MDP case can be easily adapted to the SMDP case. The necessary modification corresponds to incorporating the additional discounting when an action takes more than one time step. Algorithm 4.1 illustrates this change, which simply incorporates the duration k into the discounting in the update rule. (An additional parameter is required to communicate the duration k.) This update rule can be combined with the action-selection mechanism in Algorithm 2.6 to obtain a complete learning agent for SMDPs.

Algorithm 4.1 SMDPQLearningUpdate(s, a, r, s', k)

```
$Q^\sigma[sa] \overset{\alpha}{\leftarrow} r + \gamma^k \max_{a'} Q^\sigma[s'a']$
```

Similar adjustments to incorporate duration could apply to planning algorithms, which would also have to cope with the modified representation of the state transition function P. However, it is possible to plan for MDPs without explicitly reasoning about k. Note that the duration of a composite action only influences planning insofar as it affects how much to discount the value of the successor state, as demonstrated in the resulting Bellman equation for evaluating a policy π:

$$Q^\pi(s, a) = R(s, a) + \sum_{s'} \sum_{k=1}^{\infty} P(s, a, s', k)\gamma^k V^\pi(s'). \tag{4.1}$$

This thesis adopts a *multi-time model* (Sutton et al., 1999) of P, which avoids the explicit representation of the duration k by folding the discount factor into the transition probabilities. This approach represents P as a $|S||A| \times |S|$ matrix, preserving consistency with the representation of the state transition function for standard MDPs. The similar forms of Equations (2.3) and (4.1) suggest the appropriate matrix entries:

$$P[sa, s'] = \sum_{k=1}^{\infty} P(s, a, s', k)\gamma^k, \tag{4.2}$$

so that $P[sa, s']$ gives the marginal probability that state-action sa will eventually transition to s', appropriately discounted. Since multi-time representations of the state transition function incorporate the discount factor, the matrix form of the Bellman equation (4.1) becomes:

$$Q^\pi = R + PV^\pi. \tag{4.3}$$

Note that in this formalism P is no longer a stochastic matrix, since its rows do not sum to 1. If all the actions are primitive, then the multi-time representation would cause each row to sum to γ, since this multi-time model incorporates the discount factor. In the general case, the sum of each row is in $[0, \gamma]$.[2]

Another way to understand the matrix P is to interpret the discount factor as a $1 - \gamma$ probability of each primitive action transitioning to a zero-value (terminal) state. After the agent chooses an action, the system essentially generates an immediate reward according to R, then with probability γ transitions to a successor state according to the dynamics of the environment. Otherwise, with probability $1 - \gamma$, the history terminates and no further reward is possible. In this interpretation, the goal is to maximize the total sum

[2]The sum can only equal 0 if the associated state-action always terminates the episode, in an episodic task, or of course if $\gamma = 0$.

of rewards (without further discounting) before this stochastic termination. Then $P[sa, s']$ gives the probability that sa will eventually transition to s', and $1 - \sum_{s'} P[sa, s']$ gives the probability that sa will eventually transition to the implicit terminal state.

The multi-time representation of state transitions allows value functions for SMDPs to be computed by MDP planning algorithms, simply by folding the uncertain action durations into discounted transition probabilities in P. In particular, an optimal policy for a given SMDP can be computed simply by using standard value iteration, as defined in Algorithm 2.2[3].

4.1.1 The Options Formalism

The SMDP formalism requires only that each action, primitive or otherwise, have a distribution over successors and durations that depends only on the current state (Sutton et al., 1999). It treats each composite action as a black box, whose input is a state and whose output is a distribution over successor states and durations. An *option* is a specific formalism for describing how a composite action behaves during its execution, defined as follows. An option $o = \langle I^o, \pi^o, \beta^o \rangle$ comprises an initiation set $I^o \subseteq S$, an option policy $\pi^o : S \to A$, and a termination function $\beta^o : S \to [0, 1]$. The initiation set I^o specifies the set of states in which the option may begin execution. The option policy π^o specifies the primitive actions the option selects at any given time

[3]This usage requires a minor adjustment: removing the discount factor from Algorithm 2.2 to reflect the incorporation of γ into P

step during its execution. The termination function β^o gives the probability $\beta^o(s)$ that option o will terminate upon transitioning into state s.

An option defines a composite action by completely describing its behavior, in terms of other actions. Given a set of options to augment the primitive actions defined by the environment, an agent could execute the options as composite actions and apply an SMDP learning algorithm. Learning with options therefore need not be much different from learning with primitive actions, since MDP learning algorithms can be easily extended to the SMDP case. SMDP Q-learning remains a common baseline for learning with options, in the same way that Q-learning remains a common choice in practice for learning with primitive actions. Algorithm 4.2 adapts Algorithm 4.1 to the specific case of SMDP learning with a given set of options O. Note that this algorithm returns to the standard interface for learning algorithms, which invokes UPDATE and CHOOSE at every time step. To simplify the logic of the algorithm, the option set O includes options corresponding to the primitive actions. Each action $a \in A$ is construed as an option o such that $I^o = S$, and for all $s \in S$, $\pi^o(s) = a$ and $\beta(s) = 1$. By considering its options O, an agent therefore considers both all the primitive actions as well as whatever truly composite actions it was provided as prior knowledge.

As with the standard Q-learning algorithm discussed in Section 2.2.2, SMDP Q-learning only specifies how to compute the value function given the available data. A complete agent must combine the value function with a mechanism for selecting and executing actions. Algorithm 4.3 extends the

Algorithm 4.2 OptionQLearningUpdate(s, a, r, s')

```
$k^\sigma \leftarrow k^\sigma + 1$    {Track duration of $o^\sigma$}
if Agent terminates $o^\sigma$, with probability $\beta^{o^\sigma}(s)$ then
    $Q[x^\sigma o^\sigma] \overset{\alpha}{\leftarrow} r + \gamma^k \max_{o'} Q[s'o']$
    Uninitialize $o^\sigma$    {Signal termination to Choose}
end if
```

ϵ-greedy action selection mechanism to the options framework. Note that in both these algorithms, the algorithm state σ must be initialized with prior knowledge that includes the option set O: including definitions of the option policy π^o for each option o (in addition of course to the initation sets and termination functions).

Algorithm 4.3 OptionEpsilonGreedyChoose(s)

```
if $o^\sigma$ is uninitialized then
    if Agent explores, with probability $\epsilon$ then
        repeat
            $o^\sigma \leftarrow$ drawn uniformly at random from $O$
        until $s \in I^{o^\sigma}$
    else
        $o^\sigma \leftarrow$ chosen to maximize $Q[so^\sigma]$
    end if
    $x^\sigma \leftarrow s$    {Record state in which option initiated}
    $k^\sigma \leftarrow 0$    {Initialize duration of $o^\sigma$}
end if
$a \leftarrow \pi^{o^\sigma}(s)$    {Follow current option policy}
return $a$
```

4.1.2 Option Discovery

The options framework's simplicity and convenience belie subtleties in how precisely options improve agents' learning performance. One important subtlety arises from the formal definition of an option not as a subtask in a learning problem but as a solution to a subtask. For an illustration of how this distinction has impacted research in hierarchical RL, first consider the work on option discovery.

Most existing algorithms for discovering temporal abstractions fit the same overall pattern. They identify certain states as subgoals, whether by finding states that frequently occur in successful episodes (McGovern & Barto, 2001), that correlate with finding novel states (Şimşek & Barto, 2004), or that connect clusters of a state transition graph (Mannor et al., 2004). The agents then define options that transition to these subgoal states, obtaining the option policy using Experience Replay (Lin, 1992), a technique originally developed to speed the convergence of Q-learning. This technique works by simply applying the appropriate update rule (Algorithm 2.5) in batch fashion to saved trajectories of experience, so that each piece of data is used for more than one update. Given a newly discovered subgoal state, an agent can create an option by first defining an RL subproblem in which this state has high value. The agent then uses Experience Replay to propagate this value back through the option's state space, obtaining a local value function and the option policy. SMDP Q-learning (Sutton et al., 1999) with the resulting options, discovered online, is then shown to improve upon standard Q-learning.

Although previous work demonstrated algorithms that outperform basic Q-learning, it offered at best an incomplete picture of the benefits of temporal abstraction. Given the procedure followed by most of the existing work on option discovery, one important question concerns the relative contributions of Experience Replay and of temporal abstraction to the performance improvements. In particular, could Experience Replay alone provide the same benefit as using options whose policies are learned using Experience Replay?

McGovern and Barto briefly addressed this issue in their work on option discovery (McGovern & Barto, 2001). In their experiments, they included a condition that used Experience Replay without options, but they limited Experience Replay to the same number of value updates as they used to learn their option policies. If Experience Replay was thus applied to the entire learning problem, it performed worse than using the same number of backups to learn option policies. However, simply limiting Experience Replay to the states in the proposed option's initiation set provided the same benefit as learning the option policy, without actually creating an option.

This latter result alone should cast doubt over the contribution of temporal abstraction in this particular scenario, since the benefit seems to arise simply from focusing the efforts of Experience Replay to states near the discovered "subgoal" state. A reasonable question is whether this benefit might be achieved more easily than by subgoal discovery, which runs the risk of expending computational resources to create options that burden the learning agent with additional parameters to learn, another instance of Minton's utility problem (Minton, 1988). The problem runs even deeper, since using subgoals to focus Experience Replay to particular states only makes sense if enough computation time exists to perform Experience Replay but not enough to apply it more globally. Most evaluations of RL performance, including those in the literature on option discovery, measure reward earned as a function of the number of actions taken, not the amount of computation time.

Recent work in option discovery continues to appeal to the unproven intuition that hierarchy has intrinsic value for RL agents, despite a lack of conclusive evidence. The remainder of this chapter investigates how precisely options affect the performance of learning agents. Section 4.2 defines the experimental framework and examines more closely the use of Experience Replay to determine option policies. Section 4.3 builds upon these results to examine the role of options in determining the exploration behavior of a typical RL agent.

4.2 Experiments with Option Policies

Any general evaluation of the options framework must take care to choose representative agents and environments. This evaluation begins by adopting an agent incorporating all of the innovations presented in the canonical work on options (Sutton et al., 1999), as well as the original environment used to demostrate those innovations.

4.2.1 An SMDP Learning Algorithm

Although widely used in research involving options, basic SMDP Q-learning does not learn very efficiently. Its handling of options as black boxes is inefficient for at least two reasons. First, SMDP Q-learning updates the value function only once for each execution of an action. When all the actions are primitive, it updates the value function at every time step, just as Q-learning does. When some actions are actually options, the value function is updated less frequently, since one execution of an option may take multiple time steps but lead to only a single update.

Second, the simplicity of treating an option as a black box comes at the cost of the learning agent ignoring possibly useful information in the option's definition. In situations where the option's policy is determined by a concurrently learning agent, this absence of communication between high-level and low-level agents can be particularly inefficient.

Intra-option methods (Sutton et al., 1999) address these concerns partially, by allowing the learning agent to inspect an option's parameters: its policy, initiation set, and termination condition. This additional data permits algorithms to update the agent's estimated value of the option whenever the primitive action executed in a given state is consistent with the option's policy, since that instance sheds light on how the option would have behaved if it had been executing. Algorithm 4.4 implements intra-option model-learning (Sutton et al., 1999). Instead of updating the value function directly given each instance, it updates a model of each option (using stochastic approximation) and then uses these learned models to update the value function. This algorithm is an instance of the Dyna-Q algorithm (Sutton & Barto, 1998), adapted to include options.

In the experiments described in this chapter, the agent state will be

Algorithm 4.4 IntraOptionModelLearningUpdate(σ, r, s')

```
$Q[sa] \overset{\alpha}{\leftarrow} r + \gamma \max_{a'} Q[s'a']$        {Update value function directly from data}
for all $o \in O$ do
    if $\pi^o(s) = a$ then
        $R[so] \overset{\alpha}{\leftarrow} r + \gamma(1 - \beta^o(s'))R[s'o]$        {Update reward model}
        for all $s'' \in S$ do {Update state transition model}
            $P[so, s''] \overset{\alpha}{\leftarrow} \beta^o(s')\delta(s', s'') + (1 - \beta^o(s'))\gamma P[s'o, s'']$
        end for
    end if
                                                        {Update value function from model}
    $Q[so] \overset{\alpha}{\leftarrow} R[so] + \gamma \sum_{s'' \in S} P[so, s''] \max_{o'} Q[s''o']$
end for
if $o^\sigma$ terminates, with probability $\beta^{o^\sigma}(s')$ then
    Uninitialize $o^\sigma$                                {Signal termination to Choose}
end if
```

initialized as follows. For ϵ-greedy action selection, the probability of selecting a random action is $\epsilon = 0.1$. The option set, from which actions are chosen, varies with each experiment, but it typically contains all the primitive actions (represented as options that always terminate everywhere) as well as some composite actions given as prior knowledge. The other parameters include the learning rate $\alpha = 0.3$ and discount factor $\gamma = 0.9$. Unless otherwise specified, the value function Q and option models R and P are initialized to 0.0 in every entry. These matrices are indexed by the option set O, not A.[4]

Finally, the implemention of option learning employs the "interrupting options" technique (Sutton et al., 1999), which allows the agent to interrupt the execution of an option if it enters a state where another action has higher

[4]The notation $Q[sa]$ refers to the entry $Q[so]$ for the option o constructed to represent the primitive action a.

value than the option currently being executed. This technique is formalized in Algorithm 4.5, a minor adaptation of Algorithm 4.3, OptionEpsilonGreedyChoose.

Algorithm 4.5 InterruptingOptionEpsilonGreedyChoose(s)

```
if $o^\sigma$ is uninitialized or $\max_{o'} Q[so'] > Q[so^\sigma]$ then
    if Agent explores, with probability $\epsilon$ then
        repeat
            $o \leftarrow$ drawn uniformly at random from $O$
        until $s \in I^{o^\sigma}$
    else
        $o^\sigma \leftarrow$ chosen to maximize $Q[so^\sigma]$
    end if
end if
$a \leftarrow \pi^{o^\sigma}(s)$                    {Follow current option policy}
return $a$
```

4.2.2 The Four-Room Gridworld

For consistency with prior work, the experiments in this chapter use the simple four-room gridworld employed by Sutton, Precup, and Singh (Sutton et al., 1999), shown in Figure 4.1. Each cell in the grid represents a state that the agent may occupy. From each cell, the agent can take one of four primitive actions: left, right, up, or down. Each primitive action is stochastic, taking the agent to the desired cell with probability 0.8 and in a perpendicular direction with probability 0.2. For example, when selecting the up action, the agent would move up 80% of the time, left 10% of the time and right 10% of the time. If the effect of the movement would place the agent in a wall, then the agent remains in its current cell. The agent starts each episode at a

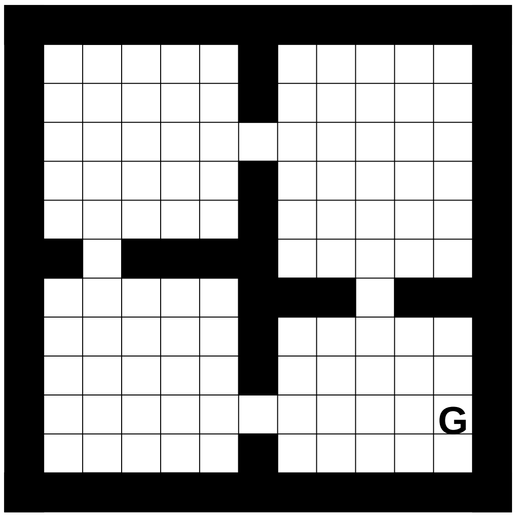


Figure 4.1: The four-room gridworld

random location in the upper left room and the goal is a state near the lower right corner of the grid-world. The immediate reward is 0 at each state in the world, except at the goal, where the reward is 1.

This environment is clearly quite simple, and it was used in prior work presumably due to its suitability for learning with temporal abstraction. In particular, the doorway states are intuitive subgoals for an agent that must navigate this world. Nevertheless, in-depth experiments reveal that even in this simple domain, originally chosen to demonstrate the effects of learning with options, the utility of temporal abstraction depends on numerous factors.

4.2.3 Options and Experience Replay

In this section we conduct experiments that reproduce the conditions used in recent work on option discovery. In particular, it compares standard

Q-learning against an agent that, in the middle of learning, introduces options with policies obtained using Experience Replay. However, it also compares against an agent that simply applies Experience Replay without creating options, at the same point in the learning process.

Both agents that use Experience Replay save each experience $\langle s_t, a_t, r_{t+1}, s_{t+1} \rangle$ at each time step. Instead of replicating the various subgoal discovery algorithms in past research, these experiments give the option-learning agent the benefit of the doubt, and allow it access to the correct subgoal states, corresponding to the doorways between rooms, after 20 episodes of learning. Each subgoal thus defines an option that terminates only at that subgoal, has an initiation set that includes every other state, and whose policy is learned using Experience Replay.

The other experimental agent is Q-learning using Experience Replay only. For this agent, saved experience is simply played back in reverse order after 20 episodes to update the value function. The agent was allowed the same number of updates that were used to learn the four option policies, so it replayed the complete set of experiences four times.

Figure 4.2 compares the learning performance of the three agents. All three algorithms exhibit the same performance until episode 20, since they all use only standard Q-learning until then. The agent that defines options exhibits a marked improvement over the agent that simply continues to use Q-learning, but the agent that just applies Experience Replay exhibits the same improvement.

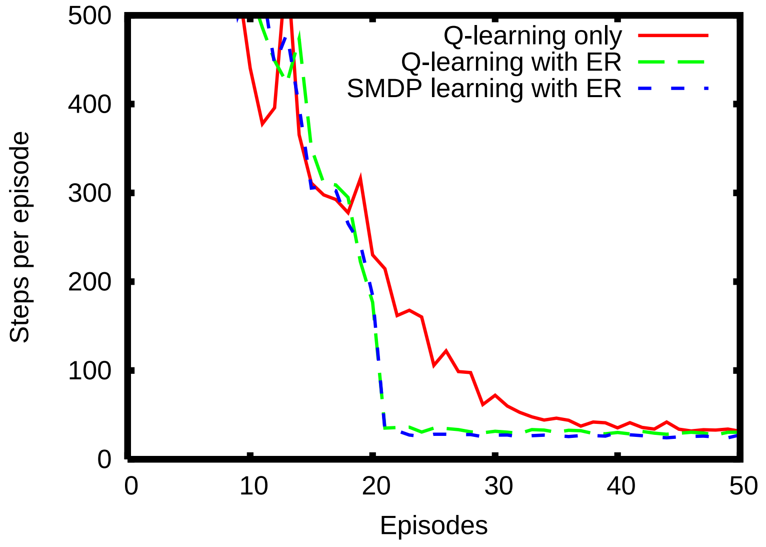


Figure 4.2: Learning performance of three agents that use standard Q-learning until episode 20, when one agent defines options using Experience Replay and another just uses Experience Replay to update its value function. Each line shows the number of steps required to reach the goal state, averaged over 50 independent runs.

4.2.4 Dynamic Option Policies

Prior work established the benefit of learning with options, at least in certain circumstances and when options are provided to the agent (Sutton et al., 1999). The automatic discovery of options, to remove the assumption that a human expert must provide this domain knowledge, remains an active area of research. However, Section 4.2.3 demonstrated that at least in one simple learning task, the typical approach of introducing options defined using Experience Replay, applied to some subgoal state found by some discovery algorithm, performs no better than simply using Experience Replay alone. This result suggests the need for a deeper understanding of how options improve learning performance.

Introducing an option to an agent's action space provides the agent with at least two forms of domain knowledge. First, an option's initiation set and termination function identify a subgoal that potentially decomposes the learning problem into subproblems. Second, an option's policy describes a solution to one of these subproblems. To isolate the benefit of temporal abstraction, these two contributions must be separated.

To this end, this thesis proposes a refinement of the option formalism, which defines an abstract action as a subproblem instead of as the solution to a subproblem. A *subtask* is defined as a "partial option" $o = \langle I^o, A^o, G^o, \beta^o \rangle$. A subtask o thus corresponds to the following problem. From an initial state $s \in I^o$, select actions from A^o in such a way as to maximize the expected cumulative reward, given that a transition into state s' terminates the subtask with probability $\beta^o(s')$ and generates a *subgoal value* of $G^o(s')$. A solution to a subtask defines an option, by specifying the option policy $\pi^o : S \rightarrow A^o$.[5]

In general, an agent may learn the option policy by recursively applying RL to learn in the *subtask MDP* $M^o = \langle S, A^o, R^o, P^o \rangle$ where the state space S is inherited from the original MDP and the reward and state transition functions are determined by the set of child actions A^o. In particular, for any $a \in A^o$, $P^o[sa, s'] = P[sa, s']$ and $R^o[sa] = R[sa] + \sum_{s' \in S} P^o[sa, s'] \beta^o(s') G^o(s')$. Note that in general, the option may only be capable of visiting a subset of S, corresponding to the states reachable from I^o given the set of child actions A^o and the termination function β^o. The subtask may thus be thought of as an agent in its own right, attempting to learn only a part of the original problem, perhaps concurrently with the agent learning when to execute this subtask. From the perspective of the higher-level agent, this subtask is just an action whose behavior may change over time. This nonstationarity may complicate learning, but it allows for a larger breadth of hierarchical learning algorithms, instead of requiring levels of abstraction to be learned strictly sequentially, bottom-up.

Figure 4.3 compares the performance of standard Q-learning against an algorithm that learns the option policies in parallel with the value function for the entire task. The option learner is given the four correct subtasks as prior knowledge, and it applies four instances of Q-learning in parallel with

[5]Prior work established the use of subgoal values to determine options (Sutton et al., 1999), but without defining a subtask as a unit of knowledge.

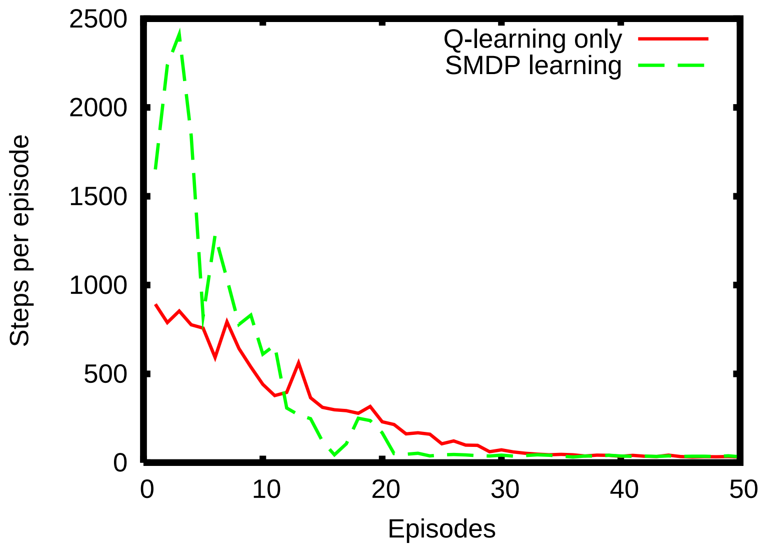


Figure 4.3: Performance of Q-learning and an algorithm that learns subtask policies in parallel with the overall task policy. Each learning curve is the average of 50 independent runs.

Intra-Option Model Learning (Algorithm 4.4) to learn the option policies. The results show a substantial decrease in performance for the agent that employs temporal abstraction, but perhaps surprisingly this poor performance does not seem to be a result of the evolving option policies. Instead, the agent exhibits some pathological initial exploratory behavior, as investigated in more detail in Section 4.3.1. While the experimental approach to learning the option policies and top-level policy could be more sophisticated, the technique is straightforward and suggests the subtlety in employing hierarchical knowledge properly.

4.3 Experiments with Options and Exploration

Section 4.2 suggests that the impact of a hierarchical approach is not straightforward, even with environments and learning agents designed to highlight its benefit. One reason for this subtlety is the conflation of subtasks and subtask solutions, but this section reveals that subtasks influence learning performance in large part by modifying the exploration behavior of the learning agent. Section 4.3.1 explicates the poor performance of the option-learning algorithm shown in Figure 4.3, showing that the premature introduction of a subtask can lead to pathological exploration behavior. Section 4.3.2 investigates how options interact with a standard technique for encouraging exploration. Finally, Section 4.3.3 suggests an alternative approach to using subtasks to change the structure of a learning problem.

4.3.1 The Initiation Set

From the perspective of evaluating the utility of temporal abstractions, prior work neglected to address the impact of when a discovered option becomes available during learning and how broadly applicable it is. For example, existing option discovery methods typically define the initiation set heuristically as the set of states historically visited a few time steps before the identified subgoal state (McGovern & Barto, 2001; Şimşek & Barto, 2004). These methods then add the newly created options to the action space in the middle of learning the current task. Evaluations of temporal abstractions in this context are constrained in at least two ways. First, temporal abstraction is only applied when the value function is already partially learned. Second, the initiation set is in general only a subset of the possible states in which the option could reasonably be defined. These constraints have practical ramifications in situations when hierarchies are being learned in one task for use in a related task (in a transfer learning setting). Perhaps more importantly, they allow only a limited understanding of the utility of temporal abstractions in general.

For an illustration of how these issues can lead to problems, consider the behavior of the subtask-learning agent whose performance is shown in Figure 4.3. It was given as prior knowledge four subtasks, corresponding to navigating to each of the four doorways from anywhere else in the environment. The addition of these four abstract actions drastically changes the apparent structure of the environment. Since the value function is initialized to 0 and the only nonzero reward occurs at the goal state, then the typical ϵ-greedy exploration policy will conduct a random walk until finding the goal at least once. However, every time the agent randomly selects one of the options, it will end up at one of the doorways, at least five steps from the goal. The agent cannot reach the goal until it randomly generates a sequence of actions that doesn't include any of the options but that does reach the goal, which becomes exponentially unlikely in the distance of the goal from the options' subgoal state.

To investigate this phenomenon, Figure 4.4 compares four learning agents. The "Q-learning" agent performs standard Q-learning, as usual. The "options" agent is given four correct options at the start of learning, including

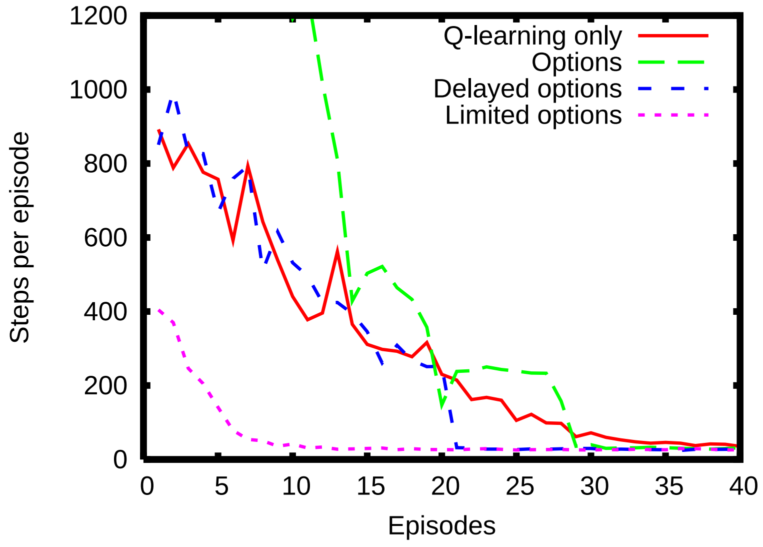


Figure 4.4: Comparison of learning agents with varying access to correct temporal abstractions. The "Q-learning" agent never uses options. The "options" agent gains immediate access to the correct options everywhere. The "delayed options" agent gains access to these options after 20 episodes. The "limited options" agent gains immediate access to these options except in the lower-right room. Each learning curve is the average of 50 independent runs.

the optimal option policy for each subtask. These subtasks all include the entire state space (except for the one subgoal of each subtask) in their initiation sets. The "delayed options" agent receives the same four options, but only after the first 20 episodes of learning. Finally, the "limited options" agent receives options immediately, but the initiation sets of these options do not include any state in the room containing the goal state.

The agent that has immediate access to the optimal option policies in all states performs the worst, for the reasons described above. The "delayed options" agent avoids the pathological exploration behavior by not invoking any options until it has a partially learned value function that allows it to learn not to select the options in states with higher value than the subgoal state. This agent performs identically to the Q-learning agent until after episode 20, at which point its performance rapidly improves to the optimal policy. Of course, this agent forfeits any possible benefit that temporal abstraction might provide in the initial stages of learning.

The "limited options" agent essentially uses prior knowledge to allow the options to execute only when navigating to a doorway might be helpful. Eliminating the state in the lower-right room from the initiation sets effectively prevents the agent from allowing an "earlier" subtask to interfere with completing the final steps of a solution. In a sense, existing option-discovery algorithms already apply this idea heuristically, by only adding to the initiation set the states experienced just before the candidate subgoal state.

Note that both of the agents that improve over Q-learning do so by artificially limiting the availability of the abstract actions. This phenomenon is nonintuitive to the extent that options can be construed as subroutines, which typically one would design to be as generally applicable as possible.

4.3.2 Optimistic Initialization

The previous sections illustrated the perhaps nonintuitive possibility that introducing options might worsen learning performance, even when the agent retains direct access to all of the primitive actions. The reason is that the addition of such temporal abstractions changes the qualitative structure of the learning problem. The random exploration mechanisms employed in practice by most implementations of RL suffice in environments where the actions exhibit some degree of symmetry, so that random walks eventually reach every region of the state space. Introducing abstract actions that only terminate at subgoals biases random walks to states near those subgoals, since a single random selection of an option can erase the effort of several primitive actions that were carrying an agent away from the subgoal states.

One way to prevent options from interfering with the exploration behavior of an agent is to use an exploration mechanism that never relies on a random walk. A simple heuristic often used to encourage exploratory actions is optimism in the face of uncertainty, in which an agent assigns optimistic values to unfamiliar state-action pairs. This idea underlies the known finite-time convergence proofs for RL (Brafman & Tennenholtz, 2002; Kearns & Singh, 1998; Strehl et al., 2006), as well as the R-MAX algorithm used throughout this thesis.

This heuristic can be applied to Q-learning simply by initializing the value function to some upper bound on the true optimal value function. This optimistic initialization causes most updates to the value function to decrease the value of state-action pairs just executed, making the agent more likely to select a different action the next time it revisits that state. This technique would prevent an agent from indefinitely returning to a subgoal state by driving the value of the corresponding option below the optimistic initial values of primitive actions that lead away from the subgoal.

Unfortunately, this technique also prevents the learning algorithm from continuing to select options when they are actually optimal! To converge to a stable optimal policy that selects an option in a certain state, the learning algorithm must first learn that every other primitive action (and option) has a lesser (or equal) value. Of course, at least one primitive action has the same value as the option: the action that the option's policy selects at that state.

Therefore, it seems that augmenting the action space of an agent with options cannot meaningfully impact learning performance when exhaustive exploration is required, such as in the general case when nothing is known about the environment a priori. Regardless of the presence of options, the agent must execute every suboptimal state-action pair enough times to learn that they have smaller values than the optimal state-action pairs.

This phenomenon is apparent in the following experiment, in which each agent uses optimistic initialization to drive its exploration. The first agent simply uses Q-learning, the second is given correct subtasks and learns the option policies dynamically, and the third is given optimal option policies as prior knowledge. To illustrate the severity of the problem, the last agent additionally receives as prior knowledge an option that navigates to the goal state optimally from any state. Therefore, one optimal policy for this agent is simply always to select this option, in every single state. Nevertheless, Figure 4.5 reveals that all three agents exhibit identical learning performance.

4.3.3 Augmentation versus Abstraction

The preceding sections demonstrate that temporal abstractions benefit RL in only limited situations. Allowing the use of options in the wrong regions of the state space at the wrong stages of learning can lead to pathologically bad exploration. Much of the benefit when options do help might be more easily obtained by applying Experience Replay. The use of optimistic initialization, the primary means for ensuring finite-time convergence to an optimal policy, completely precludes options from impacting learning performance at all. These results raise the question of how hierarchy can possibly benefit RL in the general case.

Hauskrecht et al. consider another possible application of options: to abstract, rather than augment, the MDP (Hauskrecht et al., 1998). Their work, in the context of planning given the MDP parameters, showed that planning with only an appropriate set of options can dramatically reduce the

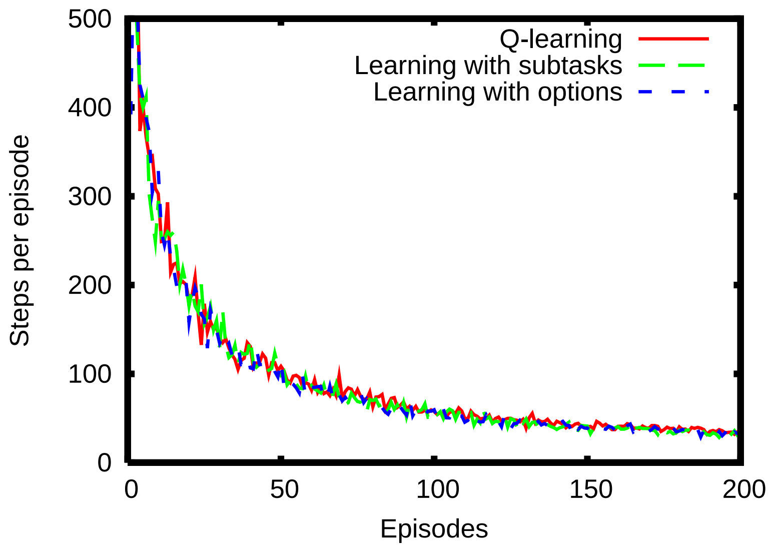


Figure 4.5: Performance of three learning agents that use optimistic initialization. The agents have identical performance, regardless of whether they use temporal abstraction. Learning curves are the average of 50 independent runs.

computational effort required to obtain an optimal policy. In contrast, they show that planning with both options and primitive actions can converge much more slowly than planning with primitive actions alone, given an optimistic initialization of the value function. These results do not translate directly to the RL context, where the emphasis is on sample complexity instead of computational complexity. For example, Section 4.3.2 already shows that learning with options in addition to primitive actions does not hurt (or help) learning performance given optimistic value-function initialization. Nevertheless, the idea of using hierarchy to reduce rather than augment the action space has received relatively little attention.[6]

[6]As discussed in Chapter 5, the less popular MAXQ framework adopts this approach.

One obvious danger in using only options to learn a task is the possibility that pruning away the primitive actions will remove the agent's ability to behave optimally. For example, an agent in the four-room gridworld that only selects from among the four subtasks that navigate to a doorway could only generate an optimal policy if the goal state were in one of the doorways. In the case of planning, analysis of the MDP can determine the goal state and create an appropriate option. For an RL agent, which typically has no prior knowledge of the goal state, the primitive actions must be available in the vicinity of the goal. In the four-room gridworld, the agent only needs access to the primitive actions in the bottom-right room.

Figure 4.6 compares two agents that ignore primitive actions outside the bottom-right room against a standard Q-learning agent. One of these two agents uses only options until it reaches the last room, after which it uses both options and primitive actions (unless it leaves the room without reaching the goal). The other agent only uses primitive actions in the last room. The agent that uses options in the last room exhibits the same pathological exploration behavior as the "options" agent in Figure 4.4: trying any of the options with an exploratory action takes the agent at least five steps away from the goal.

The agent that only uses primitive actions in the last room learns very quickly. Note that after executing any option, it can be in only one of four states. After executing a primitive action, it can be in only one of 24 states, since it only executes these actions in the bottom-right room (or its two doorways). As a result, this agent learns in a much smaller state space than the

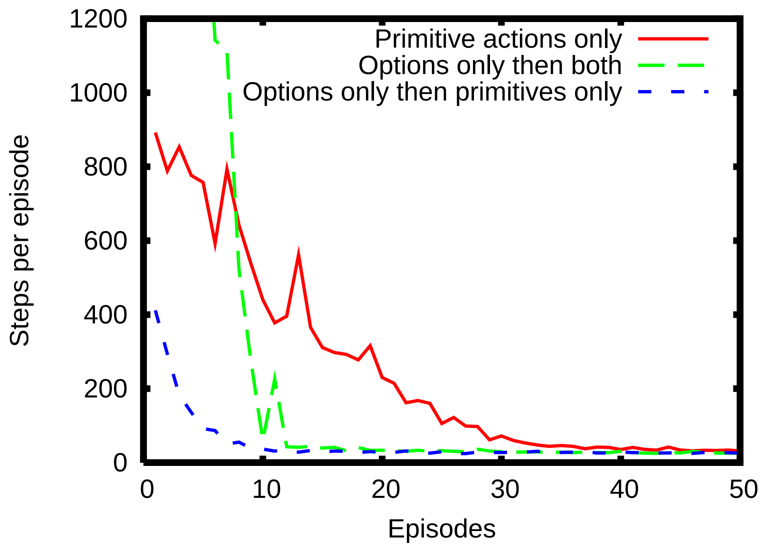


Figure 4.6: Performance of learning agents that remove primitive actions.

104-state space explored by the agents that may always execute primitive actions.

This abstracted agent learns this simple task about as efficiently as the "limited options" agent in Figure 4.4. The primary difference between the two agents is that the options produce a "soft" bias in the "limited options" agent towards the subgoal states (except the in the bottom-right room). The abstracted agent uses the options to enforce a hard bias, so that it never needs to explore any actions in any states outside of the doorway states and the last room. A key theoretical benefit of this hard bias is that it allows an abstracted agent to employ techniques such as optimistic initialization without wiping out the benefit of temporal abstraction. Since the agent cannot explore from the states that were abstracted away, it can avoid a needlessly thorough exploration of every primitive action from every state.

4.4 Discussion

The benefit of employing temporal abstraction in RL depends on a surprising number of factors. These factors include the precise learning algorithm used, when and where the abstraction becomes available, and the availability of the primitive actions or other abstract actions.

One way to construe the introduction of options into an agent's action space is as the introduction of bias in the typical random exploration mechanism. Random actions become much more likely to bring the agent to one of the indicated subgoal states. Introducing this bias too early in learning can prevent the agent from reaching the goal, which is the necessary first step in backing up positive values throughout the state space. Only later in learning can the options benefit, by backing value across multiple states at once and by biasing learning to important parts of the state space. Optimistic initialization, by removing much of the random element from exploration, eliminates this bias, beneficial or otherwise. By using options to replace primitive actions instead of to augment them, an agent can constrain exploration, including the effects of optimistic initialization.

This result highlights the difficulty in discovering truly useful options automatically. The agents described in Section 4.3.3 benefited from options because these options prevented them from ever thoroughly exploring three of the four rooms in the domain. How could an agent in this environment discover this temporal abstraction without first spending effort to explore those three rooms, thus negating the possible benefit? One possible answer is that a useful abstract action must somehow generalize from past experience. An agent that fully explores several empty rooms may soon learn just to pass through empty rooms in the future. By introducing an approriate abstract action, the agent dynamically adapts its generalization. This idea suggests an important connection between temporal abstraction and state abstraction or function approximation.

Another consequence of these results for algorithms that discover hierarchy is that candidate abstract actions may not always be evaluated in isolation. The benefit of adding one action may depend on the agent's ability to consequently remove others. This behavior depends on the action hierarchy as a whole, which determines the contexts in which each action executes and the possible state-action pairs that must be explored to learn each option's policy.

An important direction for future research is to investigate how temporal abstraction affects the computational complexity of RL. The fact that Experience Replay can subsume the benefits of using options in certain cases suggests that reducing the number of value function updates is one way in which options can help, but the work of Hauskrecht et al. indicates that options can actually increase the amount of computation required. More research is needed in the area of model-based algorithms, which more explicitly separate the gathering of data (exploration) from the computation of the value function from that data (Kearns & Singh, 1998). In particular, given the interaction between temporal abstraction and optimistic initialization, one open question is how to incorporate options into algorithms that currently offer finite-time convergence guarantees. The next chapter takes some first steps in this direction.

Chapter 5

Model-Based Hierarchies

The preceding chapter took a deeper look at the usage of hierarchy in RL. Despite the intuitive appeal of hierarchy as a mechanism for dividing problems into subproblems, the benefits of using options were shown to depend on factors ranging from the exploration mechanism to what actions were available at different stages of learning and in different regions of the state space. These results reveal the need for experiments that can clearly separate the benefits of hierarchy from the benefits of optimistic exploration or techniques for data reuse such as Experience Replay.

This chapter addresses this need by introducing the first hierarchical algorithm for the general RL setting that fully incorporates model-based techniques for efficient data use and intelligent exploration (Jong & Stone, 2008). It combines two existing algorithms, R-MAX and MAXQ. R-MAX, defined in Section 2.3.2, embodies the model-based approach of using a model of the environment to summarize all of the available data, and then using this model to seek out optimistic rewards. In some sense, R-MAX can be seen as a principled combination of Experience Replay with optimistic initializion. The MAXQ framework, discussed in Section 5.1, contributes an approach to hierarchy that emphasizes problem decomposition by formally representing the value function for an environment as a sum of subtask value functions. Prior research with MAXQ has also focussed on the need for abstracting MDPs, instead of augmenting them with additional options, which seems critical to improved learning performance in the general case.

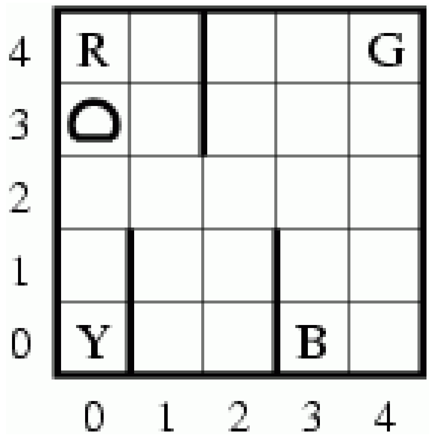
Section 5.2 combines R-MAX and MAXQ into a novel algorithm, R-MAXQ, and provides some analysis of its convergence guarantees. Section 5.3 conducts experimental comparisons among these algorithms, and Section 5.4 concludes with discussion and related work.

5.1 The MAXQ Framework

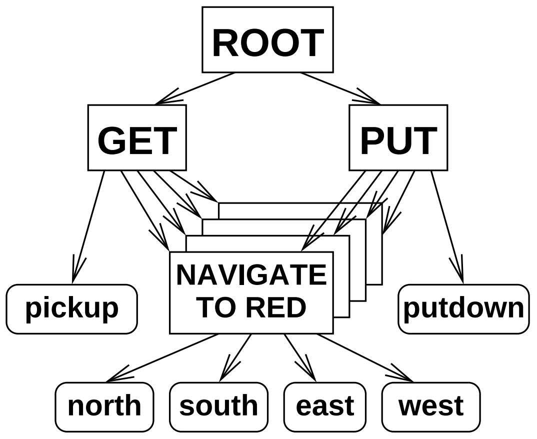
MAXQ is commonly seen as an alternative to options (Barto & Mahadevan, 2003), but it can also be seen as an elaboration of the subtask formalism defined in Section 4.2.4 to represent subproblems for which an option is a solution. In this sense, MAXQ is a particular method for generating options. One of its key ideas, discussed below, is the decomposition of an overall value function into component value functions that each estimate the cumulative reward for some interval of time. This decomposition permits different levels of hierarchy to learn more independently and then to communicate effectively across levels. MAXQ therefore takes the idea of intra-option learning a step further: instead of obtaining only policy information from a lower-level option, an agent can obtain value information from a lower-level subtask.

The MAXQ framework was first studied with the Taxi domain (Dietterich, 2000a), a simple gridworld combined with a task that yields deeper hierarchical structure than the four-room environment of Chapter 4. This domain consists of a 5×5 gridworld with four landmarks, labeled `red`, `blue`, `green`, and `yellow`, illustrated in Figure 5.1a. The agent is a taxi that must navigate this gridworld to pick up and deliver a passenger. The system has four state variables and six primitive actions. The first two state variables, `x` and `y`, give the coordinates of the taxi in the grid. The third, `passenger`, gives the current location of the passenger as one of the four landmarks or as `taxi`, if the passenger is inside the taxi. The final state variable, `destination`, denotes the landmark where the passenger must go. Four primitive actions, `north`, `south`, `east`, and `west`, each move the taxi in the indicated direction with probability 0.8 and in each perpendicular direction with probability 0.1. The `pickup` action transfers the passenger into the taxi if the taxi is at the indicated landmark. The `putdown` action ends an episode if the passenger is in the taxi and the taxi is at the desired destination. Each episode begins with the taxi in a random location, the passenger at a random landmark, and a destination chosen randomly from the remaining landmarks. Each action incurs a -1 penalty, except that unsuccessful `pickup` and `putdown` actions cost -10, and a successful `putdown` action earns a reward of 20.

The structure of the Taxi domain makes it conducive for research into hierarchical RL. The optimal policy may be described abstractly in four steps. First, navigate to the landmark where the passenger is. Second, pick up the passenger. Third, navigate to the destination landmark. Finally, put down



(a)



(b)

Figure 5.1: (a) Taxi domain, and (b) a task hierarchy for Taxi

the passenger. Navigation to each of the landmarks constitute reuseable subtasks that hierarchical algorithms can exploit. Prior work (Dietterich, 2000a) expressed this domain knowledge in the task hierarchy shown in Figure 5.1b. This hierarchy defines a navigational composite action for each of the four landmarks. These actions include the four primitive movement actions as children, and they terminate upon reaching the coordinates corresponding to the respective landmark. The `GET` and `PUT` composite actions each have all four of their navigational composite actions as children, as well as `pickup` or `putdown`, respectively. `GET` terminates when the passenger is in the taxi, and `PUT` terminates only when the episode does. The `ROOT` action only includes `GET` and `PUT` as children, and like `PUT` it defines no terminal states beyond those intrinsic to the domain. All goal reward functions give 0 reward; each subtask simply minimizes the costs earned before reaching its subgoal.

The definition in Section 4.2.4 of a subtask $o = \langle I^o, A^o, G^o, \beta^o \rangle$ is only a very slight generalization of the MAXQ definition of a task. The MAXQ literature assumes that the termination function $\beta^o : S \to [0, 1]$ is binary: for a given state, each task either always terminates or never terminates upon reaching that state. It also assumes that the initiation set $I^o = \{s \in S \mid \beta^o(s) = 0\}$ is simply the set of nonterminal states. Note that these assumptions match the usage of options in practice, with the exception of some option-discovery research that grows the initiation set incrementally (as discussed in Section 4.3.1). For consistency, this chapter maintains the subtask formalism derived from options, although the subtasks used will obey the MAXQ assumptions regarding β and I.

In addition, note that MAXQ allows the subtask set A^o to contain both primitive actions and other composite subtasks. The only restriction is that the task hierarchy cannot produce a cycle, in which a task includes itself as a descendent task. In contrast, option policies π^o typically include only primitive actions; the option formalism bears a resemblance to the idea of macro operators in planning, where there is little advantage to defining recursive macros, instead of a single flattened macro specified only in terms of primitives. Only when the behavior of an option is being learned or discovered does additional levels of hierarchy make sense!

As a concrete example, the `NAVIGATE TO RED` task in Figure 5.1b has the following components. The subtask set $A = \{\texttt{north}, \texttt{south}, \texttt{east}, \texttt{west}\}$ includes just the four movement-related primitive actions. The termination function specifies termination probability $\beta(s) = 1$ for all states s in which the taxi's x coordinate is 0 and its y coordinate is 4; $\beta(s) = 0$ for all other states. The goal value function G assigns value 0 to all states, since the only goal is to minimize penalties accrued before termination.

Moving up the hierarchy, the `GET` task includes four `NAVIGATE` tasks as subtasks, as well as the primitive action `pickup`. `GET` terminates once the passenger is inside the taxi. Finally, the `ROOT` task only invokes `GET` and `PUT` as subtasks; it does not need to reason directly about any of the primitive actions.

Each task o defines an SMDP learning problem $\left\langle S, A^o, \tilde{R}^o, \tilde{P}^o \right\rangle$ with the same state space S of the underlying MDP[1] but an action space A^o that depends on the task o. This state-action space provides the interface for the RL algorithm used to learn a solution to this SMDP problem. It also indexes the reward and state transition matrices $\tilde{R}^o$ and $\tilde{P}^o$, which combine information about the behavior of the actions A^o with the task termination function β^o and goal value function G^o:

$$\begin{aligned} \tilde{R}^o &= R^o + P^o \beta^o G^o && (5.1) \\ \tilde{P}^o &= P^o \left(1 - \beta^o\right), && (5.2) \end{aligned}$$

where β^o is represented as a $S \times S$ diagonal matrix where $\beta^o[s, s]$ is the probability of termination upon reaching s, and R^o and P^o are multi-time models

[1]Task-specific state abstraction may additionally allow the task to reason in a reduced state space, and such abstraction may be a primary motivation of hierarchical decomposition.

of the subtasks A^o. These equations take advantage of the multi-time representation of the state-transition function: $\tilde{P}^o$ represents task termination by zeroing out transition probabilities into terminal states. The reward function incorporates goal values for each state-action sa by adding the expected value of $G^o(s')$. Note that since child action $c \in A^o$ may itself be a task, the entry $P^o[sc, s']$ may describe the estimated behavior of an option, not a primitive action. R^o and P^o therefore depend on the current option policies of all the descendent tasks of o. This dependency highlights a difficulty of learning different levels of hierarchy concurrently: above the lowest level, the SMDP learning problems become nonstationary.

Nevertheless, if task o and all its descendents have current option policies, then the standard Bellman equations still hold:

$$\tilde{Q}^o = \tilde{R}^o + \tilde{P}^o \tilde{V}^o \tag{5.3}$$

$$\tilde{V}^o = \pi^o \tilde{Q}^o, \tag{5.4}$$

where the tilde indicates that these estimated values include the artificial goal values, which are not an intrinsic part of the environment. Given policies for its descendent tasks, and therefore the SMDP reward and state transition functions $\tilde{R}^o$ and $\tilde{P}^o$, standard planning algorithms can compute the recursively optimal policy for task o, which maximizes Equation (5.4). The option policy therefore maximizes the expected discounted sum of both rewards earned during execution and goal values earned upon termination. Given the option policy π^o, the "uncontaminated" task value function is given by solving

$$Q^o = R^o + P^o (1 - \beta^o) V^o \tag{5.5}$$

$$V^o = \pi^o Q^o = \pi^o (R^o + P^o (1 - \beta^o) V^o) . \tag{5.6}$$

These equations modify the standard Bellman equations only to apply task termination: terminal states have 0 value, reflecting the fact that no further reward is earned after task termination.

These task value functions simply give the expected reward accumulated during the execution of the task. For example, $V^{\texttt{GET}}[s]$ corresponds to the expected cumulative discounted reward earned by executing the `GET` task starting in state s (for some current option policy $\pi^{\texttt{GET}}$, as well as option policies for the descendent tasks). The value function $V^{\texttt{ROOT}}$ corresponds to the value function in the underlying MDP for the hierarchical policy specified by the current option policy in each task. A key idea of MAXQ is that these value functions–and therefore their corresponding RL problems–are related in specific ways determined by the structure of the task hierarchy.

The MAXQ framework defines for each task o the *completion function* $C^o : S \times A \to \mathbb{R}$, which estimates the cumulative discounted reward obtained by completing task o after first performing subtask $a \in A^o$ starting in state $s \in I^o$. In matrix notation, this function may be defined:

$$C^o = P^o V^o, \tag{5.7}$$

so it satisfies $Q^o = R^o + C^o$. The completion function $\tilde{C}^o$ including goal values is defined analogously: $\tilde{C}^o = P^o \tilde{V}^o$.

The key benefit to decomposing Q^o into the sum of R^o and C^o is that the task o need only learn C^o. The subtask rewards R^o depend only on the subtasks. In particular,

$$R^o[sc] = V^c[s] \tag{5.8}$$

for all composite tasks o, child tasks $a \in A^o$, and states $s \in S$. If c represents a primitive action a, then

$$V^a[s] = R[sa], \tag{5.9}$$

where R is the reward function of the underlying MDP. For example, suppose that in a given state s, $\pi^{\texttt{ROOT}}$ executes `GET`, $\pi^{\texttt{GET}}$ executes `RED`, and $\pi^{\texttt{RED}}$ exexecutes `north`. Then the overall value of s can be decomposed as follows:

$$\begin{aligned}
V^{\texttt{ROOT}}[s] &= Q^{\texttt{ROOT}}[s\texttt{GET}] \\
&= V^{\texttt{GET}}[s] + C^{\texttt{ROOT}}[s\texttt{GET}] \\
&= Q^{\texttt{GET}}[s\texttt{RED}] + C^{\texttt{ROOT}}[s\texttt{GET}] \\
&= V^{\texttt{RED}}[s] + C^{\texttt{GET}}[s\texttt{RED}] + C^{\texttt{ROOT}}[s\texttt{GET}] \\
&= Q^{\texttt{RED}}[s\texttt{north}] + C^{\texttt{GET}}[s\texttt{RED}] + C^{\texttt{ROOT}}[s\texttt{GET}] \\
&= V^{\texttt{north}}[s] + C^{\texttt{RED}}[s\texttt{north}] + C^{\texttt{GET}}[s\texttt{RED}] + C^{\texttt{ROOT}}[s\texttt{GET}] \\
&= R[s\texttt{north}] + C^{\texttt{RED}}[s\texttt{north}] + C^{\texttt{GET}}[s\texttt{RED}] + C^{\texttt{ROOT}}[s\texttt{GET}].
\end{aligned}$$

Each of the quantities in the final line may be estimated separately, each by the task indicated in the superscript. During policy evaluation, the overall value of s can then be estimated by summing these components. In this example, given the described policy, the value of the `ROOT` task at state s is the sum of the value of the `north` primitive action at state s, plus the value of completing the `RED` task after executing `north` at state s, plus the value of completing the `GET` task after executing `RED` at state s, plus the value of completing `ROOT` after executing `GET` at state s.

The original MAXQ-Q algorithm learns each completion function (and the reward function for primitive actions) using the same stochastic-approximation technique as Q-learning. It relies on the function EvaluateMaxNode (Algorithm 5.1) to compute $V^o[s]$ for a given task o and state s. This function uses the "internal" completion functions $\tilde{C}$ to determine each task's choice of subtask, but it computes values using C, which only incorporates environmental rewards.

Algorithm 5.1 EvaluateMaxNode(σ, o, s)

```
if $o$ is a primitive task then
    $a \leftarrow o$                         {Get primitive action encapsulated by $o$}
    return $R^\sigma[sa]$
else
                                             {Choose child using local goal values}
    $c^{\max} = \text{argmax}_{c \in A^o \mid s \in I^c}$ EvaluateMaxNode($c, s$) $+ \tilde{C}^\sigma_o[sc]$
                                             {Report value without local goal values}
    return EvaluateMaxNode($c^{\max}, s$) $+ C^\sigma_o[sc^{\max}]$
end if
```

Algorithms 5.2 and 5.3 specify the Update and Choose subroutines for MAXQ-Q. MAXQChoose recursively chooses subtasks using ϵ-greedy action selection, building a stack of currently executing tasks. For each task that terminates, MAXQUpdate uses stochastic approximation to improve the completion function estimates for composite tasks and the reward function estimates that determine the values of primitive tasks. It applies this update to each state visited during the just-terminated task, since the terminal state reached serves as a valid sample for hypothethical task executions that start after the actual execution. To this end, the agent maintains the full state history $\{s\}_i$ indexed by time steps i and task initiation times $\{i\}^o$ associated with each task o.

Algorithm 5.2 MAXQChoose(s)

```
$o \leftarrow \textsc{Top}(K^\sigma)$                                  {Currently executing task}
while $o$ is not a primitive task do
  if Agent explores with probability $\epsilon^o$ then
    $c \leftarrow$ drawn uniformly at random from $c \in A^o \mid \beta^c[s,s] = 0$
  else
    $c \leftarrow \text{argmax}_{c \in A^o \mid s \in I^c} \textsc{EvaluateMaxNode}(c, s) + \tilde{C}^\sigma_o[sc]$
  end if
  $o \leftarrow c$
  $K^\sigma \leftarrow \textsc{Push}(K^\sigma, o)$                                {Initiate task $o$}
  $i^\sigma_o \leftarrow t$                                              {Record time $o$ initiated}
end while
$s^\sigma_t \leftarrow s$                                                {Record state history}
return $a$
```

Compared to standard Q-learning with only primitive actions, MAXQ-Q with a task hierarchy must estimate many more parameters. Instead of one value $Q[sa]$ for each pair or state and action, it learns one value $C^o[sc]$ for each combination of task, subtask, and state in which those tasks may execute. Even with the domain information encoded in the task hierarchy shown in Figure 5.1b, MAXQ-Q learns the Taxi domain more slowly than Q-learning in the absence of state abstraction (Dietterich, 2000a). However, by dint of

Algorithm 5.3 MAXQUPDATE(s, ar, s')

```
$o \leftarrow \text{Top}(K^\sigma)$                                   {Task encapsulating primitive action $a$}
$R[sa] \overset{\alpha^o}{\leftarrow} r$
repeat
  $c \leftarrow o$                                                   {Now consider $o$ the child...}
  $K^\sigma \leftarrow \text{Pop}(K^\sigma)$
  $o \leftarrow \text{Top}(K^\sigma)$                                              {Parent of $c$}
  if $\beta^o[s', s'] = 1$ then {$o$ will terminate}
    $\tilde{V}^\sigma_o[s'] = G^o[s']$
    $V^\sigma_o[s'] = 0$
  else
                                   {Determine next task that $o$ will execute}
    $c' \leftarrow \text{argmax}_c \text{EvaluateMaxNode}(c, s') + \tilde{C}^\sigma_o[s'c]$
    $\tilde{V}^\sigma_o[s'] \leftarrow \text{EvaluateMaxNode}(c', s') + \tilde{C}^\sigma_o[s'c']$
    $V^\sigma_o[s'] \leftarrow \text{EvaluateMaxNode}(c', s') + C^\sigma_o[s'c']$
  end if
  for all timesteps $i \in \{i^\sigma_c \dots t^\sigma\}$ since $c$ began execution do
    $\tilde{C}^\sigma_o[s^\sigma_i c] \overset{\alpha^o}{\leftarrow} \gamma^{t-i+1}\tilde{V}^\sigma_o[s']$
    $C^\sigma_o[s^\sigma_i c] \overset{\alpha^o}{\leftarrow} \gamma^{t-i+1}V^\sigma_o[s']$
  end for
until $\beta^o[s', s'] = 0$
$t^\sigma \leftarrow t^\sigma + 1$
```

estimating the values of individual subtasks, the completion functions $\{C\}^o$ facilitate state abstraction in ways that a monolithic value function Q does not. For example, the RED task simply navigates to the red landmark, so the state representation of $C^{\texttt{RED}}$ only needs to include x and y coordinates. Dietterich identifies several forms of abstraction, which reduce the overall number of parameters estimated by MAXQ-Q for the Taxi domain to 632, compared to 3000 for Q-learning (Dietterich, 2000a). Observing that MAXQ-Q then outperforms Q-learning, he identifies state abstraction as a primary motivation for introducing hierarchy.

Options can also facilitate state abstraction, so the key difference between typical option-learning algorithms and MAXQ-Q is the latter's decomposition of the value function. This decomposition, and the resulting communication across levels of hierarchy, allows values learned for a subtask directly to influence values estimated for its parent task. The MAXQ framework more naturally handles this case of concurrent learning across levels of hierarchy, since the standard option formalism represents options as composite actions with static behavior.

5.2 Hierarchical Models for Reinforcement Learning

This section develops a new algorithm that builds upon MAXQ by extending its idea of communicating values across levels of hierarchy to communicating models across levels of hierarchy. MAXQ observes that the value function learned for a subtask SMDP is equivalent to a portion of the reward function used to plan in the parent task SMDP. MAXQ-Q still relies on stochastic approximation to learn C, which combines the state transition function and state value function, but a logical question is what property of a subtask can be used to construct the state transition function for the parent task?

After answering this question, this thesis proposes the first fully model-based hierarchical RL algorithm for the standard discounted-reward RL setting. This algorithm, R-MAXQ, combines the advantages of the model-based R-MAX algorithm and the MAXQ framework for hierarchy. In doing so, it can help address critical issues that Chapter 4 raised: what precisely is the role of hierarchy in RL, and how does it interact with the problem of intelligent exploration?

The key idea behind this model-based approach is to assume that a composite task o can query a child task c for not just $R^o[sc]$ but also for $P^o[sc]$. Then the only unknown quantity in Equation (5.3) is $\tilde{V}^o$, which can be computed using standard dynamic programming methods and stored locally.

A task o can define its reward function $R^o[sc] = R_c[s]$ and its state transition function $P^o[sc, x] = P_c[s, x]$, for all $c \in A^o$ and $s, x \in S$, if each child task c can compute its estimated rewards R_c and transition probabilities to terminal states P_c.[2] Prior research into option models (Sutton et al., 1999)

[2]The superscript in P^o is intended to connote that the state-action transition matrix P^o is part of the higher-level task o, used to compute the policy that associates o with its lower-level subtasks. The subscript in P_c is intended to connote that c is a lower-level subtask, and P_c is the relevant portion of the state-action transition matrix for a parent task of c.

defined Bellman-like equations for these quantities, which can be translated into this thesis's matrix notation as follows:

$$R_o = \pi^o \left(R^o + P^o \left(1 - \beta^o\right) R_o\right) \tag{5.10}$$

$$P_o = \pi^o P^o \left((1 - \beta^o) P_o + \beta^o\right). \tag{5.11}$$

In this notation, MAXQ's key insight is that $R_o = V^o$. For a parent task p, R^p can be constructed from the value functions V^o for each $o \in A^p$.

This thesis analogously defines P_o as the *terminal-state function* of o, hereafter denoted Ω^o, which gives the discounted probability $\Omega^o[s, s']$ that task o, when initiated in state $s \in I^o$, will terminate in state $s' \in S \setminus I^o$. Each column of Ω^o, corresponding to the discounted transition probabilities into a state s', can be thought of as a value function for a task which offers reward 1 for reaching s' (as well as terminating upon reaching this state) and reward 0 elsewhere. It follows that the terminal-state function can also be computed using standard MDP planning algorithms, as well as by solving

$$\Omega^o = \pi^o \left(P^o \beta^o + P^o \left(1 - \beta^o\right) \Omega^o\right). \tag{5.12}$$

Although prior work on option model-learning (Sutton et al., 1999) presented the original forms of Equations (5.6) and (5.12), the algorithm it proposed learned these quantities through relatively inefficient stochastic approximation. The MAXQ framework (Dietterich, 2000a) formally decomposed the overall task value function into components using Equation (5.8) to relate the reward function R^o of a parent task to the value function V^c of a child task. This value-function decomposition allows R to be computed recursively, leaving the MAXQ-Q algorithm only to learn $C^o = P^o V^o$ through stochastic approximation (along with V^a for primitive tasks only). The novel R-MAXQ algorithm takes this idea a step further and uses

$$P^o[sc, s'] = \Omega^c[s, s'] \tag{5.13}$$

to relate the state transition function of a parent task to the terminal-state functions of its child tasks.

A key strength of the new algorithm is its ability to compute directly the reward and transition models for a given composite task, using dynamic programming instead of stochastic approximation. Unlike MAXQ-Q, no explicit learning need take place at the higher levels of the hierarchy, where values are obtained from planning with models derived from the lower levels.

This model decomposition provides a way to compute policies and therefore high-level transition and reward models given lower-level transition and reward models. To ground out this process, models of the primitive actions must be available. However, if R^a and P^a are available for each primitive action a, note that we could compute the optimal policy of the system using standard (non-hierarchical) planning algorithms. Nevertheless, Section 5.3 empirically demonstrate the benefit of using hierarchies. The next section first presents the new model-based learning algorithm.

5.2.1 The R-MAXQ Algorithm

The previous section shows how to compute abstract models from primitive models, but a complete model-based RL algorithm must specify how to estimate the primitive models. This section combines this hierarchical model decomposition, inspired by the MAXQ value function decomposition, with the primitive models defined by the R-MAX algorithm (Brafman & Tennenholtz, 2002), yielding a new algorithm, R-MAXQ.

As discussed in Section 2.3.2, R-MAX begins with the maximum-likelihood estimates of the MDP reward and state transition matrices R and P (defined in terms of primitive actions only). It keeps track of the sample size for each state-action, and when this sample is smaller than some threshold, it overrides the appropriate row of these matrices with optimistic values. With the multi-time representation of P, the optimistic model can be implemented quite simply: $R[sa] = v^{\max}$ and $P[sa, s'] = 0$, for undersampled state-action sa, all successor state s', and an upper bound on the value function $v^{\max}$. MDP planning with the revised matrices directly yields the exploration policy used at the current time step.

R-MAXQ works in the same way, except it computes a hierarchical value function using its model decomposition instead of a monolithic value function using the standard MDP model. It applies optimistic rewards to the primitive action rewards R, which then propagate up the hierarchy, via each task's computed abstract reward function V^o. Each local policy implicitly exploits or explores by choosing a child task with high apparent value, which combines the task's actual value and possibly some optimistic bonus due to some reachable unknown states. No explicit reasoning about exploration is required at any of the composite tasks in the hierarchy: as in R-MAX, the planning algorithm is oblivious to its role in balancing exploration and exploitation in a learning agent. A key advantage of R-MAXQ is that its hierarchy allows it to constrain the agent's policy in a fashion that may reduce unnecessary exploratory actions, as illustrated in Section 5.3.

Algorithm 5.4 specifies the CHOOSE component of R-MAXQ. It maintains a stack of currently executing tasks, K. Each time step begins by popping tasks that have terminated and then pushing tasks according to each task's policy. The necessary models, value functions, and policies are computed by the mutually recursive functions RMAXQPLAN and RMAXQEVALUATE.

Algorithm 5.4 RMAXQCHOOSE(s)

```
$o \leftarrow$ TOP($K^\sigma$)                                   {Current executing task}
while $\beta^o[s, s] = 1$, implying task $o$ should terminate do
    $K^\sigma \leftarrow$ POP($K^\sigma$)
    $o \leftarrow$ TOP($K^\sigma$)
end while
while $o$ is not a primitive task do
    RMAXQPLAN($o$)                                                {Computes $\pi_o^\sigma$}
    $c \leftarrow$ chosen with probability $\pi_o^\sigma[s, sc]$
    $o \leftarrow c$
    $K^\sigma \leftarrow$ PUSH($K^\sigma, o$)                                 {Initiate task $o$}
end while
return $a$
```

Algorithm 5.5 computes the policy for a given composite task o by constructing an SMDP and simply applying value iteration. The rewards and transitions that comprise the MDP are obtained by combining the task's goal values G^o and termination function β^o with the results of policy evaluation for the subtasks, as defined in Equations (5.1) and (5.2).

Algorithm 5.5 RMAXQPLAN(o)

```
for all subtasks $c \in A^o$ of task $o$ do
  RMAXQEVALUATE($\sigma, c$)                    {Compute $V_c^\sigma$ and $\Omega_c^\sigma$}
end for
                  {Construct (and store) MDP from child task models}
$R_o^\sigma \leftarrow [sc] \mapsto V_c^\sigma[s]$
$P_o^\sigma \leftarrow [sc, s'] \mapsto \Omega_c^\sigma[s, s']$
            {Temporarily apply task termination and goal rewards to MDP}
$\tilde{R}_o \leftarrow R_o^\sigma + P_o^\sigma \beta^o G^o$
$\tilde{P}_o \leftarrow P_o^\sigma (1 - \beta^o)$
$\tilde{V}_o^\sigma \leftarrow$ VALUEITERATION($\tilde{R}_o, \tilde{P}_o, \epsilon, \tilde{V}_o^\sigma$)        {Plan with modified model}
$\tilde{Q}_o \leftarrow \tilde{R}_o + \gamma \tilde{P}_o \tilde{V}_o^\sigma$
$\pi^o \leftarrow$ chosen so that $\pi[s, sa] > 0 \Rightarrow a \in \operatorname{argmax}_a \tilde{Q}_o[sa]$
```

Algorithm 5.6 computes the value function and terminal-state function for a given task. For primitive tasks, these functions are equivalent to the maximum likelihood reward and state transition functions, modified with the R-MAX optimistic exploration mechanism. For composite tasks, Algorithm 5.5 is used to compute the task policy, and then policy evaluation models the behavior of the task.

Compared to other hierarchical learning algorithms, R-MAXQ by dint of its model-based approach, focuses almost exclusively on planning. Algorithms 5.4-5.6 address the computation of a hierarchical policy given a simply maximum-likelihood model of the primitive actions (including sample sizes for each state-action). In fact, to obtain a complete agent, it suffices to combine

Algorithm 5.6 RmaxqEvaluate(o)

```
if $o$ is primitive then
    $a \leftarrow o$                                   {Use $a$ for primitive actions}
    $V_a^\sigma \leftarrow [s] \mapsto \begin{cases} v^{\max} & \text{if } N^\sigma[sa] < m \\ R^\sigma[sa] & \text{otherwise} \end{cases}$
    $\Omega_a^\sigma \leftarrow [s, s^\sigma] \mapsto \begin{cases} 0 & \text{if } N^\sigma[sa] < m \\ P^\sigma[sa, s'] & \text{otherwise} \end{cases}$
else {$o$ is a composite task}
    RmaxqPlan($o$)                                     {Sets $R_o^\sigma$, $P_o^\sigma$, $\pi_o^\sigma$}
                                                       {Solve Equation (5.6)}
    $V_o^\sigma \leftarrow$ solution to $V_o^\sigma = \pi_o^\sigma \left(R_o^\sigma + P_o^\sigma \left(1 - \beta^o\right) V_o^\sigma\right).$
                                                       {Solve Equation (5.12)}
    $\Omega_o^\sigma \leftarrow$ solution to $\Omega_o^\sigma = \pi_o^\sigma \left(P_o^\sigma \beta^o + P_o \left(1 - \beta^o\right) \Omega_o^\sigma\right).$
end if
```

the RmaxqChoose function with the same MaximumLikelihoodUpdate function (Algorithm 2.9) used by R-max.

Note that this presentation of R-maxq omits many optimizations for computational efficiency. It instead focuses on completely describing the logic that determines the agent's action selection and therefore its data efficiency. Chapter 7 discusses the software architecture of a practical implementation, while the next section addresses the sample complexity of R-maxq.

5.2.2 Analysis of R-MAXQ

R-maxq probably follows an approximately optimal policy for all but a finite number of time steps. Unfortunately, this number may be exponential in the size of the hierarchy. This section provides a sketch of a proof and then closes with a brief discussion of the implications of this result.

The original R-MAX algorithm achieves efficient exploration by using an optimistic model. Its model of any given state-action pair is optimistic until it samples that state-action m times. By computing a value function from this optimistic model, the resulting policy implicitly trades off exploration (when the value computed for a given state includes optimistic rewards) and exploitation (when the value only includes estimates of the true rewards). Kakade bounded the sample complexity of RL (Kakade, 2003) by first showing that R-MAX probably only spends a finite number of time steps attempting to reach optimistic rewards (exploring). For the remaining (unbounded) number of time steps, the algorithm exploits its learned model, but its exploitation is near-optimal only if this model is sufficiently accurate. Kakade then bounded the values of m necessary to ensure the accuracy of the model with high probability.

To be precise, let an MDP with finite state space S and finite action space A be given. Let ϵ be a desired error bound, δ the desired probability of failure, and γ the discount factor. Then R-MAX applied to an arbitrary initial state will spend $O\left(\frac{m|S||A|L}{\epsilon} \log \frac{|S||A|}{\delta}\right)$ time steps exploring, with probability greater than $1 - \frac{\delta}{2}$, where $L = O\left(\frac{\log \epsilon}{1-\gamma}\right)$. Furthermore, there exists an $m \in O\left(\frac{|S|L^2}{\epsilon^2} \log \frac{|S||A|}{\delta}\right)$ such that when the agent is not exploring, $V^{\pi^*}(s_t) - V^{\pi_t}(s_t) \leq \frac{\epsilon}{1-\gamma}(R^{\max} - R^{\min})$ with probability greater than $1 - \frac{\delta}{2}$, where s_t and π_t are the current state and policy at time t, and $R^{\max}$ and $R^{\min}$ bound the reward function.

The hierarchical decomposition used by R-MAXQ complicates an analysis of its sample complexity, but essentially the same argument that Kakade used provides a loose bound. Kakade's proof (Kakade, 2003) provides the gross structure of the argument; this section will only sketch the necessary extensions. A key lemma is Kakade's ϵ-approximation condition (Lemma 8.5.4). The state transition model $\hat{\Omega}^a$ for a primitive action a (Ω^a can be thought of as the portion of the state transition function P that applies to a) is an ϵ-approximation for the true dynamics Ω^a if for all states $s \in S$, $\sum_{s' \in S} \left| \hat{\Omega}^a(s, s') - \Omega^a(s, s') \right| < \epsilon$. The ϵ-approximation condition states that if a model has the correct reward function V^a (again, the portion of the reward function R that applies to a) but only an ϵ-approximation of the transition dynamics Ω^a for each action, then for all policies π and states $s \in S$, $\left| \hat{V}^\pi(s) - V^\pi(s) \right| < \frac{\epsilon L}{1-\gamma}$.

Essentially, this condition relates the error bounds in the approximation of the transition function to the resulting error bounds in the computed value function. It allows the analysis of R-MAX to determine a sufficient value of m to achieve the desired degree of near optimality. This condition must be extended in two ways to adapt the overall proof to R-MAXQ. First, R-MAXQ violates Kakade's assumption of deterministic reward functions. While primitive actions can still be assumed to have deterministic (and thus trivial to learn) rewards, the abstract rewards of composite tasks depend on the stochastic dynamics of the environment. Define an estimated subtask reward function $\hat{V}^c$ to be a λ-approximation of the true reward function V^c if for all states $s \in S$, $\left| \hat{V}^c(s) - V^c(s) \right| < \lambda$. Then it is straightforward to adjust Kakade's derivation of the ϵ-approximation condition to show that the computed value function V^o for any given policy satisfies for all $s \in S$, $\left|\hat{V}^o(s) - V^o(s)\right| < \frac{\epsilon L}{1-\gamma} + \lambda$. In other words, introducing λ error into a previously perfect reward function (subtask value function) only increases the error of the estimated value function by λ. This extends Kakade's bounds to the error of V^o given errors in V^c and Ω^c.

The second necessary extension is a bound on the error of Ω^o given errors in V^c and Ω^c. Equation (5.12) illustrates that for every $s' \in S \setminus I^c$, $\Omega^o[\cdot, s']$ can be thought of as a value function estimating the expected cumulative discounted probability of transitioning into s'. The total error in $\Omega^o(s, \cdot)$ will thus be bounded by the sum of the errors for each $s' \in S \setminus I^c$, so it can be shown that Ω^o is an $O\left(\frac{|S \setminus I^c| \epsilon L}{1-\gamma}\right)$-approximation. The error bound for Ω^o is therefore larger than the error bound for V^o by a factor of $|S \setminus I^c|$, reflecting the fact that Ω^o can be computed as a collection of $|S \setminus I^c|$ value functions.

These results bound the errors that propagate up from the primitive actions in the hierarchy, but these bounds seem quite loose. In particular, these bounds can't rule out the possibility that each level of the hierarchy might multiply the approximation error by a factor of $\frac{|T^a|L}{1-\delta}$. Since the amount of data required varies as the inverse square of ϵ, if R-MAX requires m samples of each action at each state to achieve a certain error bound, R-MAXQ may require $m' = O\left(m \left(\frac{TL}{1-\delta}\right)^{2h}\right)$ samples of each primitive action at each state to achieve the same error bound at the root of the hierarchy, where T is the maximum number of reachable terminal states for any composite action and h is the height of the hierarchy: the number of composite tasks on the longest path from the root of the hierarchy to a primitive action (not including the root itself).

Adapting the remainder of Kakade's proof establishes that R-MAXQ will probably converge to a (recursively) near-optimal policy, although this guarantee requires exponentially more data than R-MAX in the worst case. Note that this guarantee applies to any choice of hierarchy. It remains to be seen whether it might be possible to derive tighter bounds for specific classes of action hierarchies. Furthermore, as Kakade notes in his derivation, the ϵ-approximation condition is perhaps unnecessarily stringent, since it gives the worst possible degradation in approximation quality over all possible policies (Kakade, 2003).

In practice, implementations of R-MAX use far smaller values of m than would be required to achieve useful theoretical guarantees. In this vein, note that running R-MAXQ will result in no more time spent in exploration than running R-MAX with the same value for m. The hierarchical decomposition only weakens the guarantees on the near-optimality of the policy that R-MAXQ exploits. The experiments described in the next section show that a good hierarchy can even reduce the amount of time spent exploring, with no appreciable deterioration in solution quality.

5.3 Experimental Results

This section presents empirical results, which show that R-MAXQ outperforms both of its components, R-MAX and MAXQ-Q. It discusses these outcomes in detail, to reveal how precisely the algorithm benefits from combining model-based learning and hierarchical decomposition.

The experiments with R-MAX and R-MAXQ set the threshold sample size at $m = 5$. Preliminary experiments showed that larger values of m did not signicantly improve the final policy, although of course they led to more time spent estimating the model. The only other parameter for these algorithms is the stopping criterion for the dynamic programming steps. In all cases, value iteration ran until the largest change was smaller than $\epsilon = 0.001$. R-MAXQ and the original MAXQ-Q algorithm were provided with the hierarchy shown in Figure 5.1b as prior knowledge. The implementation of MAXQ-Q used precisely the hand-tuned parameters Dietterich optimized for the initial value function, learning rates, and temperature decay (for Boltzmann exploration) for each action in the hierarchy (Dietterich, 2000a). For each condition, 100 independent trials were conducted.

5.3.1 R-MAXQ versus R-MAX

First compare the performance of R-MAXQ and R-MAX on the Taxi task. A reasonable initial hypothesis would be that R-MAXQ would perform no better than R-MAX in the absence of state abstraction, since the model-based ability to plan to explore might subsume the exploratory role that options have played in many model-free RL implementations (Şimşek & Barto, 2004; Singh et al., 2005). Figures 5.2 and 5.3 reveal that in fact the two algorithms exhibit very different learning curves. In particular, although R-MAX requires many

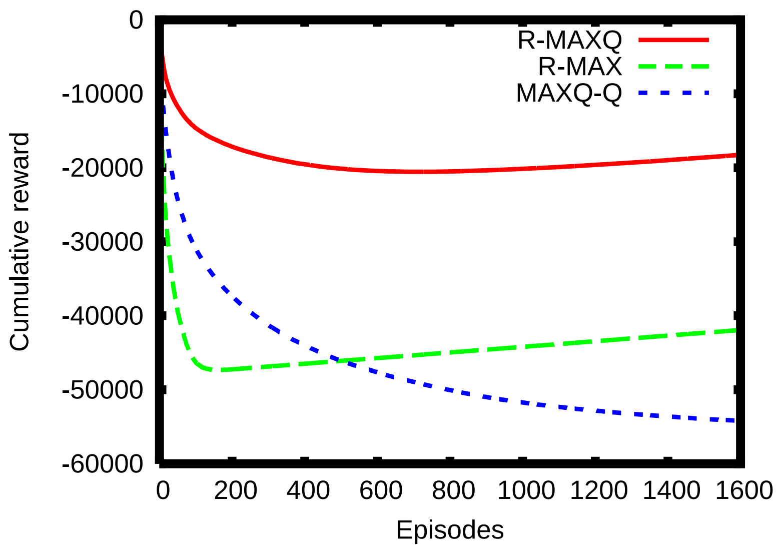


Figure 5.2: Cumulative performance of R-maxq, R-max, and maxq-q on the Taxi domain, averaged over 100 independent trials. The differences among these total costs of learning over 1600 episodes are significant ($p < 0.001$).

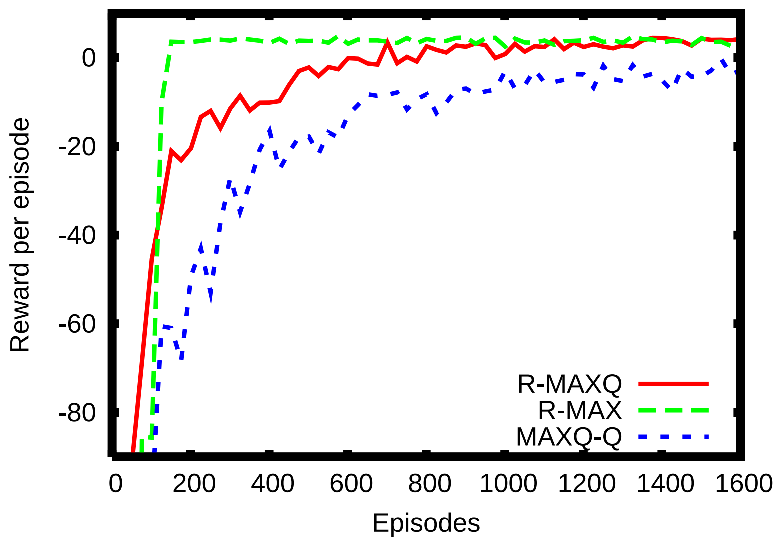


Figure 5.3: Asymptotic performance of R-maxq, R-max, and maxq-q on the Taxi domain, averaged over 100 independent trials. R-maxq and R-max each learn significantly better policies than maxq-q after 1600 episodes ($p < 0.001$).

fewer episodes to converge to an optimal policy, R-MAXQ earns much greater total reward.

The hierarchy used by R-MAXQ doesn't so much guide exploration as it constrains it. In particular, note that the hierarchical agent can never attempt the `putdown` action except at one of the four landmark locations, since the `PUT` action only becomes available when the agent is already at one of these locations, and the four navigational actions keep the agent in this reduced set of states. The agent thus only attempts the `putdown` action in 12 incorrect states, instead of the 396 explored by R-MAX. In addition, R-MAX attempts the `pickup` action in 100 states in which R-MAXQ doesn't, when the passenger is already in the car. Since the penalty for incorrect usage of these actions is -10, R-MAX loses $10(396-12+100)m = 24200$ reward due to its wasted exploration, accounting for the difference between the two algorithms in Figure 5.2. Furthermore, since the `GET` action cannot navigate to an arbitrary location, R-MAXQ can't attempt the `pickup` action in a non-landmark location until some episode randomly starts the agent there. In this case the hierarchy can only postpone, not prevent, wasted exploration. This effect explains the delayed convergence relative to R-MAX : in later episodes R-MAXQ spends time on exploration that R-MAX performed more eagerly.

5.3.2 R-MAXQ versus MAXQ-Q

Figures 5.2 and 5.3 also compare R-MAXQ with the original MAXQ-Q algorithm. Of course, this comparison isn't very fair, since a primary goal of the MAXQ framework was to create opportunities for state abstraction (Dietterich, 2000a), which these implementations did not initially exploit. In fact, Dietterich identified the condition described in Section 5.3.1, which he called shielding, as one that permits abstraction. For a more fair comparison, the implementation of MAXQ-Q was allowed to use all the state abstractions in the Taxi domain identified by Dietterich, along with his optimized parameters (Dietterich, 2000a).

In return, R-MAXQ was also allowed to enjoy an explicit form of state abstraction as prior knowledge: Dietterich's notion of max node irrelevance. Each action in the hierarchy abstracts away state variables when domain knowledge indicates that doing so would not compromise the learned model. However, whereas in MAXQ-Q a task o only reports its abstract values V^o to its parents, in R-MAXQ it must also convey the abstract transitions Ω^o. Thus a composite action could only ignore a state variable if all of its children also ignore that state variable.

In the hierarchy shown in Figure 5.1b, the four primitive movement actions and the four navigational actions can abstract away the `passenger` and `destination` state variables. `GET` and `pickup` ignore `destination`, and `PUT` and `putdown` ignore `passenger`. However, `ROOT` cannot ignore any state variables. When a child's transition function was more abstract than a parent's model, the parent assumed a very simple dynamic Bayes network (DBN) factorization (Boutilier et al., 1995). For example, $P^{\texttt{north}}$ sets `x` and `y` (each conditional on the previous values of both variables), but `passenger` and `destination` remain constant. Figure 5.4 compares the performance of the resulting algorithms. Both MAXQ-Q and R-MAXQ learn much faster with state abstraction, with the model-based nature of R-MAXQ continuing to give it an edge.

It is worthwhile to examine more closely how the hierarchy interacts with state abstraction in the Taxi domain. Consider how MAXQ-Q learns the `ROOT` action. The only values stored locally are the completion functions $C^{\texttt{ROOT}}(\cdot, \texttt{GET})$ and $C^{\texttt{ROOT}}(\cdot, \texttt{PUT})$, which have different abstract representations. The latter function is always equal to 0, since after `PUT` terminates there is nothing to complete, since the entire episode has terminated. Meanwhile, to evaluate $C^{\texttt{ROOT}}(s, \texttt{GET})$ the algorithm need only inspect the `passenger` and `destination` variables of s, since the values of these two variables before executing `GET` completely determine the remaining cost of completing `ROOT` after `GET` terminates. Hence, MAXQ-Q only learns 16 values at the `ROOT` node; to compute the value of a state it recursively queries R^a and adds the appropriate completion function (Dietterich, 2000a).

R-MAXQ doesn't apply any explicit state abstraction to `ROOT`, but note that after executing either of its two child actions, the result must be one of 12 nonterminal states: with the taxi at one of four landmarks, the passenger in the taxi, and the destination at one of the other three landmarks. Hence, value iteration in the SMDP for the `ROOT` task need only compute the values of these 12 states plus the current state. As with MAXQ-Q, the result distribution irrelevance of `GET` allows R-MAXQ to store only a small number of values

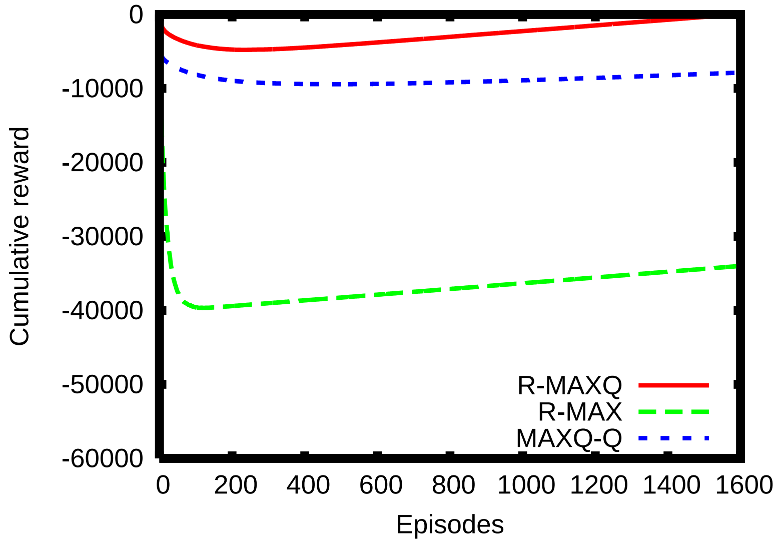


Figure 5.4: Cumulative performance of R-MAXQ, R-MAX, and MAXQ-Q on the Taxi domain, using state abstraction. (The asymptotic performance is qualitatively similar to that shown in Figure 5.3, although with faster convergence.) The differences among these total costs of learning over 1600 episodes are statistically significant ($p < 0.001$)

locally. To compute the value of a state, R-MAXQ also queries one-step values from its children and then adds the appropriate successor state values. In this sense, these 12 states can be thought of as the *completion set* of `ROOT`.

Figure 5.4 also shows the performance of standard R-MAX with the same DBN factorization as R-MAXQ applied to most of its actions (which are all primitive). Note that in the absence of shielding, `putdown` cannot safely ignore the `passenger` variable. The ability to abstract the primitive models does reduce the amount of exploration that R-MAX must perform, but the improvement is significantly smaller than that of the other algorithms. This result gives more support for motivating hierarchical decomposition with opportunities for state abstraction.

5.4 Discussion

Other algorithms have combined hierarchical RL with a model-based approach, but not in the standard framework of discounted rewards and stochastic dynamics. Diuk et al. developed a model-based MAXQ algorithm for deterministic domains, allowing them to quickly sample the effect of a composite action recursively: every action's effect can be represented as a scalar reward and a single successor state (Diuk et al., 2006). Their algorithm also uses Dietterich's approach to state abstraction, occasionally forcing it to replan, since the effect of a child action may depend on state variables not visible to the parent, making it seem nondeterministic. In contrast, R-MAXQ does not employ explicit state abstraction, allowing it to save the value functions and policies computed during one time step for all future time steps. It relies on the choice of hierarchy to decompose MDPs into smaller SMDPs, automatically achieving an effective reduction in the size of the state space considered during any one time step.

Seri and Tadepelli extended the MAXQ framework to average-reward reinforcement learning, resulting in an algorithm that learns a model to facilitate the computation of the bias for each state from the average reward of the current policy (Seri & Tadepalli, 2002). However, the computation of the average reward itself relies on stochastic approximation techniques, and their algorithm does not have any formal guarantees regarding its sample complexity.

The R-MAXQ algorithm combines the efficient model-based exploration of R-MAX with the hierarchical decomposition of MAXQ. Although the algorithm does not improve upon the known formal bounds on the sample complexity of RL, it retains a finite-time convergence guarantee. An empirical evaluation demonstrates that even a relatively simple hierarchy can improve the cumulative reward earned by constraining the exploration that the agent performs, both within individual episodes of learning and throughout an agent's experience with its environment. Even in the absence of explicit state abstraction, the structure of an action hierarchy can drastically reduce the effective state space seen by a given composite action during a single episode. Chapter 6 further investigates this implicit concept of a reduced completion set, mirroring Dietterich's explicitly abstracted completion function.

Chapter 6

Compositional Model-Based Algorithms

Preceding chapters touched upon several divergent branches of RL research, all in pursuit of an understanding into how a learning agent can effectively explore its environment. Given this connection to the fundamental issue of exploration, research is sorely needed into how these branches synergize or possibly even conflict. This hole in the RL literature is not necessarily surprising: these underlying ideas—model-based exploration, function approximation, and hierarchical decomposition—had disparate original motivations and are important enough to attract their own subcommunities of researchers. This thesis has already illuminated connections among these branches of RL, but this chapter will unify elements of each into a single coherent framework. The resulting Fitted R-MAXQ is the culminating algorithm of this thesis.

Chapter 2 covered the basic MDP formalism and the R-MAX algorithm, which offers probabilistic polynomial-time convergence to near-optimal policies. R-MAX achieves this probabilistic guarantee using a model-based approach that dates back at least to the original prioritized sweeping algorithm (Moore & Atkeson, 1993), which both summarized an agent's experience as an MDP model to reuse data efficiently and also employed optimistic exploration to improve the model. Later work formally analyzed the benefits of these intuitive motivations, introducing and later refining bounds on learning in MDPs (Kearns & Singh, 1998; Brafman & Tennenholtz, 2002; Kakade, 2003). Still, these guarantees require that the agent exhaustively explore every state. Particularly in large domains, this exploration can be impractical.

Chapter 3 addressed function approximation, a widespread RL technique necessary to cope with the large state spaces encounted in practical applications. Function approximation defines how to generalize from limited data to even unvisited states, so it directly impacts the issue of exploration. Unsurprisingly, function approximation also invalidates the known convergence guarantees for RL. This connection between exploration and generalization has nevertheless received little attention, with most researchers focusing on the question of accurately estimating value functions from a fixed data set (Lagoudakis & Parr, 2003; Riedmiller, 2005). Fitted R-max demonstrates one way to combine generalization and model-based exploration, but it only serves as a first step.

Chapters 4 and 5 examine hierarchical decomposition, another popular topic in RL. This idea has tremendous intuitive appeal: learning algorithms would presumably benefit from the same awareness of structure that seems to allow humans to cope with the extraordinary complexity of the real world. The precise benefits of hierarchy nevertheless remain unclear, even with the canonical options framework, learning algorithms, and test environments. In-depth experiments in Chapter 4 revealed that hierarchy's key role may be to bias or constrain an agent's exploration, and from this conclusion Chapter 5 developed R-MAXQ which combines hierarchical decomposition with model-based exploration.

The common thread through these chapters is exploration. Despite this connection, almost no research has investigated the possible synergies or redundancies among model-based methods, function approximation, and hierarchical decomposition. This chapter contributes a formulation of these branches of RL into a unified framework that makes their synergies apparent (Jong & Stone, 2009). Section 6.1 introduces this formulation and how it fits function approximation, the model-based R-MAX algorithm, and the hierarchical MAXQ framework. Section 6.2 discusses common properties of these components and how they inform the agent's behavior. Section 6.3 leverages this compositional formulation to define the first algorithm for model-based, continuous-state, hierarchical RL. In combining all these advantages into a single concrete algorithm, Fitted R-MAXQ constitutes the main contribution not just of this chapter but also of the entire thesis. Section 6.4 concludes with some discussion.

6.1 Model Components

As discussed above, the algorithms discussed so far in this thesis share a common thread: they impact the performance of RL by influencing an agent's exploration behavior. Furthermore, like many if not most RL algorithms, they work by changing the computation of the value function, which determines the agent's behavior. These changes are typically expressed largely in terms of modifications to the standard Bellman equations.

A distinguishing feature of the techniques addressed in this thesis is that each modified Bellman equation is equivalent to the standard Bellman equation for some (other) MDP. This property permits standard MDP planning algorithms to perform the actual value-function computation. In other words, each of these learning algorithms can be interpreted as a kind of problem reformulation: an agent can *learn* from one MDP by planning and acting in another MDP, which need not be a direct estimate of the first MDP. This chapter formalizes this notion of MDP transformation, and it describes the insights this perspective offers into exploration.

Model-based RL algorithms conform to the following general template, when examined from the perspective of choosing an action at a particular time step.

1. Estimate a model of each action from data.
2. Compute the optimal value function by planning with the action models.
3. Choose an action according to the policy implied by the value function.

Individual algorithms vary in their precise implementation of each step, but this chapter will demonstrate a class of model-based algorithms that vary only in their implementations of Step 1. Simply by varying how they construct action models from data, they achieve a wide variety of learning and exploration behaviors, despite using precisely the same planning algorithm (Step 2) and action-selection mechanism (Step 3).

We will define this class of algorithms using the agent formalism given in Chapter 2, but first we must formalize Step 1 in the above template. Assume that the agent's state σ includes a record of the data D^σ. This record contains a $|S||A|$-dimensional vector D^σ_R that records the total cumulative reward $D^\sigma_R[sa]$ earned across all executions of action a in state s. It also contains a $|S||A| \times |S|$ matrix D^σ_P that records the number of times $D^\sigma_P[sa, s']$ that state-action sa transitioned to successor state s'. Due to the Markov assumption, the data record D^σ is a sufficient statistic of the agent's experience.

This standardized representation of the data allows us to define the space of possible ways to compute action models from data. Let $\mathcal{D}$ be the space of all possible data records, $\mathcal{V}$ be the space of all possible $|S| \times 1$ value vectors, and $\mathcal{O}$ be the space of all possible $|S| \times |S|$ outcome matrices. A *value model generator* $M^a_V : \mathcal{D} \to \mathcal{V}$ maps a data record to a value vector for a specific action a, and similarly an *outcome model generator* $M^a_\Omega : \mathcal{D} \to \mathcal{O}$ maps a data record to an outcome matrix. For convenience, we refer to M^a_V and M^a_Ω as two components of a single *model generator* M^a for action a.

The model generators M^a serve as the sole meaningful parameter of the GeneratedModel algorithm, defined by Algorithms 6.1 and 6.1. GeneratedModelChoose embodies the three steps from the general template given earlier, and GeneratedModelUpdate simply updates the data record after each action.

Algorithm 6.1 GeneratedModelChoose(s, σ)

```
for all $a \in A$ do
    $V^\sigma_a \leftarrow M^a_V(D^\sigma)$
    $\Omega^\sigma_a \leftarrow M^a_\Omega(D^\sigma)$
end for
$R \leftarrow [sa] \to V^\sigma_a[s]$
$P \leftarrow [sa, s'] \to \Omega^\sigma_a[s, s']$
$V^\sigma \leftarrow$ ValueIteration$(R, P, \epsilon^\sigma, V^\sigma)$
$QgetsR + \gamma P V^\sigma$
$\pi \leftarrow$ PolicyImprovement$(Q)$
$a \leftarrow$ chosen with probability $\pi[s, sa]$
return $(a, \sigma)$
```

Algorithm 6.2 GeneratedModelUpdate(σ, s, a, r, s')

```
$D^\sigma_R[sa] \leftarrow D^\sigma_R[sa] + r$
$D^\sigma_P[sa, s'] \leftarrow D^\sigma_P[sa, s'] + 1$
return $\sigma$
```

GeneratedModel can produce a wide variety of learning and exploration behaviors by specifying appropriate values for the functions M^a, but first we give the most simple non-trivial model generator, MLE, which produces maximum-likelihood estimates for each action model:

$$\textsc{mle}^a_V(D) = [s] \mapsto \begin{cases} \frac{D^\sigma_R[sa]}{N^\sigma_a[s]} & \text{if } N^\sigma_a[s] > 0 \\ 0 & \text{otherwise} \end{cases} \tag{6.1}$$

$$\textsc{mle}^a_\Omega(D) = [s, s'] \mapsto \begin{cases} \frac{D^\sigma_P[sa,s']}{N^\sigma_a[s]} & \text{if } N^\sigma_a[s] > 0 \\ 0 & \text{otherwise} \end{cases} \tag{6.2}$$

where N^σ_a is the $|S|$-dimensional vector containing the sample size $N^\sigma_a[s] = \sum_{s'} D^\sigma_P[sa, s']$ for each state-action sa.

With $M^a = \textsc{mle}$ for all $a \in A$, the GeneratedModel algorithm learns from experience, but it won't explore intelligently. It assumes that each untried state-action has value 0, and it does not distinguish between a sample size of 1 and a sample size of 1,000,000. The R-MAX algorithm, described in more detail in Section 2.3.2, can be seen as a simple refinement that addresses this exploration issue. It maintains a set U of state-action pairs where insufficient data exists to estimate the model accurately. For state-action pairs in U, the R-MAX uses an optimistic model in which the action terminates the trajectory after earning immediate reward $V^{\max}$, an upper bound on the value function. In this manner, the "unknown" state-action pairs are seen as optimal, encouraging the agent to visit them and gather data. Otherwise, the maximum-likelihood estimate of the state-action's effects is used. In effect, the agent computes the value function using the following modified Bellman equation:

$$V^{\pi} = \pi \left(UV^{\max} + (I - U)(\bar{R} + \gamma \bar{P} V^{\pi}) \right), \tag{6.3}$$

where $\bar{R}$ and $\bar{P}$ are the maximum-likelihood reward and transition matrices and U is represented as a $|S||A| \times |S||A|$ diagonal binary matrix where $U[sa, sa] = 1$ iff $sa \in U$. R-MAX has some appealing theoretical properties, such as a probabilistic polynomial bound on the number of times it departs from a near-optimal policy (Kakade, 2003).

GENERATEDMODEL can replicate the behavior of R-MAX simply by using an alternative model generator for each action. However, R-MAX can furthermore be defined as a transformation applied to the model generated by the MLE model generator. Define a *value model transformation* $T_V : \mathcal{V} \to \mathcal{V}$ as a function that maps one $|S|$-dimensional value vector to another, and define an *outcome model transformation* $T_\Omega : \mathcal{O} \to \mathcal{O}$ as a function that maps one $|S| \times |S|$ outcome matrix to another. Then we can define R-MAX as a model transformation with the following value and outcome components for each action:

$$\text{R-MAX}^a_V(V) = U^a V^{\max} + (I - U^a)V \tag{6.4}$$

$$\text{R-MAX}^a_\Omega(\Omega) = (I - U^a)\Omega, \tag{6.5}$$

where U^a is a diagonal binary matrix specifying which state-actions are "unknown," so that $U^a[s,s] = 1$ iff $N^\sigma_a[s] \geq m$ for some threshold m. Note that the model transformation R-MAX^a depends on the sample sizes N^σ_a, not just on the model components given as arguments.

A model transformation T may be composed with a model generator M to obtain a new model generator $T \circ M$, which has the components:

$$\begin{aligned}(T \circ M)_V(D) &= T_V(M_V(D)) \\ (T \circ M)_\Omega(D) &= T_\Omega(M_\Omega(D)).\end{aligned}$$

We can therefore apply R-MAX exploration to our simple maximum-likelihood model-based RL agent by setting each model generator $M^a = \text{R-MAX}^a \circ \text{MLE}^a$. It is straightforward to show that GeneratedModelChoose and GeneratedModelUpdate together behave identically to the R-max algorithm as defined in Section 2.3.2, using RMaxChoose and MaximumLikelihoodUpdate.

Many RL algorithms can be expressed simply as model transformations in this way. Furthermore, these transformations can be composed to yield new algorithms, which ideally synthesize all the benefits of the component algorithms. This framework thus yields a class of algorithms spanned by the possible compositions of these components. The following sections briefly recap the novel algorithms presented earlier in this thesis before similarly recasting them as model components that may be composed together.

6.1.1 Function Approximation

Function approximation scales RL to environments where exhaustive exploration is infeasible or impossible (such as any domain with a continuous or otherwise infinite state space) by introducing the idea of generalization. A function approximator defines a family of value functions with some finite parameterization. Chapter 3 focused on *averagers*, which approximate the value of a given state as a weighted average of the values of a finite subset $X \subset S$. In particular, it approximates $V^\pi[s]$ as a weighted average $\sum_{x \in X} \Phi[s,x]V^\pi[x]$, for a given $|S| \times |S|$ matrix Φ. Fitted Value Iteration (Gordon, 1995) uses this approach by performing value iteration using the following approximation of V^π:

$$V^\pi = \pi\left(R + \gamma P \Phi V^\pi\right). \tag{6.6}$$

This approximation essentially projects the outcome of every action into X, so that the values of the states in X suffice to approximate the value of every state in the original state space S. The model generator FVI underscores the simplicity of this technique.

$$\textsc{fvi}_V(V) \;=\; V \tag{6.7}$$

$$\textsc{fvi}_\Omega(\Omega) = \Omega\Phi \tag{6.8}$$

Some recent algorithms have also applied function approximation to the estimated reward and transition models. The instance-based Kernel-Based Reinforcement Learning algorithm (Ormoneit & Sen, 2002) estimates a value function using the equations:

$$\begin{aligned} V^\pi &= \pi\left(\hat{R} + \gamma \hat{P} V^\pi\right) && (6.9) \\ \hat{R} &= [sa] \mapsto \frac{\sum_{x \in S} K[s,x] D_R^a[x]}{\hat{N}_a[s]} && (6.10) \\ \hat{P} &= [sa, s'] \mapsto \frac{\sum_{x \in S} K[s,x] D_P^a[x,s']}{\hat{N}_a[s]}, && (6.11) \end{aligned}$$

where $\hat{N}_a = K N_a$ is the $|S|$-dimensional vector that gives the weighted amount of data $\hat{N}_a[s] = \sum_{x \in S} K[s,x] N_a[x]$ used to approximate s, and K is the $|S| \times |S|$ kernel weighting matrix that determines the degree $K[s,x] \geq 0$ to which data at state x generalizes to the model of action a at state s.

Although KBRL was intended as an offline algorithm (its authors assumed data sampled randomly from the state-action space), its modifications to the Bellman equations can be encapsulated as a model operator:

$$\begin{aligned}
\textsc{kbrl}^a_V(V) &= \Psi_a V && (6.12)\\
\textsc{kbrl}^a_\Omega(\Omega) &= \Psi_a \Omega && (6.13)\\
\Psi_a &= [s,x] \mapsto \frac{K[s,x]N_a[x]}{\hat{N}_a[s]} && (6.14)\\
\hat{N}_a &= KN_a. && (6.15)
\end{aligned}$$

The Fitted R-MAX algorithm, described in Section 3.2.2, extends R-MAX using the model approximation of KBRL and the averaging function approximation of FVI. It can now be expressed as another instance of GENERATEDMODEL, setting $M^a = \textsc{fvi} \circ \textsc{r-max}^a \circ \textsc{kbrl}^a \circ \textsc{mle}^a$, with R-MAX using the generalized sample sizes $\hat{N}$ instead of the raw sample sizes N.

6.1.2 Hierarchy

Given the structure we perceive in the real world, it seems natural to apply hierarchy to reinforcement learning. Chapter 4 and Section 5.1 describe the two most popular frameworks for hierarchical RL, MAXQ decomposition (Dietterich, 2000a) and options (Sutton et al., 1999). Both frameworks define temporally abstract actions that represent entire sequences of primitive actions. MAXQ decomposes an overall learning problem using a given task hierarchy, where each abstract action is a task that induces its own individual learning problem. In contrast, the options framework formalizes an abstract action as a partial policy, which can be construed as a solution to a task. We will find it convenient to interpret an abstract action o in both ways, depending on context.

A task $o = \langle I^o, A^o, G^o, \beta^o \rangle$ comprises a set of state I^o in which the task may be initiated, a set of subtasks A^o, a goal function $G^o : S \to \mathbb{R}$, and a task termination function $\beta^o : S \to [0, 1]$. The termination function, along with the goal function that provides a synthetic reward for each terminal state, imposes an objective onto the system defined by the state space S and the child actions A^o, which may include both other tasks and primitive actions.[1] The task terminates with probability $\beta^o(s)$ upon reaching a state s and then awards itself an artificial goal value $G^o[s]$, where G^o is a $|S|$-dimensional vector. We can also represent β^o as a $|S| \times |S|$ diagonal matrix.

The optimal policy π^o for task o maximizes

$$\tilde{V}_o = \beta^o G^o + (I - \beta^o)\pi_o \left(R_o + \gamma P_o \tilde{V}_o \right), \tag{6.16}$$

$$\tag{6.17}$$

where R_o and P_o are the (abstract) reward and transition matrices, respectively, for the actions in A^o. By construction, if a state s is terminal, then $\tilde{V}_o[s] = G^o[s]$. Otherwise, the value is the expected discounted reward until reaching a terminal state, plus the expected goal value of that terminal state. The resulting policy π_o chooses children $c \in A^o$ in a way that maximizes a combination of one-step rewards during execution and goal values upon termination. The task value function $\tilde{V}_o$ captures this combined value, but it's

[1]We will index the child actions A^o using c instead of a to emphasize that $c \in A^o$ may be either a task/option or a primitive action.

also possible to compute the value function V_o that only includes the one-step rewards (and is not "contaminated" with the goal rewards). In particular, V_o is given by solving

$$V_o = (I - \beta^o)\pi_o \left(R_o + \gamma P_o V_o\right). \tag{6.18}$$

$$\tag{6.19}$$

Note that Equation (6.18) is the same as Equation (6.16) but without the goal rewards, so V_o only measures the rewards earned until reaching a terminal state. For a terminal state s, $V_o[s] = 0$ to simplify the computation, but in principle the value function need not be defined for terminal states, since the task o should never be invoked in those states.

As detailed in Section 5.2, a key insight of MAXQ is that V_o can be interpreted as the reward model for the option $o = \langle I^o, \pi_o, \beta^o \rangle$ that, when initiated in a state $s \in I^o$, simply selects actions according to π_o until reaching a state s' and terminating with probability $\beta^o[s', s']$. Suppose that $o \in A^p$ for some parent task p. Then we can capture the insight of MAXQ as $R_p[so] = V_o[s]$. In other words, we can use V_o to construct part of the reward vector for any MDP learning task p that includes o as an executable action.

The same recursive approach can also apply to the transition function. Just as the abstract reward function for an option specifies the expected (discounted) sum of one-step rewards earned before reaching a terminal state, the abstract transition function for an option should specify the expected (discounted) probability of terminating in each terminal state. To this end, the

$|S| \times |S|$ state outcome matrix Ω_o for an option o can be computed with the following Bellman-like equation:

$$\Omega_o = \pi^o \left(\beta^o + \gamma(I - \beta^o)P_o\Omega_o\right). \tag{6.20}$$

Note intuitively that each column of Ω_o can be interpreted as a value function for a task which gives a reward of 1 upon terminating in the state corresponding to that column. In this context, each column of β^o can be seen as the "goal reward" for reaching the corresponding terminal state. As a result, Ω_o can be computed using standard MDP planning algorithms. So if $o \in A^p$ for some parent task p, then $P_p[so, s'] = \Omega_o[s, s']$.[2]

The hierarchical decomposition thus specifies how to construct the reward and transition matrices for a task p recursively given the value functions and state outcome matrices of the options $o \in A^p$. To complete this recursive specification, we need only give the base case. For a primitive action a, the value function V_a and state outcome matrix Ω_a correspond exactly to the reward and transition models for that action.

The R-MAXQ algorithm described in Section 5.2.1 employs this recursive model-building approach to extend efficient exploration to hierarchical RL. Of course, this algorithm is not simply an instance of GeneratedModel, since that algorithm interprets each action as a primitive action, which can

[2]Remeber that P^p is a multi-time model (Sutton et al., 1999), so its rows may not sum to 1, reflecting the effect of the discount factor over time. This representation thus folds the duration of actions (typically represented explicitly in the standard SMDP formalism) into the discounted transition probabilities.

be executed directly in the environment. Nevertheless, GeneratedModel can be extended to hierarchical RL in a conceptually clean way. Consider the following template:

1. Recursively estimate a model of each task from data.
2. Compute the optimal value function by planning with the task models.
3. Choose a task according to the policy implied by the value function.
4. Recursively execute the task.

This template deviates from the one given at the beginning of Section 6.1 in two respects. First, it replaces environmental actions with tasks that must be provided in a given task hierarchy. Second, it adds a fourth step, which executes the learned policy recursively.

To clarify the recursive nature of this template, Algorithms 6.3-6.5 specify the recursive algorithm in detail as three mutually recursive functions. This presentation does not adopt the Choose/Update formalism, which would obscure the recursion, but we will give that implementation presently. The algorithm is invoked by calling Execute on the root of the task hierarchy.

The Execute procedure described in Algorithm 6.3 executes the given task or action c using the data available in the agent's state σ. For a primitive action $a = c$, it simply executes a in the environment and updates the raw data for a. For a task $o = c$, it recursively computes the option policy π_o and then chooses a subtask to execute, until reaching a terminal state.

Algorithm 6.3 GENERATEDHIERARCHICALMODELEXECUTE(σ, c)

```
if $c$ is a (primitive) action then
  $s \leftarrow$ current state
  $a \leftarrow c$
  Execute action $a$ in the environment
  $r \leftarrow$ reward
  $s' \leftarrow$ successor state
  $D_R^\sigma[sa] \leftarrow D_R^\sigma[sa] + r$
  $D_P^\sigma[sa, s'] \leftarrow D_P^\sigma[sa, s'] + 1$
else {$c$ is a (composite) task/option}
  $o \leftarrow c$
  repeat
    $\sigma \leftarrow$ PLAN($\sigma, o$)                                  {Sets $\pi_o^\sigma$}
    $c \leftarrow$ choose child action/option with probability $\pi_o^\sigma[s, sc]$
    $\sigma \leftarrow$ EXECUTE($\sigma, c$)
    $s \leftarrow$ current state
  until termination with probability $\beta^o[s, s]$
end if
return $\sigma$
```

The Plan function computes the option policy π_o for a given task o using the available data in the agent state σ. It first constructs a model of each subtask $c \in A^o$, then it invokes planning on the resulting MDP, modified to respect the task's termination function β^o and goal rewards G^o, using the special MAXQ model operator:

$$\begin{aligned}
\text{MAXQ}^o_{\tilde{V}}(V) &= \beta^o G^o + (I - \beta^o)V && (6.21)\\
\text{MAXQ}^o_{V}(V) &= (I - \beta^o)V && (6.22)\\
\text{MAXQ}^o_{\Omega}(\Omega) &= (I - \beta^o)\Omega. && (6.23)
\end{aligned}$$

Note that $\text{MAXQ}_{\tilde{V}}$ and MAXQ_{Ω} are used by Plan (Algorithm 6.4), whereas Model uses MAXQ_V and MAXQ_{Ω}.

Algorithm 6.4 Plan(σ, o)

```
for all children $c \in A^o$ do
  $\sigma \leftarrow$ Model($\sigma, c$)                {Sets $V^\sigma_c$, $\Omega^\sigma_c$}
end for
$R \leftarrow [sc] \mapsto \text{MAXQ}^o_{\tilde{V}}(V^\sigma_c)[s]$
$P \leftarrow [sc, s'] \mapsto \text{MAXQ}^o_{\Omega}(\Omega^\sigma_c)[s, s']$
$\tilde{V}^\sigma_o \leftarrow$ ValueIteration($R, P, \epsilon, \tilde{V}^\sigma_o$)
$\tilde{Q} \leftarrow R + \gamma P V^\sigma_o$
$\pi^\sigma_o \leftarrow$ PolicyImprovement($Q$)
return $\sigma$
```

The Model function (Algorithm 6.5) produces the value and state outcome models V_c and Ω_c for any action or task c. For a primitive action a, it applies that action's model generator M^a. (Recall that the model generator may incorporate any number of model operators that modify the mapping from data to model.) For a task o, the Model function first invokes Plan to obtain the option policy π_o. It then performs policy evaluation to compute the option models defined in Equations (6.18) and (6.20), which in this context only requires solving a system of linear equations. Note that the MAXQ model operator will set $V_o^\sigma[s] = 0$ for every terminal state s, to simplify the computation. MODEL sets the value of terminal states to a minimal value $V^{\min}$ to discourage the option's parent p from choosing to execute o in terminal states, removing the need to constrain the parent's option policy explicitly.

Algorithm 6.5 MODEL(σ, c)

```
if $c$ is a (primitive) action then
  $a \leftarrow c$
  $V_a^\sigma \leftarrow M_V^a(D^\sigma)$                    {Run value model generator for $a$}
  $\Omega_a^\sigma \leftarrow M_\Omega^a(D^\sigma)$                  {Run outcome model generator for $a$}
else {$c$ is a (composite) task/option}
  $o \leftarrow c$
  $\sigma \leftarrow$ PLAN($\sigma, o$)                                    {Sets $\pi_o^\sigma$}
  $R \leftarrow [sc] \mapsto \text{MAXQ}_V^o(V_c^\sigma)[s]$
  $P \leftarrow [sc, s'] \mapsto \text{MAXQ}_\Omega^o(\Omega_c^\sigma)[s, s']$
  $V_o^\sigma \leftarrow$ solution to $V_o^\sigma = \pi_o^\sigma (R + \gamma P V_o^\sigma)$
  $V_o^\sigma \leftarrow \beta^o V^{\min} + (I - \beta^o) V_o^\sigma$       {Terminal states have minimum value}
  $\Omega_o^\sigma \leftarrow$ solution to $\Omega_o^\sigma = \pi_o^\sigma (\beta^o + \gamma P \Omega_o^\sigma)$
  $\Omega_o^\sigma \leftarrow (I - \beta^o) \Omega_o^\sigma$                {Terminal states have no successors}
end if
return $\sigma$
```

Since this algorithm generalizes GENERATEDMODEL to hierarchical RL, we call it GENERATEDHIERARCHICALMODEL. It can be expressed in the CHOOSE/UPDATE framework, by explicitly representing the stack of executing tasks. Algorithm 6.6 shows the CHOOSE function; GENERATEDHIERARCHICALMODEL uses the same UPDATE function as GENERATEDMODEL.

Algorithm 6.6 GeneratedHierarchicalModelChoose(s, σ)

```
                                        {Pop terminated tasks from stack.}
  $o \leftarrow$ Top($K^\sigma$)
  while task $o$ terminated with probability $\beta^o[s, s]$ do
    $K^\sigma \leftarrow$ Pop($K^\sigma$)
    $o \leftarrow$ Top($K^\sigma$)
  end while
                                                          {Choose subtasks}
  $c \leftarrow$ Top($K^\sigma$)
  Pop($K^\sigma$)                         {Simplifies definition of loop body.}
  repeat
    $o \leftarrow c$
    Push($K^\sigma, o$)
    $\sigma \leftarrow$ Plan($\sigma, o$)                                    {Sets $\pi^\sigma_o$}
    $c \leftarrow$ chosen with probability $\pi^\sigma_o[s, sc]$
  until $c$ is a primitive action
  $a \leftarrow c$
  return $(a, \sigma)$
```

Section 6.3 describes a particular instance of GeneratedHierarchicalModel, but first the next section discusses desirable properties of model operators, including the maxq operator intrinsic to GeneratedHierarchicalModel.

6.2 Model Operators and Exploration

GeneralizedHierarchicalModel is a single RL algorithm that has incredibly wide-ranging parameters: a task hierarchy that decomposes the problem into subtasks and a set of model generators, one for each primitive action. This section discusses the properties of hierarchies and model generators that lead to effective learning.

Learning hierarchies automatically from data remains a challenging open problem. Chapter 4 begins to address this problem by investigating the question of how hierarchy can benefit learning, but hierarchical RL algorithms by and large still require human users to construct domain-specific task hierarchies.

Model generators must also be somewhat domain-specific, but this chapter defines several model operators, based on existing effective RL algorithms, which form useful building blocks. Section 6.2.1 analyzes these model operators and how they guide learning behavior by effectively reducing the size of the MDP that must be explored. Section 6.2.2 demonstrates the synergies between this approach to generating models and key existing techniques for reducing the effective size of MDPs using factored state representations and state abstraction. Finally, Section 6.2.3 ties the discussion of effective model generators back to the question of effective task hierarchies.

6.2.1 Models and Completion Sets

Section 6.1 defined a model operator only as a function that maps models to other models, but all the model operators defined so far fall into one of two clear families of such functions. The first kind of model operator performs *model splicing*, in which certain states of the given model (rows of the reward vector and transition matrix) are replaced with a fixed alternative. Model splicing operators "multiplex" between two models, using the alternative model for states in a certain set, represented as a binary diagonal matrix. For example, the R-MAX operator splices a reward vector V with an upper bound $V^{\max}$ of the value function, which is used for unknown states: R-MAX$_V(V) = UV^{\max} + (I - U)V$. Similarly, the MAXQ operator splices a reward vector V with either a goal reward function G or with the constant zero function: MAXQ$_{\tilde{V}}(V) = \beta G + (I - \beta)V$ and MAXQ$_V(V) = (I - \beta)V$. Both R-MAX and MAXQ splice the transition function Ω with the transition function that always transitions to the special, implicit terminal state (and therefore has zero probability of transitioning to any explicit state). Both operators bias the agent towards certain states, either states where the agent wants more data or terminal states for the current subtask.

The second kind of model operator performs *transition composition*, which multiplies the transition matrix Ω with an averager, as defined in Section 3.1. However, in this context, the averager can more intuitively be thought of as just another transition matrix: if Φ is an averager, then Φ is a $|S| \times |S|$ matrix in which $\Phi[s, x] \geq 0$ for all s, x and $\sum_{x \in S} \Phi[s, x] \leq 1$ for all s. Note that just as a discounted transition matrix Ω allows for transitions to a special, implicit terminal state with probability $1 - \sum_x \Omega[s, x]$, an averager Φ may include this special, implicit state in its average for s.

For example, consider the model operator KBRL, which transforms a reward vector R into ΨR and a transition function Ω into $\Psi\Omega$. As used in KBRL and the Fitted R-MAX algorithm, R and Ω correspond to the maximum likelihood estimates for some primitive action. In environments with large or infinite state spaces, this model will be essentially undefined for most states and for those states where it is defined, the estimate will be based on a sample size of 1. The averager Ψ determines the contribution $\Psi[s, x]$ of the maximum likelihood action model at state x for approximating the action model at state s. The composed transition matrix ΨR essentially transitions a given state s first to a state x where the maximum likelihood model is defined, and then it transitions from x to a successor state s' according to that model.

As a more complete example, consider the Fitted R-MAX algorithm originally defined in Section 3.2.2. As shown in Section 6.1.1, it is simply an instance of GENERATEDMODEL that sets each action's model generator to $M^a = \text{FVI} \circ \text{R-MAX}^a \circ \text{KBRL}^a \circ \text{MLE}^a$. In particular, it replaces the maximum likelihood transition function Ω_a with $(I - U_a)\Psi_a\Omega_a\Phi$. When planning from a given state s, the algorithm assumes that each action produces a sequence of transitions:

1. Transitions to the implicit terminal state if s is unknown (R-MAX)
2. Transitions to a state where data is available (KBRL)
3. Transitions according to the maximum likelihood data for that state (MLE)
4. Transitions to a state in the finite sample X (FVI)

An important advantage of this framework is that each transformation to the transition matrix Ω has a clear intuitive purpose while also preserving the crucial property that Ω is a (possibly discounted) transition matrix.

This property allows the final model to be given to an off-the-shelf planning algorithm. In principle, the GeneratedModel and GeneratedHierarchicalModel agents may take advantage of any advances in MDP planning, independent of the agent learning behaviors implemented via the model generators.

Each model operator defined above serves a distinct purpose, but we've already seen that r-max and maxq share a common technique that involves fixing the value of certain state-actions, to bias the agent's policy. Another important mechanism for transforming models is exemplified by the fvi operator, which right-multiplies the transition matrix Ω with an averager Φ that forces the action to transition into a given state set $X \subset S$. This simple modification underlies the Fitted Value Iteration algorithm (Gordon, 1995), which permits approximate planning in very large or infinite state spaces. By assuming that every action transitions into X, the algorithm can effectively plan in the reduced state space $X \cup \{s\}$, since only states in X are reachable from s given the modified model.

Given a set of actions (typically the available subtasks A^o for a task o or simply the environmental actions A for a flat MDP), we define the *completion set* of a state s to be the set of all states reachable from s. To be precise a state s' is reachable from s if there exists a sequence of one or more actions that, when executed in sequence (effectively composing their transition matrices), give a nonzero probability of transitioning to s'.

The fvi model operator forces the completion set for every state to be a subset of a fixed state sample X. In the context of the original Fitted Value Iteration algorithm, this approximation (ostensibly of the value function, approximating V with ΦV), offered a primarily computational benefit. In particular, since in effect no action transitioned to a state outside of X, it suffices to define, represent, and store the computed value function V only on X. Meanwhile, Q remains defined over the entire original state space, allowing the approximated optimal action to be computed at any state. Fortunately, an agent need only compute Q outside of X for its current state.

The idea of enforcing small completion sets play a crucial role in hierarchical RL, as well. Consider the Model subroutine of GeneratedHierarchicalModel, given in Algorithm 6.5. This function can be construed as another kind of model generator, in that it produces an action model given a set of data, although it is implicitly parameterized by a given task hierarchy. It computes V_o and Ω_o for a task o using Equations (6.18) and (6.20). It is straightforward to show that the resulting transition function Ω_o only transitions to terminal states s, for which $\beta^o[s, s] > 0$. It follows that a task o that defines few terminal states helps reduce the completion sets of any tasks that include o as a subtask. Such parent tasks therefore reap a similar benefit to computational efficiency as fitted value iteration.

The tendency for tasks to funnel the agent into terminal states also has important consequences for sample complexity, as discussed in more detail in Section 5.3.1 in the context of the R-maxq algorithm. The task hierarchy described there for the Taxi domain only permits the `putdown` primitive action to execute in the completion set of the `ROOT` task, removing the need to explore those actions in most of the state space. That hierarchy also only permits the PICKUP primitive action in initial state or the completion set of the `ROOT` task, effectively postponing a comparable amount of exploration until it becomes necessary. In both cases, the agent avoids unnecessary exploration because it confines its planning and therefore its optimistic exploration to the completion set of the current state.

6.2.2 Abstraction and Factored MDPs

Reducing the effective size of the state space has been a longstanding fundamental principle underlying the development of planning and RL algorithms. In this respect, the ability of certain model operators to enforce small completion sets is nothing new. State abstraction and state factorization in particular have long histories in the RL community.

The formal presentation of GENERATEDHIERARCHICALMODEL to this point has made no assumptions about the structure of the state space. It has focused on temporal abstraction, not state abstraction. This section demonstrates how the algorithm can benefit from state abstractions made possible by factored state spaces. Far from being redundant with the state space reductions discussed in the previous section, the benefits of state abstraction will be shown to synergize very effectively with those of temporal abstraction.

Since the RL community largely grew out of the popularity of model-free algorithms such as Q-learning, state abstraction has mostly been implemented with respect to the value function, which serves as the primary data structure and representation of knowledge in such algorithms. This approach stores learned values as a function of some abstract state space, and it assumes the value of a concrete state s is equal to the value of the abstract state $\chi(s)$, where χ is an abstraction function mapping concrete states to abstract states. Typically, the state s is a vector of feature values, and the abstraction function χ projects states into a subspace that ignores irrelevant features.

This approach, which groups together states that are assumed to have the same value, tends to limit the usefulness of state abstraction in practice. One difficulty is that the value of a state s depends not just on intrinsic properties of that state but also on the value of every state reachable from s. This restriction limits the number of states that can be safely abstracted together. Furthermore, the sufficient and necessary conditions for abstracting together two states are not clear, in part due to these relationships among values. The most widely adopted approaches rely on fairly strong sufficient conditions that imply prior knowledge about the underlying MDP (Li et al., 2006).

The most popular of these conditions for state abstraction, stochastic bisimulation, calls an abstraction function valid if, for any two states that it abstracts together, those two states have the same rewards and the same distribution over abstract successor states for every action. In essence, it must be possible to define the model accurately using only the abstract state space. As suggested above, this condition is quite strong.

One way to weaken this condition is to represent the model as a dynamic Bayes network (DBN), which assumes the state space $S = S_1 \times S_2 \times \cdots S_k$ is factored. Then for a given action, the value s_i^{t+1} of state variable S_i at time $t+1$ depends on the values at time t of some fixed subset of the state variables.[3] Each action allows for a different DBN describing which state variables at time t are relevant (Markov) to which state variables at time $t+1$.

This structured representation of the model opens up the possibility of a structured representation of the value function (Boutilier et al., 1995), such as a decision tree. In the general case, the tree must still incorporate every state feature, but the hope is that in many cases only a few features suffice to determine the value. In practice, this value function representation does not scale, and approximate representations must be used (Koller & Parr, 2000).

Facilitating state abstraction is a primary motivation of MAXQ and its decomposition of the value function (Dietterich, 2000b). By computing the value of any given state as the sum of task-specific value functions, MAXQ permits "dynamic" state abstraction that employs different state representations in different contexts. However, the MAXQ Q-learning algorithm still relies on relatively inefficient stochastic approximation of the value function: instead of constructing a model of each task, it modifies each task's value function at most once per primitive action executed.

The GENERATEDHIERARCHICALMODEL algorithm combines the dy-

[3]Note that for convenience we identify a state variable with the set of values it may take.

namic state abstraction of MAXQ with the efficient exploration of model-based algorithms such as R-MAX. Essentially, each task can employ its own abstraction function. If all we care about is sample complexity, data efficiency, and efficient exploration, no modification to the given algorithm is necessary! It suffices to define the appropriate model generator for each primitive action.

Suppose the state variables can be partitioned into two sets $\mathcal{X}$ and $\mathcal{Y}$, so that $S = S_{\mathcal{X}} \times S_{\mathcal{Y}}$, where $S_{\mathcal{X}} = \prod_{X \in \mathcal{X}}$ is the state space projected onto the state variables $\mathcal{X}$. Then a transition function Ω *ignores* $\mathcal{Y}$ if both of the following are true for all $x, x' \in S_{\mathcal{X}}$ and $y, y' \in S_{\mathcal{Y}}$:

1. $\Omega[xy, x'y] = \Omega[xy', x'y']$
2. $\Omega[xy, x'y'] > 0 \Rightarrow y = y'$.

The first condition states that Ω modifies the values of the state variables in $\mathcal{X}$ in a way that does not depend on the values of the state variables in $\mathcal{Y}$. The second condition states that Ω does not modify the values of the state variables in $\mathcal{Y}$. Similarly, a reward vector V ignores $\mathcal{Y}$ if $V[xy] = V[xy']$ for all $x \in S_{\mathcal{X}}$ and $y, y' \in S_{\mathcal{Y}}$.

This notion of ignored state variables bears some resemblance to the DBN formalism in that it provides for action-specific state abstraction that depends on a factorization of the state space. It cannot describe conditional independence relationships among individual state variables. Instead it partitions the state variables into a relevant set $\mathcal{X}$ and an ignored set $\mathcal{Y}$, but this coarser degree of specification suffices to enable efficient learning, as illustrated later in this chapter.

State variable ignorance is more flexible than stochastic bisimulation in some ways, since it permits a different abstraction for each action. However, it does make stronger assumptions about the behavior of each action within each abstract state. In particular, stochastic bisimulation would allow each action to modify the ignored state features $\mathcal{Y}$ arbitrarily.

This latter point is also what sets state variable ignorance apart from the primary conditions for abstraction that Dietterich specified for the original MAXQ decomposition. Since he focused on computing only values hierarchically, not models, he could afford to make weaker assumptions. However, the domains he investigated in his experiments also satisfied the stronger condition of state variable ignorance.

A model generator that ignores a subset $\mathcal{Y}$ of the state variables can generalize data fully among states that differ only on the features $\mathcal{Y}$. For example, given an action model that ignores $\mathcal{Y}$, the R-MAX algorithm only requires $m|S_{\mathcal{X}}|$ instances of that action, instead of $m|S|$ instances. As discussed in Chapter 5, the R-MAXQ algorithm extends the optimistic exploration of R-MAX to hierarchical RL in such a way that exploration remains driven purely by uncertainty in the primitive action models, propagated up the hierarchy. Since (implicitly) exploratory actions at upper levels in the hierarchy only occur due to a lack of sufficient data at the lower levels, the benefits of state abstraction at the lower levels also propagate naturally and automatically up the hierarchy.

Furthermore, it can be shown that if $\mathcal{Y}$ is ignored by every child task $c \in A^o$ for some task o, then for any option policy π^o, the task value function V^o also ignores $\mathcal{Y}$. Intuitively, $\mathcal{Y}$ cannot affect the value function if it affects neither the reward vector nor the transition matrix that determine the value function. More formally, a straightforward proof by induction shows that if two states differ only on $\mathcal{Y}$, then for a fixed policy, the expected reward is the same at all future time steps. Since the computation of the task termination function Ω^o is equivalent to the computation of a value function for each possible terminal state, Ω^o also ignores $\mathcal{Y}$.

The action model for task o that GeneratedHierarchicalModel reports to any parent tasks thus ignores $\mathcal{Y}$ if all the child tasks ignore $\mathcal{Y}$. Although this fact does not directly inform the sample complexity of the algorithm, it affects the algorithm's computational complexity. In particular, regardless of the actual size of the state space, the Plan procedure (Algorithm 6.4) need only represent the abstract state space $S_{\mathcal{X}}$. Note that since a state xy' is not reachable from xy if $y \neq y'$ and all the available actions ignore $\mathcal{Y}$, state variable ignorance effectively bounds the size of the completion set for a task. Similarly, the computed action model for option o can by represented in the abstract state space. Chapter 7 discusses some of the implementation details of representing and manipulating such abstract models.

6.2.3 State Abstraction and Temporal Abstraction

The hierarchical model-based approach to state abstraction described in the previous section seems more constrained than the model-free state abstraction Dietterich employed in the original MAXQ algorithm. Using his five conditions for state abstraction, Dietterich was able to show that every component value function in his decomposition used a small number of state variables (Dietterich, 2000a). In contrast, a task o can only ignore state variables $\mathcal{Y}$ if all of its child tasks also ignore $\mathcal{Y}$. The state variables $\mathcal{X}$ necessary to represent the value function V^o is the union of all the state variables relevant to any primitive action that can possibly be executed during the execution of o. It follows that the root task of the hierarchy will generally include every state variable.

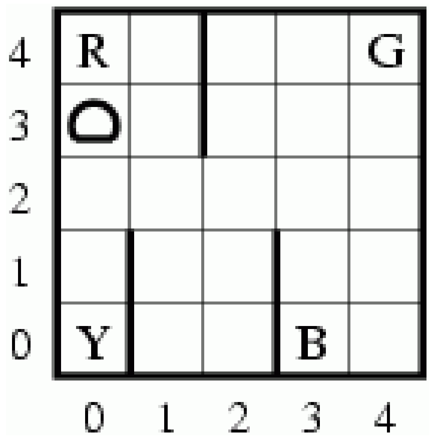
Fortunately, this fact does not imply that computation for the root task must use the full concrete state space S. Although the computed action model for a given task will tend to use more state variables than its subtasks, the task's termination function typically tightly constrains the set of states reachable using that task. As you go higher up a task hierarchy, the number of ignored state variables decreases, but the number of reachable states decreases. The complexity of planning for each task can be quantified precisely using the maximum size of the completion set for that task, as defined in Section 6.2.1. For primitive action models, small completion sets arise from state abstraction: ignoring most of the state variables. At upper levels of the task hierarchy, small completion sets arise from temporal abstraction: constraining planning to the terminal states of the subtasks.

Consider the Taxi domain defined in Section 5.1 and shown again in Figure 6.1a. Recall that it contained four state variables: `x`, `y`, `passenger`, and `destination`. Each coordinate has five values, the `destination` takes one of four landmark values, and `passenger` indicates that the passenger is at one of the four landmarks or in the taxi, for a total of $5 \times 5 \times 4 \times 5 = 500$ states, although 100 of these are terminal, with `passenger` = `destination`, leaving 400 non-terminal states. The taxi agent must deliver the passenger to the destination landmark using six primitive actions, which can be arranged into the task hierarchy reprinted in Figure 6.1b.

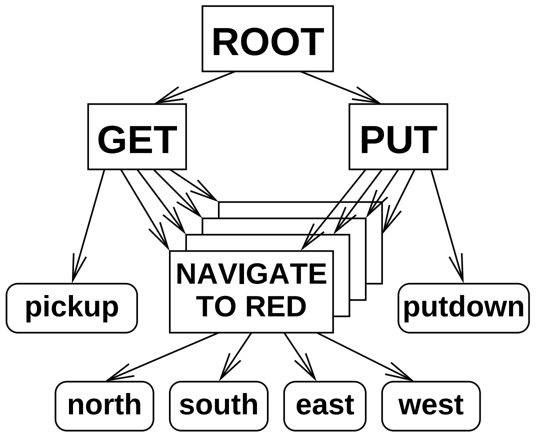
Table 6.1 describes the sizes of the state spaces for each task in the hierarchy. The column "Totel model size" describes the number of states where the action model for each task is defined. Primitive actions must be modeled at every state that is non-terminal in the environment, while tasks must be modeled at every state that is non-terminal in either the environment or

Task	Total model size		Completions		Max completion set
	Concrete	Abstract	Concrete	Abstract	
`putdown`	400	400	—	—	—
`pickup`	400	100	—	—	—
`n/s/e/w`	400	25	—	—	—
`NAVIGATE`	384	24	384	24	24
`GET`	300	100	80	20	4
`PUT`	100	100	16	16	4
`ROOT`	400	400	12	12	1

Table 6.1: State representations in the Taxi task hierarchy



(a)



(b)

Figure 6.1: (a) Taxi domain, and (b) a task hierarchy for Taxi

that task. State abstraction forces the model at two states to be essentially equivalent, allowing a more compact representation of both the model and the value function. For tasks, the column "Completions" describes the number of reachable non-terminal states during the execution of that task. The GeneratedHierarchicalModel only needs to compute the value function V_o at these states. Finally, the "Max completion set size" column describes the maximum number of reachable non-terminal states for any given initial state in which the task might be executed. This number bounds the number of states the planner must consider during any one execution of the task.

For example, each `NAVIGATE` task (one for each of the four landmarks) benefits directly from state abstraction. The only primitive actions reachable from `NAVIGATE` all ignore the `passenger` and `destination` state variables, allowing the computation of the policy $\pi_{\texttt{NAVIGATE}}$ to consider only the 24 abstract states corresponding to the 24 coordinates outside of the task's goal coordinates. Similarly, the action model $V_{\texttt{NAVIGATE}}$ and $\Omega_{\texttt{NAVIGATE}}$ depends on only those 24 abstract states.

In contrast, the ROOT task derives no direct benefit from state abstraction. In particular, the value function for the overall ROOT task depends on all four state variables. However, if you only consider the successor states that result after executing one of its child tasks, GET and PUT, the ROOT task only considers 12 states! After executing GET, the taxi must be at one of the four landmark locations, with the passenger in the taxi, and with the destination at one of the other three landmark locations. After executing PUT, the ROOT task must itself terminate. At this level of the hierarchy, the complexity of planning and the size of the action model is constrained by temporal abstraction, not state abstraction.

GeneratedHierarhicalModel can thus use a task hierarchy to decompose a problem with a large state space into a collection of subtasks, each with a much smaller state space. These state-space reductions arise from a blend of state abstraction and temporal abstraction, with lower levels of the hierarchy exploiting state abstraction, upper levels exploiting temporal abstraction, and intermediate levels exploiting both to a lesser degree. This hierarchical form of abstraction thus plays an essential role in facilitating efficient planning given a factored MDP model. As discussed in Section 5.3, it also plays an important role in constraining the exploration that determines how the agent learns the factored model of the primitive actions.

6.3 Fitted R-MAXQ

The R-maxq algorithm described in Section 5.2.1 already includes all the benefits described in the preceding section: it combines data-efficient model-based exploration with hierarchical decomposition. The GeneratedHierarchicalModel algorithm generalizes R-maxq, allowing us to combine it with other model operators. In particular, it is now straightforward to combine R-maxq with the Fitted R-max algorithm, which incorporates the function approimation of fitted value iteration and the model generalization techniques of kernel-based reinforcement learning (KBRL).

In keeping with the algorithms developed in Chapters 3 and 5, we refer to this novel instance of GeneratedHierarchicalModel as Fitted R-maxq. To define this algorithm, it suffices to specify the model generators provided to GeneratedHierarchicalModel to be the same provided to GeneratedModel to obtain Fitted R-max. For each primitive action a, the model generator $M^a = \text{FVI} \circ \text{R-MAX}^a \circ \text{KBRL}^a \circ \text{MLE}^a$. Note that these model generators define how Fitted R-maxq reasons about primitive actions in any environment. To obtain the benefits of hierarchical decomposition, the algorithm must also be given a domain-specific task hierarchy.

For concreteness, Algorithm 6.7 specifies Fitted R-maxq in a more traditional form, which eschews the Choose and Update formalism and instantiates our choice of model generator. The algorithm still requires a task hierarchy, so it takes as its sole input the root task. This presentation assumes a continuing task, but the modifications for episodic tasks are straightforward.

Algorithm 6.7 FITTED-R-MAXQ(`ROOT`)

```
$K \leftarrow$ [ROOT]                                   {Initialize task stack to ROOT}
loop
  for all tasks $o$ do {propagate models and values from the bottom up}
    $R_o \leftarrow [sc] \mapsto V_c[s]$
    $P_o \leftarrow [sc, s'] \mapsto \Omega_c[s, s']$
    $\pi_o \leftarrow$ optimize $\tilde{V}_o = \beta^o G^o + (I - \beta^o)\pi_o(R_o + \gamma P_o \tilde{V}_o)$
    $V_o \leftarrow$ solution to $V_o = (I - \beta^o)\pi_o\,(R_o + \gamma P_o V_o)$
    $\Omega_o \leftarrow$ solution to $\Omega_o = \pi_o\,(\beta^o + \gamma(I - \beta^o)P_o\Omega_o)$
  end for
  $s \leftarrow$ current state
  $c \leftarrow$ TOP($K^\sigma$)
  POP($K^\sigma$)                              {Simplifies definition of loop body.}
  repeat {execute task $c$ recursively}
    $o \leftarrow c$
    PUSH($K^\sigma, o$)
    $\sigma \leftarrow$ PLAN($\sigma, o$)                                  {Sets $\pi_o^\sigma$}
    $c \leftarrow$ chosen with probability $\pi_o^\sigma[s, sc]$
  until $c$ is a primitive action
  $a \leftarrow c$
  Execute action $a$
  $r \leftarrow$ one-step reward
  $s' \leftarrow$ successor state
  $N_a[s] \leftarrow N_a[s] + 1$                      {Update primitive action model}
  $D_a^R[s] \leftarrow D_a^R[s] + r$
  $D_a^P[s, s'] \leftarrow D_a^P[s, s'] + 1$
  $\hat{N}_a \leftarrow \Psi N_a$
  $V_a \leftarrow [s] \mapsto \begin{cases} \frac{(\Psi D_a^R)[s]}{\hat{N}_a[s]} & \text{if } \hat{N}_a[s] > m \\ V^{\max} & \text{otherwise} \end{cases}$
  $\Omega_a \leftarrow [s, s'] \mapsto \begin{cases} \frac{(\Psi D_a^P \Phi)[s,s']}{\hat{N}_a[s]} & \text{if } \hat{N}_a[s] > m \\ V^{\max} & \text{otherwise} \end{cases}$
  $o \leftarrow$ TOP($K^\sigma$)                      {Pop terminated tasks from stack.}
  while task $o$ terminated with probability $\beta^o[s, s]$ do
    $K^\sigma \leftarrow$ POP($K^\sigma$)
    $o \leftarrow$ TOP($K^\sigma$)
  end while
end loop
```

6.3.1 Experiments

In order to exercise the full capabilities of Fitted R-MAXQ, we introduce a new domain, Flag & Puddle World, which combines aspects of the Flag World domain (Dearden et al., 1999) and the RL benchmark environment Puddle World, which was previously described in Section 3.3.2.2. Puddle World already has a continuous state space, and no modifications are necessary to enable model-based reasoning (which is purely an algorithmic issue), but we introduce some of the structure from Flag World that encourages hierarchical reasoning. We intend our extensions to give the overall task more structure of the sort found in real-world tasks.

First, we recapitulate the original Puddle World environment, depicted in Figure 6.2. The agent must navigate the unit square to reach a goal state in the upper-right corner, which terminates each episode. Four primitive actions move the agent 0.05 in each of the four cardinal directions, with some Gaussian noise ($\sigma = 0.01$) added to each of the two state variables after every action. Each action incurs a -1 penalty until reaching the goal, but each time step spent in a puddle incurs an additional penalty between 0 and -40, depending on the proximity to the middle of the puddle.

We modify this environment by removing the goal state in the corner and instead giving the agent a set of four different flags it must collect in each episode, as shown in Figure 6.3. Each flag can only be collected in the neighborhood (within distance 0.1) of a specific spot in the unit square, which is initially unknown. Four binary state variables track whether each flag

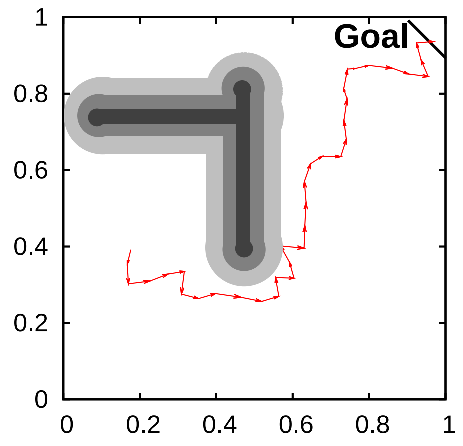


Figure 6.2: A trajectory in the original Puddle World environment.

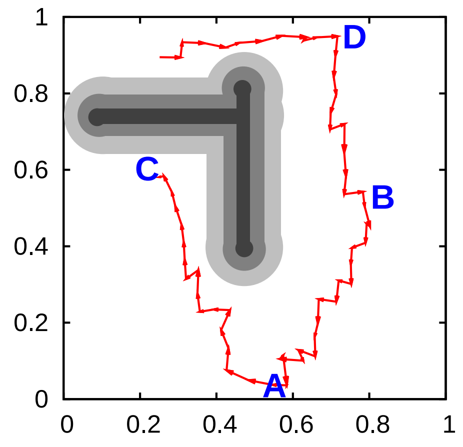


Figure 6.3: A trajectory in the Flag & Puddle World environment. The agent collects flags D, B, A, then C, but any order is permissible.

has been collected, and a distinct primitive action allows the agent to search the current location for a particular flag and collect it if present. In each episode, the agent begins in a random location, so over time it must learn the locations of the flags, the locations of the costly puddles, and how to harvest all four flags as cheaply as possible. This environment has six state variables, two continuous and four binary, and eight primitive actions: four movement actions and the four collection actions.

We ran each algorithm tested for 50 independent trials, using for each algorithm the same set of 50 configurations of flag locations, generated uniformly at random but with no location inside of a puddle. For each configuration, we generated a fixed sequence of 500 start states, again uniformly at random. Each trial lasted for 500 episodes, and we limited each episode to 1000 time steps.

6.3.2 Algorithm Configurations

We compare several different instantiations of our compositional framework for model-based RL. In this section, we describe the precise configuration of each model generator and model operator used. We use value iteration with prioritized sweeping (Moore & Atkeson, 1993) both to compute the optimal policy π_o given a model (R_o, P_o) and also to evaluate π^o to obtain the abstract model (V_o, Ω_o). We used a discount factor of $\gamma = 1$, since the task is episodic.

The maximum-likelihood model generator MLE has no parameters, but to apply finite algorithms in PuddleWorld we used a discretization of the unit square into a 16 × 16 grid. This discretization seems quite coarse, but finer grids only lead to more excessive exploration without a concomitant increase in policy quality.

For the instance-based model approximation of KBRL, we adopted a Gaussian kernel function:

$$K[s,x] = e^{-(d(s,x)/b)^2}, \tag{6.24}$$

where $d(s,x)$ is the Euclidean distance between s and x, and $b = \frac{1}{16}$ is a bandwidth parameter that controls the breadth of generalization, chosen using coarse optimization. To compute K efficiently, we stored the instances in a cover tree (Beygelzimer et al., 2006) and rounded down to zero any value of $K[s,x] < 0.01$. Finally, we adopt the "relative transition model" defined by Jong and Stone (2007a), which modifies Equation (6.13) by using the vector displacement that resulted from each state-action instead of the absolute successor state that resulted:

$$\textsc{kbrl}_P(\Omega) = [s,s'] \mapsto \frac{1}{\hat{N}_a[s]} \sum_{x \in S} \Psi[s,x] N_a[x] \sum_{x' \in S} D_a^P[x,x'] \delta(s', s + (x' - x)). \tag{6.25}$$

All of the algorithms we tested rely on the R-MAX approach to exploration. We set $V^{\max} = 0$, since all the immediate rewards in Puddle World are negative. When used with MLE, we defined $U_a = \{s \in S \mid N_a[s] < 2\}$. Since the stochasticity in Puddle World is relatively benign, gathering more data for each state-action didn't improve the final policy quality but resulted in much more expensive exploration. When used with KBRL, we defined

$U_a = \{s \in S \mid \hat{N}_a[s] < 1\}$. This low threshold seemed adequate since KBRL must typically generalize from several instances to reach a kernel weight of 1.

For FVI, we defined the averager Φ using linear interpolation over a uniformly spaced grid, with a resolution of $\frac{1}{16}$. This function approximation scheme therefore approximates the value of a point in the unit square (for a particular setting of the binary state variables) as an interpolation between the four surrounding points. Again, increasing the resolution did not improve the quality of the learned policy, but it did increase the computational burden of planning.

For the hierarchical algorithms, we defined a simple task hierarchy for the Flag & Puddle World that corresponds to the prior knowledge that the four flag-collection actions are independent of one another.[4] For each flag, we define a task o such that the child actions A^o include the four movement primitives and the action that collects that flag. The termination function β^o specifies termination at all states where the flag's boolean is set, and $G^o = 0$. The root of the hierarchy has these four tasks as children; it cannot execute any primitive actions directly.

Note that once the agent has learned enough of the dynamics of the Flag & Puddle World, the optimal high-level behavior depends on collecting the flags in the best order, a small instance of the Travelling Salesman Problem

[4]This hierarchy therefore imparts less domain knowledge than the hierarchy Dietterich provided for learning in the Taxi domain (Dietterich, 2000a), where the possible passenger coordinates were all known a priori.

(TSP). The value $V^o[s]$ of the subtask o for a certain flag gives the agent's "distance" to that flag in its current state s. The optimal low-level behavior depends on minimizing this cost by selecting movement actions appropriately to navigate to the flag while avoiding the puddles. Learning the optimal task values in the Flag & Puddle World therefore corresponds to recovering the cost matrix for the embedded TSP instance.

Finally, all the algorithms benefitted from state abstraction. The four primitive movement actions neither depend on nor affect the boolean state variables for the four flags. Similarly, each subtask that collects a given flag only depends on the agent's coordinates and only affects the boolean state variable corresponding to that flag.

6.3.3 Results

Figure 6.4 shows learning curves for four algorithms: R-MAX, R-MAXQ, Fitted R-MAX, and our combination of these algorithms, Fitted R-MAXQ. All four algorithms converge to statistically the same policy quality after only 25-30 episodes, but they incur very different exploration costs before getting there.

Figure 6.5 integrates under the curves in Figure 6.4 to show the total learning costs. Note that both figures only show the first several episodes, to focus on the period of learning when the algorithms' performance differs. Note that the benefit of adding both hierarchical decomposition and function approximation to R-MAX is greater than the sum of the benefits for adding

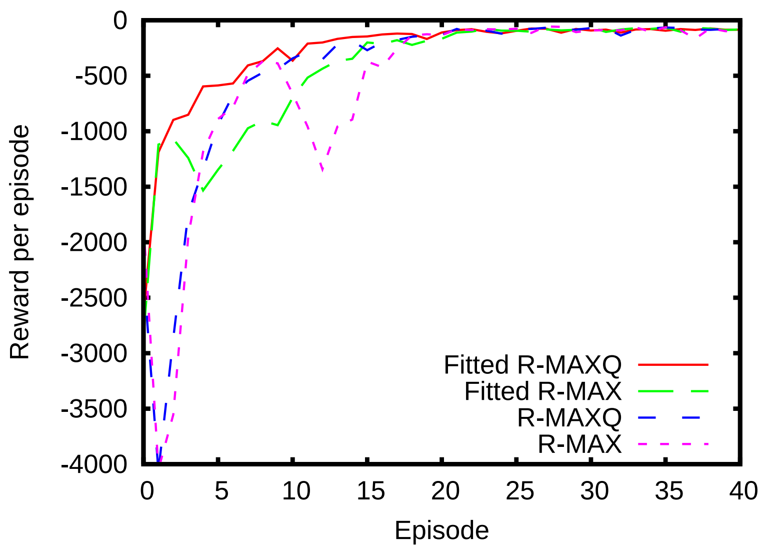


Figure 6.4: Reward per episode for variations of R-MAX with and without function approximation and hierarchical decomposition. After 40 episodes, the differences among the quality in learned policies is not statistically significant.

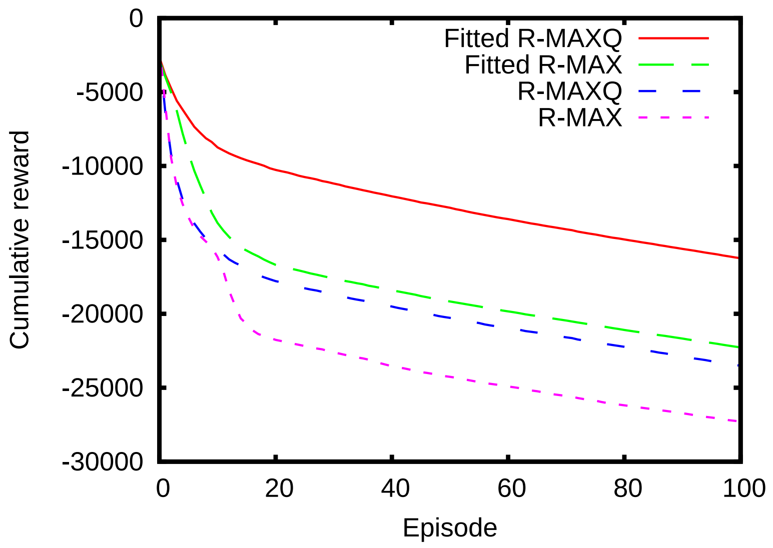


Figure 6.5: Cumulative reward for variations of R-MAX with and without function approximation and hierarchical decomposition. After 100 episodes, the differences among learning costs are all statistically significant ($p < 0.0001$).

each innovation by itself!

An inspection of the behavior of Fitted R-MAXQ reveals that it outperforms the other algorithms largely by avoiding excessive exploration in the puddles. Consider a state in the middle of a puddle that is in the set of unknown states U_a for some primitive action a. The R-MAX operator will assign this state the optimistic value $V^{\max}$, but this value does not guarantee that the agent will attempt to reach this state. If the predicted cost of completing the current task is smaller than the predicted cost of wading through the puddle to the unknown state, the agent will choose to ignore the unknown state and instead exploit a path through known states. In the non-hierarchical case, the current task is always to complete all the remaining activities, which may have a rather high cost. In the hierarchical case, the current task is to complete a particular one of the activities, which is more likely to have a lower cost than wading into the puddle. In a sense, the hierarchical decomposition limits the optimism applied to unknown states, which R-MAXQ models as terminating only the current task, not the entire episode. Meanwhile, discretization interferes with the accurate prediction of the costs of exploring versus exploiting. The coarse discretization we used is very effective in most of the state space, where the dynamics are the same, but not near the puddles, where the immediate reward varies quickly as a function of the coordinates.

6.4 Discussion

This chapter synthesized into a cohesive whole all of the ideas and algorithms explored in this thesis. For the first time, a single reinforcement learning algorithm combines model-based exploration, function approximation, and hierarchical decomposition. Each of these ideas confers a distinct benefit and helps compensate for the others' shortcomings. The optimistic exploration of R-MAX efficiently explores a given state space, but this notion of efficiency guards against the worst-case situation wherein each state may have arbitrarily different dynamics. The function approximation permits generalization across similar states, thereby putting even infinitely large problems within reach, but the tendency of value functions to incorporate long-term distinctions across all states and actions limits the accuracy of the approximation. Hierarchy can decompose large problems into smaller ones, both reducing the scope that approximations must cover and constraining exploration, but until now the hierarchical RL literature has focused almost exclusively on decomposing value functions, not action models.

These disparate algorithmic ideas can be unified by casting them all as transformations applied to a common space of models. A model-based RL algorithm can be seen as a simple combination of a planning algorithm with a model generator, which produces a model given data. Transformations applied to these generated models can dramatically alter the behavior of the resulting learning agent. This formalism is general enough to represent the optimistic exploration of R-MAX, the function approximation of fitted value iteration, the model approximation implicit in KBRL, and the goal-attainment behavior of typical options. Even hierarchy fits naturally into this framework. Although Section 6.1.2 defined a distinct algorithm GeneratedHierarchicalModel to handle the hierarchical case, a given task hierarchy can be seen itself as a model generator, producing a task model that depends on the models generated by the hierarchies root by its subtasks. This thesis defined this space of algorithms but has only begun to explore the possible model operators and their particular configurations, which give rise to concrete learning algorithms.

Already the model operators defined in this thesis give additional insights into the original algorithms that inspired them. For example, a consistent, universal notation highlights the deep similarities between the goal-oriented learning in maxq task hierarchies and the bias of R-max towards unknown states, and also between the "funneling" effect of subtasks and the class of function approximators known as averagers. This latter connection underscores the importance of the completion set size: the number of distinct states reachable from the agent's current state.

The principal branches of RL research that serve as the foundation for this thesis can all be better understood by examining their relationship to the completion set, which defines the search space for the planning that must occur at any one time step to behave near-optimally at that time step. The optimistic exploration of R-max provides a mechanism for evaluating a particular search space relatively efficiently but prioritizing exactness first. Function approximation provides a mechanism for generalizing the exact evaluations of a small set of states to a much larger set, albeit while introducing approximation error. Hierarchy provides a mechanism for decomposing a state space into smaller state spaces, which can each be evaluated and approximated independently. The temporal abstraction inherent in hierarchy also plays an important role in facilitating state abstraction, which can further reduce the size of the state space or reveal that the completion sets for two distinct states are essentially the same.

These insights may prove particularly valuable in the still developing field of hierarchy discovery. For example, they suggest that an effective task hierarchy minimizes the size of each task's completion set while permitting the root task to achieve near-optimal values. Individual tasks may provide very different roles in this sense. Tasks such as `NAVIGATE` in the Taxi domain provide an "abstract interface" to the set of primitive movement actions that preserves their shared state representation but yields a much smaller completion set. Such tasks allow other tasks, like `GET` in the Taxi domain, to incorporate such primitive actions with others with richer state representations, without drastically increasing the effective size of the state space.

Chapter 7

Software Architecture

Chapter 6 formally defined a novel algorithm that combines model-learning, function approximation, and hierarchical decomposition. This definition focused on the exploration behavior of the resulting learning agent by including the full equations and models used. The experiments throughout this thesis addressed the sample complexity of various learning algorithms: how much data do they require to converge to good policies, or how much reward will they earn per time step compared to one another? These algorithms culminate in Fitted R-MAXQ, which combines several algorithmic ideas by defining its policy at each time step as the recursive solution of a hierarchy of MDPs derived from data. How can an implementation of Fitted R-MAXQ avoid recomputing each value function from scratch at each time step, while always taking advantage of all available data? How can it approximate the models of the primitive actions from nearby instances both on-line and efficiently? How can this algorithm be applied to large or infinite problems when its definition relies on vectors and matrices indexed by the state space?

This chapter answers these questions and describes a practical implementation of Fitted R-MAXQ, written in C++ using RL-Glue 3.04 (Tanner &

White, 2009) and publicly available in the RL-Library[1] at `http://library.rl-community.org/wiki/Fitted_R-MAXQ`. Section 7.1 describes the custom state and action representations employed by Fitted R-MAXQ, including their representation as C++ objects.[2] Section 7.2 shows how Fitted R-MAXQ assembles these building blocks into a complete implementation. Section 7.3 gives concrete instructions for using Fitted R-MAXQ, and Section 7.4 concludes with some discussion.

7.1 State and Action Representations

This thesis has described crucial quantities such as the value function V and the transition function Ω as vectors and matrices. These representations clearly do not scale to infinite or even to very large state spaces. They also seem incompatible with the structured state spaces common to realistic environments, which comprise multiple state variables. The implementation of Fitted R-MAXQ addresses these issues by adopting an instance-based state representation and embracing an object-oriented representation of functions defined over the state space.

To represent any given environment state, Fitted R-MAXQ defines the type `StateVector` as an array of (double-precision) floating-point values.[3] Each element of a `StateVector` specifies the value of a particular state vari-

[1]`http://library.rl-community.org/`

[2]In this chapter, Fitted R-MAXQ refers to the specific implementation of the algorithm described in Section 6.3 and formally defined in Algorithm 6.7.

[3]The implementation uses the `vector` container in the C++ Standard Library.

able. This state representation allows for discrete state variables simply by ignoring the extra precision: the averagers that govern generalization among states can assign each state variable a different weight. A function defined over states is represented as a collection of objects, indexed by state instances, each of which represents the evaluation of the function at that state. Roughly speaking, these objects correspond to an entire row of a matrix or column vector. For example, one object might represent $V[s]$ for a particular state s. The entire function V is then represented by an object that can produce a child object $V[s]$ for any given state s.[4]

To reduce the number of function evaluation objects that it must construct, a function over states maintains a container of existing objects, indexed by state instances. When given a state equivalent to one of these instances, the function returns the existing function evaluation object. Fitted R-MAXQ therefore essentially flattens the nominal state space, comprising all possible values of type `StateVector`, into a literal state space comprising the addresses of every state instance in memory. To this end, Fitted R-MAXQ defines the type `StateVectorRef` as a smart pointer[5] to a `StateVector` object. Any collection of data indexed by state can then be represented using an associative container, such as the standard `map` or `unordered_map`,[6] where `StateVectorRef` is the key type.

[4]Section 7.1.1 will discuss why Fitted R-MAXQ represents $V[s]$ as an object, not simply a floating-point value.

[5]The implementation uses the `shared_ptr` class in the TR1 extensions, implemented in the Boost library.

[6]The `unordered_map` container appears in TR1 and is implemented in Boost.

Similarly, transition matrices can be represented with nested associative containers, both indexed by state instances. The outer container maps state instances to rows of the transition matrix, where each row is represented as a mapping from state instances to floating-point transition probabilities. By design Fitted R-MAXQ only needs to store a small number of entries in each row of a matrix, but matrix multiplication requires fast iteration through the row to compute the inner product of the row with a column vector. Each row of a matrix is thus stored as a `vector<pair<StateVectorRef, double> >`, treated as an associative array mapping `StateVectorRef` keys to floating-point values. This representation of a transition matrix is tantamount to an adjacency list representation of the state transition graph. It takes advantage of the fact that in most realistic problems, transition matrices tend to be very sparse. The matrix as a whole is then just another mapping from states to function evaluation objects, where in this case each object represents an entire row of the matrix.

Representing vectors and matrices as collections of objects indexed by state instances does not allow the agent to store the entire value function or model at once for a sizeable state space, but an essential trait of Fitted R-MAXQ is its use of approximation to limit the number of instances involved in the computation at any given time step: the number of instances in the completion set. The function evaluation at a particular state is constructed lazily when needed, and Fitted R-MAXQ uses reference counting to deallocate the state instance and function evaluations at that state as soon as they are no longer needed. Because this reference counting includes the state instances themselves, the literal state space is only as large as needed at any moment in time. The number of references to a particular function evaluation object is generally equal to the number of other function evaluation objects, known as observers, whose values depend on the first object.

7.1.1 Observer Pattern

The most common mathematical operation in Fitted R-MAXQ is computing a new column vector as the product of a matrix and an existing column vector. For example, an approximate value function $\hat{V}$ is the product $\hat{V} = \Phi V$ of a $|S| \times |S|$ averager Φ and a $|S|$-dimensional column vector V. Although some quantities in this thesis included the product of matrices, the result was always multiplied with a vector, so associativity allows the implementation to produce a vector as the outcome of each multiplication.[7] Since the result $\hat{V}$ is a vector, it is again represented as a collection of objects $\hat{V}[s]$ indexed by states s. Each object's job is to compute a particular entry (row) of the column vector, which is defined as the inner product of the appropriate row of the matrix and original vector. To this end, the object representing $\hat{V}[s]$ maintains references to $\Phi[s, \cdot]$ and to each $V[x]$ with $\Phi[s, x] > 0$.

If we only needed to compute $\hat{V}[s]$ once, it would not be necessary to maintain these references. The goal of the implementation is to propagate

[7]The sole exception is the computation of the abstract transition matrix Ω_o for a task given its policy π_o: solving Equation (6.20).

changes in the data efficiently to all of the quantities that must be computed. Each object therefore keeps track of those other objects whose value depends on it. For example, $\hat{V}[s]$ *observes* each $V[x]$ such that $\Phi[s, x] > 0$ (as well as observing $\Phi[s, \cdot]$). When any $V[x]$ changes its computed value, it sends a notification to any observers so that they can update their own values. Since these objects maintain reference counts, they can be safely deallocated (and no longer need to propagate their values) when they have no observers.

With this observer pattern, the computation of the entire hierarchy of value functions becomes an event-driven system. Any change to the data for a primitive action propagates to all the stored intermediary values for every quantity in the overall computation. The need to store each such intermediary value underscores the importance of using state and temporal abstraction to minimize the number of represented states at each task in the hierarchy!

7.1.2 Action Effects

State abstraction plays a crucial role in reducing the number of state objects that must be created. For functions over states such as V, the role of state abstraction is straightforward. Whenever the computation requires the object $V[s]$, it suffices to substitute $V[\chi(s)]$, where $\chi : S \to S$ is the abstraction function mapping each state into a subset of S. However, for a transition matrix Ω, the object representing the row $\Omega[\chi(s), \cdot]$ can only be substituted for $\Omega[s, \cdot]$ if the transition matrix will only be multiplied with vectors that adopt the same state abstraction. As discussed in Section 6.2.3, the level of state abstraction necessary decreases as you ascend the hierarchy, replacing the benefits of state abstraction with temporal abstraction. The interface between two tasks that adopt different levels of state abstraction poses a problem: how should the quantities communicated between the tasks be represented?

As a concrete example in the Taxi domain (Section 5.1), consider the `GET` task, which has `NAVIGATE(RED)` as a subtask. The state representation for the `GET` task includes the taxi's coordinates as well as the passenger's location, but the `NAVIGATE` subtasks only require the taxi's coordinates. Suppose s_1 and s_2 are two states that share the same values for the taxi's coordinates but different values for the passenger's location. The transition model for `NAVIGATE(RED)`, could compute $\Omega_{\texttt{NAVIGATE(RED)}}[s_1, \cdot]$ and $\Omega_{\texttt{NAVIGATE(RED)}}[s_2, \cdot]$ separately, but doing so would be inefficient, since s_1 and s_2 share the same abstract state in `NAVIGATE(RED)`. Conversely, if $\Omega_{\texttt{NAVIGATE(RED)}}[s_1, \cdot] = \Omega_{\texttt{NAVIGATE(RED)}}[s_2, \cdot]$, then `GET` would believe that s_1 and s_2 share the same distribution over successor states, even though these two states differ in their values for the passenger's location, a state variable that `NAVIGATE(RED)` does not affect.

For the predicted transition function Ω that must be communicated between tasks, the implementation of Fitted R-MAXQ adopts *relative transitions*. With this representation, each row $\Omega[s, \cdot]$ comprises a distribution over relative effects, instead of absolute successor states. The space of possible *effects* $E : S \to S$ is the set of functions that map states to successor states. Therefore, the agent plans with task o in state s by predicting the effect $e \in E$

with probability $\Omega_o[s, e]$, with the resulting $s' = e(s)$. The probability that o will transition s to s' is then $\sum_{e|e(s)=s'} \Omega_o[s, e]$.

This approach allows the predicted successor states to be relative to the current state, independent of the representation of the current state. Effects can represent the standard transition matrix by allowing only constant functions, in which each effect produces an absolute successor state, regardless of the current state. More generally, they permit the implementation to decouple the representation of the current state in Ω from the representation of the successor state. For example, every effect in the `NAVIGATE(RED)` task only modifies the taxi's coordinates in a given state. The transition model for `NAVIGATE(RED)` can therefore be safely used by a parent task with any state representation that includes the taxi's coordinates.

The effect representation also permits the vector transitions discussed in Section 3.2.2. In particular, when observing a transition from state s to s', instead of recording s' as the absolute successor of s, the agent records the vector effect $s' - s$. This approach corresponds to the inductive assumption that the action produced a relative change, not an absolute transition. In this way, vector effects can help generalize data from primitive actions to the models for primitive tasks.

7.2 Hierarchical Dynamic Programming

The main organization unit of Fitted R-MAXQ is the task. Every task, whether primitive or composite, conforms to a common base interface. Most importantly, each task is responsible for computing its *declarative behavior*: a model describing what the task will do if executed, comprising V_o and Ω_o. Each task also computes its *procedural behavior*, which specifies how the agent executes that task. For a primitive task, the procedural behavior is simply a primitive action that the agent executes for one time step. For a composite task, the procedural behavior is an option: the initiation set I^o in which the task may execute, the policy π_o that the agent should (recursively) follow when executing the task, and the termination function β^o that describes when to cease executing the task.

The bulk of each task is a pair of functions defined over state instances, one for the declarative behavior and one for the procedural behavior. For any state s, a task o must be able to produce a model object that defines the declarative behavior of the state-action so by implementing the `StateActionModel` interface. This interface defines methods for obtaining both the floating-point value $V_o[s]$ and the row vector $\Omega_o[s, \cdot]$, represented as an array of effects and their probabilities. (See Section 7.1.2 for a discussion of the representing of Ω as a $|S| \times |E|$ matrix that transitions state to relative effects.)

Each task o also produces a policy object that defines the procedural behavior of the state-action so by implementing the `StatePolicy` interface. This interface simply provides a reference to a subtask $c \in A^o$ that maximizes $\pi_o[s, sc]$. Both the `StateActionModel` and `StatePolicy` interfaces also support the observer pattern, so that other objects can elect to receive notifications whenever a particular object implementing these interfaces changes its value.

7.2.1 Primitive Tasks

Primitive tasks correspond to a fixed primitive action, part of the agent-environment interface, so their procedural behavior is trivial. Their initiation set includes every state, they terminate in every state, and the policy pi_o specifies the primitive action at every state. To compute the declarative behavior at each state, Fitted R-MAXQ implements the model generalization used in KBRL and Fitted R-MAX, described in Section 3.2.

The primitive task for an action $a \in A$ stores all the data the agent has collected for that action. It stores for each state instance s an object of type `PrimitiveStateActionData`, which contains the number of times $N_a[s]$ that action a was executed in state s, the cumulative reward $D_R[sa]$, and the effect counts $D_P[sa, \cdot]$. Given a new experience $\langle s, a, r, s' \rangle$, the primitive task increments $N_a[s]$, adds r to $D_R[sa]$, and increments $D_P[sa, e]$, where $e = x \mapsto x + (s' - s)$ is the effect that adds to any state the difference between s^prime and s. The data for a particular instance s allows the primitive task easily to compute the maximum likelihood model for the state-action sa. However, note that for most environments with very large state spaces, the agent may only visit each state once, so this maximum likelihood model is likely to be trivial.

A primitive task modifies the maximum likelihood model by applying a $|S| \times |S|$ model averager matrix Ψ, where Ψ normalizes a kernel function

K (Equation (6.14)). In Fitted R-MAXQ, this kernel function is a Gaussian probability density function:

$$K(s, x) = \exp\left(-\frac{d(s, x)^2}{\sigma^2}\right), \tag{7.1}$$

where d is a distance function for states and σ controls the breadth of generalization. The implementation uses a Euclidean distance function, where each state dimension is scaled to the interval $(0, 1)$, but it allows for state abstraction by specifying state variables that do not contribute to the distance. The generalization breadth σ remains a free parameter of the algorithm.

The exact computation of Ψ as described above would be very inefficient, since it would assign nonzero weight to the data at state x for any given state s. To reduce the number of predicted effects for each state s, Fitted R-MAXQ ignores the tails of the Gaussian by setting to 0 any value of the kernel function K or the averager Ψ below a certain threshold.[8] Bounding the minimum kernel value effectively limits the allowable distance between s and x. Bounding the minimum probability in the averager, by only including the instances with the highest kernel weights, effectively limits the approximation to the nearest neighbors of s among the instances.

Individual rows of the Ψ must be computed efficiently on demand, requiring the efficient lookup of instances near a given state s. Fitted R-MAXQ employs a custom implementation of the Cover Tree algorithm (Beygelzimer

[8]The experiments in this thesis used a minimum value of 0.01 for both the kernel function and the averager probabilities.

et al., 2006), which supports both nearest-neighbor lookups in $O(\log(n))$ time and incremental insertion (also in logarithmic time), which is essential for updating the approximation after each time step.

Given a state s, the primitive task constructs a `PrimitiveStateActionModel` object that implements the `StateActionModel` interface and therefore represents both $V_a[s]$ and $\Omega_a[s, \cdot]$. This object observes a `KernelAverage` object representing $\Psi[s, \cdot]$, as well as each `PrimitiveStateActionData` object for the instances x such that $\Psi[s, x] > 0$. When creating a new `PrimitiveStateActionData` object due to a new piece of experience with the environment, each row of Ψ may change, notifying the `PrimitiveStateActionModel` objects observing them to recompute their values.

7.2.2 Composite Tasks

Both the declarative and procedural behavior of a composite task depend on the behavior of its subtasks. The computation of the declarative behavior, which includes the computation of the procedural behavior, proceeds in three phases. First, it constructs an MDP $\langle S_o, A_o, R_o, P_o \rangle$ from the subtask models. Second, it computes the policy π_o. Finally, it computes the predicted model (R_o, Ω_o). The composite task has a module for each of these three phases, with each module producing objects, indexed by state instances, which together define the quantity the module computes.

For any given state s, the `MDP` module constructs a `MDPState` object, which includes a `MDPStateAction` object for each subtask. Each `MDPStateAction` computes $R_o[sa]$ and $P_o[sa, \cdot]$ given $V_a[s]$ and $\Omega_a[s, \cdot]$. The `MDP` module therefore assembles the action models for each subtask into a single MDP, in part by resolving a distribution $\Omega_a[s, \cdot]$ over effects into a distribution $P_o[sa, \cdot]$ over successor states. Due to the representation of effects, even if a subtask a returns the same `StateActionModel` object for two different states, the corresponding `MDPStateAction` objects in the parent task may be distinct.

Special care must be taken to limit the size of the MDP, since the planner and predictor modules will require `MDPState` objects for every state reachable from the current state in the MDP. For this reason, Fitted R-MAXQ employs fitted value iteration, which effectively constrains all successor states in the MDP to a finite sample $X \subset S$[9], as described in Sections 3.1 and 6.1.1. In short, it implements the value-function approximation $\hat{V} = \Phi V$ by composing the transition function P with the averager Φ. Due to the use of relative effects, Fitted R-MAXQ modifies the computation of P_o near the beginning of Algorithm 6.7 as follows

$$P_o[sc, s'] \leftarrow \sum_e \Omega_c[s, e]\Phi[e(s), s'], \tag{7.2}$$

for each $s, s' \in S$ and $c \in A^o$. In the implementation of Fitted R-MAXQ, the `MDPStateAction` representing $P_o[sc, \cdot]$ (and $R_o[sc]$) therefore observes the `StateActionModel` that represents $\Omega_c[s, \cdot]$ (and $V_c[s]$) and, for each effect e such that $\Omega_c[s, e] > 0$, both the object that represents $\Phi[e(s), \cdot]$ and the `MDPState` for each s' such that $\Phi[e(s), s'] > 0$.

[9]The implementation also allows self-transitions, so technically the set of possible successor states for a state s is $X \cup \{s\}$.

The object that represents each row of Φ in Fitted R-MAXQ is an INTERPOLATIONAVERAGE, which approximates a given state as a weighted average of states sampled from S in a fixed grid.[10] In particular, it approximates a state s with the points of the hybercube containing s, with the weights obtained by linear interpolation along each state dimension. Note that this approximation differs from simple discretization in that even two nearby states will have distinct weights for the surrounding hypercube, allowing their values to remain distinct.

Fitted R-MAXQ instantiates `MDPState` objects lazily and deallocates them when they no longer have any observers. As a result, the only `MDPState` objects that reside in memory at any given time step correspond either to states in the finite sample X or to the current state s. Even though X could be very large, Fitted R-MAXQ only constructs `MDPState` objects for those states reachable from s, and it only permanently retains those objects corresponding to states in a strongly connected component of the state transition graph (which create cycles of observers).

The `Planner` module, given a non-terminal state s, constructs a `DecisionState` object that implements the `StatePolicy` interface and represents both $\pi_o[s, s\cdot]$

[10]The experiments in Section 6.3 sample each state variable at intervals of $\frac{1}{16}$ the nominal range of the state variable.

and $\tilde{V}[s]$. It computes

$$\begin{aligned}\tilde{V}_o[s] &= \beta^o[s,s]G^o[s] + (1-\beta^o[s,s])\max_{c\in A^o}\tilde{Q}_o[sc] \\ \tilde{Q}_o[sc] &= R_o[sc] + \gamma\sum_{s'} P_o[sc,s']\tilde{V}_o[s'].\end{aligned}$$

The `DecisionState` object for state s contains a `DecisionState::Action` object for each child task $c \in A^o$ (that includes s in its initiation set), which represents $\tilde{Q}_o[sc]$. The `DecisionState` computes $\tilde{V}_o[s]$ as the maximum of these $\tilde{Q}_o[sc]$ values, and it chooses a child task c that achieves this maximum to return as the policy action. The `DecisionState::Action` object for state s and child task c observes the `MDPStateAction` object that represents $R_o[sc]$ and $P_o[sc,\cdot]$, as well as the `DecisionState` objects for each s' such that $P_o[sc,s'] > 0$.[11]

In the general case, the graph of dependencies and observers among `DecisionState` objects is cyclical, requiring familiar dynamic programming methods, such as value iteration. The Fitted R-MAXQ implementation uses prioritized sweeping (Moore & Atkeson, 1993): each `DecisionState::Action` object bounds the error in its computed value, given the values of its immediate successors. Recomputing the value of $\tilde{Q}_o[sc]$ resets its local error bound to zero, but if the enclosing `DecisionState` must change $\tilde{V}_o[s]$, then it also propagates the magnitude of the change in value to each `DecisionState::Action` object that observes $\tilde{V}_o[s]$, increasing the error bound on those objects. The

[11]For terminal states, the `Planner` module constructs `ValueState` objects that contain only a value, not a policy action. The value of the terminal state is determined using only the goal function.

`Planner` module uses a priority queue to repeatedly recompute the value of the`DecisionState::Action` object with the largest error bound, until the largest error bound falls beneath a threshold ϵ.

The `Predictor` module computes $V_o[s]$ and $\Omega_o[s, \cdot]$ for a given state s, given the `DecisionState` that represents $\pi_o[s, s\cdot]$ (from the `Planner` module) and the `MDPStateAction` that represents $R_o[s\cdot]$ and $P_o[s\cdot, \cdot]$ (from the MDP module). The computation of the predicted value $V_o[s]$ resembles the computation of $\tilde{V}_o[s]$, except that it does not include the goal function G^o and replaces the maximization over subtasks with the policy action, so

$$V_o[s] = (1 - \beta^o[s, s]) \sum_{a \in A^o} \pi_o[s, sa] \left(R_o[sa] + \gamma \sum_{s'} P_o[sa, s'] V_o[s'] \right).$$

Otherwise, the Predictor module uses the same prioritized sweeping computation of values. The computation of $\Omega_o[s, \cdot]$ follows the same approach

$$\Omega_o[s, \cdot] = \sum_{a \in A^o} \pi_o[s, sa] \left(\beta^o[s, \cdot] + \gamma \sum_{s'} P_o[sa, s'] \Omega[s', \cdot] \right).$$

In Fitted R-MAXQ, $\pi_o[s, sa]$ is binary, so this quantity serves as an indicator function for the policy action. Note that because the columns of Ω are now indexed by effects, in this equation the columns of β^o are also indexed by effects. Since each composite task executes until reaching a terminal state, Fitted R-MAXQ uses absolute (non-relative) effects for composite tasks. Therefore, $\beta^o[s, e] = 1$ if $e(x) = s$ for all $x \in S$ and $\beta^p[s, e] = 0$ otherwise. For composite tasks, the notion of effects is only necessary to handle changes in state representation, not to improve generalization of the model. Note that the row vector $\Omega_o[s, \cdot]$ is stored as an adjacency list, so that summing two vectors simply requires merging the two lists (which must be sorted by effect, requiring a total ordering of possible effects) and summing the probabilities appropriately.

7.3 Using Fitted R-MAXQ

Fitted R-MAXQ adopts the RL-Glue interface (Tanner & White, 2009), permitting its use with a wide variety of environments that also adopt RL-Glue. This interface governs the interaction between agents and environments, permitting experiments that incorporate independently developed implementations of either, even when written in different programming languages. Ideally, the interface communicates to an agent everything it needs to know about the environment: the number of state variables, the ranges of the state variables, and the number of actions. Fitted R-MAXQ allows the user to specify important domain knowledge, such as state and temporal abstractions, when it is available.

In RL-Glue version 3, the user compiles the agent code, the environment code, and the code running an experiment into three separate binary executables. To run an experiment, the user launches an executable of each type before launching the standard `rl_glue` program, which initiates socket-based connections to the first three processes and initiates the experiment. The source code for this thesis includes a separate environment implementation for each problem domain used, and the simple `SingleRun` experiment runs the connected agent in the connected environment for 1000 episodes of at most 1000 time steps each. The included makefile, with the default invocation `make all`, will compile the experiment and each agent and environment.

All of the agents share the same implementation of the RL-Glue agent methods, defined in `fittedrmaxq.cc`. The agent initialization function of each instance of Fitted R-MAXQ links to a different implementation of the function `get_task_hierarchy`. This function constructs the task hierarchy, given the RL-Glue task specification, which describes the low-level state and action representation. Each agent instance, including examples of Fitted R-MAX and R-MAXQ, has its own source file implementing `get_task_hierarchy`, which defines algorithm parameters and possibly domain-specific prior knowledge. To produce a task hierarchy, this function calls `PrimitiveTask::create` and `CompositeTask::create`. These static object methods create individual task instances,[12] which may be assembled into the appropriate task hierarchy.

The constructor for primitive tasks takes five arguments:

primitive The value a of the RL-Glue action that the primitive task represents

threshold The weighted sample size m of the generalized data for state-action sa that is required before disabling optimistic exploration of sa

maxval The optimistic value $V^{\max}$ to assign $R_a[s]$ when the weighted sample size for sa is smaller than the threshold m

[12]These methods initialize the smart pointers appropriately; the user should not call the class constructors directly.

model_approximator The averager Ψ to use when generalizing data from state instances to approximate $R_a[s]$ and $\Omega_a[s, \cdot]$

precondition A predicate over state instances that determines whether the primitive task is available, to support environments in which not every primitive action is available in every state

The constructor for composite tasks takes eight arguments:

subtasks_begin An input iterator that allows the user to specify the subtasks A^o. The constructor dereferences and increments this iterator to obtain each subtask reference.

subtasks_end The constructor should stop dereferencing the input iterator once it becomes equal to this value.

initiation_set The predicate I^o over state instances that determines whether the composite task is available

termination_condition The predicate over state instances that determines the termination function β^o

goal_function The goal function G^o

averager The averager Φ to use when generalizing values from state instances to a given state

gamma The discount factor γ used by the `Planner` and `Predictor` modules during value iteration

epsilon The threshold on the Bellman error used by the `Planner` and `Predictor` modules to determine when to terminate value iteration

The following sections show how appropriate choices for the task hierarchy, including the parameters of each task, allow Fitted R-MAXQ to replicate any of the new algorithms that this thesis defines. By way of example, the makefile includes the invocation for running Fitted R-MAXQ in the Flag & Puddle World, by running the command `make run`.

7.3.1 R-MAX

Fitted R-MAXQ can reproduce the behavior of R-MAX (Brafman & Tennenholtz, 2002) simply by using no function approximation and a flat task hierarchy, which only contains a primitive task for each action available to the agent and a single composite task, which directly chooses primitive tasks to maximize expected reward in the environment. To this end, the list of possible averagers includes `ProjectionAverager` objects, which implements only state abstraction, not approximation. It "approximates" a given state s with the projection of s into the abstract state space for the primitive action a. For the purposes of Fitted R-MAXQ, an abstract state space is defined by a list of state variable indices that specifies the relevant state variables, as defined in Section 6.2.2.

The sole composite task for R-MAX is the root task of the hierarchy. Its initiation set should include every state, and it should not expicitly terminate at any state. (For episodic tasks, states that terminate an episode are represented implicitly as the absence of a successor state.) The root task also uses the `ProjectionAverager`, since no value approximation is required.

The `ProjectionAverager` is intended for discrete state spaces where generalization is not desired. For large state spaces, the agent will not acquire enough data to progress beyond its optimistic exploration phase. To allow R-MAX to be applied to continuous state spaces using the common technique of discretizing the state space, the implementation also provides a `DiscretizationAverager`, which resembles the `InterpolationAverager` in that it approximates each state using a uniform grid over the state space. Unlike the `InterpolationAverager`, the `DiscretizationAverager` assigns all the approximation weight to the single hypercube corner closest to the given state.

To run the standard R-MAX algorithm in the (discrete) Taxi domain, connect `rl_glue` to `Rmax`, `Taxi`, and `SingleRun`. The file `Rmax.cc` defines a flat task hierarchy (in which the only composite task is the root) using the `ProjectionAverager` and no domain-specific knowledge. The file `Rmax-FPWorld.cc` contains an instance of R-MAX that uses a `DiscretizationAverager` to learn in the Flag & Puddle World, along with domain-specific state abstractions for each primitive action. This agent can be run by invoking `rl_glue` with `Rmax-FPWorld`, `FlagPuddleWorld`, and `SingleRun`.

7.3.2 Fitted R-MAX

Fitted R-MAX (Jong & Stone, 2007b; Jong & Stone, 2007a), described in Section 3.2.2, extends R-MAX by replacing the averagers used in Section 7.3.1 with averagers that implement generalization. In particular, the primitive tasks use `KernelAverager`s (Section 7.2.1) to model a given state-action using data generalized from nearby states, and the composite task uses an `InterpolationAverager` (Section 7.2.2) in effect to generalize the value function from a uniform grid of states.

A `KernelAverager` weights these states as a function of their distance. It assumes a Euclidean distance function, but it allows the user to specify different scale factors for each state dimension. Its constructor accepts a `StateVariables` argument, which is a `map` that associates indices in a `StateVector` to the constant that should be multiplied with that state variable to obtain the scaled Euclidean space. Furthermore, the map may omit a `StateVector` index to indicate that the corresponding state variable is irrelevant and should not contribute to the distance.

The `KernelAverager` also accepts a generalization breadth that determines the standard deviation of the Gaussian used to compute the averager weights from the distances. The choice of scale factors for each dimension and this generalization breadth determine the generalization behavior of the primitive action. Finally, the `KernelAverager` also takes two parameters that help trade off accuracy and efficiency: the minimum nonzero Gaussian weight (before normalization) and averager weight (after normalization, which effectively bound the maximum generalization distance and number of nearest neighbors.

The `InterpolationAverager`, which provides a fitted approximation of the value function by producing a fitted approximation of the MDP, only requires two parameters: a `StateVariables` mapping that determines the scaled Euclidean state space and a resolution factor that determines the granularity of the grid sampled from the state space. The `StateVariables` parameter given to the `InterpolationAverager` should include all the `StateVector` indices used by any of the primitive tasks. The resolution factor is an integer i that causes the `InterpolationAverager` to create a grid leaving $\frac{1}{2^i}$ between each point.

Fitted R-max may be run by connecting `rl_glue` with `FittedRmax`, `MountainCar` (or `PuddleWorld`), and `SingleRun`. The file `FittedRmax.cc` defines a task hierarchy which includes only one composite task, which uses an `InterpolationAverager` to approximate values, and `KernelAverager` instances to approximate models.

7.3.3 R-MAXQ and Fitted R-MAXQ

R-maxq (Jong & Stone, 2008), described in Section 5.2.1, extends R-max along a different dimension than does Fitted R-max. It constructs the root composite task using other composite tasks, not just primitive tasks. Each composite task may contribute different goal states and a different state representation, defined by its own averager. Additional layers of the task hierarchy can more gradually increase the number of state variables as reasoning progresses up the hierarchy, while introducing subgoal states that result in a compensating decrease in the size of the completion set for each subtask MDP.

The file `Rmaxq-Taxi.cc` contains an example of how to construct such a task hierarchy, including domain-specific subtasks and state abstractions. This code is linked into the agent `Rmaxq-Taxi`, which can be connected to `rl_glue` along with `Taxi` and `SingleRun`.

Fitted R-MAXQ is the result of applying hierarchy to Fitted R-MAX instead of to R-MAX. An example task hierarchy for the Flag & Puddle World is given in `FittedRmaxq-FPWorld.cc`. To run this agent and environment, connect `rl_glue` with `FittedRmaxq-FPWorld`, `FlagPuddleWorld`, and `SingleRun`.

7.4 Discussion

This thesis has focused on the sample complexity of learning in environments where structured generalization is possible. The model-based methods it developed are computationally intensive, compared to popular model-free methods such as Q-learning and SARSA. Even though such computationally cheap algorithms are convenient for running a large number of experiments, the algorithms in this thesis adopt the perspective that CPU time is not the limiting factor in most prospective real-world applications of RL. Samples and data are the scarce resource: it doesn't matter how quickly an RL algorithm can process millions of interactions with the environment if it must learn a good policy after only thousands of interactions.

That said, after designing an algorithm such as Fitted R-MAXQ designed to learn as efficiently as possible given limited data, it is important next to optimize its implementation to learn as efficiently as possible given realistic computational resources. An algorithm with appealing theoretical properties can only ever be of academic interest if its implementation would require effectively infinite time, memory, or space. The Bayesian approach to exploration outlined in Section 2.3.1 sits at one extreme of the space of algorithms. It is in some sense optimal but requires arbitrarily accurate estimates of non-parametric probability density functions in extremely high-dimensional spaces, which remains well beyond our ability to implement.

This chapter demonstrated that Fitted R-MAXQ can be effectively implemented, if some care is taken to minimize unnecessary computation. The implementation of Fitted R-MAXQ available at `http://library.rl-community.org/wiki/Fitted_R-MAXQ` uses lazy evaluation to compute only the portion of the action models and value function necessary to choose an action at the current state. It caches the intermediary results of these computations and propagates changes efficiently through a network of dependencies. Reference counting ensures that the implemention only spends time updating values that are either directly necessary for the current state or part of a strongly connected component of the fitted MDP (and therefore expensive to reconstruct repeatedly).

Chapter 8

Discovery

This thesis has focused on algorithms that learn efficiently by taking advantage of state abstractions and temporal abstractions that allow an agent to generalize from data in a structured manner. These algorithms assume the preexistence of these abstractions, but can they be discovered automatically? This question motivated the development of Fitted R-MAXQ and the other algorithms described in the preceding chapters. These algorithms provide a firm foundation for evaluating candidate abstractions in a manner that incorporates the structured state spaces found in dynamic Bayes networks (DBNs), the structured action spaces found in hierarchical RL, and the focus on sample complexity found in model-based methods.

This chapter describes the preliminary research into abstraction discovery (Jong & Stone, 2005) that made these needs apparent. Section 8.1 defines a novel criterion for state abstraction designed for ease of discovery. Section 8.2 develops an algorithm for discovering and deploying abstractions using this criterion. Section 8.3 provides an experimental validation of these methods. Section 8.4 discusses these results in the context of this thesis and delineates directions for future work.

8.1 Policy Irrelevance

Humans can cope with an unfathomably complex world due to their ability to focus on pertinent information while ignoring irrelevant detail. In contrast, most of the research into artificial intelligence relies on fixed problem representations. Typically, the researcher must engineer a feature space rich enough to allow the algorithm to find a solution but small enough to achieve reasonable efficiency. The agent must consider enough aspects of each situation to inform its choices without spending resources worrying about minutiae. In practice, the complexity of this state representation is a key factor limiting the application of standard RL algorithms to real-world problems.

State abstraction maps two distinct states in the original state representation to a single abstract state if an agent should treat the two states in exactly the same way. The agent can still learn optimal behavior if the MDP that formalizes the underlying domain obeys certain conditions: the relevant states must share the same local behavior in the abstract state space (Dean & Givan, 1997; Ravindran & Barto, 2003). However, this prior research only directly applies in a planning context, in which the MDP model is given, or if the user manually determines that the conditions hold and supplies the corresponding state abstraction to the RL algorithm.

This chapter proposes an alternative basis to state abstraction that is more conducive to automatic discovery. Intuitively, if it is possible to behave optimally while ignoring a certain aspect of the state representation, then an agent has reason to ignore that aspect during learning. Recognizing that discovering structure tends to be slower than learning an optimal behavior policy (Thrun & Schwartz, 1995), this approach suggests a knowledge-transfer framework, in which policies learned in one domain are analyzed to discover abstractions that might improve learning in similar domains. To test whether abstraction is possible in a given region of the state space, this chapter gives two statistical methods that trade off computational and sample complexity.

Some care must be taken when applying the discovered abstractions, since the criteria defined in this section for use in discovery are strictly weaker than those given in prior work on state abstraction, typically based on either equivalence in values or in transition probabilities (Li et al., 2006). Transferring abstractions from one domain to another may also introduce generalization error. To preserve convergence to an optimal policy, the state abstractions are encapsulated in options, temporal abstractions that construe sequences of primitive actions as constituting a single abstract action (Sutton et al., 1999). In contrast to previous work with temporal abstraction, this method discovers abstract actions intended just to simplify the state representation, not to achieve a certain goal state. RL agents equipped with these abstract actions thus learn when to apply state abstraction the same way they learn when to execute any other action.

8.1.1 Defining Irrelevance

Without loss of generality, assume that the state space of an MDP is the cartesian product of (the domains of) n state variables $\mathcal{X} = \{X_1, \ldots, X_n\}$

and m state variables $\mathcal{Y} = \{Y_1, \ldots, Y_m\}$, so $S = X_1 \times \cdots \times X_n \times Y_1 \times \cdots \times Y_m$. The term $[s]_{\mathcal{X}}$ denotes the projection of s onto $\mathcal{X}$, and $s' \models [s]_{\mathcal{X}}$ denote that s' agrees with s on every state variable in $\mathcal{X}$. The goal is to determine when it is safe to abstract away $\mathcal{Y}$. This chapter introduces a novel approach to state abstraction called *policy irrelevance*. Intuitively, if an agent can behave optimally while ignoring a state variable, then it should abstract that state variable away. More formally, $\mathcal{Y}$ is policy irrelevant at s if some optimal policy specifies the same action for every s' such that $s' \models [s]_{\mathcal{X}}$:

$$\exists_a \forall_{s' \models [s]_{\mathcal{X}}} \forall_{a'}\ Q^*[s'a] \geq Q^*[s'a'], \tag{8.1}$$

where Q^* is the state-action value function for an optimal policy π^*. If $\mathcal{Y}$ is policy irrelevant for every s, then $\mathcal{Y}$ is policy irrelevant for the entire domain.

Consider the illustrative toy domain shown in Figure 8.1. It has just four nonterminal states described by two state variables, X and Y. It has two deterministic actions, represented by the solid and dashed arrows respectively. When $X = 1$, both actions terminate the episode but determine the final reward, as indicated in the figure. This domain has two optimal policies, one

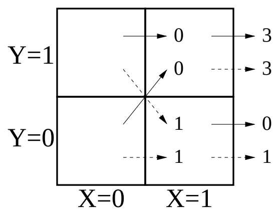


Figure 8.1: A domain with four nonterminal states and two actions. When $X = 1$ both actions transition to an absorbing state, not shown.

of which we can express without Y: take the solid arrow when $X = 0$ and the dashed arrow when $X = 1$. We thus say that Y is policy irrelevant across the entire domain.

Note however that we cannot simply aggregate the four states into two states. As McCallum pointed out, the state distinctions sufficient to represent the optimal policy are not necessarily sufficient to learn the optimal policy (McCallum, 1995). In this example, observe that if we treat $X = 1$ as a single abstract state, then in $X = 0$ we will learn to take the dashed arrow, since it transitions to the same abstract state as the solid arrow but earns a greater immediate reward. We demonstrate how to circumvent this problem while still benefitting from the abstraction in Section 8.2.2.

8.1.2 Testing Irrelevance

Given access to the transition and reward functions, an agent can evaluate the policy irrelevance of a candidate set of state variables $\mathcal{Y}$ by solving the MDP using a method, such as policy iteration, that can yield the set of optimal actions $\pi^*(s) \subseteq A$ at each state s, where $\pi^*(s)$ is the entire set $\{a \in A \mid Q^*[sa] = V^*[s]\}$. Then $\mathcal{Y}$ is policy irrelevant at s if some action is in each of these sets for each assignment to $\mathcal{Y}$: $\bigcap_{s' \models [s]_{\mathcal{X}}} \pi^*(s') \neq \emptyset$.

However, testing policy irrelevance in an RL context is trickier if the domain has more than one optimal policy, which is often the case for domains that contain structure or symmetry. Most current RL algorithms focus on finding a single optimal action at each state, not all the optimal actions. For example, Figure 8.2 shows the Q values learned from a run of Q-learning[1]. Even though the state variable Y is actually policy irrelevant, from this data we would conclude that an agent must know the value of Y to behave optimally when $X = 1$. In this trial the learning algorithm explored enough to find an optimal policy but not enough to converge to accurate Q values for every state-action pair. This phenomenon seems quite common in practice, but even with sufficient exploration the inherent stochasticity of the domain may disguise state variable irrelevance. This chapter proposes two methods for detecting policy irrelevance in a manner robust to this variability.

8.1.2.1 Statistical Hypothesis Testing

Hypothesis testing is a method for drawing inferences about the true distributions underlying sample data. This section describes how to apply this method to the problem of inferring policy irrelevance. To this end, it interprets an RL algorithm's learned value $Q[sa]$ as a random variable, whose distribution depends on both the learning algorithm and the domain. Ideally,

[1]No discounting, learning rate 0.25, Boltzmann exploration with starting temperature 50, cooling rate 0.95, for 50 episodes

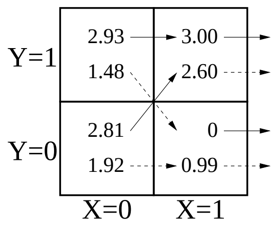


Figure 8.2: The domain of Figure 8.1 with some learned Q values.

an agent could then directly test the hypothesis that Equation 8.1 holds, but this hypothesis lacks an appropriate test statistic. Instead, assume that for a reasonable RL algorithm, the means of these distributions share the same relationships as the corresponding true Q values: $\overline{Q}[sa] \geq \overline{Q}[sa'] \equiv Q^*[sa] \geq Q^*[sa']$, where $\overline{Q}$ is the algorithm's learned state-action value function. The agent can then test propositions of the form

$$\overline{Q}[sa] \geq \overline{Q}[sa'], \tag{8.2}$$

using a standard procedure such as a one-sided paired t-test or Wilcoxon signed ranks test (Degroot, 1986). These tests output for each hypothesis of the form in Equation 8.2 its significance level $p_{s,a,a'}$. If $\overline{Q}[sa] = \overline{Q}[sa']$ then this value is a uniformly random number from the interval $(0, 1)$. Otherwise, $p_{s,a,a'}$ will tend towards 1 if Equation 8.2 is true and towards 0 if it is false. These values may be combined in a straightforward way to obtain a confidence measure for the hypothesis that Equation 8.1 holds:

$$p = \max_a \min_{s' \models [s]_X} \min_{a' \neq a} p_{s',a,a'}. \tag{8.3}$$

Figure 8.3 shows these p values for the toy domain depicted in Figure 8.1, given data from 25 independent trials of Q-learning. It uses the Wilcoxon signed-ranks test, which unlike the t-test does not assume that $Q[sa]$ is Gaussian. Figure 8.3a reveals "random" looking values, so it is reasonable to accept the hypothesis that Y is policy irrelevant for both values of X. In Figure 8.3b the values of p are very close to 0, so the agent must reject the

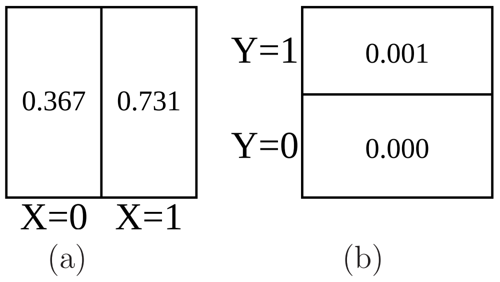


Figure 8.3: The value of p for each of the two abstract states when testing the policy irrelevance of (a) Y and (b) X.

hypothesis that X is policy irrelevant for either value of Y. This chapter uses 0.05 as a threshold for rejecting the hypothesis that Equation 8.1 holds. If p exceeds 0.05 for every s, then $\mathcal{Y}$ is irrelevant across the entire domain. In practice this number seems quite conservative, since in those cases when the hypothesis is false, empirical data yields p values orders of magnitude smaller.

8.1.2.2 Monte Carlo Simulation

The hypothesis testing approach is computationally efficient, but it requires a large amount of data. This section explores an alternative approach designed to conserve experience data when interaction with the domain is expensive. It draws upon work in Bayesian MDP models (Dearden et al., 1999) to reason more directly about the distribution of each $Q[sa]$. This technique regards each row $P[sa, \cdot]$ of the transition matrix as the parameters of a multinomial distribution over successor states. Bayesian estimation techniques define a probability distribution over the values of a given row, which may be conditioned given data from the corresponding state-action using Bayes' theorem. The Dirichlet distribution has been shown to be a conjugate prior family for multinomial distributions, so that the result of conditioning any Dirichlet prior with multinomial data is another Dirichlet distribution.

The Dirichlet distribution is parameterized by a real-valued vector, which in this context has dimensionality $1 \times |S|$. To estimate the entire transition matrix, we therefore maintain a $|S||A| \times |S|$ matrix α, where $\alpha[sa, s']$ gives the relative weight of s' when sampling a distribution over successors $P[sa, \cdot]$ for a particular state-action. If we construe each $P[sa, s']$ as a random variable, when we draw a transition matrix P from the Dirichlet, the mean $E\left[P[sa, s']\right] = \frac{\alpha[sa,s']}{\hat{n}_{sa}}$, where $\hat{n}_{sa} = \sum_s' \alpha[sa, s']$. In this sense, α can be interpreted as a (real-valued) histogram of transition counts. In fact, given a transition from sa to s' as data, the posterior parameters are obtained simply by incrementing $\alpha[sa, s']$ by 1. What this Bayesian approach offers beyond maximum-likelihood approaches is the variance of each model parameter, $\text{Var}\left[P[sa, s']\right] = \frac{p_{sas'}(1-p_{sas'})}{\hat{n}_{sa}}$, where $p_{sas'} = E\left[P[sa, s']\right]$. After conditioning on state transition data from a run of some arbitrary RL algorithm, the matrix α specifies Dirichlet distributions for each state-action. The variance of these distributions goes to 0 and their mean converges to the true transition function as the amount of data increases.[2]

Given such a Bayesian model of the domain, the agent can apply Monte Carlo simulation to make probabilistic statements about the Q values. Given

[2]It is also possible to build a Bayesian model of the reward function, but all the domains studied this this thesis use deterministic rewards.

the conditioned matrix α, it can draw a transition matrix from the collection of Dirichlets by randomly generating each entry of the transition matrix $P[sa, s'] \sim \text{Gamma}(\alpha[sa, s'], 1)$. The agent can sample several transition matrixes and perform MDP planning with each to obtain a random sample for each $Q^*[sa]$. Then it can estimate the probability that $Q^*[sa] \geq Q^*[sa']$ holds as the fraction of the sample in which the inequality is true. This probability estimate can be used in the same way that the hypothesis-testing approach uses significance levels to obtain a confidence measure for the policy irrelevance of $\mathcal{Y}$ at some s:

$$p = \max_a \min_{s' \models [s]_{\mathcal{X}}} \min_{a' \neq a} \Pr(Q^*[s'a] \geq Q^*[s'a']). \tag{8.4}$$

This method seems to yield qualitatively similar results to the hypothesis testing method. Experiments show that it almost always obtains a value of $p = 0$ for cases in which $\mathcal{Y}$ actually is relevant; it obtains a value near 1 when only one action is optimal; it obtains a uniformly random number in $(0, 1)$ when more than one action is optimal. Although it achieves similar results using less data, this method incurs a higher computational cost due to the need to solve multiple MDPs.[3]

[3]This cost can be ameliorated somewhat by initializing each MDP's value function with the value function for the maximum likelihood MDP, as in (Strens, 2000).

8.2 Abstraction Discovery

The techniques described in Section 8.1.2 both involve two stages of computation. In the first stage, they acquire samples of state-action values, either by solving the task repeatedly or by solving sampled MDPs repeatedly. In the second stage, they use this data to test the relevance of arbitrary sets of state variables at arbitrary states. Any one of these tests in the second stage is very cheap relative to the cost of the first stage, but the number of possible tests is astronomical. A practical implementation must limit both the sets of state variables it tests and the states at which it tests those state variables. This section describes a concrete method for the discovery and application of state abstractions.

8.2.1 Discovering Irrelevance

First consider the sets of state variables. It is straightforward to prove that if $\mathcal{Y}$ is policy irrelevant at s, then every subset of $\mathcal{Y}$ is also policy irrelevant at s.[4] A corollary is that an agent only need to test the policy irrelevance of $\{Y_1, \ldots, Y_k\}$ at s if both $\{Y_1, \ldots, Y_{k-1}\}$ and $\{Y_k\}$ are policy irrelevant at s. This observation suggests an inductive procedure that first tests each individual state variable for policy irrelevance and then tests increasingly larger sets only as necessary. This inductive process will continue only so long as it finds increasingly powerful abstractions.

[4]The converse is not necessarily true. Consider a duplicate of an otherwise always relevant state variable. Then each copy of the state variable is always policy irrelevant given the remainder of the state representation, but the pair of them is not.

The agent can afford to test every state variable at a given state, since the number of variables is relatively small. In contrast, the total number of states is quite large: exponential in the number of variables. This chapter hence adopts an heuristic approach, which performs tests for policy irrelevance only on states visited during some trajectories through the task. For these states, the agent then determines what sets of state variables are policy irrelevant, as described above. The outcome of these tests specifies what state features the agent "paid attention to" while applying its learned policy during those trajectories. For each set of state variables, the agent then constructs a binary classification problem with a training set comprising the visited states. An appropriate classification algorithm then allows the agent to generalize the results of the irrelevance tests to states besides those visited during the test trajectories. The next section shows how the agent can ensure that the classifiers' generalization errors do not lead to the application of unsafe abstractions.

8.2.2 Exploiting Irrelevance

Section 8.2.1 describes how to represent as a learned classifier the region of the state space where a given set of state variables is policy irrelevant. A straightforward approach to state abstraction would simply aggregate together all those states in this region that differ only on the irrelevant variables. However, this approach may prevent an RL algorithm from learning the correct value function and therefore the optimal policy. Section 8.1.1 gave a simple example of such an abstraction failure, even with perfect knowledge of policy irrelevance. Generalizing the learned classifier from visited states in one domain to unvisited states in a similar domain introduces another source of error. A solution to all of these problems is to encapsulate each learned state abstraction inside a temporal abstraction. In particular, the agent can apply each state space aggregation only inside a learned option: an abstract action that may persist for multiple time steps in the original MDP (Sutton et al., 1999).[5]

Formally, for a set of state variables $\mathcal{Y}$ that is policy irrelevant over some $S' \subseteq S$, the agent constructs an option $o = \langle \pi, \mathcal{I}, \beta \rangle$, comprising an option policy $\pi : [S']_{\mathcal{X}} \rightarrow A$, an initiation set $\mathcal{I} \subset S$, and a termination condition $\beta : S \rightarrow [0, 1]$. Once an agent executes an option o from a state in $\mathcal{I}$, it always executes primitive action $\pi(s)$ at each state s, until terminating with probability $\beta(s)$. The experiments in this chapter set $\mathcal{I} = S'$ and $\beta(s) = 0.01$ for $s \in \mathcal{I}$ and $\beta(s) = 1$ otherwise.[6] Since $\mathcal{Y}$ is policy irrelevant over S', the agent chooses an option policy π equal to the projection onto $[S']_{\mathcal{X}}$ of an optimal policy for the original MDP. An agent augmented with such options can behave optimally in the original MDP by executing one of these options whenever possible.

Although the discovery of these abstractions is interesting in its own

[5]Chapter 4 discusses options in more detail.

[6]The nonzero termination probability for $s \in \mathcal{I}$ serves as a probabilistic timeout to escape from bad abstractions.

right, their utility becomes most apparent when an agent can transfer the discovered options to novel domains, for which it does not yet have access to an optimal policy. To transfer an option to a new domain, the agent simply copies the initiation set and termination condition. This straightforward approach suffices for domains that share precisely the same state space as the original domain. Even when the state space changes, representating $\mathcal{I}$ and β as a learned classifier gives some hope for reasonable generalization. The agent can also copy the option policy π, if it expects the optimal behavior from the original domain to remain optimal in the new domain.

This chapter assumes only that the policy irrelevance remains the same. The implemented agent relearns the option policy concurrently with the learning of the high-level policy, which chooses among the original primitive actions and the discovered options. Each option establishes an RL subproblem with state space $[\mathcal{I}]_{\mathcal{X}}$ and the same action space A. Whenever an option terminates in a state s, the agent augments the reward from the environment with a subgoal reward equal to the current estimate of the optimal high-level value function evaluated at s. The option can be construed not as learning to achieve a subgoal but as learning to behave while ignoring certain state variables. In other words, the option adopts the goals of the high-level agent, but learns in a reduced state space.

Since each option is just another action for the high-level agent to select, RL algorithms will learn to disregard options as suboptimal in those states where the corresponding abstractions are unsafe. The options that correspond to safe state abstractions join the set of optimal actions at each appropriate state. The smaller state representation should allow the option policies to converge quickly, so RL algorithms will learn to exploit these optimal policy fragments instead of uncovering the whole optimal policy the hard way. The next section illustrates this process.

8.3 Experimental Results

This chapter's experiments use Dietterich's Taxi domain (Dietterich, 2000a), illustrated in Figure 8.4 and previously discussed in Section 5.1. Recall that this domain has four state variables. The first two correspond to the taxi's current position in the grid world. The third indicates the passenger's current location, at one of the four labeled positions (Red, Green, Blue, and Yellow) or inside the taxi. The fourth indicates the labeled position where the passenger would like to go. The domain therefore has $5 \times 5 \times 5 \times 4 = 500$ possible states. At each time step, the taxi may move north, move south, move east, move west, attempt to pick up the passenger, or attempt to put down the passenger. Actions that would move the taxi through a wall or off the grid have no effect. Every action has a reward of -1, except illegal attempts to pick up or put down the passenger, which have reward -10. The agent receives a reward of +20 for achieving a goal state, in which the passenger is at the destination (and not inside the taxi). This thesis uses the stochastic version of the domain: whenever the taxi attempts to move, the resulting motion occurs in a random perpendicular direction with probability 0.2. Furthermore, once

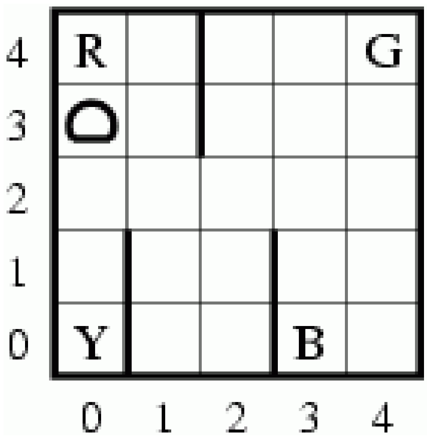


Figure 8.4: The Taxi domain.

the taxi picks up the passenger and begins to move, the destination changes with probability 0.3.

This domain's representation requires all four of its state variables in general, but it still affords opportunity for abstraction. In particular, note that the passenger's destination is only relevant once the agent has picked up the passenger.

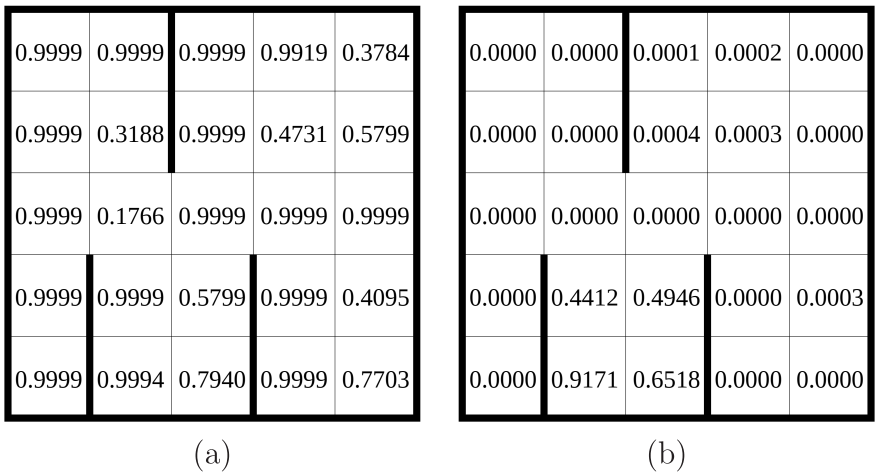
8.3.1 The Wilcoxon signed ranks test

The experimental agent applied the methodology described in Sections 8.1 and 8.2 to the task of discovering this abstraction, as follows. First, it ran 25 independent trials of Q-learning to obtain samples of Q^*. For each trial, it set the learning rate $\alpha = 0.25$ and used ϵ-greedy exploration with $\epsilon = 0.1$. Learning to convergence required about 75000 time steps for each trial.

This data allows the agent to compute the policy irrelevance of any state variable at any state. For example, consider again the passenger's destination. To demonstrate the typical behavior of the testing procedure, Figure 8.5a shows the outcome for every location in the domain, when the passenger is waiting at the upper left corner (the Red landmark), using the Wilcoxon signed-ranks test. The nonzero p values at every state imply that the passenger's destination is policy irrelevant in this case. Note that the values are extremely close to 1 whenever the agent has only one optimal action to get to the upper left corner, which the procedure can then identify confidently. The squares with intermediate values are precisely the states in which more than one optimal action exists. Now consider Figure 8.5b, which shows the output of the same test when the passenger is inside the taxi. The p values are extremely close to 0 in every state except for the four at the bottom middle, where due to the layout of the domain the agent can always behave optimally by moving north.

8.3.2 Monte Carlo Simulation

Another implementation of the abstraction-discovering agent used the Monte Carlo approach on the Taxi domain. This agent used R-MAX (Brafman & Tennenholtz, 2002) (described in more detail in Section 2.3.2), with $m = 10$ to ensure that the Bayesian model had at least ten samples for each reachable state-action. The agent explored for 40,000 time steps in the original Taxi domain, enough to ensure that it completed its exploration in practice. The agent assumed that the reward function was deterministic, so it knew all the



(a)

0.9999	0.9999	0.9999	0.9919	0.3784
0.9999	0.3188	0.9999	0.4731	0.5799
0.9999	0.1766	0.9999	0.9999	0.9999
0.9999	0.9999	0.5799	0.9999	0.4095
0.9999	0.9994	0.7940	0.9999	0.7703

(b)

0.0000	0.0000	0.0001	0.0002	0.0000
0.0000	0.0000	0.0004	0.0003	0.0000
0.0000	0.0000	0.0000	0.0000	0.0000
0.0000	0.4412	0.4946	0.0000	0.0003
0.0000	0.9171	0.6518	0.0000	0.0000

Figure 8.5: The results of the Wilcoxon signed-ranks test for determining the policy irrelevance of the passenger's destination in the Taxi domain. We show the result of the test for each possible taxi location for (a) a case when the passenger is not yet in the taxi and (b) the case when the passenger is inside the taxi.

one step rewards after visiting each state-action pair at least once. In general, if the agent does not make this assumption, then it must choose some prior distribution over rewards for each state-action pair. Since the Taxi domain has a deterministic reward function, this complication may be avoided in the work reported here. [7]

Furthermore, the agent initialized to 0 each parameter of the Dirichlet prior distributions over the multinomial transition distributions. This prior distribution is not formally a Dirichlet distribution, which assumes that each parameter is positive. However, the agent can still sample from these distribu-

[7]Dearden, Friedman, and Andre (Dearden et al., 1999) model rewards as multinomial distributions the same way they model transition functions, but as a result they must supply a priori some finite set that contains all possible rewards.

tions by assuming that unobserved state transitions have probability 0. This form of prior, known as an improper prior, has the advantage of yielding a Bayesian model whose mean is identical to the maximum likelihood model, and it is slightly more computationally efficient than the approach of Dearden et al. (1999).

After the exploration phase, the agent sampled 100 MDPs from the learned Bayesian model, then it solved each of these using value iteration. The following results examine the same two cases as in Section 8.3.1. Figure 8.6a shows for each of the 25 taxi locations the maximum probability at which some action is optimal across all passenger destinations, given that the passenger is still waiting at the upper left landmark. In other words, each cell contains the quantity $\max_a \min_{s' \models [s]_{\mathcal{X}}} \Pr(Q[s'a]) = V[s'])$, where $\mathcal{X}$ corresponds to the taxi location and passenger location.

Although these estimated probabilities do not convey the same formal meaning as the significance values that statistical hypothesis tests output, they may be interpreted in a somewhat similar fashion. Consider the taxi location with the smallest estimated probability, which is 0.20. If the null hypothesis states that some action is optimal at that location across all passenger destinations, the Monte Carlo simulation gives no reason to reject that hypothesis, since at least one action was optimal in 20 of the 100 sampled MDPs.

Figure 8.6b shows the estimated probabilities for the second case, when the passenger is inside the taxi. Note that for all the locations where the passenger destination is in fact relevant, no action was optimal across passenger

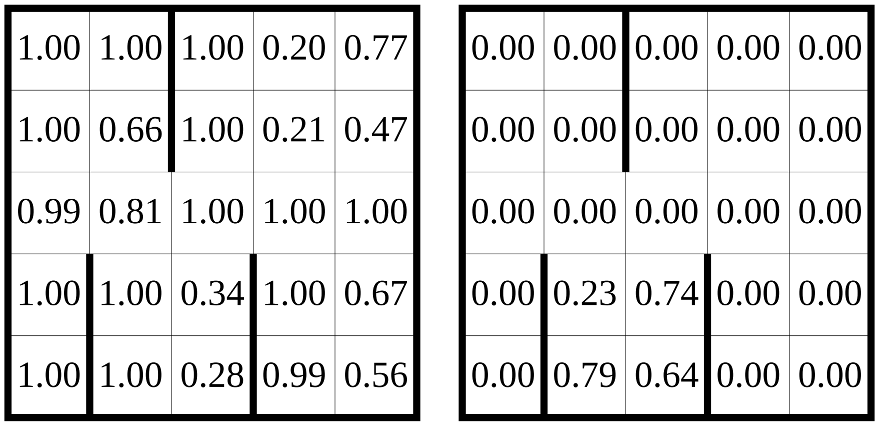


Figure 8.6: The results of Monte Carlo for determining the policy irrelevance of the passenger's destination in the Taxi domain. We show the result of the test for each possible taxi location for (a) a case when the passenger is not yet in the taxi and (b) the case when the passenger is inside the taxi.

destinations in any of the 100 sampled MDPs. One can easily imagine setting a probability threshold similar in meaning to the significance level of statistical hypothesis tests, then rejecting the null hypothesis only when the estimated probability falls below that threshold. In ten trials, a threshold of 0.05 never caused any false negatives but did lead the algorithm erroneously to classify the passenger's destination as relevant in three instances out of 1000. (In each trial, the destination is irrelevant for each combination of four passenger locations and 25 taxi locations.)

The principal cost of the Monte Carlo approach is computational. The process of learning the Bayesian model, sampling 100 MDPs, and performing value iteration until convergence 100 times required 335 seconds on a 2.8 GHz Pentium 4 CPU, in contrast to the 9 seconds required to run 25 instances of

Q-Learning and to apply the Wilcoxon signed ranks test. On the other hand, the Monte Carlo approach makes more efficient use of the data, requiring only 40000 steps of direct experience with the environment instead of $25 \times 75000 = 1875000$ steps. Thus one method emphasizes computational efficiency and the other sample complexity.

8.3.3 Generalization of Discovered Abstractions

Once policy-irrelevant variables have been found at each state, regardless of the method used to test for policy irrelevance, the next step for an agent is to generalize and apply these abstraction. Rather than compute the outcome of the test for every subset of state variables at every state, the agent followed the approach described in Section 8.2.1 and sampled 20 trajectories from the domain using one of the learned policies. It tested each individual state variable at each state visited. The following results used the hypothesis testing approach. The agent then created a binary classification problem for each variable, using the visited states as the training set. For the positive examples, it took each state at which the hypothesis test returns a p value above the conservative threshold of 0.05. Finally, it applied a simple rule-learning classifier to each problem: the Incremental Reduced Error Pruning (IREP) algorithm, as described in (Cohen, 1995). A typical set of induced rules follows:

1. Taxi's x-coordinate:

   (a) $y = 1 \wedge$ passenger in taxi $\wedge$ destination Red
       $\Rightarrow$ *policy irrelevant*

   (b) otherwise, *policy relevant*

2. Taxi's y-coordinate:

   (a) $x = 4 \wedge$ passenger in taxi $\Rightarrow$ *policy irrelevant*

   (b) otherwise, *policy relevant*

3. Passenger's destination:

   (a) passenger in taxi $\Rightarrow$ *policy relevant*

   (b) otherwise, *policy irrelevant*

4. Passenger's location *and* destination

   (a) $(x = 1 \wedge y = 2) \vee (x = 1 \wedge y = 1)$
       $\Rightarrow$ *policy irrelevant*

   (b) otherwise, *policy relevant*

The sets of state variables not mentioned either had no positive training examples or induced an empty rule set, which classifies the state variables as relevant at every state. Rule set 3 captures the abstraction that motivated the analysis of this domain, specifying that the passenger's destination is policy relevant only when the passenger is in the taxi. The other three rules classify state variables as usually relevant, except in narrow cases. For example, rule

1a holds because the Red destination is in the upper half of the map, $y = 1$ specifies that the taxi is in the lower half, and all the obstacles in this particular map are vertical. Rule 2a is an example of an overgeneralization. When holding the passenger on the rightmost column, it is usually optimal just to go left, unless the passenger wants to go the Green landmark in the upper-right corner.

The generalization performance of these learned abstractions were tested on 10×10 instances of the Taxi domain with randomly generated obstacles, running both horizontally and vertically. These instances placed one landmark near each corner and otherwise had the same dynamics as the original domain. Each abstraction was implemented as an option, as discussed in Section 8.2.2. Since the locations of the landmarks moved, agents could not simply transfer option policies from the original Taxi domain. In each experiment, the agent used Q-learning with ϵ-greedy exploration[8] to learn both the option policies and the high-level policy that chose when to apply each option and thus each state abstraction.[9] To improve learning efficiency, the agent also employed off-policy training (Sutton et al., 1999), as follows. Whenever a primitive action a was executed from a state s, it updated $Q[sa]$ for the high-level agent as well as for every option that includes s in its initiation set. Whenever an option o terminated, it updated $Q[so]$ for every state s visited during the execution

[8] $\epsilon = 0.1$ and $\alpha = 0.25$

[9] In general, SMDP Q-learning is necessary to learn the high-level policy, since the actions may last for more than one time step. However, this algorithm reduces to standard Q-learning in the absence of discounting, which the Taxi domain does not require.

of o. Each state-action estimate in the system therefore received exactly one update for each timestep the action executed in the state. Figure 8.7 compares the learning performance of this system to a Q-learner without abstraction. The abstractions allowed the experimental Q-learner to converge much faster to an optimal policy, despite estimating a strict superset of the parameters of the baseline Q-learner.

8.4 Discussion

This chapter addressed the problem of discovering state abstractions automatically, given only prior experience in a similar domain. It defined a condition for abstraction in terms of the relevance of state variables for expressing an optimal policy, and it described two statistical methods for testing this condition for a given state and set of variables. One method applies efficient statistical hypothesis tests to Q-values obtained from independent runs of an RL algorithm. The other method applies Monte Carlo simulation to a learned Bayesian model to conserve experience data. Then it exhibited an efficient algorithm to use one of these methods to discover what sets of state variables are irrelevent over what regions of the state space. Finally, this chapter showed that encapsulating these learned state abstractions inside temporal abstractions allows an RL algorithm to benefit from the abstractions while preserving convergence to an optimal policy.

The research described in this chapter directly motivated the contributions of earlier chapters in this thesis. This approach to abstraction discovery

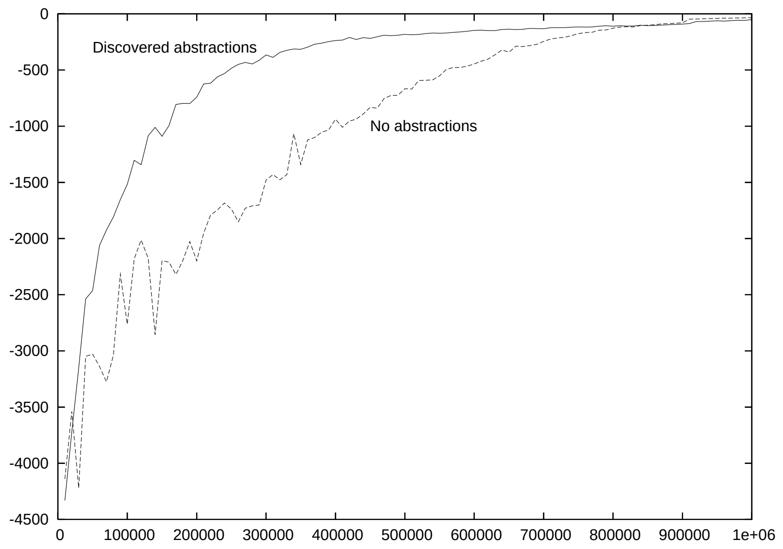


Figure 8.7: The average reward per episode earned by agents with learned abstractions encapsulated as options and only primitive actions, respectively, on a 10×10 version of the Taxi domain. The reward is averaged over 10000-step intervals. The results are the average of 25 independent trials.

suffers from some practical limitations, which suggested the need for an RL agent that combines model-based exploration, function approximation, and hierarchical decomposition. The development of such an agent in this thesis provides a necessary foundation for investigating abstraction discovery in more realistic contexts.

The benefits of model-based RL become apparent when considering the two methods for testing state variable relevance. The model-free method given in Section 8.1.2.1 requires several independently learned near-optimal value functions to analyze for potential state abstractions. In contrast, the model-based method given in Section 8.1.2.2 demonstrates the efficiency of methods that reason explicitly about the parameters of the underlying MDP and the uncertainty in estimates of these parameters. Model-based RL agents therefore seem a more promising context for the practical discovery of abstractions.

Another limitation is the reliance on transfer learning. The experiments in Section 8.3 discover abstractions for use in a target environment by first converging to optimal behavior in a related environment and then transferring appropriate knowledge. Ideally, abstraction discovery could speed learning in the target environment without requiring an easy-to-learn but sufficiently similar source environment. However, discovering a good representation seems a harder problem than learning with a given representation, in the general case. Continuous or otherwise infinitely large environments provide a more natural setting for abstraction discovery, since they necessarily require generalization or approximation. In such an environment, an agent might learn successively better policies by adopting increasingly fine approximations. Over time, this agent would generalize less and less, requiring more and more data. This kind of agent creates an opportunity for abstraction discovery, with a coarsely approximated model or value function, to reduce the data requirements of a finer approximation. Although many researchers explore new algorithms first in small, discrete state spaces, this opportunity suggests a benefit to developing discovery algorithms compatible with function approximation.

Finally, the experimental agent in this chapter encapsulated partial state abstractions inside temporal abstractions, namely options. Agents that incorporate model learning and function approximation, for the reasons discussed above, must therefore integrate these mechanisms with hierarchical decomposition. The details of this integration, which comprise the preceding chapters of this thesis, were also informed by the ideas described in this section. Fitted R-MAX, defined in Chapter 3, depends on a generalization breadth parameter, which directly corresponds to the resolution of function approximation. Chapters 4 and 5 investigated how hierarchy can benefit an agent, showing that instead of duplicating the ability of model-based RL to guide exploration, hierarchical constraints can help R-MAXQ reduce the number of model parameters that require estimation. Furthermore, this benefit is fully compatible with state abstraction, which can further reduce this number.

Chapters 6 and 7 describe the Fitted R-MAXQ algorithm, which combines all of these ideas into a single agent which is uniquely suited to serve as the foundation for ongoing research into abstraction discovery. Its support for factored state spaces and hierarchical decomposition permit the agent to experiment with the synergies between state abstractions and temporal abstractions. Its model-based approach to exploration and instance-based function approximation allow the agent to reuse data efficiently, so that several candidate abstractions or degrees of approximation may be evaluated in parallel. Fitted R-MAXQ not only seeks to learn with the best possible data efficiency, but it does so in a manner designed to support critical future work in discovering representations and generalizations. Chapter 10 outlines some potential next steps for this research into discovery, incorporating the other contributions of this thesis.

Chapter 9

Related Work

Successful learning in more complex environments remains the primary challenge of RL researchers. This thesis describes one approach to expanding the reach of RL, by extending the reach of model-based methods, which are known to be efficient in exploring small domains where generalization is not necessary. These extensions include function approximation, necessary for environments with continuous state, and hierarchical decomposition, which allows an agent's knowledge more closely to match the structure of realistic problems. This chapter will survey other efforts to expand the scope of RL, paying particular attention to methods that address the issues of exploration and generalization in environments with structured dynamics or infinite state spaces.

9.1 Exploration

The capacity for Q-learning (Watkins, 1989) to converge to optimal behavior, despite its stark simplicity, fueled rapidly growing interest in RL. This growth lead to the field's inclusion in undergraduate Artificial Intelligence textbooks (Russell & Norvig, 1995) and the release of a widely accepted standard text of its own (Sutton et al., 1999). However, the asymptotic convergence of Q-learning tends to be slow in practice. In contrast, the Prioritized Sweeping algorithm explicitly sought to improve the empirical efficiency of RL, by constructing a model and guiding the agent to insufficiently explored states (Moore & Atkeson, 1993). It offered an arbitrary threshold to determine the sufficient amount of data, without giving any theoretical guarantees. The algorithm's name refers to its mechanism for reducing computation time, by updating the values of states during planning in an order dependent on the propagated Bellman error.

Kearns and Singh (1998) produced the first upper bound on the sample complexity of RL by defining and analyzing another model-based algorithm, Explicit Explore or Exploit (E^3). Brafman and Tennenholtz (2002) demonstrated that a much simpler algorithm, R-MAX, achieves the same convergence guarantees, in the more general setting of stochastic games. In the MDP setting, R-MAX reduces to Prioritized Sweeping (although Brafman and Tennenholtz, focusing only on sample complexity, do not specify the details of how to compute the value function given the model). Kakade (2003) later improved upon the sample complexity bounds of RL by contributing a new analysis of R-MAX.

This thesis builds upon model-based RL in part because R-MAX continues to have among the best known sample complexity among exact algorithms for finite MDPs. It seems no coincidence that the algorithms that explicitly reason about the dynamics of the environment are generally the only ones that can guarantee finite-time convergence to a near-optimal policy with some probability. One notable exception is Delayed Q-learning, which attains such guarantees while being nominally model-free (Strehl et al., 2006). However, this algorithm is model-free only in the sense that its space complexity is sub-quadratic with respect to the size of the state space: it never constructs a complete transition matrix. Instead, it recalls the last m successor states for each state-action. Delayed Q-learning otherwise employs the same general strategy as R-MAX, initializing values optimistically and only updating values when sufficient data exists to approximate the dynamics of a given state-action. The implementation of Fitted R-MAXQ described in Chapter 7 is also model-free in this sense, since the number of instances used to approximate the dynamics for any given state-action is bounded by a constant.

Other model-based approaches reason about uncertainty in more nuanced ways. For example, Model-Based Interval Estimation constructs confidence intervals around the estimated MDP parameters (Strehl & Littman, 2005), allowing an agent to decrease its optimism in a more gradual way than in R-MAX. Bayesian approaches explicitly define probability distributions over MDPs, which are conditioned given experience and sampled to perform inference or planning (Dearden et al., 1999; Strens, 2000). These approaches tend to be quite computationally intensive, but they clarify the prior knowledge and inductive assumptions a learning agent makes. These other techniques for constructing models from data might serve as intriguing alternatives to R-MAX, for use as a foundation for synthesis with function approximation and hierarchical decomposition. In addition, Bayesian RL in principle could define prior distributions over hierarchical structures or approximation schemes.

Another potential modification to Fitted R-MAXQ is a replacement of its planning component, fitted value iteration conjoined with prioritized sweeping. Sparse sampling, which produces near-optimal plans in time that does not depend on the size of the MDP (Kearns et al., 1999), may be particularly useful for planning with large models. Although in principle model-based RL decomposes the RL problem neatly into a model-estimation step and a planning step, planning techniques may inform the estimation process. For example, the concept of averagers and composing transition functions used by fitted value iteration (Gordon, 1995) informed the model approximation of Fitted R-MAX. Another planning algorithm, Topological Value Iteration, improves planning speed by identifying the strongly connected components in the transition graph of an MDP (Dai & Goldsmith, 2007; Dai et al., 2008). A topological perspective on planning might help refine the notion of completion set: Fitted R-MAX can be seen as defining an approximation of a continuous MDP that constrains all strongly connected components to a finite sample of the state space.

9.2 Generalization

One prerequisite for applying RL to real-world problems is the capacity to cope with continuous state spaces. Early efforts to tackle such problems focused on generalizing the standard tabular representation of the value function using function approximators, such as tile coding (Sutton, 1996) or neural networks (Tesauro, 1995; Tesauro et al., 2007). This approach allows the straightforward reuse of classic algorithms, such as Q-learning. A key downside is that such straightforward generalizations sacrifice even the weak (asymptotic) guarantees of these algorithms (Boyan & Moore, 1995).

Nevertheless, the bulk of research using continuous-state problems still employs model-free algorithms, seeking more robust approximation methods that can safely capture the inherent structure of continuous value functions. Some of the greatest contributions in this branch of research defined algorithms, such as Least Square Policy Iteration (Lagoudakis & Parr, 2003) and Neural Fitted Q Iteration (Riedmiller, 2005), which at least prevent the computation of the value function from diverging, as well as providing empirically good performance, at least when given the correct basis functions or neural network learning parameters. These improvements led naturally to research attempting to define or discover better basis functions and function representations (Glaubius & Smart, 2004; Mahadevan, 2005; Keller et al., 2006; Whiteson & Stone, 2006). Given the difficulty of defining appropriate value function approximations, some successful approaches decline to reason about the MDP dynamics and value function at all, instead searching the space of policies directly as an optimization problem (Stanley & Miikkulainen, 2002).

All of these methods focus on how to estimate the value function accurately given the available data, but they don't reason explicitly about how to obtain this data. Most rely on some variant of random exploration, often using the epsilon-greedy strategy of choosing a random action at a given time step with some fixed probability and otherwise following the current estimated optimal policy.

Kakade et al. (2003) do address the sample complexity of RL in metric state spaces. They offer a theoretical bound that assumes both the existence of an approximate planner that guarantees near-optimal policies given a generative model and also the existence of a local modeling algorithm, which can produce a generative model with bounded error given sufficient data within the neighborhood of a given state. Fitted R-max can be seen as a practical approximation of this theoretical algorithm, using fitted value iteration as the approximate planner and a locally weighted instance-based model based on Kernel Based Reinforcement Learning (KBRL).

Ormoneit and Sen (2002) originally presented KBRL as a largely theoretical instance-based algorithm, which demonstrated asymptotic convergence to the optimal value function given data sampled uniformly from the state space. Considering recent results demonstrating the near-optimality of fitted value iteration (Munos & Szepesvári, 2008), Fitted R-max thus seems a good foundation for the search for algorithms that can learn near-optimally in the presence of generalization.

Other researchers have applied model-learning to continuous-state RL, but rarely with any attention paid to exploration and never in the general case of stochastic transitions. For example, Atkeson et al. (1997) learned deterministic models using Locally Weighted Regression, which computes the successor for a given state with linear regression that weights the data in the same way that Fitted R-max weights nearby instances. Despite the absence of convergence guarantees, such model-based methods have proven efficient in practice. TEXPLORE is one recent algorithm that forsakes such guarantees in favor of proven supervised learning methods to produce models, which when combined with explicit exploration show faster learning in practice (Hester & Stone, 2010).

More recently, Parr et al. (2008); Sutton et al. (2008) drew connections between the popular class of linear function approximators and models, showing the equivalence of the approximate value functions produced by LSTD and LSPI with the exact value functions resulting from planning with an approximate model in feature space. These results parallel the observation of Gordon (1995) that the approximate value function produced by fitted value iteration is equivalent to the exact value function resulting from planning with a model fitted with averagers. This parallel suggests an alternative extension of R-max to continuous state spaces, tying optimism to features of the state space instead of to neighborhoods of the state space.

9.3 Structure

Given the difficulty of learning accurate value functions in realistic applications, many researchers have attempted to imbue into their learning agents some notion of the structure that humans can intuitively perceive in the world. One particularly appealing form of structure is the hierarchical decomposition addressed in this thesis. Hierarchical methods have long been a significant portion of the RL literature. Barto and Mahadevan (2003) provided an excellent survey of the currently popular frameworks: options (Sutton et al., 1999), MAXQ (Dietterich, 2000a) and HAMs (Parr & Russell, 1998). The variety in these frameworks underscores the lack of agreement in the precise purpose of hierarchical decomposition. The standard options framework grew out of the idea of macro-operators in planning (Korf, 1985), presupposing the existence of partial solutions that can be packaged into option policies. MAXQ and HAMs instead learn hierarchical policies from the ground up and construe hierarchy as a mechanism for dynamically applying state abstractions (Dietterich, 2000b; Andre & Russell, 2002). Most ongoing work in hierarchical RL still uses the original model-free formulations of these algorithms.

One exception is Hierarchical H-learning (Seri & Tadepalli, 2002), which adapts the MAXQ framework to the less commonly studied average-reward formalism for RL. In this formalism, the goal is to learn policies that maximize the average reward per time step, instead of the cumulative discounted reward or the total reward over some finite horizon. Hierarchical H-learning is also model-based, in the sense that it learns transition functions for each subtask and a reward function for primitive actions. However, to learn the transition model, it uses maximum likelihood estimation given the observed behavior of each subtask as they learn. Unlike R-MAXQ this approach does not take into account the nonstationary dynamics of subtasks during learning, although in practice the behavior of the subtasks presumably converge. Seri and Tadepalli also define a version of their algorithm that employs optimistic initialization to encourage exploration, but neither Hierarchical H-learning nor its base algorithm, H-learning, have any known convergence guarantees.

The Deterministic Sample-Based Hierarchical Learner (DSHL) also applies model learning to the MAXQ framework, although it only handles the deterministic case (Diuk et al., 2006). Without stochasticity, the model learning reduces to remembering the successor state for each state-action in a finite environment. DSHL does plan within each task by computing a model and applying a planner, but the abstract model at one level of the hierarchy bears no direct relationship to the abstract model at another level. When considering the effect of a subtask, DSHL computes the exact successor state in the full state space by sampling a trajectory with the hierarchical policy for that subtask, recursing down to the memorized model of the primitive actions. Furthermore, to take advantage of the same hierarchy of state abstractions that (Dietterich, 2000a) developed for MAXQ, the DSHL must replan at the beginning of each episode (or after visiting a state-action for the first time), since the precise rewards and transition probabilities in an abstract model may depend on the current concrete state.[1] In contrast, R-MAXQ avoids the need to replan by replacing state abstraction with subgoals (which induce small completion sets) at higher levels of a task hierarchy. This technique allows R-MAXQ to

[1]For example, the standard hierarchy for the Taxi domain abstracts away the taxi's current coordinates in the `ROOT` task, but the value of the `GET` subtask depends on these coordinates. The model-free MAXQ-Q algorithm sidesteps this problem by decomposing this value into components, one of which is a function of the subtask and has an independent state representation.

define stable models of each task recursively in terms of the subtask models.

Hauskrecht et al. (1998) independently studied hierarchical models similar to those of R-MAXQ in the planning setting. They studied using option-like macro-actions to speed up the computation of the optimal value function for a given MDP. For a given partition of a state space into clusters, they defined abstract multi-time models, following Sutton et al. (1999), for options that navigate to the cluster boundaries. With experiments similar to those in Chapter 4, they demonstrated that augmenting the base MDP with these macro-actions could worsen planning speed, depending on the initialization of the value function. Their computation of the value function converged most quickly by planning in the abstract MDP whose actions comprised only macro-actions and whose states correspond only to the boundary states (completion set) of those actions, as in R-MAXQ.

Other attempts to apply structure to RL often involve using tree representations that decompose the state space hierarchically, if not the action space. The U-Tree algorithm (McCallum, 1995) lazily defines state distinctions using statistical testing, much in the spirit of the preliminary work in Chapter 8. This algorithm is particularly notable in that it also applies to the partially observable MDP case, although even in the standard RL context it cannot guarantee escaping a “local minimum” in state representation. Hester and Stone (2009) learn deterministic models in continuous state spaces using a decision tree to predict the change in each state variable due to a given action, while employing domain-specific heuristics to promote exploration and reduce overgeneralization. Other researchers have studied the problem of planning with models that incorporate state abstraction using Dynamic Bayes Networks (DBNs), using structured value-function representations instead of hierarchically decomposed value functions to manage the convolution of state representations (Boutilier et al., 1995; Koller & Parr, 2000). Such planning techniques could potentially be combined with R-MAX exploration in the RL setting.

Finally, this thesis was inspired by a deluge of efforts to discover task hierarchies for RL. Some of this work originated in attempts to reuse portions of learned policies for agents that encounter a sequence or distribution of environments throughout their lifetimes (Thrun & Schwartz, 1995; Kalmár & Szepesvári, 1999). The emergence of the options framework inspired a wave of methods that define subgoals using some notion of novelty (Şimşek & Barto, 2004; Butz, 2004) or bottleneck states (McGovern & Barto, 2001; Stolle & Precup, 2002; Mannor et al., 2004). More recently, some researchers induced MAXQ task hierarchies by analyzing what state variables change in agent trajectories (Hengst, 2002; Mehta et al., 2008). All of this research relied on model-free learning in finite domains. A key contribution of this thesis is a foundational algorithm, Fitted R-MAXQ, that fully integrates hierarchical RL with principled exploration, using model-based methods, and the ability to generalize in continuous environments, using function approximation.

Chapter 10

Conclusion

10.1 Thesis Contributions

This thesis brought together several key ingredients that each independently sustain entire branches of research into Reinforcement Learning. These branches all share the same broad goal: to allow learning agents to succeed in real-world environments. Achieving this goal would allow us to deploy artificial intelligence in the same way as human intelligence: by beginning with a powerful learning mechanism (including an appropriate inductive bias) and allowing experience with the world to construct the necessary knowledge.

The first ingredient is directed exploration, which allows an agent to change its goal temporarily from maximizing extrinsic environmental rewards to gathering more data. Too many RL algorithms focus exclusively on the estimation of statistical parameters given fixed data, without considering the active learning question of what data would be valuable. The second ingredient is generalization, which allows an agent to draw conclusions even about situations it has never before encountered. Both of these faculties are essential to reinforcement learning in practice: model-based methods that enable directed exploration are known to have excellent data efficiency, and function approximators that enable generalization of values are required to obtain any sort of reasonable behavior in continuous environments. Nevertheless, almost no research has explicitly addressed the tension between exploration, which sees a given state as novel enough to require new data, and generalization, which sees a given state as familiar enough to apply existing data. This thesis contributed an algorithm, Fitted R-MAX, which applies both model-based exploration and function approximation to learn efficiently in continuous state spaces (Chapter 3).

Although function approximation allows RL to generalize across infinite state spaces, it does not eliminate the curse of dimensionality. Nearly all realistic problems have natural representations that include multiple state variables, and requiring an agent to learn a combinatorial number of parameters. Hierarchical decomposition constitutes an appealing mechanism for managing the complexity of such problems, but little agreement exists as to how precisely hierarchy can help. This thesis contributed both a study of the utility of hierarchical RL methods (Chapter 4) and an integration of such decompositions with model-based methods, in the form of R-MAXQ (Chapter 5). This algorithm relies on optimism in the face of uncertainty to encourage the exploration of novel states, eliminating one popular motivation for applying a hierarchy. This thesis instead suggested that key roles for hierarchy include preventing unnecessary exploration and ameliorating the curse of dimensionality, by permitting smaller state representations at each level of the hierarchy. At lower levels, state abstractions can reduce the dimensionality of learning;

at higher levels, planning need only consider subgoal states.

Finally, this thesis developed a powerful synthesis of all three of these ingredients: model-based exploration, function approximation, and hierarchical decomposition (Chapter 6). It reifies these ingredients as composable operators that modify Bellman equations and therefore action models. The result of composing all the newly defined operators is the Fitted R-MAXQ algorithm, for which the source code is publicly available[1] (Chapter 7). This algorithm takes advantage of domain knowledge, in the form of a task hierarchy, to learn near-optimal policies with excellent data efficiency, even in continuous environments with rich structure.

10.2 Future Work

This thesis makes no claim that Fitted R-MAXQ is the solution to RL or even necessarily that this algorithm is the best solution for some real-world problem. It still requires a task hierarchy to avoid exploring every neighborhood of the state space, and its online updates to the approximate model are very computationally intensive, given the nearest-neighbor lookups of previous instances. With respect to the latter concern, the speed of computers continues to grow rapidly, but data remains as expensive as ever, particularly for the control problems and agent-based systems relevant to RL. Even though modern computer systems achieve high performance by emphasizing paral-

[1]The most recent version is available in the RL-Library at `http://library.rl-community.org/wiki/Fitted_R-MAXQ`.

lelism over raw clock speed, limiting the classes of computationally intensive algorithms that can benefit from advances in hardware, Fitted R-MAXQ is a promising candidate for a multi-threaded implementation. Each primitive action model depends only on the data for that primitive action, and the decomposition of the value function into the local nodes of a hierarchy creates further opportunities for parallel computation.

A primary motivation for the development of Fitted R-MAXQ was to serve as a foundation for research into the automatic discovery of abstractions and hierarchical structure (Chapter 8). By design, many of the algorithm's features make it an attractive candidate for such work. Reasoning explicitly about models may guide abstraction discovery, in much the same way that model uncertainty can drive exploration. The algorithm's hierarchical exploration policy is purely a function of the current data and task hierarchy, so a discovery algorithm can freely change the hierarchy or evaluate two hierarchies in parallel, in an "off-policy" fashion. Function approximation allows Fitted R-MAXQ to scale to more compelling problems, and varying degrees of generalization create more opportunities for discovery. Policies learned with a coarse approximation can determine abstractions that permit more efficient learning with otherwise finer approximations.

The preliminary abstraction discovery technique that this thesis contributed suggests some concrete steps towards investigating abstraction discovery using Fitted R-MAXQ.

Discover primitive action state representations Fitted R-MAXQ assumes that the state representation of each primitive action is known, and it defines the state representations of abstract actions recursively, using the union of each subtasks' state features. To remove the need for this prior knowledge, the agent must automatically discover the relevant state features of the primitive actions. One approach to this problem is to apply statistical hypothesis testing to determine whether the successor state distribution is identical for two given states, although this approach requires sufficient data for each state-action to test.

Propagate uncertainty hierarchically Fitted R-MAXQ combines exploration and hierarchical structure by propagating optimistic values up the hierarchy. Other exploration methods that reason more explicitly about model uncertainty, such as Model-Based Interval Estimation (Strehl & Littman, 2005) and Bayesian RL (Dearden et al., 1999), may both help refine the exploration behavior of Fitted R-MAXQ and also provide measures of uncertainty that can guide abstraction discovery.

Generate tasks as sets of actions Chapters 4 and 5 suggest that a task's utility depends as much on its subtasks as its goal. Given known uncertainty in each primitive action model and using some of the techniques described in Chapter 8, an agent can find small subsets of the action space that include a near-optimal action for long segments of near-optimal trajectories through the state space. This technique can also apply to action spaces that include abstract actions, if the uncertainty in their behavior has been propagated up from primitive actions.

Generate termination predicates for tasks Given a set of actions, an agent can create a task by determining the conditions under which the agent might limit its behavior to those actions. As in Chapter 8, the agent might learn a classifier given sampled trajectories. The agent can specify the task's goal function, which determines the task policy within the state region identified by the learned classifier, by examining at which states near-optimal trajectories exist the region.

Abstraction evaluation As discussed in Chapter 5, the primary effect of both forms of abstraction is to reduce the amount of exploration the agent attempts.[2] The agent must trade off this benefit against the possibility that the abstractions only appear safe due to a small sample size. In episodic tasks, the agent can confirm that its average return doesn't suffer when adopting the abstractions. In general, the agent might occasionally explore the space of representations by temporarily relaxing the abstractions it currently exploits. This ongoing verification process may be particularly important in the presence of generalization, especially if the agent gradually decreases the approximation granularity used by Fitted R-MAXQ. Abstractions that seem safe at a coarse granularity may prevent effective learning at a finer granularity.

[2]This benefit assumes the agent abstacts its action space using subtasks instead of merely augmenting it

Empirical validation One challenge for Fitted R-MAXQ is to demonstrate effectiveness on more complex problems. A desire to extend RL to more realistic domains motivated this combination of models, approximation, and hierarchy, but it has only been applied to environments artifically designed to include elements of hierarchical structure yet remain particularly tractable. Some compelling problem domains that exist outside of RL research and which seem within the reach of Fitted R-MAXQ include robot navigation and computer games. For example, hierarchies and policies discovered in a simulator could guide a surveillance robot's patrol pattern, or a Pac-Man agent's subtasks might include `CollectPowerPellet` or `FleeFromNearbyGhost`.

In the longer term, the discovery of abstract actions with learned termination predicates (and possibly initiation sets) could potentially join RL with classical planning algorithms. Model-based approaches already join together RL with MDP planning, and Fitted R-MAXQ demonstrates that a domain with stochastic actions in a continuous state space can be abstracted with subtasks that deterministically (eventually) achieve discrete subgoals. Although the present algorithm represents these subgoals as concrete states sampled from the original state space, one can envision defining propositions that correspond to the preconditions and postconditions of these abstract actions.

A scientific agent of this kind has been the guiding vision for this thesis. Such an agent doesn't just estimate parameters given data. It actively explores to gather the raw data from which it constructs models that facilitate planning.

The resulting policies induce abstract values and state transitions that serve as the foundation for another layer of planning. In this manner, perhaps, an agent can rise from low-level experience to high-level understanding of its world.

Bibliography

Andre, D., & Russell, S. J. (2002). State abstraction for programmable reinforcement learning agents. *Proceedings of the Eighteenth National Conference on Artificial Intelligence* (pp. 119–125).

Atkeson, C. G., Moore, A. W., & Schaal, S. (1997). Locally weighted learning for control. *Artificial Intelligence Review*, *11*, 75–113.

Barto, A. G., & Mahadevan, S. (2003). Recent advances in hierarchical reinforcement learning. *Discrete-Event Systems*, *13*, 41–77. Special Issue on Reinforcement Learning.

Beygelzimer, A., Kakade, S., & Langford, J. (2006). Cover trees for nearest neighbor. *Proceedings of the Twenty-Third International Conference on Machine Learning*.

Boutilier, C., Dearden, R., & Goldszmidt, M. (1995). Exploiting structure in policy construction. *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence* (pp. 1104–1111).

Boyan, J. A., & Moore, A. W. (1995). Generalization in reinforcement learning: Safely approximating the value function. *Advances in Neural Information Processing Systems 7*.

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.

Butz, M. (2004). Effective online detection of task-independent landmarks. *Proceedings for the ICML-04 Workshop on Predictive Representations of World Knowledge*.

Cohen, W. W. (1995). Fast effective rule induction. *Proceedings of the Twelfth International Conference on Machine Learning* (pp. 115–123).

Dai, P., & Goldsmith, J. (2007). Topological value iteration algorithm for Markov decision processes. *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*.

Dai, P., Strehl, A. L., & Goldsmith, J. (2008). Expediting RL by using graphical structures. *Proceedings of the Seventh International Joint Conference on Autonomous Agents and Multiagent Systems*.

Dean, T., & Givan, R. (1997). Model minimization in Markov decision processes. *Proceedings of the Fourteenth National Conference on Artificial Intelligence*.

Dearden, R., Friedman, N., & Andre, D. (1999). Model based Bayesian exploration. *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence* (pp. 150–159).

Degroot, M. H. (1986). *Probability and statistics*. Addison-Wesley Pub Co. 2nd edition.

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

Dietterich, T. G. (2000b). An overview of MAXQ hierarchical reinforcement learning. *Proceedings of the Fourth Symposium on Abstraction, Reformulation and Approximation*.

Diuk, C., Strehl, A. L., & Littman, M. L. (2006). A hierarchical approach to efficient reinforcement learning in deterministic domains. *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems*.

Dutech, A., Edmunds, T., Kok, J., Lagoudakis, M., Littman, M., Riedmiller, M., Russell, B., Scherrer, B., Sutton, R., Timmer, S., Vlassis, N., White, A., & Whiteson, S. (2005). Reinforcement learning benchmarks and bake-offs II. http://www.cs.rutgers.edu/˜mlittman/topics/nips05-mdp/bakeoffs05.pdf.

Glaubius, R., & Smart, W. D. (2004). Manifold representations for value-function approximation. *Proceedings of the Nineteenth National Conference on Artificial Intelligence*.

Gordon, G. J. (1995). Stable function approximation in dynamic programming. *Proceedings of the Twelfth International Conference on Machine Learning*.

Hauskrecht, M., Meuleau, N., Kaelbling, L. P., Dean, T., & Boutilier, C. (1998). Hierarchical solution of Markov decision processes using macro-actions. *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence* (pp. 220–229).

Hengst, B. (2002). Discovering hierarchy in reinforcement learning with HEXQ. *Proceedings of the Nineteenth International Conference on Machine Learning*.

Hester, T., & Stone, P. (2009). Generalized model learning for reinforcement learning in factored domains. *Proceedings of the Eighth International Joint Conference on Autonomous Agents and Multiagent Systems*.

Hester, T., & Stone, P. (2010). Real time targeted exploration in large domains. *The Ninth International Conference on Development and Learning*.

Jong, N. K., Hester, T., & Stone, P. (2008). The utility of temporal abstraction in reinforcement learning. *Proceedings of the Seventh International Joint Conference on Autonomous Agents and Multiagent Systems*.

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

Jong, N. K., & Stone, P. (2007a). Model-based exploration in continuous state spaces. *Proceedings of the Seventh Symposium on Abstraction, Reformulation and Approximation*.

Jong, N. K., & Stone, P. (2007b). Model-based function approximation in reinforcement learning. *Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multiagent Systems.*

Jong, N. K., & Stone, P. (2008). Hierarchical model-based reinforcement learning: R-MAX + MAXQ. *Proceedings of the Twenty-Fifth International Conference on Machine Learning.*

Jong, N. K., & Stone, P. (2009). Compositional models for reinforcement learning. *Proceedings of the European Conference on Machine Learning and Practice of Knowledge Discovery in Databases.*

Kakade, S., Kearns, M., & Langford, J. (2003). Exploration in metric state spaces. *Proceedings of the Twentieth International Conference on Machine Learning.*

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

Kalmár, Z., & Szepesvári, C. (1999). *An evaluation criterion for macro learning and some results* (Technical Report TR-99-01). Mindmaker Ltd., Budapest 1121, Konkoly Th. M. u. 29-33, HUNGARY.

Kearns, M., Mansour, Y., & Ng, A. Y. (1999). A sparse sampling algorithm for near-optimal planning in large Markov decision processes. *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence* (pp. 1324–1331).

Kearns, M., & Singh, S. (1998). Near-optimal reinforcement learning in polynomial time. *Proceedings of the Fifteenth International Conference on Machine Learning* (pp. 260–268).

Keller, P. W., Mannor, S., & Precup, D. (2006). Automatic basis function construction for approximate dynamic programming and reinforcement learning. *Proceedings of the Twenty-Third International Conference on Machine Learning*.

Koller, D., & Parr, R. (2000). Policy iteration for factored MDPs. *Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence* (pp. 326–334).

Korf, R. E. (1985). *Learning to solve problems by seaching for macro-operators*. Boston, MA: Pitman.

Lagoudakis, M. G., & Parr, R. (2003). Least-squares policy iteration. *Journal of Machine Learning Research*, *4*, 1107–1149.

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*.

Lin, L.-J. (1992). Self-improving reactive agents based on reinforcement learning, planning, and teaching. *Machine Learning*, *8*, 293–321.

Littman, M. L., Dean, T. L., & Kaelbling, L. P. (1995). On the complexity of solving Markov decision problems. *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*.

Mahadevan, S. (2005). Samuel meets Amarel: Automating value function approximation using global state space analysis. *Proceedings of the Twentieth National Conference on Artificial Intelligence*.

Mannor, S., Menache, I., Hoze, A., & Klein, U. (2004). Dynamic abstraction in reinforcement learning via clustering. *Proceedings of the Twenty-First International Conference on Machine Learning* (pp. 560–567).

McCallum, A. K. (1995). *Reinforcement learning with selective perception and hidden state*. Doctoral dissertation, University of Rochester.

McGovern, A., & Barto, A. G. (2001). Automatic discovery of subgoals in reinforcement learning using diverse density. *Proceedings of the Eighteenth International Conference on Machine Learning* (pp. 361–368).

Mehta, N., Ray, S., Tadepalli, P., & Dietterich, T. (2008). Automatic discovery and transfer of MAXQ hierarchies. *Proceedings of the Twenty-Fifth International Conference on Machine Learning*.

Minton, S. (1988). Quantitative results concerning the utility of explanation-based learning. *Proceedings of the Seventh National Conference on Artificial Intelligence* (pp. 564–569).

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., & Szepesvári, C. (2008). Finite-time bounds for fitted value iteration. *Journal of Machine Learning Research*, *1*, 815–857.

Ormoneit, D., & Sen, Ś. (2002). Kernel-based reinforcement learning. *Machine Learning*, *49*, 161–178.

Parr, R., Li, L., Taylor, G., Painter-Wakefield, C., & Littman, M. (2008). An analysis of linear models, linear value-function approximation, and feature selection for reinforcement learning. *Proceedings of the Twenty-Fifth International Conference on Machine Learning.*

Parr, R., & Russell, S. (1998). Reinforcement learning with hierarchies of machines. *Advances in Neural Information Processing Systems 10.*

Ravindran, B., & Barto, A. G. (2003). SMDP homomorphisms: An algebraic approach to abstraction in semi-Markov decision processes. *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence.*

Riedmiller, M. (2005). Neural fitted Q iteration – first experiences with a data efficient neural reinforcement learning method. *Proceedings of the European Conference on Machine Learning.*

Russell, S. J., & Norvig, P. (1995). *Artificial intelligence: A modern approach.* Prentice-Hall, Inc.

Seri, S., & Tadepalli, P. (2002). Model-based hierarchical average-reward reinforcement learning. *Proceedings of the Nineteenth International Conference on Machine Learning* (pp. 562–569).

Şimşek, Ö., & Barto, A. G. (2004). Using relative novelty to identify useful temporal abstractions in reinforcement learning. *Proceedings of the Twenty-First International Conference on Machine Learning* (pp. 751–758).

Singh, S., Barto, A. G., & Chentanez, N. (2005). Intrinsically motivated reinforcement learning. *Advances in Neural Information Processing Systems 17*.

Stanley, K. O., & Miikkulainen, R. (2002). Efficient reinforcement learning through evolving neural network topologies. *Proceedings of the Genetic and Evolutionary Computation Conference*.

Stolle, M., & Precup, D. (2002). Learning options in reinforcement learning. *Proceedings of the Fifth Symposium on Abstraction, Reformulation and Approximation*.

Strehl, A. L., Li, L., Wiewiora, E., Langford, J., & Littman, M. L. (2006). PAC model-free reinforcement learning. *Proceedings of the Twenty-Third International Conference on Machine Learning* (pp. 881–888).

Strehl, A. L., & Littman, M. L. (2005). A theoretical analysis of model-based interval estimation. *Proceedings of the Twenty-Second International Conference on Machine Learning* (pp. 857–864).

Strens, M. (2000). A Bayesian framework for reinforcement learning. *Proceedings of the Seventeenth International Conference on Machine Learning* (pp. 943–950).

Sutton, R. S. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. *Advances in Neural Information Processing Systems 8*.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning: An introduction*. Cambridge, MA: MIT Press.

Sutton, R. S., Precup, D., & Singh, S. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, *112*, 181–211.

Sutton, R. S., Szepesvári, C., Geramifard, A., & Bowling, M. H. (2008). Dyna-style planning with linear function approximation and prioritized sweeping. *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence* (pp. 528–536).

Tanner, B., & White, A. (2009). RL-Glue: Language-independent software for reinforcement-learning experiments. *Journal of Machine Learning Research*, *10*, 2133–2136.

Tesauro, G. (1995). Temporal difference learning and TD-Gammon. *Communications of the ACM*, *38*.

Tesauro, G., Jong, N. K., Das, R., & Bennani, M. N. (2007). On the use of hybrid reinforcement learning for autonomic resource allocation. *Cluster Computing*, *10*.

Thrun, S., & Schwartz, A. (1995). Finding structure in reinforcement learning. *Advances in Neural Information Processing Systems 7*.

Watkins, C. (1989). *Learning from delayed rewards*. Doctoral dissertation, University of Cambridge.

Whiteson, S., & Stone, P. (2006). Evolutionary function approximation for reinforcement learning. *Journal of Machine Learning Research*, *7*, 877–917.

Vita

Nicholas Kenneth Jong was born in Summit, New Jersey, to Kenneth and Nancy Jong. He attended Bridgewater-Raritan High School, graduating in 1998. At Carnegie Mellon University, he earned a Bachelor of Science degree with majors in Computer Science and Psychology and minors in English and Mathematics. Later that year, in the fall of 2002, he joined the PhD program at the Department of Computer Sciences in the University of Texas at Austin, where he served as a teaching assistant, research assistant, and assistant instructor. He also worked at the IBM TJ Watson Research Center during a summer internship. Currently, he works in Cupertino, California, as a software enginner for Apple.

Permanent address: 777 Mountain View Avenue
Bridgewater, New Jersey 08807

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.