Journal of Artificial Intelligence Research 13 (2000) 227-303 Submitted 11/99; published 11/00

Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition

Thomas G. Dietterich TGD@CS.ORST.EDU
Department of Computer Science, Oregon State University
Corvallis, OR 97331

Abstract

This paper presents a new approach to hierarchical reinforcement learning based on decomposing the target Markov decision process (MDP) into a hierarchy of smaller MDPs and decomposing the value function of the target MDP into an additive combination of the value functions of the smaller MDPs. The decomposition, known as the MAXQ decomposition, has both a procedural semantics—as a subroutine hierarchy—and a declarative semantics—as a representation of the value function of a hierarchical policy. MAXQ unifies and extends previous work on hierarchical reinforcement learning by Singh, Kaelbling, and Dayan and Hinton. It is based on the assumption that the programmer can identify useful subgoals and define subtasks that achieve these subgoals. By defining such subgoals, the programmer constrains the set of policies that need to be considered during reinforcement learning. The MAXQ value function decomposition can represent the value function of any policy that is consistent with the given hierarchy. The decomposition also creates opportunities to exploit state abstractions, so that individual MDPs within the hierarchy can ignore large parts of the state space. This is important for the practical application of the method. This paper defines the MAXQ hierarchy, proves formal results on its representational power, and establishes five conditions for the safe use of state abstractions. The paper presents an online model-free learning algorithm, MAXQ-Q, and proves that it converges with probability 1 to a kind of locally-optimal policy known as a recursively optimal policy, even in the presence of the five kinds of state abstraction. The paper evaluates the MAXQ representation and MAXQ-Q through a series of experiments in three domains and shows experimentally that MAXQ-Q (with state abstractions) converges to a recursively optimal policy much faster than flat Q learning. The fact that MAXQ learns a representation of the value function has an important benefit: it makes it possible to compute and execute an improved, non-hierarchical policy via a procedure similar to the policy improvement step of policy iteration. The paper demonstrates the effectiveness of this non-hierarchical execution experimentally. Finally, the paper concludes with a comparison to related work and a discussion of the design tradeoffs in hierarchical reinforcement learning.

1. Introduction

The area of Reinforcement Learning (Bertsekas & Tsitsiklis, 1996; Sutton & Barto, 1998) studies methods by which an agent can learn optimal or near-optimal plans by interacting directly with the external environment. The basic methods in reinforcement learning are based on the classical dynamic programming algorithms that were developed in the late 1950s (Bellman, 1957; Howard, 1960). However, reinforcement learning methods offer two important advantages over classical dynamic programming. First, the methods are online. This permits them to focus their attention on the parts of the state space that are important and to ignore the rest of the space. Second, the methods can employ function approximation algorithms (e.g., neural networks) to represent their knowledge. This allows them to generalize across the state space so that the learning time scales much better.

Despite recent advances in reinforcement learning, there are still many shortcomings. The biggest of these is the lack of a fully satisfactory method for incorporating hierarchies into reinforcement learning algorithms. Research in classical planning has shown that hierarchical methods such as hierarchical task networks (Currie & Tate, 1991), macro actions (Fikes, Hart, & Nilsson, 1972; Korf, 1985), and state abstraction methods (Sacerdoti, 1974; Knoblock, 1990) can provide exponential reductions in the computational cost of finding good plans. However, all of the basic algorithms for probabilistic planning and reinforcement learning are "flat" methods—they treat the state space as one huge flat search space. This means that the paths from the start state to the goal state are very long, and the length of these paths determines the cost of learning and planning, because information about future rewards must be propagated backward along these paths.

Many researchers (Singh, 1992; Lin, 1993; Kaelbling, 1993; Dayan & Hinton, 1993; Hauskrecht, et al., 1998; Parr & Russell, 1998; Sutton, Precup, & Singh, 1998) have experimented with different methods of hierarchical reinforcement learning and hierarchical probabilistic planning. This research has explored many different points in the design space of hierarchical methods, but several of these systems were designed for specific situations. We lack crisp definitions of the main approaches and a clear understanding of the relative merits of the different methods.

This paper formalizes and clarifies one approach and attempts to understand how it compares with the other techniques. The approach, called the MAXQ method, provides a hierarchical decomposition of the given reinforcement learning problem into a set of subproblems. It simultaneously provides a decomposition of the value function for the given problem into a set of value functions for the subproblems. Hence, it has both a declarative semantics (as a value function decomposition) and a procedural semantics (as a subroutine hierarchy).

The decomposition into subproblems has many advantages. First, policies learned in subproblems can be shared (reused) for multiple parent tasks. Second, the value functions learned in subproblems can be shared, so when the subproblem is reused in a new task, learning of the overall value function for the new task is accelerated. Third, if state abstractions can be applied, then the overall value function can be represented compactly as the sum of separate terms that each depends on only a subset of the state variables. This more compact representation of the value function will require less data to learn, and hence, learning will be faster.

Previous research shows that there are several important design decisions that must be made when constructing a hierarchical reinforcement learning system. To provide an overview of the results in this paper, let us review these issues and see how the MAXQ method approaches each of them.

The first issue is how to specify subtasks. Hierarchical reinforcement learning involves breaking the target Markov decision problem into a hierarchy of subproblems or subtasks. There are three general approaches to defining these subtasks. One approach is to define each subtask in terms of a fixed policy that is provided by the programmer (or that has been learned in some separate process). The "option" method of Sutton, Precup, and Singh (1998) takes this approach. The second approach is to define each subtask in terms of a non-deterministic finite-state controller. The Hierarchy of Abstract Machines (HAM) method of Parr and Russell (1998) takes this approach. This method permits the programmer to provide a "partial policy" that constrains the set of permitted actions at each point, but does not specify a complete policy for each subtask. The third approach is to define each subtask in terms of a termination predicate and a local reward function. These define what it means for the subtask to be completed and what the final reward should be for completing the subtask. The MAXQ method described in this paper follows this approach, building upon previous work by Singh (1992), Kaelbling (1993), Dayan and Hinton (1993), and Dean and Lin (1995).

An advantage of the "option" and partial policy approaches is that the subtask can be defined in terms of an amount of effort or a course of action rather than in terms of achieving a particular goal condition. However, the "option" approach (at least in the simple form described in this paper), requires the programmer to provide complete policies for the subtasks, which can be a difficult programming task in real-world problems. On the other hand, the termination predicate method requires the programmer to guess the relative desirability of the different states in which the subtask might terminate. This can also be difficult, although Dean and Lin show how these guesses can be revised automatically by the learning algorithm.

A potential drawback of all hierarchical methods is that the learned policy may be suboptimal. The hierarchy constrains the set of possible policies that can be considered. If these constraints are poorly chosen, the resulting policy will be suboptimal. Nonetheless, the learning algorithms that have been developed for the "option" and partial policy approaches guarantee that the learned policy will be the best possible policy consistent with these constraints.

The termination predicate method suffers from an additional source of suboptimality. The learning algorithm described in this paper converges to a form of local optimality that we call *recursive optimality*. This means that the policy of each subtask is locally optimal given the policies of its children. But there might exist better hierarchical policies where the policy for a subtask must be locally suboptimal so that the overall policy is optimal. For example, a subtask of buying milk might be performed suboptimally (at a more distant store) because the larger problem also involves buying film (at the same store). This problem can be avoided by careful definition of termination predicates and local reward functions, but this is an added burden on the programmer. (It is interesting to note that this problem of recursive optimality has not been noticed previously. This is because previous work focused on subtasks with a single terminal state, and in such cases, the problem does not arise.)

The second design issue is whether to employ state abstractions within subtasks. A subtask employs state abstraction if it ignores some aspects of the state of the environment. For example, in many robot navigation problems, choices about what route to take to reach a goal location are independent of what the robot is currently carrying. With few exceptions, state abstraction has not been explored previously. We will see that the MAXQ method creates many opportunities to exploit state abstraction, and that these abstractions can have a huge impact in accelerating learning. We will also see that there is an important design tradeoff: the successful use of state abstraction requires that subtasks be defined in terms of termination predicates rather than using the option or partial policy methods. This is why the MAXQ method must employ termination predicates, despite the problems that this can create.

The third design issue concerns the non-hierarchical "execution" of a learned hierarchical policy. Kaelbling (1993) was the first to point out that a value function learned from a hierarchical policy could be evaluated incrementally to yield a potentially much better non-hierarchical policy. Dietterich (1998) and Sutton, et al. (1999) generalized this to show how arbitrary subroutines could be executed non-hierarchically to yield improved policies. However, in order to support this non-hierarchical execution, extra learning is required. Ordinarily, in hierarchical reinforcement learning, the only states where learning is required at the higher levels of the hierarchy are states where one or more of the subroutines could terminate (plus all possible initial states). But to support non-hierarchical execution, learning is required in all states (and at all levels of the hierarchy). In general, this requires additional exploration as well as additional computation and memory. As a consequence of the hierarchical decomposition of the value function, the MAXQ method is able to support either form of execution, and we will see that there are many problems where the improvement from non-hierarchical execution is worth the added cost.

The fourth and final issue is what form of learning algorithm to employ. An important advantage of reinforcement learning algorithms is that they typically operate online. However, finding online algorithms that work for general hierarchical reinforcement learning has been difficult, particularly within the termination predicate family of methods. Singh's method relied on each subtask having a unique terminal state; Kaelbling employed a mix of online and batch algorithms to train her hierarchy; and work within the "options" framework usually assumes that the policies for the subproblems are given and do not need to be learned at all. The best previous online algorithms are the HAMQ Q learning algorithm of Parr and Russell (for the partial policy method) and the Feudal Q algorithm of Dayan and Hinton. Unfortunately, the HAMQ method requires "flattening" the hierarchy, and this has several undesirable consequences. The Feudal Q algorithm is tailored to a specific kind of problem, and it does not converge to any well-defined optimal policy.

In this paper, we present a general algorithm, called MAXQ-Q, for fully-online learning of a hierarchical value function. This algorithm enables all subtasks within the hierarchy to be learned simultaneously and online. We show experimentally and theoretically that the algorithm converges to a recursively optimal policy. We also show that it is substantially faster than "flat" (i.e., non-hierarchical) Q learning when state abstractions are employed.

The remainder of this paper is organized as follows. After introducing our notation in Section 2, we define the MAXQ value function decomposition in Section 3 and illustrate it with a simple example Markov decision problem. Section 4 presents an analytically tractable version of the MAXQ-Q learning algorithm called the MAXQ-0 algorithm and proves its convergence to a recursively optimal policy. It then shows how to extend MAXQ-0 to produce the MAXQ-Q algorithm, and shows how to extend the theorem similarly. Section 5 takes up the issue of state abstraction and formalizes a series of five conditions under which state abstractions can be safely incorporated into the MAXQ representation. State abstraction can give rise to a hierarchical credit assignment problem, and the paper briefly discusses one solution to this problem. Finally, Section 7 presents experiments with three example domains. These experiments give some idea of the generality of the MAXQ representation. They also provide results on the relative importance of temporal and state abstractions and on the importance of non-hierarchical execution. The paper concludes with further discussion of the design issues that were briefly described above, and in particular, it addresses the tradeoff between the method of defining subtasks (via termination predicates) and the ability to exploit state abstractions.

Some readers may be disappointed that MAXQ provides no way of learning the structure of the hierarchy. Our philosophy in developing MAXQ (which we share with other reinforcement learning researchers, notably Parr and Russell) has been to draw inspiration from the development of Belief Networks (Pearl, 1988). Belief networks were first introduced as a formalism in which the knowledge engineer would describe the structure of the networks and domain experts would provide the necessary probability estimates. Subsequently, methods were developed for learning the probability values directly from observational data. Most recently, several methods have been developed for learning the structure of the belief networks from data, so that the dependence on the knowledge engineer is reduced.

In this paper, we will likewise require that the programmer provide the structure of the hierarchy. The programmer will also need to make several important design decisions. We will see below that a MAXQ representation is very much like a computer program, and we will rely on the programmer to design each of the modules and indicate the permissible ways in which the modules can invoke each other. Our learning algorithms will fill in "implementations" of each module in such a way that the overall program will work well. We believe that this approach will provide a practical tool for solving large real-world MDPs. We also believe that it will help us understand the structure of hierarchical learning algorithms. It is our hope that subsequent research will be able to automate most of the work that we are currently requiring the programmer to do.

2. Formal Definitions

We begin by introducing definitions for Markov Decision Problems and Semi-Markov Decision Problems.

2.1 Markov Decision Problems

We employ the standard definition for Markov Decision Problems (also known as Markov decision processes). In this paper, we restrict our attention to situations in which an agent is interacting with a fully-observable stochastic environment. This situation can be modeled as a Markov Decision Problem (MDP) $\langle S, A, P, R, P_0 \rangle$ defined as follows:

- S: the finite set of states of the environment. At each point in time, the agent can observe the complete state of the environment.
- A: a finite set of actions. Technically, the set of available actions depends on the current state s, but we will suppress this dependence in our notation.
- P: When an action $a \in A$ is performed, the environment makes a probabilistic transition from its current state s to a resulting state s' according to the probability distribution $P(s'|s, a)$.
- R: Similarly, when action a is performed and the environment makes its transition from s to s', the agent receives a real-valued (possibly stochastic) reward r whose expected value is $R(s'|s, a)$. To simplify the notation, it is customary to treat this reward as being given at the time that action a is initiated, even though it may in general depend on s' as well as on s and a.
- P_0: The starting state distribution. When the MDP is initialized, it is in state s with probability $P_0(s)$.

A *policy*, π, is a mapping from states to actions that tells what action $a = \pi(s)$ to perform when the environment is in state s.

We will consider two settings: episodic and infinite-horizon.

In the episodic setting, all rewards are finite and there is at least one zero-cost absorbing terminal state. An absorbing terminal state is a state in which all actions lead back to the same state with probability 1 and zero reward. For technical reasons, we will only consider problems where all deterministic policies are "proper"—that is, all deterministic policies have a non-zero probability of reaching a terminal state when started in an arbitrary state. (We believe this condition can be relaxed, but we have not verified this formally.) In the episodic setting, the goal of the agent is to find a policy that maximizes the expected cumulative reward. In the special case where all rewards are non-positive, these problems are referred to as stochastic shortest path problems, because the rewards can be viewed as costs (i.e., lengths), and the policy attempts to move the agent along the path of minimum expected cost.

In the infinite horizon setting, all rewards are also finite. In addition, there is a discount factor γ, and the agent's goal is to find a policy that minimizes the infinite discounted sum of future rewards.

The value function V^π for policy π is a function that tells, for each state s, what the expected cumulative reward will be of executing policy π starting in state s. Let r_t be a random variable that tells the reward that the agent receives at time step t while following policy π. We can define the value function in the episodic setting as

$$V^\pi(s) = E\left\{r_t + r_{t+1} + r_{t+2} + \cdots | s_t = s, \pi\right\}.$$

In the discounted setting, the value function is

$$V^\pi(s) = E\left\{r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots \middle| s_t = s, \pi\right\}.$$

We can see that this equation reduces to the previous one when $\gamma = 1$. However, in infinite-horizon MDPs this sum may not converge when $\gamma = 1$.

The value function satisfies the Bellman equation for a fixed policy:

$$V^\pi(s) = \sum_{s'} P(s'|s,\pi(s)) \left[R(s'|s,\pi(s)) + \gamma V^\pi(s')\right].$$

The quantity on the right-hand side is called the *backed-up value* of performing action a in state s. For each possible successor state s', it computes the reward that would be received and the value of the resulting state and then weights those according to the probability of ending up in s'.

The optimal value function V^* is the value function that simultaneously maximizes the expected cumulative reward in all states $s \in S$. Bellman (1957) proved that it is the unique solution to what is now known as the Bellman equation:

$$V^*(s) = \max_a \sum_{s'} P(s'|s,a) \left[R(s'|s,a) + \gamma V^*(s')\right]. \tag{1}$$

There may be many optimal policies that achieve this value. Any policy that chooses a in s to achieve the maximum on the right-hand side of this equation is an optimal policy. We will denote an optimal policy by π^*. Note that all optimal policies are "greedy" with respect to the backed-up value of the available actions.

Closely related to the value function is the so-called *action-value function*, or Q function (Watkins, 1989). This function, $Q^\pi(s,a)$, gives the expected cumulative reward of performing action a in state s and then following policy π thereafter. The Q function also satisfies a Bellman equation:

$$Q^\pi(s,a) = \sum_{s'} P(s'|s,a) \left[R(s'|s,a) + \gamma Q^\pi(s',\pi(s'))\right].$$

The optimal action-value function is written $Q^*(s,a)$, and it satisfies the equation

$$Q^*(s,a) = \sum_{s'} P(s'|s,a) \left[R(s'|s,a) + \gamma \max_{a'} Q^*(s',a')\right]. \tag{2}$$

Note that any policy that is greedy with respect to Q^* is an optimal policy. There may be many such optimal policies—they differ only in how they break ties between actions with identical Q^* values.

An *action order*, denoted ω, is a total order over the actions within an MDP. That is, ω is an anti-symmetric, transitive relation such that $\omega(a_1, a_2)$ is true iff a_1 is strictly preferred to a_2. An *ordered greedy policy*, π_ω is a greedy policy that breaks ties using ω. For example, suppose that the two best actions at state s are a_1 and a_2, that $Q(s,a_1) = Q(s,a_2)$, and that $\omega(a_1,a_2)$. Then the ordered greedy policy π_ω will choose a_1: $\pi_\omega(s) = a_1$. Note that although there may be many optimal policies for a given MDP, the ordered greedy policy, π^*_ω, is unique.

2.2 Semi-Markov Decision Processes

In order to introduce and prove some of the properties of the MAXQ decomposition, we need to consider a simple generalization of MDPs—the semi-Markov decision process.

A discrete-time *semi-Markov Decision Process* (SMDP) is a generalization of the Markov Decision Process in which the actions can take a variable amount of time to complete. In particular, let the random variable N denote the number of time steps that action a takes when it is executed in state s. We can extend the state transition probability function to be the joint distribution of the result states s' and the number of time steps N when action a is performed in state s: $P(s', N|s, a)$. Similarly, the expected reward can be changed to be $R(s', N|s, a)$.[1]

It is straightforward to modify the Bellman equation to define the value function for a fixed policy π as

$$V^\pi(s) = \sum_{s',N} P(s', N|s, \pi(s)) \left[R(s', N|s, \pi(s)) + \gamma^N V^\pi(s') \right].$$

The only change is that the expected value on the right-hand side is taken with respect to both s' and N, and γ is raised to the power N to reflect the variable amount of time that may elapse while executing action a.

Note that because expectation is a linear operator, we can write each of these Bellman equations as the sum of the expected reward for performing action a and the expected value of the resulting state s'. For example, we can rewrite the equation above as

$$V^\pi(s) = \overline{R}(s, \pi(s)) + \sum_{s',N} P(s', N|s, \pi(s)) \gamma^N V^\pi(s'). \tag{3}$$

where $\overline{R}(s, \pi(s))$ is the expected reward of performing action $\pi(s)$ in state s, and the expectation is taken with respect to s' and N.

All of the results given in this paper can be generalized to apply to discrete-time semi-Markov Decision Processes. A consequence of this is that whenever this paper talks of executing a primitive action, it could just as easily talk of executing a hand-coded open-loop "subroutine". These subroutines would not be learned, and nor could their execution be interrupted as discussed below in Section 6. But in many applications (e.g., robot control with limited sensors), open-loop controllers can be very useful (e.g., to hide partial-observability). For an example, see Kalmár, Szepesvári, and A. Lörincz (1998).

Note that for the episodic case, there is no difference between a MDP and a Semi-Markov Decision Process, because the discount factor γ is 1, and therefore neither the optimal policy nor the optimal value function depend on the amount of time each action takes.

2.3 Reinforcement Learning Algorithms

A reinforcement learning algorithm is an algorithm that tries to construct an optimal policy for an unknown MDP. The algorithm is given access to the unknown MDP via the following

1. This formalization is slightly different from the standard formulation of SMDPs, which separates $P(s'|s, a)$ and $F(t|s, a)$, where F is the cumulative distribution function for the probability that a will terminate in t time units, and t is real-valued rather than integer-valued. In our case, it is important to consider the joint distribution of s' and N, but we do not need to consider actions with arbitrary real-valued durations.

reinforcement learning protocol. At each time step t, the algorithm is told the current state s of the MDP and the set of actions $A(s) \subseteq A$ that are executable in that state. The algorithm chooses an action $a \in A(s)$, and the MDP executes this action (which causes it to move to state s') and returns a real-valued reward r. If s is an absorbing terminal state, the set of actions $A(s)$ contains only the special action reset, which causes the MDP to move to one of its initial states, drawn according to P_0.

In this paper, we will make use of two well-known learning algorithms: Q learning (Watkins, 1989; Watkins & Dayan, 1992) and SARSA(0) (Rummery & Niranjan, 1994). We will apply these algorithms to the case where the action value function $Q(s,a)$ is represented as a table with one entry for each pair of state and action. Every entry of the table is initialized arbitrarily.

In Q learning, after the algorithm has observed s, chosen a, received r, and observed s', it performs the following update:

$$Q_t(s,a) := (1-\alpha_t)Q_{t-1}(s,a) + \alpha_t[r + \gamma \max_{a'} Q_{t-1}(s',a')],$$

where α_t is a learning rate parameter.

Jaakkola, Jordan and Singh (1994) and Bertsekas and Tsitsiklis (1996) prove that if the agent follows an "exploration policy" that tries every action in every state infinitely often and if

$$\lim_{T\to\infty} \sum_{t=1}^{T} \alpha_t = \infty \quad \text{and} \quad \lim_{\mathrm{T}\to\infty} \sum_{\mathrm{t}=1}^{\mathrm{T}} \alpha_{\mathrm{t}}^2 < \infty \tag{4}$$

then Q_t converges to the optimal action-value function Q^* with probability 1. Their proof holds in both settings discussed in this paper (episodic and infinite-horizon).

The SARSA(0) algorithm is very similar. After observing s, choosing a, observing r, observing s', and choosing a', the algorithm performs the following update:

$$Q_t(s,a) := (1-\alpha_t)Q_{t-1}(s,a) + \alpha_t[r + \gamma Q_{t-1}(s',a')],$$

where α_t is a learning rate parameter. The key difference is that the Q value of the chosen action a', $Q(s',a')$, appears on the right-hand side in the place where Q learning uses the Q value of the best action. Singh, et al. (1998) provide two important convergence results: First, if a fixed policy π is employed to choose actions, SARSA(0) will converge to the value function of that policy provided α_t decreases according to Equations (4). Second, if a so-called GLIE policy is employed to choose actions, SARSA(0) will converge to the value function of the optimal policy, provided again that α_t decreases according to Equations (4). A GLIE policy is defined as follows:

Definition 1 *A GLIE (Greedy in the Limit with Infinite Exploration) policy is any policy satisfying*

1. *Each action is executed infinitely often in every state that is visited infinitely often.*
2. *In the limit, the policy is greedy with respect to the Q-value function with probability 1.*

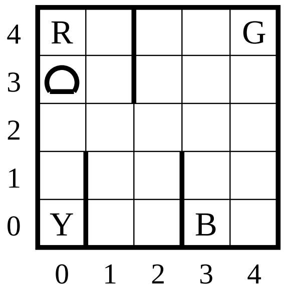

Figure 1: The Taxi Domain.

3. The MAXQ Value Function Decomposition

At the center of the MAXQ method for hierarchical reinforcement learning is the MAXQ value function decomposition. MAXQ describes how to decompose the overall value function for a policy into a collection of value functions for individual subtasks (and subsubtasks, recursively).

3.1 A Motivating Example

To make the discussion concrete, let us consider the following simple example. Figure 1 shows a 5-by-5 grid world inhabited by a taxi agent. There are four specially-designated locations in this world, marked as R(ed), B(lue), G(reen), and Y(ellow). The taxi problem is episodic. In each episode, the taxi starts in a randomly-chosen square. There is a passenger at one of the four locations (chosen randomly), and that passenger wishes to be transported to one of the four locations (also chosen randomly). The taxi must go to the passenger's location (the "source"), pick up the passenger, go to the destination location (the "destination"), and put down the passenger there. (To keep things uniform, the taxi must pick up and drop off the passenger even if he/she is already located at the destination!) The episode ends when the passenger is deposited at the destination location.

There are six primitive actions in this domain: (a) four navigation actions that move the taxi one square North, South, East, or West, (b) a Pickup action, and (c) a Putdown action. There is a reward of -1 for each action and an additional reward of $+20$ for successfully delivering the passenger. There is a reward of -10 if the taxi attempts to execute the Putdown or Pickup actions illegally. If a navigation action would cause the taxi to hit a wall, the action is a no-op, and there is only the usual reward of -1.

To simplify the examples throughout this section, we will make the six primitive actions deterministic. Later, we will make the actions stochastic in order to create a greater challenge for our learning algorithms.

We seek a policy that maximizes the total reward per episode. There are 500 possible states: 25 squares, 5 locations for the passenger (counting the four starting locations and the taxi), and 4 destinations.

This task has a simple hierarchical structure in which there are two main sub-tasks: Get the passenger and Deliver the passenger. Each of these subtasks in turn involves the subtask of navigating to one of the four locations and then performing a Pickup or Putdown action.

This task illustrates the need to support temporal abstraction, state abstraction, and subtask sharing. The temporal abstraction is obvious—for example, the process of navigating to the passenger's location and picking up the passenger is a temporally extended action that can take different numbers of steps to complete depending on the distance to the target. The top level policy (get passenger; deliver passenger) can be expressed very simply if these temporal abstractions can be employed.

The need for state abstraction is perhaps less obvious. Consider the subtask of getting the passenger. While this subtask is being solved, the destination of the passenger is completely irrelevant—it cannot affect any of the nagivation or pickup decisions. Perhaps more importantly, when navigating to a target location (either the source or destination location of the passenger), only the target location is important. The fact that in some cases the taxi is carrying the passenger and in other cases it is not is irrelevant.

Finally, support for subtask sharing is critical. If the system could learn how to solve the navigation subtask once, then the solution could be shared by both the "Get the passenger" and "Deliver the passenger" subtasks. We will show below that the MAXQ method provides a value function representation and learning algorithm that supports temporal abstraction, state abstraction, and subtask sharing.

To construct a MAXQ decomposition for the taxi problem, we must identify a set of individual subtasks that we believe will be important for solving the overall task. In this case, let us define the following four tasks:

- Navigate(t). In this subtask, the goal is to move the taxi from its current location to one of the four target locations, which will be indicated by the formal parameter t.
- Get. In this subtask, the goal is to move the taxi from its current location to the passenger's current location and pick up the passenger.
- Put. The goal of this subtask is to move the taxi from the current location to the passenger's destination location and drop off the passenger.
- Root. This is the whole taxi task.

Each of these subtasks is defined by a subgoal, and each subtask terminates when the subgoal is achieved.

After defining these subtasks, we must indicate for each subtask which other subtasks or primitive actions it should employ to reach its goal. For example, the Navigate(t) subtask should use the four primitive actions North, South, East, and West. The Get subtask should use the Navigate subtask and the Pickup primitive action, and so on.

All of this information can be summarized by a directed acyclic graph called the *task graph*, which is shown in Figure 2. In this graph, each node corresponds to a subtask or a primitive action, and each edge corresponds to a potential way in which one subtask can "call" one of its child tasks. The notation $formal/actual$ (e.g., $t/source$) tells how a formal parameter is to be bound to an actual parameter.

Now suppose that for each of these subtasks, we write a policy (e.g., as a computer program) to achieve the subtask. We will refer to the policy for a subtask as a "subroutine", and we can view the parent subroutine as invoking the child subroutine via ordinary

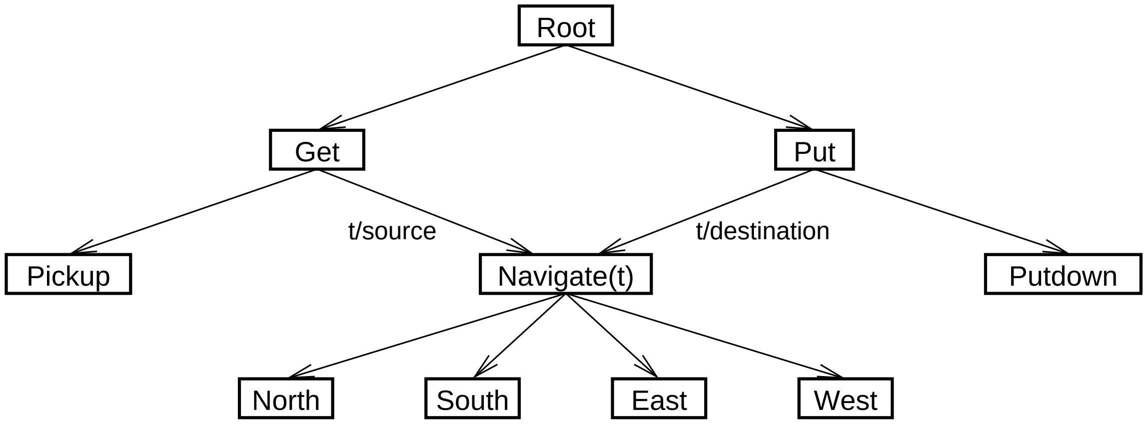

Figure 2: A task graph for the Taxi problem.

subroutine-call-and-return semantics. If we have a policy for each subtask, then this gives us an overall policy for the Taxi MDP. The Root subtask executes its policy by calling subroutines that are policies for the Get and Put subtasks. The Get policy calls subroutines for the Navigate(t) subtask and the Pickup primitive action. And so on. We will call this collection of policies a *hierarchical policy.* In a hierarchical policy, each subroutine executes until it enters a terminal state for its subtask.

3.2 Definitions

Let us formalize the discussion so far.

The MAXQ decomposition takes a given MDP M and decomposes it into a finite set of subtasks $\{M_0, M_1, \ldots, M_n\}$ with the convention that M_0 is the root subtask (i.e., solving M_0 solves the entire original MDP M).

Definition 2 *An* unparameterized subtask *is a three-tuple, $\langle T_i, A_i, \tilde{R}_i\rangle$, defined as follows:*

1. *T_i is a* termination predicate *that partitions S into a set of active states, S_i, and a set of terminal states, T_i. The policy for subtask M_i can only be executed if the current state s is in S_i. If, at any time that subtask M_i is being executed, the MDP enters a state in T_i, then M_i terminates immediately (even if it is still executing a subtask, see below).*

2. *A_i is a* set of actions *that can be performed to achieve subtask M_i. These actions can either be primitive actions from A, the set of primitive actions for the MDP, or they can be other subtasks, which we will denote by their indexes i. We will refer to these actions as the "children" of subtask i. The sets A_i define a directed graph over the subtasks $M_0, \ldots, M_n$, and this graph may not contain any cycles. Stated another way, no subtask can invoke itself recursively either directly or indirectly.*

   If a child subtask M_j has formal parameters, then this is interpreted as if the subtask occurred multiple times in A_i, with one occurrence for each possible tuple of actual *values that could be bound to the formal parameters. The set of actions A_i may differ from one state to another and from one set of actual parameter values to another, so technically, A_i is a function of s and the actual parameters. However, we will suppress this dependence in our notation.*

3. $\tilde{R}_i(s')$ *is the* pseudo-reward function, *which specifies a (deterministic)* pseudo-reward *for each transition to a terminal state $s' \in T_i$. This pseudo-reward tells how desirable each of the terminal states is for this subtask. It is typically employed to give goal terminal states a pseudo-reward of 0 and any non-goal terminal states a negative reward. By definition, the pseudo-reward $\tilde{R}_i(s)$ is also zero for all* non-terminal *states s. The pseudo-reward is only used during learning, so it will not be mentioned further until Section 4.*

Each primitive action a from M is a primitive subtask in the MAXQ decomposition such that a is always executable, it always terminates immediately after execution, and its pseudo-reward function is uniformly zero.

If a subtask has formal parameters, then each possible binding of actual values to the formal parameters specifies a distinct subtask. We can think of the values of the formal parameters as being part of the "name" of the subtask. In practice, of course, we implement a parameterized subtask by parameterizing the various components of the task. If b specifies the actual parameter values for task M_i, then we can define a parameterized termination predicate $T_i(s, b)$ and a parameterized pseudo-reward function $\tilde{R}_i(s', b)$. To simplify notation in the rest of the paper, we will usually omit these parameter bindings. However, it should be noted that if a parameter of a subtask takes on a large number of possible values, this is equivalent to creating a large number of different subtasks, each of which will need to be learned. It will also create a large number of candidate actions for the parent task, which will make the learning problem more difficult for the parent task as well.

Definition 3 *A* hierarchical policy, *π, is a set containing a policy for each of the subtasks in the problem: $\pi = \{\pi_0, \ldots, \pi_n\}$.*

Each subtask policy π_i takes a state and returns the name of a primitive action to execute or the name of a subroutine (and bindings for its formal parameters) to invoke. In the terminology of Sutton, Precup, and Singh (1998), a subtask policy is a deterministic "option", and its probability of terminating in state s (which they denote by $\beta(s)$) is 0 if $s \in S_i$, and 1 if $s \in T_i$.

In a parameterized task, the policy must be parameterized as well so that π takes a state and the bindings of formal parameters and returns a chosen action and the bindings (if any) of its formal parameters.

Table 1 gives a pseudo-code description of the procedure for executing a hierarchical policy. The hierarchical policy is executed using a stack discipline, similar to ordinary programming languages. Let K_t denote the contents of the pushdown stack at time t. When a subroutine is invoked, its name and actual parameters are pushed onto the stack. When a subroutine terminates, its name and actual parameters are popped off the stack. Notice (line 16) that if *any* subroutine on the stack terminates, then all subroutines below

Table 1: Pseudo-Code for Execution of a Hierarchical Policy.

```
Procedure ExecuteHierarchicalPolicy(π)
s_t is the state of the world at time t
K_t is the state of the execution stack at time t

Let t = 0; K_t = empty stack; observe s_t
push (0, nil) onto stack K_t (invoke the root task with no parameters)

repeat
    while top(K_t) is not a primitive action
        Let (i, f_i) := top(K_t), where
            i is the name of the "current" subroutine, and
            f_i gives the parameter bindings for i
        Let (a, f_a) := π_i(s, f_i), where
            a is the action and f_a gives the parameter bindings chosen by policy π_i
        push (a, f_a) onto the stack K_t
    end // while

    Let (a, nil) := pop(K_t) be the primitive action on the top of the stack.
    Execute primitive action a, observe s_{t+1}, and receive reward R(s_{t+1}|s_t, a)

    If any subtask on K_t is terminated in s_{t+1} then
        Let M' be the terminated subtask that is highest (closest to the root) on the stack.
        while top(K_t) ≠ M' do pop(K_t)
        pop(K_t)

    K_{t+1} := K_t is the resulting execution stack.
until K_{t+1} is empty

end ExecuteHierarchicalPolicy
```

it are immediately aborted, and control returns to the subroutine that had invoked the terminated subroutine.

It is sometimes useful to think of the contents of the stack as being an additional part of the state space for the problem. Hence, a hierarchical policy implicitly defines a mapping from the current state s_t and current stack contents K_t to a primitive action a. This action is executed, and this yields a resulting state s_{t+1} and a resulting stack contents K_{t+1}. Because of the added state information in the stack, the hierarchical policy is non-Markovian with respect to the original MDP.

Because a hierarchical policy maps from states s and stack contents K to actions, the value function for a hierarchical policy must assign values to combinations of states s and stack contents K.

Definition 4 *A* hierarchical value function, *denoted $V^\pi(\langle s, K\rangle)$, gives the expected cumulative reward of following the hierarchical policy π starting in state s with stack contents K.*

This hierarchical value function is exactly what is learned by Ron Parr's (1998b) HAMQ algorithm, which we will discuss below. However, in this paper, we will focus on learning only the *projected value functions* of each of the subtasks $M_0, \ldots, M_n$ in the hierarchy.

Definition 5 *The* projected value function *of hierarchical policy π on subtask M_i, denoted $V^\pi(i,s)$, is the expected cumulative reward of executing π_i (and the policies of all descendents of M_i) starting in state s until M_i terminates.*

The purpose of the MAXQ value function decomposition is to decompose $V(0,s)$ (the projected value function of the root task) in terms of the projected value functions $V(i,s)$ of all of the subtasks in the MAXQ decomposition.

3.3 Decomposition of the Projected Value Function

Now that we have defined a hierarchical policy and its projected value function, we can show how that value function can be decomposed hierarchically. The decomposition is based on the following theorem:

Theorem 1 *Given a task graph over tasks $M_0, \ldots, M_n$ and a hierarchical policy π, each subtask M_i defines a semi-Markov decision process with states S_i, actions A_i, probability transition function $P_i^\pi(s', N|s,a)$, and* expected *reward function $\overline{R}(s,a) = V^\pi(a,s)$, where $V^\pi(a,s)$ is the projected value function for child task M_a in state s. If a is a primitive action, $V^\pi(a,s)$ is defined as the expected immediate reward of executing a in s: $V^\pi(a,s) = \sum_{s'} P(s'|s,a)R(s'|s,a)$.*

Proof: Consider all of the subroutines that are descendents of task M_i in the task graph. Because all of these subroutines are executing fixed policies (specified by hierarchical policy π), the probability transition function $P_i^\pi(s', N|s,a)$ is a well defined, stationary distribution for each child subroutine a. The set of states S_i and the set of actions A_i are obvious. The interesting part of this theorem is the fact that the expected reward function $\overline{R}(s,a)$ of the SMDP is the projected value function of the child task M_a.

To see this, let us write out the value of $V^\pi(i,s)$:

$$V^\pi(i,s) = E\{r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots | s_t = s, \pi\} \tag{5}$$

This sum continues until the subroutine for task M_i enters a state in T_i.

Now let us suppose that the first action chosen by π_i is a subroutine a. This subroutine is invoked, and it executes for a number of steps N and terminates in state s' according to $P_i^\pi(s', N|s,a)$. We can rewrite Equation (5) as

$$V^\pi(i,s) = E\left\{ \sum_{u=0}^{N-1} \gamma^u r_{t+u} \; + \sum_{u=N}^{\infty} \gamma^u r_{t+u} \middle| \, s_t = s, \pi \right\} \tag{6}$$

The first summation on the right-hand side of Equation (6) is the discounted sum of rewards for executing subroutine a starting in state s until it terminates, in other words, it is $V^\pi(a,s)$, the projected value function for the child task M_a. The second term on the right-hand side of the equation is the value of s' for the current task i, $V^\pi(i,s')$, discounted by γ^N, where s' is the current state when subroutine a terminates. We can write this in the form of a Bellman equation:

$$V^\pi(i,s) = V^\pi(\pi_i(s), s) + \sum_{s',N} P_i^\pi(s', N|s, \pi_i(s))\gamma^N V^\pi(i,s') \tag{7}$$

This has the same form as Equation (3), which is the Bellman equation for an SMDP, where the first term is the expected reward $\overline{R}(s, \pi(s))$. **Q.E.D.**

To obtain a hierarchical decomposition of the projected value function, let us switch to the action-value (or Q) representation. First, we need to extend the Q notation to handle the task hierarchy. Let $Q^\pi(i, s, a)$ be the expected cumulative reward for subtask M_i of performing action a in state s and then following hierarchical policy π until subtask M_i terminates. Action a may be either a primitive action or a child subtask. With this notation, we can re-state Equation (7) as follows:

$$Q^\pi(i, s, a) = V^\pi(a, s) + \sum_{s', N} P_i^\pi(s', N|s, a)\gamma^N Q^\pi(i, s', \pi(s')), \tag{8}$$

The right-most term in this equation is the expected discounted reward of *completing* task M_i after executing action a in state s. This term only depends on i, s, and a, because the summation marginalizes away the dependence on s' and N. Let us define $C^\pi(i, s, a)$ to be equal to this term:

Definition 6 *The* completion function, $C^\pi(i, s, a)$, *is the expected discounted cumulative reward of* completing *subtask M_i after invoking the subroutine for subtask M_a in state s. The reward is discounted back to the point in time where a begins execution.*

$$C^\pi(i, s, a) = \sum_{s', N} P_i^\pi(s', N|s, a)\gamma^N Q^\pi(i, s', \pi(s')) \tag{9}$$

With this definition, we can express the Q function recursively as

$$Q^\pi(i, s, a) = V^\pi(a, s) + C^\pi(i, s, a). \tag{10}$$

Finally, we can re-express the definition for $V^\pi(i, s)$ as

$$V^\pi(i, s) = \begin{cases} Q^\pi(i, s, \pi_i(s)) & \text{if } i \text{ is composite} \\ \sum_{s'} P(s'|s, i)R(s'|s, i) & \text{if } i \text{ is primitive} \end{cases} \tag{11}$$

We will refer to equations (9), (10), and (11) as the *decomposition equations* for the MAXQ hierarchy under a fixed hierarchical policy π. These equations recursively decompose the projected value function for the root, $V^\pi(0, s)$ into the projected value functions for the individual subtasks, $M_1, \ldots, M_n$ and the individual completion functions $C^\pi(j, s, a)$ for $j = 1, \ldots, n$. The fundamental quantities that must be stored to represent the value function decomposition are just the C values for all non-primitive subtasks and the V values for all primitive actions.

To make it easier for programmers to design and debug MAXQ decompositions, we have developed a graphical representation that we call the *MAXQ graph*. A MAXQ graph for the Taxi domain is shown in Figure 3. The graph contains two kinds of nodes, Max nodes and Q nodes. The Max nodes correspond to the subtasks in the task decomposition—there is one Max node for each primitive action and one Max node for each subtask (including the Root) task. Each primitive Max node i stores the value of $V^\pi(i, s)$. The Q nodes correspond to the actions that are available for each subtask. Each Q node for parent task i, state s

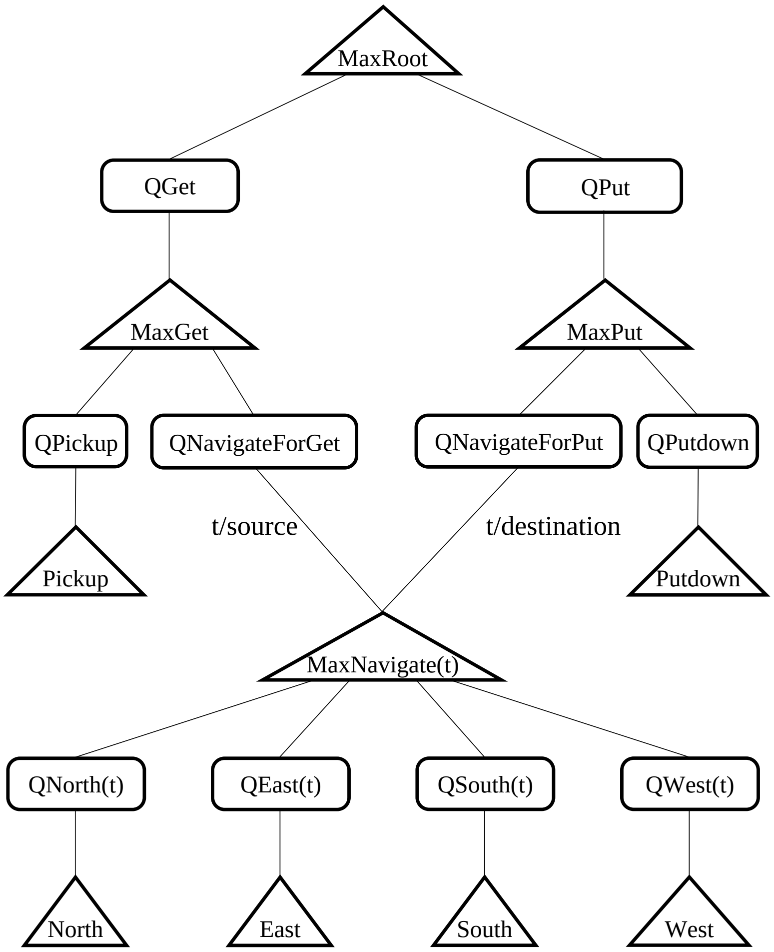

Figure 3: A MAXQ graph for the Taxi Domain.

and subtask a stores the value of $C^\pi(i, s, a)$. The children of any node are *unordered*—that is, the order in which they are drawn in Figure 3 does not imply anything about the order in which they will be executed. Indeed, a child action may be executed multiple times before its parent subtask is completed.

In addition to storing information, the Max nodes and Q nodes can be viewed as performing parts of the computation described by the decomposition equations. Specifically, each Max node i can be viewed as computing the projected value function $V^\pi(i, s)$ for its subtask. For primitive Max nodes, this information is stored in the node. For composite Max nodes, this information is obtained by "asking" the Q node corresponding to $\pi_i(s)$. Each Q node with parent task i and child task a can be viewed as computing the value of $Q^\pi(i, s, a)$. It does this by "asking" its child task a for its projected value function $V^\pi(a, s)$ and then adding its completion function $C^\pi(i, s, a)$.

As an example, consider the situation shown in Figure 1, which we will denote by s_1. Suppose that the passenger is at R and wishes to go to B. Let the hierarchical policy we are evaluating be an optimal policy denoted by π (we will omit the superscript * to reduce the clutter of the notation). The value of this state under π is 10, because it will cost 1 unit to move the taxi to R, 1 unit to pickup the passenger, 7 units to move the taxi to B, and 1 unit to putdown the passenger, for a total of 10 units (a reward of -10). When the passenger is delivered, the agent gets a reward of $+20$, so the net value is $+10$.

Figure 4 shows how the MAXQ hierarchy computes this value. To compute the value $V^\pi(\mathsf{Root}, s_1)$, $\mathsf{MaxRoot}$ consults its policy and finds that $\pi_{\mathsf{Root}}(s_1)$ is Get. Hence, it "asks" the Q node, QGet to compute $Q^\pi(\mathsf{Root}, s_1, \mathsf{Get})$. The completion cost for the Root task after performing a Get, $C^\pi(\mathsf{Root}, s_1, \mathsf{Get})$, is 12, because it will cost 8 units to deliver the customer (for a net reward of $20 - 8 = 12$) after completing the Get subtask. However, this is just the reward *after* completing the Get, so it must ask MaxGet to estimate the expected reward of performing the Get itself.

The policy for MaxGet dictates that in s_1, the $\mathsf{Navigate}$ subroutine should be invoked with t bound to R, so MaxGet consults the Q node, $\mathsf{QNavigateForGet}$ to compute the expected reward. $\mathsf{QNavigateForGet}$ knows that after completing the $\mathsf{Navigate}$(R) task, one more action (the Pickup) will be required to complete the Get, so $C^\pi(\mathsf{MaxGet}, s_1, \mathsf{Navigate}(R)) = -1$. It then asks $\mathsf{MaxNavigate}(R)$ to compute the expected reward of performing a $\mathsf{Navigate}$ to location R.

The policy for $\mathsf{MaxNavigate}$ chooses the North action, so $\mathsf{MaxNavigate}$ asks QNorth to compute the value. QNorth looks up its completion cost, and finds that $C^\pi(\mathsf{Navigate}, s_1, \mathsf{North})$ is 0 (i.e., the $\mathsf{Navigate}$ task will be completed after performing the North action). It consults $\mathsf{MaxNorth}$ to determine the expected cost of performing the North action itself. Because $\mathsf{MaxNorth}$ is a primitive action, it looks up its expected reward, which is -1.

Now this series of recursive computations can conclude as follows:

- $Q^\pi(\mathsf{Navigate}(R), s_1, \mathsf{North}) = -1 + 0$
- $V^\pi(\mathsf{Navigate}(R), s_1) = -1$
- $Q^\pi(\mathsf{Get}, s_1, \mathsf{Navigate}(R)) = -1 + -1$
  (-1 to perform the $\mathsf{Navigate}$ plus -1 to complete the Get.
- $V^\pi(\mathsf{Get}, s_1) = -2$
- $Q^\pi(\mathsf{Root}, s_1, \mathsf{Get}) = -2 + 12$
  (-2 to perform the Get plus 12 to complete the Root task and collect the final reward).

The end result of all of this is that the value of $V^\pi(\mathsf{Root}, s_1)$ is decomposed into a sum of C terms plus the expected reward of the chosen primitive action:

$$\begin{aligned} V^\pi(\mathsf{Root}, s_1) &= V^\pi(\mathsf{North}, s_1) + C^\pi(\mathsf{Navigate}(R), s_1, \mathsf{North}) + \\ &\quad C^\pi(\mathsf{Get}, s_1, \mathsf{Navigate}(R)) + C^\pi(\mathsf{Root}, s_1, \mathsf{Get}) \\ &= -1 + 0 + -1 + 12 \\ &= 10 \end{aligned}$$

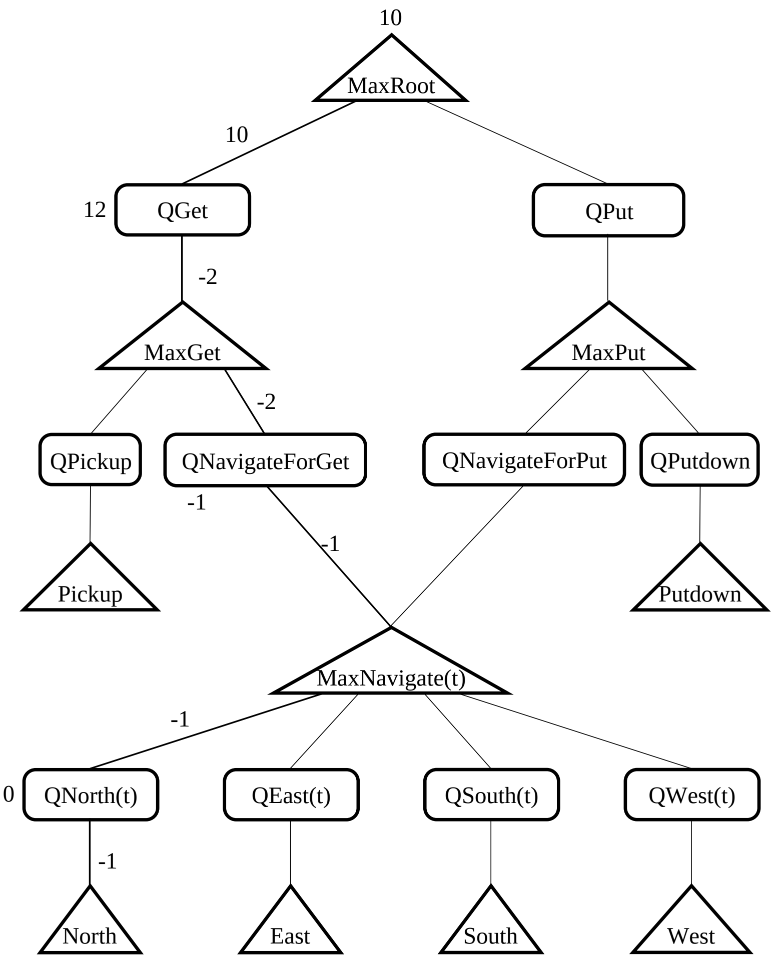

Figure 4: Computing the value of a state using the MAXQ hierarchy. The C value of each Q node is shown to the left of the node. All other numbers show the values being returned up the graph.

In general, the MAXQ value function decomposition has the form

$$V^\pi(0, s) \quad = \quad V^\pi(a_m, s) + C^\pi(a_{m-1}, s, a_m) + \ldots + C^\pi(a_1, s, a_2) + C^\pi(0, s, a_1), \quad (12)$$

where $a_0, a_1, \ldots, a_m$ is the "path" of Max nodes chosen by the hierarchical policy going from the Root down to a primitive leaf node. This is summarized graphically in Figure 5.

We can summarize the presentation of this section by the following theorem:

Theorem 2 *Let $\pi = \{\pi_i; i = 0, \ldots, n\}$ be a hierarchical policy defined for a given MAXQ graph with subtasks $M_0, \ldots, M_n$, and let $i = 0$ be the root node of the graph. Then there exist values for $C^\pi(i, s, a)$ (for internal Max nodes) and $V^\pi(i, s)$ (for primitive, leaf Max*

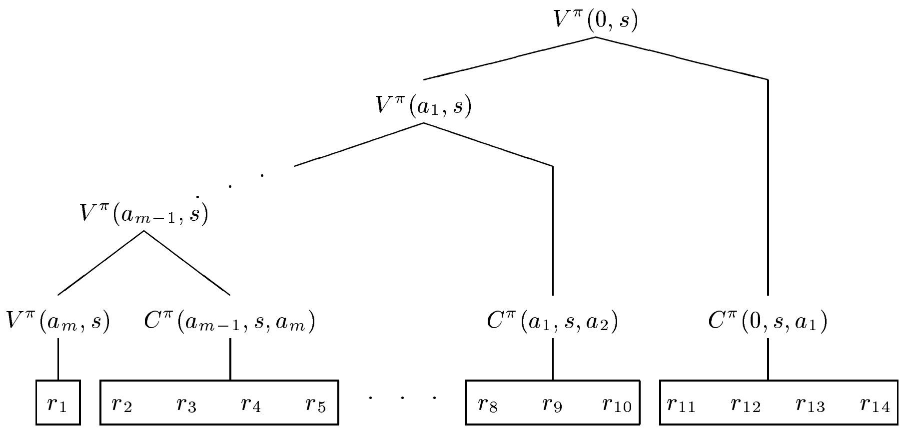

Figure 5: The MAXQ decomposition; $r_1, \ldots, r_{14}$ denote the sequence of rewards received from primitive actions at times $1, \ldots, 14$.

nodes) such that $V^\pi(0, s)$ (as computed by the decomposition equations (9), (10), and (11)) is the expected discounted cumulative reward of following policy π starting in state s.

Proof: The proof is by induction on the number of levels in the task graph. At each level i, we compute values for $C^\pi(i, s, \pi(s))$ (or $V^\pi(i, s)$, if i is primitive) according to the decomposition equations. We can apply the decomposition equations again to compute $Q^\pi(i, s, \pi(s))$ and apply Equation (8) and Theorem 1 to conclude that $Q^\pi(i, s, \pi(s))$ gives the value function for level i. When $i = 0$, we obtain the value function for the entire hierarchical policy. **Q. E. D.**

It is important to note that this representation theorem does not mention the pseudo-reward function, because the pseudo-reward is used only during learning. This theorem captures the *representational power* of the MAXQ decomposition, but it does not address the question of whether there is a learning algorithm that can find a given policy. That is the subject of the next section.

4. A Learning Algorithm for the MAXQ Decomposition

This section presents the central contributions of the paper. First, we discuss what optimality criteria should be employed in hierarchical reinforcement learning. Then we introduce the MAXQ-0 learning algorithm, which can learn value functions (and policies) for MAXQ hierarchies in which there are no pseudo-rewards (i.e., the pseudo-rewards are zero). The central theoretical result of the paper is that MAXQ-0 converges to a recursively optimal policy for the given MAXQ hierarchy. This is followed by a brief discussion of ways of accelerating MAXQ-0 learning. The section concludes with a description of the MAXQ-Q learning algorithm, which handles non-zero pseudo-reward functions.

4.1 Two Kinds of Optimality

In order to develop a learning algorithm for the MAXQ decomposition, we must consider exactly what we are hoping to achieve. Of course, for any MDP M, we would like to find an optimal policy π^*. However, in the MAXQ method (and in hierarchical reinforcement learning in general), the programmer imposes a hierarchy on the problem. This hierarchy constrains the space of possible policies so that it may not be possible to represent the optimal policy or its value function.

In the MAXQ method, the constraints take two forms. First, within a subtask, only some of the possible primitive actions may be permitted. For example, in the taxi task, during a Navigate(t), only the North, South, East, and West actions are available—the Pickup and Putdown actions are not allowed. Second, consider a Max node M_j with child nodes $\{M_{j_1}, \ldots, M_{j_k}\}$. The policy learned for M_j must involve executing the learned policies of these child nodes. When the policy for child node M_{j_i} is executed, it will run until it enters a state in T_{j_i}. Hence, any policy learned for M_j must pass through some subset of these terminal state sets $\{T_{j_1}, \ldots, T_{j_k}\}$.

The HAM method shares these same two constraints and in addition, it imposes a partial policy on each node, so that the policy for any subtask M_i must be a deterministic refinement of the given non-deterministic initial policy for node i.

In the "option" approach, the policy is even further constrained. In this approach, there are only two non-primitive levels in the hierarchy, and the subtasks at the lower level (i.e., whose children are all primitive actions) are given complete policies by the programmer. Hence, any learned policy at the upper level must be constructed by "concatenating" the given lower level policies in some order.

The purpose of imposing these constraints on the policy is to incorporate prior knowledge and thereby reduce the size of the space that must be searched to find a good policy. However, these constraints may make it impossible to learn the optimal policy.

If we can't learn the optimal policy, the next best target would be to learn the best policy that is consistent with (i.e., can be represented by) the given hierarchy.

Definition 7 *A* hierarchically optimal policy *for MDP M is a policy that achieves the highest cumulative reward among all policies consistent with the given hierarchy.*

Parr (1998b) proves that his HAMQ learning algorithm converges with probability 1 to a hierarchically optimal policy. Similarly, given a fixed set of options, Sutton, Precup, and Singh (1998) prove that their SMDP learning algorithm converges to a hierarchically optimal value function. Incidentally, they also show that if the primitive actions are also made available as "trivial" options, then their SMDP method converges to the optimal policy. However, in this case, it is hard to say anything formal about how the options speed the learning process. They may in fact hinder it (Hauskrecht et al., 1998).

Because the MAXQ decomposition can represent the value function of any hierarchical policy, we could easily construct a modified version of the HAMQ algorithm and apply it to learn hierarchically optimal policies for the MAXQ hierarchy. However, we decided to pursue an even weaker form of optimality, for reasons that will become clear as we proceed. This form of optimality is called recursive optimality.

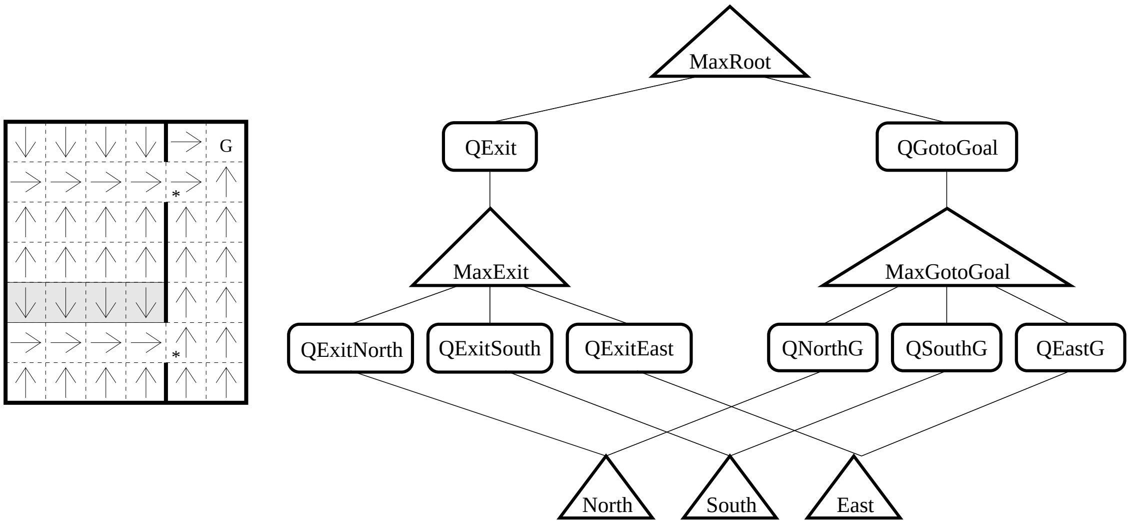

Figure 6: A simple MDP (left) and its associated MAXQ graph (right). The policy shown in the left diagram is recursively optimal but not hierarchically optimal. The shaded cells indicate points where the locally-optimal policy is not globally optimal.

Definition 8 *A* recursively optimal policy *for Markov decision process M with MAXQ decomposition $\{M_0, \ldots, M_k\}$ is a hierarchical policy $\pi = \{\pi_0, \ldots, \pi_k\}$ such that for each subtask M_i, the corresponding policy π_i is optimal for the SMDP defined by the set of states S_i, the set of actions A_i, the state transition probability function $P^\pi(s', N|s, a)$, and the reward function given by the* sum *of the original reward function $R(s'|s, a)$ and the pseudo-reward function $\tilde{R}_i(s')$.*

Note that the state transition probability distribution, $P^\pi(s', N|s, a)$ for subtask M_i is defined by the locally optimal policies $\{\pi_j\}$ of all subtasks that are descendents of M_i in the MAXQ graph. Hence, recursive optimality is a kind of local optimality in which the policy at each node is optimal given the policies of its children.

The reason to seek recursive optimality rather than hierarchical optimality is that recursive optimality makes it possible to solve each subtask without reference to the context in which it is executed. This context-free property makes it easier to share and re-use subtasks. It will also turn out to be essential for the successful use of state abstraction.

Before we proceed to describe our learning algorithm for recursive optimality, let us see how recursive optimality differs from hierarchical optimality.

It is easy to construct examples of policies that are recursively optimal but not hierarchically optimal (and vice versa). Consider the simple maze problem and its associated MAXQ graph shown in Figures 6. Suppose a robot starts somewhere in the left room, and it must reach the goal G in the right room. The robot has three actions, North, South, and East, and these actions are deterministic. The robot receives a reward of -1 for each move. Let us define two subtasks:

- Exit. This task terminates when the robot exits the left room. We can set the pseudo-reward function $\tilde{R}$ to be 0 for the two terminal states (i.e., the two states indicated by *'s).
- GotoGoal. This task terminates when the robot reaches the goal G.

The arrows in Figure 6 show the locally optimal policy within each room. The arrows on the left seek to exit the left room by the shortest path, because this is what we specified when we set the pseudo-reward function to 0. The arrows on the right follow the shortest path to the goal, which is fine. However, the resulting policy is neither hierarchically optimal nor optimal.

There exists a hierarchical policy that would always exit the left room by the upper door. The MAXQ value function decomposition can represent the value function of this policy, but such a policy would not be locally optimal (because, for example, the states in the "shaded" region would not follow the shortest path to a doorway). Hence, this example illustrates both a recursively optimal policy that is not hierarchically optimal and a hierarchically optimal policy that is not recursively optimal.

If we consider for a moment, we can see a way to fix this problem. The value of the upper starred state under the optimal hierarchical policy is -2 and the value of the lower starred state is -6. Hence, if we changed $\tilde{R}$ to have these values (instead of being zero), then the recursively-optimal policy would be hierarchically optimal (and globally optimal). In other words, if the programmer can guess the right values for the terminal states of a subtask, then the recursively optimal policy will be hierarchically optimal.

This basic idea was first pointed out by Dean and Lin (1995). They describe an algorithm that makes initial guesses for the values of these starred states and then updates those guesses based on the computed values of the starred states under the resulting recursively-optimal policy. They proved that this will converge to a hierarchically optimal policy. The drawback of their method is that it requires repeated solution of the resulting hierarchical learning problem, and this does not always yield a speedup over just solving the original, flat problem.

Parr (1998a) proposed an interesting approach that constructs a *set* of different $\tilde{R}$ functions and computes the recursively optimal policy under each of them for each subtask. His method chooses the $\tilde{R}$ functions in such a way that the hierarchically optimal policy can be approximated to any desired degree. Unfortunately, the method is quite expensive, because it relies on solving a series of linear programming problems each of which requires time polynomial in several parameters, including the number of states $|S_i|$ within the subtask.

This discussion suggests that while, in principle, it is possible to learn good values for the pseudo-reward function, in practice, we must rely on the programmer to specify a single pseudo-reward function, $\tilde{R}$, for each subtask. If the programmer wishes to consider a small number of alternative pseudo-reward functions, they can be handled by defining a small number of subtasks that are identical except for their $\tilde{R}$ functions, and permitting the learning algorithm to choose the one that gives the best recursively-optimal policy.

In our experiments, we have employed the following simplified approach to defining $\tilde{R}$. For each subtask M_i, we define two predicates: the termination predicate, T_i, and a goal predicate, G_i. The goal predicate defines a subset of the terminal states that are "goal states", and these have a pseudo-reward of 0. All other terminal states have a fixed constant pseudo-reward (e.g., -100) that is set so that it is always better to terminate in a goal state than in a non-goal state. For the problems on which we have tested the MAXQ method, this worked very well.

In our experiments with MAXQ, we have found that it is easy to make mistakes in defining T_i and G_i. If the goal is not defined carefully, it is easy to create a set of subtasks that lead to infinite looping. For example, consider again the problem in Figure 6. Suppose we permit a fourth action, West, in the MDP and let us define the termination and goal predicates for the right hand room to be satisfied iff either the robot reaches the goal or it exits the room. This is a very natural definition, since it is quite similar to the definition for the left-hand room. However, the resulting locally-optimal policy for this room will attempt to move to the nearest of these three locations: the goal, the upper door, or the lower door. We can easily see that for all but a few states near the goal, the only policies that can be constructed by MaxRoot will loop forever, first trying to leave the left room by entering the right room, and then trying to leave the right room by entering the left room. This problem is easily fixed by defining the goal predicate G_i for the right room to be true if and only if the robot reaches the goal G. But avoiding such "undesired termination" bugs can be hard in more complex domains.

In the worst case, it is possible for the programmer to specify pseudo-rewards such that the recursively optimal policy can be made arbitrarily worse than the hierarchically optimal policy. For example, suppose that we change the original MDP in Figure 6 so that the state immediately to the left of the upper doorway gives a large negative reward $-L$ whenever the robot visits that square. Because rewards everywhere else are -1, the hierarchically-optimal policy exits the room by the lower door. But suppose the programmer has chosen instead to force the robot to exit by the upper door (e.g., by assigning a pseudo-reward of $-10L$ for leaving via the lower door). In this case, the recursively-optimal policy will leave by the upper door and suffer the large $-L$ penalty. By making L arbitrarily large, we can make the difference between the hierarchically-optimal policy and the recursively-optimal policy arbitrarily large.

4.2 The MAXQ-0 Learning Algorithm

Now that we have an understanding of recursively optimal policies, we present two learning algorithms. The first one, called MAXQ-0, applies only in the case when the pseudo-reward function $\tilde{R}$ is always zero. We will first prove its convergence properties and then show how it can be extended to give the second algorithm, MAXQ-Q, which works with general pseudo-reward functions.

Table 2 gives pseudo-code for MAXQ-0. MAXQ-0 is a recursive function that executes the current exploration policy starting at Max node i in state s. It performs actions until it reaches a terminal state, at which point it returns a count of the total number of primitive actions that have been executed. To execute an action, MAXQ-0 calls itself recursively (line 9). When the recursive call returns, it updates the value of the completion function for node i. It uses the count of the number of primitive actions to appropriately discount the value of the resulting state s'. At leaf nodes, MAXQ-0 updates the estimated one-step expected reward, $V(i, s)$. The value $\alpha_t(i)$ is a "learning rate" parameter that should be gradually decreased to zero in the limit.

Table 2: The MAXQ-0 learning algorithm.

```
   function MAXQ-0(MaxNode i, State s)
1  if i is a primitive MaxNode
2      execute i, receive r, and observe result state s'
3      V_{t+1}(i, s) := (1 − α_t(i)) · V_t(i, s) + α_t(i) · r_t
4      return 1
5  else
6      let count = 0
7      while T_i(s) is false do
8          choose an action a according to the current exploration policy π_x(i, s)
9          let N = MAXQ-0(a, s) (recursive call)
10         observe result state s'
11         C_{t+1}(i, s, a) := (1 − α_t(i)) · C_t(i, s, a) + α_t(i) · γ^N V_t(i, s')
12         count := count + N
13         s := s'
14         end
15     return count
   end MAXQ-0

16 // Main program
17 initialize V(i, s) and C(i, s, j) arbitrarily
18 MAXQ-0(root node 0, starting state s_0)
```

There are three things that must be specified in order to make this algorithm description complete.

First, to keep the pseudo-code readable, Table 2 does not show how "ancestor termination" is handled. Recall that after each action, the termination predicates of all of the subroutines on the calling stack are checked. If the termination predicate of any one of these is satisfied, then the calling stack is unwound up to the highest terminated subroutine. In such cases, no C values are updated in any of the subroutines that were interrupted except as follows. If subroutine i had invoked subroutine j, and j's termination condition is satisfied, then subroutine i can update the value of $C(i, s, j)$.

Second, we must specify how to compute $V_t(i, s')$ in line 11, since it is not stored in the Max node. It is computed by the following modified versions of the decomposition equations:

$$V_t(i, s) = \begin{cases} \max_a Q_t(i, s, a) & \text{if } i \text{ is composite} \\ V_t(i, s) & \text{if } i \text{ is primitive} \end{cases} \quad (13)$$

$$Q_t(i, s, a) = V_t(a, s) + C_t(i, s, a). \quad (14)$$

These equations reflect two important changes compared with Equations (10) and (11). First, in Equation (13), $V_t(i, s)$ is defined in terms of the Q value of the *best* action a, rather than of the action chosen by a fixed hierarchical policy. Second, there are no π superscripts, because the current value function, $V_t(i, s)$, is not based on a fixed hierarchical policy π.

To compute $V_t(i, s)$ using these equations, we must perform a complete search of all paths through the MAXQ graph starting at node i and ending at the leaf nodes. Table 3

Table 3: Pseudo-code for Greedy Execution of the MAXQ Graph.

```
function EvaluateMaxNode(i, s)
    if i is a primitive Max node
        return ⟨V_t(i, s), i⟩
    else
        for each j ∈ A_i,
            let ⟨V_t(j, s), a_j⟩ = EvaluateMaxNode(j, s)
        let j^hg = argmax_j V_t(j, s) + C_t(i, s, j)
        return ⟨V_t(j^hg, s), a_{j^hg}⟩
end // EvaluateMaxNode
```

gives pseudo-code for a recursive function, EVALUATEMAXNODE, that implements a depth-first search. In addition to returning $V_t(i, s)$, EVALUATEMAXNODE also returns the action at the leaf node that achieves this value. This information is not needed for MAXQ-0, but it will be useful later when we consider non-hierarchical execution of the learned recursively-optimal policy.

This search can be computationally expensive, and a problem for future research is to develop more efficient methods for computing the best path through the graph. One approach is to perform a best-first search and use bounds on the values within subtrees to prune useless paths through the MAXQ graph. A better approach would be to make the computation incremental, so that when the state of the environment changes, only those nodes whose values have changed as a result of the state change are re-considered. It should be possible to develop an efficient bottom-up method similar to the RETE algorithm (and its successors) that is used in the SOAR architecture (Forgy, 1982; Tambe & Rosenbloom, 1994).

The third thing that must be specified to complete our definition of MAXQ-0 is the exploration policy, π_x. We require that π_x be an ordered GLIE policy.

Definition 9 *An* ordered GLIE policy *is a GLIE policy (Greedy in the Limit with Infinite Exploration) that converges in the limit to an* ordered greedy policy*, which is a greedy policy that imposes an arbitrary fixed order ω on the available actions and breaks ties in favor of the action a that appears earliest in that order.*

We need this property in order to ensure that MAXQ-0 converges to a uniquely-defined recursively optimal policy. A fundamental problem with recursive optimality is that in general, each Max node i will have a choice of many different locally optimal policies given the policies adopted by its descendent nodes. These different locally optimal policies will all achieve the same locally optimal value function, but they can give rise to different probability transition functions $P(s', N|s, i)$. The result will be that the Semi-Markov Decision Problems defined at the next level above node i in the MAXQ graph will differ depending on which of these various locally optimal policies is chosen by node i. These differences may lead to better or worse policies at higher levels of the MAXQ graph, even though they make no difference inside subtask i. In practice, the designer of the MAXQ graph will need to design the pseudo-reward function for subtask i to ensure that all locally optimal policies are equally valuable for the parent subroutine. But to carry out our formal analysis, we will just rely on an arbitrary tie-breaking mechanism.[2] If we establish a fixed ordering over the Max nodes in the MAXQ graph (e.g., a left-to-right depth-first numbering), and break ties in favor of the lowest-numbered action, then this defines a unique policy at each Max node. And consequently, by induction, it defines a unique policy for the entire MAXQ graph. Let us call this policy π_r^*. We will use the r subscript to denote recursively optimal quantities under an ordered greedy policy. Hence, the corresponding value function is V_r^*, and C_r^* and Q_r^* denote the corresponding completion function and action-value function. We now prove that the MAXQ-0 algorithm converges to π_r^*.

Theorem 3 *Let $M = \langle S, A, P, R, P_0 \rangle$ be either an episodic MDP for which all deterministic policies are proper or a discounted infinite horizon MDP with discount factor γ. Let H be a MAXQ graph defined over subtasks $\{M_0, \ldots, M_k\}$ such that the pseudo-reward function $\tilde{R}_i(s')$ is zero for all i and s'. Let $\alpha_t(i) > 0$ be a sequence of constants for each Max node i such that*

$$\lim_{T\to\infty} \sum_{t=1}^{T} \alpha_t(i) = \infty \quad \textit{and} \quad \lim_{T\to\infty} \sum_{t=1}^{T} \alpha_t^2(i) < \infty \tag{15}$$

Let $\pi_x(i, s)$ be an ordered GLIE policy at each node i and state s and assume that the immediate rewards are bounded. Then with probability 1, algorithm MAXQ-0 converges to π_r^*, the unique recursively optimal policy for M consistent with H and π_x.

Proof: The proof follows an argument similar to those introduced to prove the convergence of Q learning and $SARSA(0)$ (Bertsekas & Tsitsiklis, 1996; Jaakkola et al., 1994). We will employ the following result from stochastic approximation theory, which we state without proof:

Lemma 1 *(Proposition 4.5 from Bertsekas and Tsitsiklis, 1996) Consider the iteration*

$$r_{t+1}(i) := (1 - \alpha_t(i))r_t(i) + \alpha_t(i)((Ur_t)(i) + w_t(i) + u_t(i)).$$

Let $\mathcal{F}_t = \{r_0(i), \ldots, r_t(i), w_0(i), \ldots, w_{t-1}(i), \alpha_0(i), \ldots, \alpha_t(i), \forall i\}$ be the entire history of the iteration.

If

(a) The $\alpha_t(i) \geq 0$ satisfy conditions (15)

(b) For every i and t, the noise terms $w_t(i)$ satisfy $E[w_t(i)|\mathcal{F}_t] = 0$

(c) Given any norm $||\cdot||$ on $\mathcal{R}^n$, there exist constants A and B such that $E[w_t^2(i)|\mathcal{F}_t] \leq A + B||r_t||^2$.

(d) There exists a vector r^*, a positive vector ξ, and a scalar $\beta \in [0, 1)$, such that for all t,

$$||Ur_t - r^*||_\xi \leq \beta||r_t - r^*||_\xi$$

2. Alternatively, we could break ties by using a stochastic policy that chose randomly among the tied actions.

(e) *There exists a nonnegative random sequence θ_t that converges to zero with probability 1 and is such that for all t*

$$|u_t(i)| \leq \theta_t(||r_t||_\xi + 1)$$

then r_t converges to r^* with probability 1. The notation $||\cdot||_\xi$ denotes a weighted maximum norm

$$||A||_\xi = \max_i \frac{|A(i)|}{\xi(i)}.$$

The structure of the proof of Theorem 3 will be inductive, starting at the leaves of the MAXQ graph and working toward the root. We will employ a different time clock at each node i to count the number of update steps performed by MAXQ-0 at that node. The variable t will always refer to the time clock of the current node i.

To prove the base case for any primitive Max node, we note that line 3 of MAXQ-0 is just the standard stochastic approximation algorithm for computing the expected reward for performing action a in state s, and therefore it converges under the conditions given above.

To prove the recursive case, consider any composite Max node i with child node j. Let $P_t(s', N|s, j)$ be the transition probability distribution for performing child action j in state s at time t (i.e., while following the exploration policy in all descendent nodes of node j). By the inductive assumption, MAXQ-0 applied to j will converge to the (unique) recursively optimal value function $V_r^*(j, s)$ with probability 1. Furthermore, because MAXQ-0 is following an ordered GLIE policy for j and its descendents, they will all converge to executing a greedy policy with respect to their value functions, so $P_t(s', N|s, j)$ will converge to $P_r^*(s', N|s, j)$, the unique transition probability function for executing child j under the locally optimal policy π_r^*. What remains to be shown is that the update assignment for C (line 11 of the MAXQ-0 algorithm) converges to the optimal C_r^* function with probability 1.

To prove this, we will apply Lemma 1. We will identify the x in the lemma with a state-action pair (s, a). The vector r_t will be the completion-cost table $C_t(i, s, a)$ for all s, a and fixed i after t update steps. The vector r^* will be the optimal completion-cost $C_r^*(i, s, a)$ (again, for fixed i). Define the mapping U to be

$$(UC)(i, s, a) = \sum_{s'} P_r^*(s', N|s, a)\gamma^N \left(\max_{a'}[C(i, s', a') + V_r^*(a', s')]\right)$$

This is a C update under the MDP M_i assuming that all descendent value functions, $V_r^*(a, s)$, and transition probabilities, $P_r^*(s', N|s, a)$, have converged.

To apply the lemma, we must first express the C update formula in the form of the update rule in the lemma. Let $\overline{s}$ be the state that results from performing a in state s. Line 11 can be written

$$\begin{aligned} C_{t+1}(i, s, a) &:= (1 - \alpha_t(i)) \cdot C_t(i, s, a) + \alpha_t(i) \cdot \gamma^{\overline{N}} \left(\max_{a'}[C_t(i, \overline{s}, a') + V_t(a', \overline{s})]\right) \\ &:= (1 - \alpha_t(i)) \cdot C_t(i, s, a) + \alpha_t(i) \cdot [(UC_t)(i, s, a) + w_t(i, s, a) + u_t(i, s, a)] \end{aligned}$$

where

$$
\begin{aligned}
w_t(i,s,a) &= \gamma^{\overline{N}} \left(\max_{a'} [C_t(i,\overline{s},a') + V_t(a',\overline{s})] \right) - \\
&\quad \sum_{s',N} P_t(s',N|s,a) \gamma^N \left(\max_{a'} [C_t(i,s',a') + V_t(a',s')] \right) \\
u_t(i,s,a) &= \sum_{s',N} P_t(s',N|s,a) \gamma^N \left(\max_{a'} [C_t(i,s',a') + V_t(a',s')] \right) - \\
&\quad \sum_{s',N} P_r^*(s',N|s,a) \gamma^N \left(\max_{a'} [C_t(i,s',a') + V_r^*(a',s')] \right)
\end{aligned}
$$

Here $w_t(i,s,a)$ is the difference between doing an update at node i using the single *sample point* $\overline{s}$ drawn according to $P_t(s',N|s,a)$ and doing an update using the full distribution $P_t(s',N|s,a)$. The value of $u_t(i,s,a)$ captures the difference between doing an update using the current probability transitions $P_t(s',N|s,a)$ and current value functions of the children $V_t(a',s')$ and doing an update using the optimal probability transitions $P_r^*(s',N|s,a)$ and the optimal values of the children $V_r^*(a',s')$.

We now verify the conditions of Lemma 1.

Condition (a) is assumed in the conditions of the theorem with $\alpha_t(s,a) = \alpha_t(i)$.

Condition (b) is satisfied because $\overline{s}$ is sampled from $P_t(s',N|s,a)$, so the expected value of the difference is zero.

Condition (c) follows directly from the fact that $|C_t(i,s,a)|$ and $|V_t(i,s)|$ are bounded. We can show that these are bounded for both the episodic case and the discounted case as follows. In the episodic case, we have assumed all policies are proper. Hence, all trajectories terminate in finite time with a finite total reward. In the discounted case, the infinite sum of future rewards is bounded if the one-step rewards are bounded. The values of C and V are computed as temporal averages of the cumulative rewards received over a finite number of (bounded) updates, and hence, their means, variances, and maximum values are all bounded.

Condition (d) is the condition that U is a weighted max norm pseudo-contraction. We can derive this by starting with the weighted max norm for Q learning. It is well known that Q is a weighted max norm pseudo-contraction (Bertsekas & Tsitsiklis, 1996) in both the episodic case where all deterministic policies are proper (and the discount factor $\gamma = 1$) and in the infinite horizon discounted case (with $\gamma < 1$). That is, there exists a positive vector ξ and a scalar $\beta \in [0,1)$, such that for all t,

$$||TQ_t - Q^*||_\xi \le \beta ||Q_t - Q^*||_\xi, \tag{16}$$

where T is the operator

$$(TQ)(s,a) = \sum_{s',N} P(s',N|s,a) \gamma^N [R(s'|s,a) + \max_{a'} Q(s',a')].$$

Now we will show how to derive the pseudo-contraction for the C update operator U. Our plan is to show first how to express the U operator for learning C in terms of the T operator for updating Q values. Then we will replace TQ in the pseudo-contraction equation for Q

learning with UC, and show that U is a weighted max-norm pseudo-contraction under the same weights ξ and the same β.

Recall from Eqn. (10) that $Q(i,s,a) = C(i,s,a) + V(a,s)$. Furthermore, the U operator performs its updates using the optimal value functions of the child nodes, so we can write this as $Q_t(i,s,a) = C_t(i,s,a) + V^*(a,s)$. Now once the children of node i have converged, the Q-function version of the Bellman equation for MDP M_i can be written as

$$Q(i,s,a) = \sum_{s',N} P_r^*(s',N|s,a)\gamma^N [V_r^*(a,s) + \max_{a'} Q(i,s',a')].$$

As we have noted before, $V_r^*(a,s)$ plays the role of the immediate reward function for M_i. Therefore, for node i, the T operator can be rewritten as

$$(TQ)(i,s,a) = \sum_{s',N} P_r^*(s'|s,a)\gamma^N [V_r^*(a,s) + \max_{a'} Q(i,s',a')].$$

Now we replace $Q(i,s,a)$ by $C(i,s,a) + V_r^*(a,s)$, and obtain

$$(TQ)(i,s,a) = \sum_{s',N} P_r^*(s',N|s,a)\gamma^N (V_r^*(a,s) + \max_{a'}[C(i,s',a') + V_r^*(a',s')]).$$

Note that $V_r^*(a,s)$ does not depend on s' or N, so we can move it outside the expectation and obtain

$$\begin{aligned}(TQ)(i,s,a) &= V_r^*(a,s) + \sum_{s',N} P_r^*(s',N|s,a)\gamma^N (\max_{a'}[C(i,s',a') + V_r^*(a',s')]) \\ &= V_r^*(a,s) + (UC)(i,s,a)\end{aligned}$$

Abusing notation slightly, we will express this in vector form as $TQ(i) = V_r^* + UC(i)$. Similarly, we can write $Q_t(i,s,a) = C_t(i,s,a) + V_r^*(a,s)$ in vector form as $Q_t(i) = C_t(i) + V_r^*$.

Now we can substitute these two formulas into the max norm pseudo-contraction formula for T, Eqn. (16) to obtain

$$||V_r^* + UC_t(i) - (C_r^*(i) + V_r^*)||_\xi \le \beta ||V_r^* + C_t(i) - (C_r^*(i) + V_r^*)||_\xi.$$

Thus, U is a weighted max-norm pseudo-contraction,

$$||UC_t(i) - C_r^*(i)||_\xi \le \beta ||C_t(i) - C_r^*(i)||_\xi,$$

and condition (d) is satisfied.

Finally, it is easy to verify (e), the most important condition. By assumption, the ordered GLIE policies in the child nodes converge with probability 1 to locally optimal policies for the children. Therefore $P_t(s',N|s,a)$ converges to $P_r^*(s',N|s,a)$ for all s', N, s, and a with probability 1 and $V_t(a,s)$ converges with probability 1 to $V_r^*(a,s)$ for all child actions a. Therefore, $|u_t|$ converges to zero with probability 1. We can trivially construct a sequence $\theta_t = |u_t|$ that bounds this convergence, so

$$|u_t(s,a)| \le \theta_t \le \theta_t(||C_t(s,a)||_\xi + 1).$$

We have verified all of the conditions of Lemma 1, so we can conclude that $C_t(i)$ converges to $C_r^*(i)$ with probability 1. By induction, we can conclude that this holds for all nodes in the MAXQ including the root node, so the value function represented by the MAXQ graph converges to the unique value function of the recursively optimal policy π_r^*. **Q.E.D.**

The most important aspect of this theorem is that it proves that Q learning can take place at all levels of the MAXQ hierarchy simultaneously—the higher levels do not need to wait until the lower levels have converged before they begin learning. All that is necessary is that the lower levels eventually converge to their (locally) optimal policies.

4.3 Techniques for Speeding Up MAXQ-0

Algorithm MAXQ-0 can be extended to accelerate learning in the higher nodes of the graph by a technique that we call "all-states updating". When an action a is chosen for Max node i in state s, the execution of a will move the environment through a sequence of states $s = s_1, \ldots, s_N, s_{N+1} = s'$. Because all of our subroutines are Markovian, the same resulting state s' would have been reached if we had started executing action a in state s_2, or s_3, or any state up to and including s_N. Hence, we can execute a version of line 11 in MAXQ-0 for each of these intermediate states as shown in this replacement pseudo-code:

```
11a    for j from 1 to N do
11b        C_{t+1}(i, s_j, a) := (1 − α_t(i)) · C_t(i, s_j, a) + α_t(i) · γ^{(N+1−j)} max_{a'} Q_t(i, s', a')
11c    end // for
```

In our implementation, as each composite action is executed by MAXQ-0, it constructs a linked list of the sequence of primitive states that were visited. This list is returned when the composite action terminates. The parent Max node can then process each state in this list as shown above. The parent Max node concatenates the state lists that it receives from its children and passes them to its parent when it terminates. All experiments in this paper employ all-states updating.

Kaelbling (1993) introduced a related, but more powerful, method for accelerating hierarchical reinforcement learning that she calls "all-goals updating." To understand this method, suppose that for each primitive action, there are several composite tasks that could have invoked that primitive action. In all-goals updating, whenever a primitive action is executed, the equivalent of line 11 of MAXQ-0 is applied in every composite task that could have invoked that primitive action. Sutton, Precup, and Singh (1998) prove that each of the composite tasks will converge to the optimal Q values under all-goals updating. Furthermore, they point out that the exploration policy employed for choosing the primitive actions can be different from the policies of *any* of the subtasks being learned.

It is straightforward to implement a simple form of all-goals updating within the MAXQ hierarchy for the case where composite tasks invoke primitive actions. Whenever one of the primitive actions a is executed in state s, we can update the $C(i, s, a)$ value for all parent tasks i that can invoke a.

However, additional care is required to implement all-goals updating for non-primitive actions. Suppose that by executing the exploration policy, the following sequence of world states and actions has been obtained: $s_0, a_0, s_1, \ldots, a_{k-1}, s_{k-1}, a_k, s_{k+1}$. Let j be a composite task that is terminated in state s_{k+1}, and let $s_{k-n}, a_{k-n}, \ldots, a_{k-1}, a_k$ be a sequence of actions that *could have been executed* by subtask j and its children. In other words, suppose it is possible to "parse" this state-action sequence in terms of a series of subroutine calls and returns for one invocation of subtask j. Then for each possible parent task i that invokes j, we can update the value of $C(i, s_{k-n}, j)$. Of course, in order for these updates to be useful, the exploration policy must be an ordered GLIE policy that will converge to the recursively optimal policy for subtask j and its descendents. We cannot follow an arbitrary exploration policy, because this would not produce accurate samples of result states drawn according to $P^*(s', N|s, j)$. Hence, unlike the simple case described by Sutton, Precup, and Singh, the exploration policy cannot be different from the policies of the subtasks being learned.

Although this considerably reduces the usefulness of all-goals updating, it does not completely eliminate it. A simple way of implementing non-primitive all-goals updating would be to perform MAXQ-Q learning as usual, but whenever a subtask j was invoked in state s and returned, we could update the value of $C(i, s, j)$ for all potential calling subtasks i. We have not implemented this, however, because of the complexity involved in identifying the possible actual parameters of the potential calling subroutines.

4.4 The MAXQ-Q Learning Algorithm

Now that we have shown the convergence of MAXQ-0, let us design a learning algorithm that can work with arbitrary pseudo-reward functions, $\tilde{R}_i(s')$. We could just add the pseudo-reward into MAXQ-0, but this would have the effect of changing the MDP M to have a different reward function. The pseudo-rewards "contaminate" the values of all of the completion functions computed in the hierarchy. The resulting learned policy will not be recursively optimal for the original MDP.

This problem can be solved by learning one completion function for use "inside" each Max node and a separate completion function for use "outside" the Max node. The quantities used "inside" a node will be written with a tilde: $\tilde{R}$, $\tilde{C}$, and $\tilde{Q}$. The quantities used "outside" a node will be written without the tilde.

The "outside" completion function, $C(i, s, a)$ is the completion function that we have been discussing so far in this paper. It computes the expected reward for completing task M_i after performing action a in state s and then following the learned policy for M_i. It is computed without any reference to $\tilde{R}_i$. This completion function will be used by parent tasks to compute $V(i, s)$, the expected reward for performing action i starting in state s.

The second completion function $\tilde{C}(i, s, a)$ is a completion function that we will use only "inside" node i in order to discover the locally optimal policy for task M_i. This function will incorporate rewards both from the "real" reward function, $R(s'|s, a)$, and from the pseudo-reward function, $\tilde{R}_i(s')$. It will also be used by EVALUATEMAXNODE in line 6 to choose the best action j^{hg} to execute. Note, however, that EVALUATEMAXNODE will still return the "external" value $V_t(j^{hg}, s)$ of this chosen action.

We will employ two different update rules to learn these two completion functions. The $\tilde{C}$ function will be learned using an update rule similar to the Q learning rule in line 11 of MAXQ-0. But the C function will be learned using an update rule similar to SARSA(0)—its purpose is to learn the value function for the policy that is discovered by optimizing $\tilde{C}$. Pseudo-code for the resulting algorithm, MAXQ-Q is shown in Table 4.

The key step is at lines 15 and 16. In line 15, MAXQ-Q first updates $\tilde{C}$ using the value of the greedy action, a^*, in the resulting state. This update includes the pseudo-reward $\tilde{R}_i$.

Table 4: The MAXQ-Q learning algorithm.

function MAXQ-Q(MaxNode i, State s)

1 **let** $seq = ()$ be the sequence of states visited while executing i
2 **if** i is a primitive MaxNode
3 execute i, receive r, and observe result state s'
4 $V_{t+1}(i, s) := (1 - \alpha_t(i)) \cdot V_t(i, s) + \alpha_t(i) \cdot r_t$
5 push s onto the beginning of seq
6 **else**
7 **let** $count = 0$
8 **while** $T_i(s)$ is false **do**
9 choose an action a according to the current exploration policy $\pi_x(i, s)$
10 **let** $childSeq$ = MAXQ-Q(a, s), where $childSeq$ is the sequence of states visited while executing action a. (in reverse order)
11 observe result state s'
12 **let** $a^* = \text{argmax}_{a'}\ [\tilde{C}_t(i, s', a') + V_t(a', s')]$
13 **let** $N = 1$
14 **for each** s **in** $childSeq$ **do**
15 $\tilde{C}_{t+1}(i, s, a) := (1 - \alpha_t(i)) \cdot \tilde{C}_t(i, s, a) + \alpha_t(i) \cdot \gamma^N [\tilde{R}_i(s') + \tilde{C}_t(i, s', a^*) + V_t(a^*, s)]$
16 $C_{t+1}(i, s, a) := (1 - \alpha_t(i)) \cdot C_t(i, s, a) + \alpha_t(i) \cdot \gamma^N [C_t(i, s', a^*) + V_t(a^*, s')]$
17 $N := N + 1$
18 **end** // for
19 **append** $childSeq$ onto the front of seq
20 $s := s'$
21 **end** // while
22 **end** // else
23 **return** seq
end MAXQ-Q

Then in line 16, MAXQ-Q updates C using this *same greedy action* a^*, even if this would not be the greedy action according to the "uncontaminated" value function. This update, of course, does not include the pseudo-reward function.

It is important to note that whereever $V_t(a, s)$ appears in this pseudo-code, it refers to the "uncontaminated" value function of state s when executing the Max node a. This is computed recursively in exactly the same way as in MAXQ-0.

Finally, note that the pseudo-code also incorporates all-states updating, so each call to MAXQ-Q returns a list of all of the states that were visited during its execution, and the updates of lines 15 and 16 are performed for each of those states. The list of states is ordered most-recent-first, so the states are updated starting with the last state visited and working backward to the starting state, which helps speed up the algorithm.

When MAXQ-Q has converged, the resulting recursively optimal policy is computed at each node by choosing the action a that maximizes $\tilde{Q}(i, s, a) = \tilde{C}(i, s, a) + V(a, s)$ (breaking ties according to the fixed ordering established by the ordered GLIE policy). It is for this reason that we gave the name "Max nodes" to the nodes that represent subtasks (and learned policies) within the MAXQ graph. Each Q node j with parent node i stores both $\tilde{C}(i, s, j)$ and $C(i, s, j)$, and it computes both $\tilde{Q}(i, s, j)$ and $Q(i, s, j)$ by invoking its child Max node j. Each Max node i takes the maximum of these Q values and computes either $V(i, s)$ or computes the best action, a^* using $\tilde{Q}$.

Corollary 1 *Under the same conditions as Theorem 3, MAXQ-Q converges to the unique recursively optimal policy for MDP M defined by MAXQ graph H, pseudo-reward functions $\tilde{R}$, and ordered GLIE exploration policy π_x.*

Proof: The argument is identical to, but more tedious than, the proof of Theorem 3. The proof of convergence of the $\tilde{C}$ values is identical to the original proof for the C values, but it relies on proving convergence of the "new" C values as well, which follows from the same weighted max norm pseudo-contraction argument. **Q.E.D.**

5. State Abstraction

There are many reasons to introduce hierarchical reinforcement learning, but perhaps the most important reason is to create opportunities for state abstraction. When we introduced the simple taxi problem in Figure 1, we pointed out that within each subtask, we can ignore certain aspects of the state space. For example, while performing a MaxNavigate(t), the taxi should make the same navigation decisions regardless of whether the passenger is in the taxi. The purpose of this section is to formalize the conditions under which it is safe to introduce such state abstractions and to show how the convergence proofs for MAXQ-Q can be extended to prove convergence in the presence of state abstraction. Specifically, we will identify five conditions that permit the "safe" introduction of state abstractions.

Throughout this section, we will use the taxi problem as a running example, and we will see how each of the five conditions will permit us to reduce the number of distinct values that must be stored in order to represent the MAXQ value function decomposition. To establish a starting point, let us compute the number of values that must be stored for the taxi problem *without* any state abstraction.

The MAXQ representation must have tables for each of the C functions at the internal nodes and the V functions at the leaves. First, at the six leaf nodes, to store $V(i, s)$, we must store 500 values at each node, because there are 500 states; 25 locations, 4 possible destinations for the passenger, and 5 possible current locations for the passenger (the four special locations and inside the taxi itself). Second, at the root node, there are two children, which requires $2 \times 500 = 1000$ values. Third, at the MaxGet and MaxPut nodes, we have 2 actions each, so each one requires 1000 values, for a total of 2000. Finally, at MaxNavigate(t), we have four actions, but now we must also consider the target parameter t, which can take four possible values. Hence, there are effectively 2000 combinations of states and t values for each action, or 8000 total values that must be represented. In total, therefore, the MAXQ representation requires 14,000 separate quantities to represent the value function.

To place this number in perspective, consider that a flat Q learning representation must store a separate value for each of the six primitive actions in each of the 500 possible states, for a total of 3,000 values. Hence, we can see that without state abstraction, the MAXQ representation requires more than four times the memory of a flat Q table!

5.1 Five Conditions that Permit State Abstraction

We now introduce five conditions that permit the introduction of state abstractions. For each condition, we give a definition and then prove a lemma which states that if the condition is satisfied, then the value function for some corresponding class of policies can be represented abstractly (i.e., by abstract versions of the V and C functions). For each condition, we then provide some rules for identifying when that condition can be satisfied and give examples from the taxi domain.

We begin by introducing some definitions and notation.

Definition 10 *Let M be a MDP and H be a MAXQ graph defined over M. Suppose that each state s can be written as a vector of values of a set of state variables. At each Max node i, suppose the state variables are partitioned into two sets X_i and Y_i, and let χ_i be a function that projects a state s onto only the values of the variables in X_i. Then H combined with χ_i is called a* state-abstracted MAXQ graph.

In cases where the state variables can be partitioned, we will often write $s = (x, y)$ to mean that a state s is represented by a vector of values for the state variables in X and a vector of values for the state variables in Y. Similarly, we will sometimes write $P(x', y', N|x, y, a)$, $V(a, x, y)$, and $\tilde{R}_a(x', y')$ in place of $P(s', N|s, a)$, $V(a, s)$, and $\tilde{R}_a(s')$, respectively.

Definition 11 (Abstract Policy) *An* abstract hierarchical policy *for MDP M with state-abstracted MAXQ graph H and associated abstraction functions χ_i, is a hierarchical policy in which each policy π_i (corresponding to subtask M_i) satisfies the condition that for any two states s_1 and s_2 such that $\chi_i(s_1) = \chi_i(s_2)$, $\pi_i(s_1) = \pi_i(s_2)$. (When π_i is a stochastic policy, such as an exploration policy, this is interpreted to mean that the probability distributions for choosing actions are the same in both states.)*

In order for MAXQ-Q to converge in the presence of state abstractions, we will require that at all times t its (instantaneous) exploration policy is an abstract hierarchical policy. One way to achieve this is to construct the exploration policy so that it only uses information from the relevant state variables in deciding what action to perform. Boltzmann exploration based on the (state-abstracted) Q values, ϵ-greedy exploration, and counter-based exploration based on abstracted states are all abstract exploration policies. Counter-based exploration based on the full state space is not an abstract exploration policy.

Now that we have introduced our notation, let us describe and analyze the five abstraction conditions. We have identified three different kinds of conditions under which abstractions can be introduced. The first kind involves eliminating irrelevant variables within a subtask of the MAXQ graph. Under this form of abstraction, nodes toward the leaves of the MAXQ graph tend to have very few relevant variables, and nodes higher in the graph have more relevant variables. Hence, this kind of abstraction is most useful at the lower levels of the MAXQ graph.

The second kind of abstraction arises from "funnel" actions. These are macro actions that move the environment from some large number of initial states to a small number of resulting states. The completion cost of such subtasks can be represented using a number of values proportional to the number of resulting states. Funnel actions tend to appear higher in the MAXQ graph, so this form of abstraction is most useful near the root of the graph.

The third kind of abstraction arises from the structure of the MAXQ graph itself. It exploits the fact that large parts of the state space for a subtask may not be reachable because of the termination conditions of its ancestors in the MAXQ graph.

We begin by describing two abstraction conditions of the first type. Then we will present two conditions of the second type. And finally, we describe one condition of the third type.

5.1.1 Condition 1: Max Node Irrelevance

The first condition arises when a set of state variables is irrelevant to a Max node.

Definition 12 (Max Node Irrelevance) *Let M_i be a Max node in a MAXQ graph H for MDP M. A set of state variables Y is* irrelevant for node *i if the state variables of M can be partitioned into two sets X and Y such that for* any *stationary abstract hierarchical policy π executed by the* descendents *of i, the following two properties hold:*

- *the state transition probability distribution $P^\pi(s', N|s, a)$ at node i can be factored into the product of two distributions:*

$$P^\pi(x', y', N|x, y, a) = P^\pi(y'|x, y, a) \cdot P^\pi(x', N|x, a), \tag{17}$$

  where y and y' give values for the variables in Y, and x and x' give values for the variables in X.

- *for any pair of states $s_1 = (x, y_1)$ and $s_2 = (x, y_2)$ such that $\chi(s_1) = \chi(s_2) = x$, and any child action a, $V^\pi(a, s_1) = V^\pi(a, s_2)$ and $\tilde{R}_i(s_1) = \tilde{R}_i(s_2)$.*

Note that the two conditions must hold for all stationary abstract policies π executed by all of the descendents of the subtask i. We will discuss below how these rather strong requirements can be satisfied in practice. First, however, we prove that these conditions are sufficient to permit the C and V tables to be represented using state abstractions.

Lemma 2 *Let M be an MDP with full-state MAXQ graph H, and suppose that state variables Y_i are irrelevant for Max node i. Let $\chi_i(s) = x$ be the associated abstraction function that projects s onto the remaining relevant variables X_i. Let π be any abstract hierarchical policy. Then the action-value function Q^π at node i can be represented compactly, with only one value of the completion function $C^\pi(i, s, j)$ for each equivalence class of states s that share the same values on the relevant variables.*

Specifically $Q^\pi(i, s, j)$ can be computed as follows:

$$Q^\pi(i, s, j) = V^\pi(j, \chi_i(s)) + C^\pi(i, \chi_i(s), j)$$

where

$$C^\pi(i, x, j) = \sum_{x', N} P^\pi(x', N|x, j) \cdot \gamma^N [V^\pi(\pi(x'), x') + \tilde{R}_i(x') + C^\pi(i, x', \pi(x'))],$$

where $V^\pi(j', x') = V^\pi(j', x', y_0)$, $\tilde{R}_i(x') = \tilde{R}_i(x', y_0)$, and $\pi(x) = \pi(x, y_0)$ for some arbitrary value y_0 for the irrelevant state variables Y_i.

Proof: Define a new MDP $\chi_i(M_i)$ at node i as follows:

- States: $X = \{x \mid \chi_i(s) = x, \text{ for some } s \in S\}$.
- Actions: A.
- Transition probabilities: $P^\pi(x', N|x, a)$
- Reward function: $V^\pi(a, x) + \tilde{R}_i(x')$

Because π is an abstract policy, its decisions are the same for all states s such that $\chi_i(s) = x$ for some x. Therefore, it is also a well-defined policy over $\chi_i(M_i)$. The action-value function for π over $\chi_i(M_i)$ is the unique solution to the following Bellman equation:

$$Q^\pi(i, x, j) = V^\pi(j, x) + \sum_{x', N} P^\pi(x', N|x, j) \cdot \gamma^N [\tilde{R}_i(x') + Q^\pi(i, x', \pi(x'))] \tag{18}$$

Compare this to the Bellman equation over M_i:

$$Q^\pi(i, s, j) = V^\pi(j, s) + \sum_{s', N} P^\pi(s', N|s, j) \cdot \gamma^N [\tilde{R}_i(s') + Q^\pi(i, s', \pi(s'))] \tag{19}$$

and note that $V^\pi(j, s) = V^\pi(j, \chi(s)) = V^\pi(j, x)$ and $\tilde{R}_i(s') = \tilde{R}_i(\chi(s')) = \tilde{R}_i(x')$. Furthermore, we know that the distribution P^π can be factored into separate distributions for Y_i and X_i. Hence, we can rewrite (19) as

$$Q^\pi(i, s, j) = V^\pi(j, x) + \sum_{y'} P(y'|x, y, j) \sum_{x', N} P^\pi(x', N|x, j) \cdot \gamma^N [\tilde{R}_i(x') + Q^\pi(i, s', \pi(s'))]$$

The right-most sum does not depend on y or y', so the sum over y' evaluates to 1, and can be eliminated to give

$$Q^\pi(i, s, j) = V^\pi(j, x) + \sum_{x', N} P^\pi(x', N|x, j) \cdot \gamma^N [\tilde{R}_i(x') + Q^\pi(i, s', \pi(s'))]. \tag{20}$$

Finally, note that equations (18) and (20) are identical except for the expressions for the Q values. Since the solution to the Bellman equation is unique, we must conclude that

$$Q^\pi(i, s, j) = Q^\pi(i, \chi(s), j).$$

We can rewrite the right-hand side to obtain

$$Q^\pi(i, s, j) = V^\pi(j, \chi(s)) + C^\pi(i, \chi(s), j),$$

where

$$C^\pi(i, x, j) = \sum_{x', N} P(x', N|x, j) \cdot \gamma^N [V^\pi(\pi(x'), x') + \tilde{R}_i(x') + C^\pi(i, x', \pi(x'))].$$

Q.E.D.

Of course we are primarily interested in being able to discover and represent the *optimal* policy at each node i. The following corollary shows that the optimal policy is an abstract policy, and hence, that it can be represented abstractly.

Corollary 2 *Consider the same conditions as Lemma 2, but with the change that the abstract hierarchical policy π is executed only by the descendents of node i, but not by node i. Let ω be an ordering over actions. Then the optimal ordered policy π^*_ω at node i is an abstract policy, and its action-value function can be represented abstractly.*

Proof: Define the policy ρ^*_ω to be the optimal ordered policy over the abstract MDP $\chi(M)$, and let $Q^*(i,x,j)$ be the corresponding optimal action-value function. Then by the same argument given above, Q^* is also a solution to the optimal Bellman equation for the original MDP. This means that the policy π^*_ω defined by $\pi^*_\omega(s) = \rho^*(\chi(s))$ is an optimal ordered policy, and by construction, it is an abstract policy. **Q.E.D.**

As stated, the Max node irrelevance condition appears quite difficult to satisfy, since it requires that the state transition probability distribution factor into X and Y components for all possible abstract hierarchical policies. However, in practice, this condition is often satisfied.

For example, let us consider the Navigate(t) subtask. The source and destination of the passenger are irrelevant to the achievement of this subtask. Any policy that successfully completes this subtask will have the same value function regardless of the source and destination locations of the passenger. By abstracting away the passenger source and destination, we obtain a huge savings in space. Instead of requiring 8000 values to represent the C functions for this task, we require only 400 values (4 actions, 25 locations, 4 possible values for t).

The advantages of this form of abstraction are similar to those obtained by Boutilier, Dearden and Goldszmidt (1995) in which belief network models of actions are exploited to simplify value iteration in stochastic planning. Indeed, one way of understanding the conditions of Definition 12 is to express them in the form of a decision diagram, as shown in Figure 7. The diagram shows that the irrelevant variables Y do not affect the rewards either directly or indirectly, and therefore, they do not affect either the value function or the optimal policy.

One rule for noticing cases where this abstraction condition holds is to examine the subgraph rooted at the given Max node i. If a set of state variables is irrelevant to the leaf state transition probabilities and reward functions and also to all pseudo-reward functions and termination conditions in the subgraph, then those variables satisfy the Max Node Irrelevance condition:

Lemma 3 *Let M be an MDP with associated MAXQ graph H, and let i be a Max node in H. Let X_i and Y_i be a partition of the state variables for M. A set of state variables Y_i is irrelevant to node i if*

- *For each primitive leaf node a that is a descendent of i,*

$$P(x',y'|x,y,a) = P(y'|x,y,a)P(x'|x,a) \text{ and}$$
$$R(x',y'|x,y,a) = R(x'|x,a),$$

- *For each internal node j that is equal to node i or is a descendent of i, $\tilde{R}_j(x',y') = \tilde{R}_j(x')$ and the termination predicate $T_j(x',y')$ is true iff $T_j(x')$.*

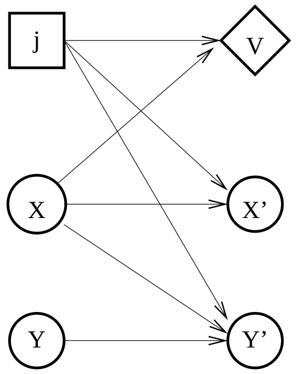

Figure 7: A dynamic decision diagram that represents the conditions of Definition 12. The probabilistic nodes X and Y represent the state variables at time t, and the nodes X' and Y' represent the state variables at a later time $t+N$. The square action node j is the chosen child subroutine, and the utility node V represents the value function $V(j,x)$ of that child action. Note that while X may influence Y', Y cannot affect X', and therefore, it cannot affect V.

Proof: We must show that any abstract hierarchical policy will give rise to an SMDP at node i whose transition probability distribution factors and whose reward function depends only on X_i. By definition, any abstract hierarchical policy will choose actions based only upon information in X_i. Because the primitive probability transition functions factor into an independent component for X_i and since the termination conditions at all nodes below i are based only on the variables in X_i, the probability transition function $P_i(x', y', N|x, y, a)$ must also factor into $P_i(y'|x, y, a)$ and $P_i(x', N|x, a)$. Similarly, all of the reward functions $V(j, x, y)$ must be equal to $V(j, x)$, because all rewards received within the subtree (either at the leaves or through pseudo-rewards) depend only on the variables in X_i. Therefore, the variables in Y_i are irrelevant for Max node i. **Q.E.D.**

In the Taxi task, the primitive navigation actions, North, South, East, and West only depend on the location of the taxi and not on the location of the passenger. The pseudo-reward function and termination condition for the MaxNavigate(t) node only depend on the location of the taxi (and the parameter t). Hence, this lemma applies, and the passenger source and destination are irrelevant for the MaxNavigate node.

5.1.2 Condition 2: Leaf Irrelevance

The second abstraction condition describes situations under which we can apply state abstractions to leaf nodes of the MAXQ graph. For leaf nodes, we can obtain a stronger result than Lemma 2 by using a slightly weaker definition of irrelevance.

Definition 13 (Leaf Irrelevance) *A set of state variables Y is* irrelevant for a primitive action *a of a MAXQ graph if for all states s the expected value of the reward function,*

$$V(a,s) = \sum_{s'} P(s'|s,a) R(s'|s,a)$$

does not depend on any of the values of the state variables in Y. In other words, for any pair of states s_1 and s_2 that differ only in their values for the variables in Y,

$$\sum_{s'_1} P(s'_1|s_1,a) R(s'_1|s_1,a) = \sum_{s'_2} P(s'_2|s_2,a) R(s'_2|s_2,a).$$

If this condition is satisfied at leaf a, then the following lemma shows that we can represent its value function $V(a,s)$ compactly.

Lemma 4 *Let M be an MDP with full-state MAXQ graph H, and suppose that state variables Y are irrelevant for leaf node a. Let $\chi(s) = x$ be the associated abstraction function that projects s onto the remaining relevant variables X. Then we can represent $V(a,s)$ for any state s by an abstracted value function $V(a,\chi(s)) = V(a,x)$.*

Proof: According to the definition of Leaf Irrelevance, any two states that differ only on the irrelevant state variables have the same value for $V(a,s)$. Hence, we can represent this unique value by $V(a,x)$. **Q.E.D.**

Here are two rules for finding cases where Leaf Irrelevance applies. The first rule shows that if the probability distribution factors, then we have Leaf Irrelevance.

Lemma 5 *Suppose the probability transition function for primitive action a, $P(s'|s,a)$, factors as $P(x',y'|x,y,a) = P(y'|x,y,a)P(x'|x,a)$ and the reward function satisfies $R(s'|s,a) = R(x'|x,a)$. Then the variables in Y are irrelevant to the leaf node a.*

Proof: Plug in to the definition of $V(a,s)$ and simplify.

$$\begin{aligned}
V(a,s) &= \sum_{s'} P(s'|s,a) R(s'|s,a) \\
&= \sum_{x',y'} P(y'|x,y,a) P(x'|x,a) R(x'|x,a) \\
&= \sum_{y'} P(y'|x,y,a) \sum_{x'} P(x'|x,a) R(x'|x,a) \\
&= \sum_{x'} P(x'|x,a) R(x'|x,a)
\end{aligned}$$

Hence, the expected reward for the action a depends only on the variables in X and not on the variables in Y. **Q.E.D.**

The second rule shows that if the reward function for a primitive action is constant, then we can apply state abstractions even if $P(s'|s,a)$ does not factor.

Lemma 6 *Suppose $R(s'|s,a)$ (the reward function for action a in MDP M) is always equal to a constant r_a. Then the entire state s is irrelevant to the primitive action a.*

Proof:

$$\begin{aligned} V(a,s) &= \sum_{s'} P(s'|s,a)R(s'|s,a) \\ &= \sum_{s'} P(s'|s,a)r_a \\ &= r_a. \end{aligned}$$

This does not depend on s, so the entire state is irrelevant to the primitive action a. **Q.E.D.**

This lemma is satisfied by the four leaf nodes North, South, East, and West in the taxi task, because their one-step reward is a constant (-1). Hence, instead of requiring 2000 values to store the V functions, we only need 4 values—one for each action. Similarly, the expected rewards of the Pickup and Putdown actions each require only 2 values, depending on whether the corresponding actions are legal or illegal. Hence, together, they require 4 values, instead of 1000 values.

5.1.3 Condition 3: Result Distribution Irrelevance

Now we consider a condition that results from "funnel" actions.

Definition 14 (Result Distribution Irrelevance). *A set of state variables Y_j is* irrelevant for the result distribution of action j *if, for all abstract policies π executed by node j and its descendents in the MAXQ hierarchy, the following holds: for all pairs of states s_1 and s_2 that differ only in their values for the state variables in Y_j,*

$$P^\pi(s', N|s_1, j) = P^\pi(s', N|s_2, j)$$

for all s' and N.

If this condition is satisfied for subtask j, then the C value of its parent task i can be represented compactly:

Lemma 7 *Let M be an MDP with full-state MAXQ graph H, and suppose that the set of state variables Y_j is irrelevant to the result distribution of action j, which is a child of Max node i. Let χ_{ij} be the associated abstraction function: $\chi_{ij}(s) = x$. Then we can define an abstract completion cost function $C^\pi(i, \chi_{ij}(s), j)$ such that for all states s,*

$$C^\pi(i, s, j) = C^\pi(i, \chi_{ij}(s), j).$$

Proof: The completion function for fixed policy π is defined as follows:

$$C^\pi(i, s, j) = \sum_{s', N} P(s', N|s, j) \cdot \gamma^N Q^\pi(i, s'). \tag{21}$$

Consider any two states s_1 and s_2, such that $\chi_{ij}(s_1) = \chi_{ij}(s_2) = x$. Under Result Distribution Irrelevance, their transition probability distributions are the same. Hence, the right-hand sides of (21) have the same value, and we can conclude that

$$C^\pi(i, s_1, j) = C^\pi(i, s_2, j).$$

Therefore, we can define an abstract completion function, $C^\pi(i, x, j)$ to represent this quantity. **Q.E.D.**

In undiscounted cumulative reward problems, the definition of result distribution irrelevance can be weakened to eliminate N, the number of steps. All that is needed is that for all pairs of states s_1 and s_2 that differ only in the irrelevant state variables, $P^\pi(s'|s_1, j) = P^\pi(s'|s_2, j)$ (for all s'). In the undiscounted case, Lemma 7 still holds under this revised definition.

It might appear that the result distribution irrelevance condition would rarely be satisfied, but we often find cases where the condition is true. Consider, for example, the Get subroutine for the taxi task. No matter what location the taxi has in state s, the taxi will be at the passenger's starting location when the Get finishes executing (i.e., because the taxi will have just completed picking up the passenger). Hence, the starting location is irrelevant to the resulting location of the taxi, and $P(s'|s_1, \mathsf{Get}) = P(s'|s_2, \mathsf{Get})$ for all states s_1 and s_2 that differ only in the taxi's location.

Note, however, that if we were maximizing discounted reward, the taxi's location would not be irrelevant, because the probability that Get will terminate in exactly N steps would depend on the location of the taxi, which could differ in states s_1 and s_2. Different values of N will produce different amounts of discounting in (21), and hence, we cannot ignore the taxi location when representing the completion function for Get.

But in the undiscounted case, by applying Lemma 7, we can represent $C(\mathsf{Root}, s, \mathsf{Get})$ using 16 distinct values, because there are 16 equivalence classes of states (4 source locations times 4 destination locations). This is much less than the 500 quantities in the unabstracted representation.

Note that although state variables Y may be irrelevant to the result distribution of a subtask j, they may be important *within* subtask j. In the Taxi task, the location of the taxi is critical for representing the value of $V(\mathsf{Get}, s)$, but it is irrelevant to the result state distribution for Get, and therefore it is irrelevant for representing $C(\mathsf{Root}, s, \mathsf{Get})$. Hence, the MAXQ decomposition is essential for obtaining the benefits of result distribution irrelevance.

"Funnel" actions arise in many hierarchical reinforcement learning problems. For example, abstract actions that move a robot to a doorway or that move a car onto the entrance ramp of a freeway have this property. The Result Distribution Irrelevance condition is applicable in all such situations as long as we are in the undiscounted setting.

5.1.4 Condition 4: Termination

The fourth condition is closely related to the "funnel" property. It applies when a subtask is guaranteed to cause its parent task to terminate in a goal state. In a sense, the subtask is funneling the environment into the set of states described by the goal predicate of the parent task.

Lemma 8 (Termination). *Let M_i be a task in a MAXQ graph such that for all states s where the goal predicate $G_i(s)$ is true, the pseudo-reward function $\tilde{R}_i(s) = 0$. Suppose there is a child task a and state s such that for all hierarchical policies π,*

$$\forall\, s' \; P_i^\pi(s', N|s, a) > 0 \;\Rightarrow\; G_i(s').$$

(i.e., every possible state s' that results from applying a in s will make the goal predicate, G_i, true.)

Then for any policy executed at node i, the completion cost $C(i, s, a)$ is zero and does not need to be explicitly represented.

Proof: When action a is executed in state s, it is guaranteed to result in a state s' such that $G_i(s)$ is true. By definition, goal states also satisfy the termination predicate $T_i(s)$, so task i will terminate. Because $G_i(s)$ is true, the terminal pseudo-reward will be zero, and hence, the completion function will always be zero. **Q.E.D.**

For example, in the Taxi task, in all states where the taxi is holding the passenger, the Put subroutine will succeed and result in a goal terminal state for Root. This is because the termination predicate for Put (i.e., that the passenger is at his or her destination location) implies the goal condition for Root (which is the same). This means that $C(\mathsf{Root}, s, \mathsf{Put})$ is uniformly zero, for all states s where Put is not terminated.

It is easy to detect cases where the Termination condition is satisfied. We only need to compare the termination predicate T_a of a subtask with the goal predicate G_i of the parent task. If the first implies the second, then the termination lemma is satisfied.

5.1.5 Condition 5: Shielding

The shielding condition arises from the structure of the MAXQ graph.

Lemma 9 (Shielding). *Let M_i be a task in a MAXQ graph and s be a state such that in all paths from the root of the graph down to node M_i there is a subtask j (possibly equal to i) whose termination predicate $T_j(s)$ is true, then the Q nodes of M_i do not need to represent C values for state s.*

Proof: In order for task i to be executed in state s, there must exist some path of ancestors of task i leading up to the root of the graph such that all of those ancestor tasks are not terminated. The condition of the lemma guarantees that this is false, and hence that task i cannot be executed in state s. Therefore, no C values need to be represented. **Q.E.D.**

As with the Termination condition, the Shielding condition can be verified by analyzing the structure of the MAXQ graph and identifying nodes whose ancestor tasks are terminated.

In the Taxi domain, a simple example of this arises in the Put task, which is terminated in all states where the passenger is not in the taxi. This means that we do not need to represent $C(\mathsf{Root}, s, \mathsf{Put})$ in these states. The result is that, when combined with the Termination condition above, we do not need to explicitly represent the completion function for Put at all!

5.1.6 Dicussion

By applying these five abstraction conditions, we obtain the following "safe" state abstractions for the Taxi task:

- North, South, East, and West. These terminal nodes require one quantity each, for a total of four values. (Leaf Irrelevance).

- Pickup and Putdown each require 2 values (legal and illegal states), for a total of four. (Leaf Irrelevance.)
- $\mathsf{QNorth}(t)$, $\mathsf{QSouth}(t)$, $\mathsf{QEast}(t)$, and $\mathsf{QWest}(t)$ each require 100 values (four values for t and 25 locations). (Max Node Irrelevance.)
- QNavigateForGet requires 4 values (for the four possible source locations). (The passenger destination is Max Node Irrelevant for MaxGet, and the taxi starting location is Result Distribution Irrelevant for the Navigate action.)
- QPickup requires 100 possible values, 4 possible source locations and 25 possible taxi locations. (Passenger destination is Max Node Irrelevant to MaxGet.)
- QGet requires 16 possible values (4 source locations, 4 destination locations). (Result Distribution Irrelevance.)
- QNavigateForPut requires only 4 values (for the four possible destination locations). (The passenger source and destination are Max Node Irrelevant to MaxPut; the taxi location is Result Distribution Irrelevant for the Navigate action.)
- QPutdown requires 100 possible values (25 taxi locations, 4 possible destination locations). (Passenger source is Max Node Irrelevant for MaxPut.)
- QPut requires 0 values. (Termination and Shielding.)

This gives a total of 632 distinct values, which is much less than the 3000 values required by flat Q learning. Hence, we can see that by applying state abstractions, the MAXQ representation can give a much more compact representation of the value function.

A key thing to note is that with these state abstractions, the value function is decomposed into a sum of terms such that no single term depends on the entire state of the MDP, even though the value function as a whole does depend on the entire state of the MDP. For example, consider again the state described in Figures 1 and 4. There, we showed that the value of a state s_1 with the passenger at R, the destination at B, and the taxi at (0,3) can be decomposed as

$$\begin{aligned} V(\mathsf{Root}, s_1) &= V(\mathsf{North}, s_1) + C(\mathsf{Navigate}(R), s_1, \mathsf{North}) + \\ &\quad C(\mathsf{Get}, s_1, \mathsf{Navigate}(R)) + C(\mathsf{Root}, s_1, \mathsf{Get}) \end{aligned}$$

With state abstractions, we can see that each term on the right-hand side only depends on a subset of the features:

- $V(\mathsf{North}, s_1)$ is a constant
- $C(\mathsf{Navigate}(R), s_1, \mathsf{North})$ depends only on the taxi location and the passenger's source location.
- $C(\mathsf{Get}, s_1, \mathsf{Navigate}(R))$ depends only on the source location.
- $C(\mathsf{Root}, s_1, \mathsf{Get})$ depends only on the passenger's source and destination.

Without the MAXQ decomposition, no features are irrelevant, and the value function depends on the entire state.

What prior knowledge is required on the part of a programmer in order to identify these state abstractions? It suffices to know some qualitative constraints on the one-step reward functions, the one-step transition probabilities, and termination predicates, goal predicates, and pseudo-reward functions within the MAXQ graph. Specifically, the Max Node Irrelevance and Leaf Irrelevance conditions require simple analysis of the one-step transition function and the reward and pseudo-reward functions. Opportunities to apply the Result Distribution Irrelevance condition can be found by identifying "funnel" effects that result from the definitions of the termination conditions for operators. Similarly, the Shielding and Termination conditions only require analysis of the termination predicates of the various subtasks. Hence, applying these five conditions to introduce state abstractions is a straightforward process, and once a model of the one-step transition and reward functions has been learned, the abstraction conditions can be checked to see if they are satisfied.

5.2 Convergence of MAXQ-Q with State Abstraction

We have shown that state abstractions can be safely introduced into the MAXQ value function decomposition under the five conditions described above. However, these conditions only guarantee that the value function of any fixed *abstract* hierarchical policy can be represented—they do not show that recursively optimal policies can be represented, nor do they show that the MAXQ-Q learning algorithm will find a recursively optimal policy when it is forced to use these state abstractions. The goal of this section is to prove these two results: (a) that the ordered recursively-optimal policy is an abstract policy (and, hence, can be represented using state abstractions) and (b) that MAXQ-Q will converge to this policy when applied to a MAXQ graph with safe state abstractions.

Lemma 10 *Let M be an MDP with full-state MAXQ graph H and abstract-state MAXQ graph $\chi(H)$ where the abstractions satisfy the five conditions given above. Let ω be an ordering over all actions in the MAXQ graph. Then the following statements are true:*

- *The unique ordered recursively-optimal policy π_r^* defined by M, H, and ω is an abstract policy (i.e., it depends only on the relevant state variables at each node; see Definition 11),*
- *The C and V functions in $\chi(H)$ can represent the projected value function of π_r^*.*

Proof: The five abstraction lemmas tell us that if the ordered recursively-optimal policy is abstract, then the C and V functions of $\chi(H)$ can represent its value function. Hence, the heart of this lemma is the first claim. The last two forms of abstraction (Shielding and Termination) do not place any restrictions on abstract policies, so we ignore them in this proof.

The proof is by induction on the levels of the MAXQ graph, starting at the leaves. As a base case, let us consider a Max node i all of whose children are primitive actions. In this case, there are no policies executed *within* the children of the Max node. Hence if variables Y_i are irrelevant for node i, then we can apply our abstraction lemmas to represent the value function of any policy at node i—not just abstract policies. Consequently, the value function of any optimal policy for node i can be represented, and it will have the property that

$$Q^*(i, s_1, a) = Q^*(i, s_2, a) \tag{22}$$

for any states s_1 and s_2 such that $\chi_i(s_1) = \chi_i(s_2)$.

Now let us impose the action ordering ω to compute the optimal *ordered* policy. Consider two actions a_1 and a_2 such that $\omega(a_1, a_2)$ (i.e., ω prefers a_1), and suppose that there is a "tie" in the Q^* function at state s_1 such that the values

$$Q^*(i, s_1, a_1) = Q^*(i, s_1, a_2)$$

and they are the only two actions that maximize Q^* in this state. Then the optimal ordered policy must choose a_1. Now in all other states s_2 such that $\chi_i(s_1) = \chi_i(s_2)$, we have just established in (22) that the Q^* values will be the same. Hence, the same tie will exist between a_1 and a_2, and hence, the optimal ordered policy must make the same choice in all such states. Hence, the optimal ordered policy for node i is an abstract policy.

Now let us turn to the recursive case at Max node i. Make the inductive assumption that the ordered recursively-optimal policy is abstract within all descendent nodes and consider the locally optimal policy at node i. If Y is a set of state variables that are irrelevant to node i, Corollary 2 tells us that $Q^*(i, s_1, j) = Q^*(i, s_2, j)$ for all states s_1 and s_2 such that $\chi_i(s_1) = \chi_i(s_2)$. Similarly, if Y is a set of variables irrelevant to the result distribution of a particular action j, then Lemma 7 tells us the same thing. Hence, by the same ordering argument given above, the ordered optimal policy at node i must be abstract. By induction, this proves the lemma. **Q.E.D.**

With this lemma, we have established that the combination of an MDP M, an abstract MAXQ graph H, and an action ordering defines a unique recursively-optimal ordered abstract policy. We are now ready to prove that MAXQ-Q will converge to this policy.

Theorem 4 *Let $M = \langle S, A, P, R, P_0 \rangle$ be either an episodic MDP for which all deterministic policies are proper or a discounted infinite horizon MDP with discount factor $\gamma < 1$. Let H be an unabstracted MAXQ graph defined over subtasks $\{M_0, \ldots, M_k\}$ with pseudo-reward functions $\tilde{R}_i(s')$. Let $\chi(H)$ be a state-abstracted MAXQ graph defined by applying state abstractions χ_i to each node i of H under the five conditions given above. Let $\pi_x(i, \chi_i(s))$ be an abstract ordered GLIE exploration policy at each node i and state s whose decisions depend only on the "relevant" state variables at each node i. Let π_r^* be the unique recursively-optimal hierarchical policy defined by π_x, M, and $\tilde{R}$. Then with probability 1, algorithm MAXQ-Q applied to $\chi(H)$ converges to π_r^* provided that the learning rates $\alpha_t(i)$ satisfy Equation (15) and the one-step rewards are bounded.*

Proof: Rather than repeating the entire proof for MAXQ-Q, we will only describe what must change under state abstraction. The last two forms of state abstraction refer to states whose values can be inferred from the structure of the MAXQ graph, and therefore do not need to be represented at all. Since these values are not updated by MAXQ-Q, we can ignore them. We will now consider the first three forms of state abstraction in turn.

We begin by considering primitive leaf nodes. Let a be a leaf node and let Y be a set of state variables that are Leaf Irrelevant for a. Let $s_1 = (x, y_1)$ and $s_2 = (x, y_2)$ be two states that differ only in their values for Y. Under Leaf Irrelevance, the probability transitions $P(s'_1|s_1, a)$ and $P(s'_2|s_2, a)$ need not be the same, but the expected reward of performing a in both states must be the same. When MAXQ-Q visits an abstract state x, it does not "know" the value of y, the part of the state that has been abstracted away. Nonetheless, it draws a sample according to $P(s'|x, y, a)$, receives a reward $R(s'|x, y, a)$, and updates its estimate of $V(a, x)$ (line 4 of MAXQ-Q). Let $P_t(y)$ be the probability that MAXQ-Q is visiting (x, y) given that the unabstracted part of the state is x. Then Line 4 of MAXQ-Q is computing a stochastic approximation to

$$\sum_{s',N,y} P_t(y) P_t(s', N|x, y, a) R(s'|x, y, a).$$

We can write this as

$$\sum_{y} P_t(y) \sum_{s',N} P_t(s', N|x, y, a) R(s'|x, y, a).$$

According to Leaf Irrelevance, the inner sum has the same value for all states s such that $\chi(s) = x$. Call this value $r_0(x)$. This gives

$$\sum_{y} P_t(y) r_0(x),$$

which is equal to $r_0(x)$ for any distribution $P_t(y)$. Hence, MAXQ-Q converges under Leaf Irrelevance abstractions.

Now let us turn to the two forms of abstraction that apply to internal nodes: Max Node Irrelevance and Result Distribution Irrelevance. Consider the SMDP defined at each node i of the abstracted MAXQ graph at time t during MAXQ-Q. This would be an ordinary SMDP with transition probability function $P_t(x', N|x, a)$ and reward function $V_t(a, x) + \tilde{R}_i(x')$ except that when MAXQ-Q draws samples of state transitions, they are drawn according to the distribution $P_t(s', N|s, a)$ over the original state space. To prove the theorem, we must show that drawing (s', N) according to this second distribution is equivalent to drawing (x', N) according to the first distribution.

For Max Node Irrelevance, we know that for all abstract policies applied to node i and its descendents, the transition probability distribution factors as

$$P(s', N|s, a) = P(y'|x, y, a) P(x', N|x, a).$$

Because the exploration policy is an abstract policy, $P_t(s', N|s, a)$ factors in this way. This means that the Y_i components of the state cannot affect the X_i components, and hence, sampling from $P_t(s', N|s, a)$ and discarding the Y_i values gives samples for $P_t(x', N|x, a)$. Therefore, MAXQ-Q will converge under Max Node Irrelevance abstractions.

Finally, consider Result Distribution Irrelevance. Let j be a child of node i, and suppose Y_j is a set of state variables that are irrelevant to the result distribution of j. When the SMDP at node i wishes to draw a sample from $P_t(x', N|x, j)$, it does not "know" the current value of y, the irrelevant part of the current state. However, this does not matter, because Result Distribution Irrelevance means that for all possible values of y, $P_t(x', y', N|x, y, j)$ is the same. Hence, MAXQ-Q will converge under Result Distribution Irrelevance abstractions.

In each of these three cases, MAXQ-Q will converge to a locally-optimal ordered policy at node i in the MAXQ graph. By Lemma 10, this produces a locally-optimal ordered policy for the unabstracted SMDP at node i. Hence, by induction, MAXQ-Q will converge to the unique ordered recursively optimal policy π_r^* defined by MAXQ-Q H, MDP M, and ordered exploration policy π_x. **Q.E.D.**

5.3 The Hierarchical Credit Assignment Problem

There are still some situations where we would like to introduce state abstractions but where the five properties described above do not permit them. Consider the following modification of the taxi problem. Suppose that the taxi has a fuel tank and that each time the taxi moves one square, it costs one unit of fuel. If the taxi runs out of fuel before delivering the passenger to his or her destination, it receives a reward of -20, and the trial ends. Fortunately, there is a filling station where the taxi can execute a Fillup action to fill the fuel tank.

To solve this modified problem using the MAXQ hierarchy, we can introduce another subtask, Refuel, which has the goal of moving the taxi to the filling station and filling the tank. MaxRefuel is a child of MaxRoot, and it invokes Navigate(t) (with t bound to the location of the filling station) to move the taxi to the filling station.

The introduction of fuel and the possibility that we might run out of fuel means that we must include the current amount of fuel as a feature in representing every C value (for internal nodes) and V value (for leaf nodes) throughout the MAXQ graph. This is unfortunate, because our intuition tells us that the amount of fuel should have no influence on our decisions inside the Navigate(t) subtask. That is, either the taxi will have enough fuel to reach the target t (in which case, the chosen navigation actions do not depend on the fuel), or else the taxi will not have enough fuel, and hence, it will fail to reach t regardless of what navigation actions are taken. In other words, the Navigate(t) subtask should not need to worry about the amount of fuel, because even if there is not enough fuel, there is no action that Navigate(t) can take to get more fuel. Instead, it is the top-level subtasks that should be monitoring the amount of fuel and deciding whether to go refuel, to go pick up the passenger, or to go deliver the passenger.

Given this intuition, it is natural to try abstracting away the "amount of remaining fuel" within the Navigate(t) subtask. However, this doesn't work, because when the taxi runs out of fuel and a -20 reward is given, the QNorth, QSouth, QEast, and QWest nodes cannot "explain" why this reward was received—that is, they have no consistent way of setting their C tables to predict when this negative reward will occur, because their C values ignore the amount of fuel in the tank. Stated more formally, the difficulty is that the Max Node Irrelevance condition is not satisfied because the one-step reward function $R(s'|s,a)$ for these actions depends on the amount of fuel.

We call this the *hierarchical credit assignment problem.* The fundamental issue here is that in the MAXQ decomposition all information about rewards is stored in the leaf nodes of the hierarchy. We would like to separate out the basic rewards received for navigation (i.e., -1 for each action) from the reward received for exhausting fuel (-20). If we make the reward at the leaves only depend on the location of the taxi, then the Max Node Irrelevance condition will be satisfied.

One way to do this is to have the programmer manually decompose the reward function and indicate which nodes in the hierarchy will "receive" each reward. Let $R(s'|s,a) = \sum_i R(i,s'|s,a)$ be a decomposition of the reward function, such that $R(i,s'|s,a)$ specifies that part of the reward that must be handled by Max node i. In the modified taxi problem, for example, we can decompose the reward so that the leaf nodes receive all of the original penalties, but the out-of-fuel rewards must be handled by MaxRoot. Lines 15 and 16 of the MAXQ-Q algorithm are easily modified to include $R(i,s'|s,a)$.

In most domains, we believe it will be easy for the designer of the hierarchy to decompose the reward function. It has been straightforward in all of the problems we have studied. However, an interesting problem for future research is to develop an algorithm that can solve the hierarchical credit assignment problem autonomously.

6. Non-Hierarchical Execution of the MAXQ Hierarchy

Up to this point in the paper, we have focused exclusively on representing and learning hierarchical policies. However, often the optimal policy for a MDP is not strictly hierarchical. Kaelbling (1993) first introduced the idea of deriving a non-hierarchical policy from the value function of a hierarchical policy. In this section, we exploit the MAXQ decomposition to generalize her ideas and apply them recursively at all levels of the hierarchy. We will describe two methods for non-hierarchical execution.

The first method is based on the dynamic programming algorithm known as policy iteration. The policy iteration algorithm starts with an initial policy π^0. It then repeats the following two steps until the policy converges. In the *policy evaluation* step, it computes the value function V^{π_k} of the current policy π_k. Then, in the *policy improvement step*, it computes a new policy, π_{k+1} according to the rule

$$\pi_{k+1}(s) := \operatorname*{argmax}_a \sum_{s'} P(s'|s,a)[R(s'|s,a) + \gamma V^{\pi_k}(s')]. \tag{23}$$

Howard (1960) proved that if π_k is not an optimal policy, then π_{k+1} is guaranteed to be an improvement. Note that in order to apply this method, we need to know the transition probability distribution $P(s'|s,a)$ and the reward function $R(s'|s,a)$.

If we know $P(s'|s,a)$ and $R(s'|s,a)$, we can use the MAXQ representation of the value function to perform one step of policy iteration. We start with a hierarchical policy π and represent its value function using the MAXQ hierarchy (e.g., π could have been learned via MAXQ-Q). Then, we can perform one step of policy improvement by applying Equation (23) using $V^\pi(0,s')$ (computed by the MAXQ hierarchy) to compute $V^\pi(s')$.

Corollary 3 *Let $\pi^g(s) = \operatorname{argmax}_a \sum_{s'} P(s'|s,a)[R(s'|s,a) + \gamma V^\pi(0,s)]$, where $V^\pi(0,s)$ is the value function computed by the MAXQ hierarchy and a is a primitive action. Then, if π was not an optimal policy, π^g is strictly better for at least one state in S.*

Proof: This is a direct consequence of Howard's policy improvement theorem. **Q.E.D.**

Unfortunately, we can't iterate this policy improvement process, because the new policy, π^g is very unlikely to be a hierarchical policy (i.e., it is unlikely to be representable in

Table 5: The procedure for executing the one-step greedy policy.

```
  procedure ExecuteHGPolicy(s)
1     repeat
2         Let ⟨V(0, s), a⟩ := EvaluateMaxNode(0, s)
3         execute primitive action a
4         Let s be the resulting state
  end // ExecuteHGPolicy
```

terms of local policies for each node of the MAXQ graph). Nonetheless, one step of policy improvement can give very significant improvements.

This approach to non-hierarchical execution ignores the internal structure of the MAXQ graph. In effect, the MAXQ hierarchy is just viewed as a way to represent V^π—any other representation would give the same one-step improved policy π^g.

The second approach to non-hierarchical execution borrows an idea from Q learning. One of the great beauties of the Q representation for value functions is that we can compute one step of policy improvement without knowing $P(s'|s,a)$, simply by taking the new policy to be $\pi^g(s) := \text{argmax}_a Q(s,a)$. This gives us the same one-step greedy policy as we computed above using one-step lookahead. With the MAXQ decomposition, we can perform these policy improvement steps *at all levels of the hierarchy.*

We have already defined the function that we need. In Table 3 we presented the function EvaluateMaxNode, which, given the current state s, conducts a search along all paths from a given Max node i to the leaves of the MAXQ graph and finds the path with the best value (i.e., with the maximum sum of C values along the path, plus the V value at the leaf). This is equivalent to computing the best action greedily at each level of the MAXQ graph. In addition, EvaluateMaxNode returns the primitive action a at the end of this best path. This action a would be the first primitive action to be executed if the learned hierarchical policy were executed starting in the current state s. Our second method for non-hierarchical execution of the MAXQ graph is to call EvaluateMaxNode in each state, and execute the primitive action a that is returned. The pseudo-code is shown in Table 5.

We will call the policy computed by ExecuteHGPolicy the *hierarchical greedy policy*, and denote it π^{hg*}, where the superscript * indicates that we are computing the greedy action at each time step. The following theorem shows that this can give a better policy than the original, hierarchical policy.

Theorem 5 *Let G be a MAXQ graph representing the value function of hierarchical policy π (i.e., in terms of $C^\pi(i,s,j)$, computed for all i, s, and j). Let $V^{hg}(0,s)$ be the value computed by* ExecuteHGPolicy *(line 2), and let π^{hg*} be the resulting policy. Define V^{hg*} to be the value function of π^{hg*}. Then for all states s, it is the case that*

$$V^\pi(s) \leq V^{hg}(0,s) \leq V^{hg*}(s). \tag{24}$$

Proof: (sketch) The left inequality in Equation (24) is satisfied by construction by line 6 of EvaluateMaxNode. To see this, consider that the original hierarchical policy, π, can be viewed as choosing a "path" through the MAXQ graph running from the root to one of the leaf nodes, and $V^\pi(0, s)$ is the sum of the C^π values along this chosen path (plus the V^π value at the leaf node). In contrast, EvaluateMaxNode performs a traversal of *all* paths through the MAXQ graph and finds the *best* path, that is, the path with the largest sum of C^π (and leaf V^π) values. Hence, $V^{hg}(0, s)$ must be at least as large as $V^\pi(0, s)$.

To establish the right inequality, note that by construction $V^{hg}(0, s)$ is the value function of a policy, call it π^{hg}, that chooses one action greedily at each level of the MAXQ graph (recursively), and then follows π thereafter. This is a consequence of the fact that line 6 of EvaluateMaxNode has C^π on its right-hand side, and C^π represents the cost of "completing" each subroutine by following π, not by following some other, greedier, policy. (In Table 3, C^π is written as C_t.) However, when we execute ExecuteHGPolicy (and hence, execute π^{hg*}), we have an opportunity to improve upon π and π^{hg} at each time step. Hence, $V^{hg}(0, s)$ is an underestimate of the actual value of π^{hg*}. **Q.E.D.**

Note that this theorem only works in one direction. It says that if we can find a state where $V^{hg}(0, s) > V^\pi(s)$, then the greedy policy, π^{hg*}, will be strictly better than π. However, it could be that π is not an optimal policy and yet the structure of the MAXQ graph prevents us from considering an action (either primitive or composite) that would improve π. Hence, unlike the policy improvement theorem of Howard (where all primitive actions are always eligible to be chosen), we do not have a guarantee that if π is suboptimal, then the hierarchically greedy policy is a strict improvement.

In contrast, if we perform one-step policy improvement as discussed at the start of this section, Corollary 3 guarantees that we will improve the policy. So we can see that in general, neither of these two methods for non-hierarchical execution is always better than the other. Nonetheless, the first method only operates at the level of individual primitive actions, so it is not able to produce very large improvements in the policy. In contrast, the hierarchical greedy method can obtain very large improvements in the policy by changing which actions (i.e., subroutines) are chosen near the root of the hierarchy. Hence, in general, hierarchical greedy execution is probably the better method. (Of course, the value functions of both methods could be computed, and the one with the better estimated value could be executed.)

Sutton, et al. (1999) have simultaneously developed a closely-related method for non-hierarchical execution of macros. Their method is equivalent to ExecuteHGPolicy for the special case where the MAXQ hierarchy has only one level of subtasks. The interesting aspect of ExecuteHGPolicy is that it permits greedy improvements at all levels of the tree to influence which action is chosen.

Some care must be taken in applying Theorem 5 to a MAXQ hierarchy whose C values have been learned via MAXQ-Q. Being an online algorithm, MAXQ-Q will not have correctly learned the values of *all* states at all nodes of the MAXQ graph. For example, in the taxi problem, the value of $C(\mathsf{Put}, s, \mathsf{QPutdown})$ will not have been learned very well except at the four special locations R, G, B, and Y. This is because the Put subtask cannot be executed until the passenger is in the taxi, and this usually means that a Get has just been completed, so the taxi is at the passenger's source location. During exploration, both children of Put will be tried in such states. The PutDown will usually fail (and receive a negative reward), whereas the Navigate will eventually succeed (perhaps after lengthy exploration) and take the taxi to the destination location. Now because of all-states updating, the values for $C(\mathsf{Put}, s, \mathsf{Navigate}(t))$ will have been learned at all of the states along the path to the passenger's destination, but the C values for the Putdown action will only be learned for the passenger's source and destination locations. Hence, if we train the MAXQ representation using hierarchical execution (as in MAXQ-Q), and then switch to hierarchically-greedy execution, the results will be quite bad. In particular, we need to introduce hierarchically-greedy execution early enough so that the exploration policy is still actively exploring. (In theory, a GLIE exploration policy never ceases to explore, but in practice, we want to find a good policy quickly, not just asymptotically).

Of course an alternative would be to use hierarchically-greedy execution from the very beginning of learning. However, remember that the higher nodes in the MAXQ hierarchy need to obtain samples of $P(s', N|s, a)$ for each child action a. If the hierarchical greedy execution interrupts child a before it has reached a terminal state (i.e., because at some state along the way, another subtask appears better to EvaluateMaxNode), then these samples cannot be obtained. Hence, it is important to begin with purely hierarchical execution during training, and make a transition to greedy execution at some point.

The approach we have taken is to implement MAXQ-Q in such a way that we can specify a number of primitive actions L that can be taken hierarchically before the hierarchical execution is "interrupted" and control returns to the top level (where a new action can be chosen greedily). We start with L set very large, so that execution is completely hierarchical—when a child action is invoked, we are committed to execute that action until it terminates. However, gradually, we reduce L until it becomes 1, at which point we have hierarchical greedy execution. We time this so that it reaches 1 at about the same time our Boltzmann exploration cools to a temperature of 0.1 (which is where exploration effectively has halted). As the experimental results will show, this generally gives excellent results with very little added exploration cost.

7. Experimental Evaluation of the MAXQ Method

We have performed a series of experiments with the MAXQ method with three goals in mind: (a) to understand the expressive power of the value function decomposition, (b) to characterize the behavior of the MAXQ-Q learning algorithm, and (c) to assess the relative importance of temporal abstraction, state abstraction, and non-hierarchical execution. In this section, we describe these experiments and present the results.

7.1 The Fickle Taxi Task

Our first experiments were performed on a modified version of the taxi task. This version incorporates two changes to the task described in Section 3.1. First, each of the four navigation actions is noisy, so that with probability 0.8 it moves in the intended direction, but with probability 0.1 it instead moves to the right (of the intended direction) and with probability 0.1 it moves to the left. The purpose of this change is to create a more realistic and more difficult challenge for the learning algorithms. The second change is that after the taxi has picked up the passenger and moved one square away from the passenger's source location, the passenger changes his or her destination location with probability 0.3. The purpose of this change is to create a situation where the optimal policy is not a hierarchical policy so that the effectiveness of non-hierarchical execution can be measured.

We compared four different configurations of the learning algorithm: (a) flat Q learning, (b) MAXQ-Q learning without any form of state abstraction, (c) MAXQ-Q learning with state abstraction, and (d) MAXQ-Q learning with state abstraction and greedy execution. These configurations are controlled by many parameters. These include the following: (a) the initial values of the Q and C functions, (b) the learning rate (we employed a fixed learning rate), (c) the cooling schedule for Boltzmann exploration (the GLIE policy that we employed), and (d) for non-hierarchical execution, the schedule for decreasing L, the number of steps of consecutive hierarchical execution. We optimized these settings separately for each configuration with the goal of matching or exceeding (with as few primitive training actions as possible) the best policy that we could code by hand. For Boltzmann exploration, we established an initial temperature and then a cooling rate. A separate temperature is maintained for each Max node in the MAXQ graph, and its temperature is reduced by multiplying by the cooling rate each time that subtask terminates in a goal state.

The process of optimizing the parameter settings for each algorithm is time-consuming, both for flat Q learning and for MAXQ-Q. The most critical parameter is the schedule for cooling the temperature of Boltzmann exploration: if this is cooled too rapidly, then the algorithms will converge to a suboptimal policy. In each case, we tested nine different cooling rates. To choose the different cooling rates for the various subtasks, we started by using fixed policies (e.g., either random or hand-coded) for all subtasks except the subtasks closest to the leaves. Then, once we had chosen schedules for those subtasks, we allowed their parent tasks to learn their policies while we tuned their cooling rates, and so on. One nice effect of our method of cooling the temperature only when a subtask terminates is that it naturally causes the subtasks higher in the MAXQ graph to cool more slowly. This meant that good results could often be obtained just by using the same cooling rate for all Max nodes.

The choice of learning rate is easier, since it is determined primarily by the degree of stochasticity in the environment. We only tested three or four different rates for each configuration. The initial values for the Q and C functions were set based on our knowledge of the problems—no experiments were required.

We took more care in tuning these parameters for these experiments than one would normally take in a real application, because we wanted to ensure that each method was compared under the best possible conditions. The general form of the results (particularly the speed of learning) is the same for wide ranges of the cooling rate and learning rate parameter settings.

The following parameters were selected based on the tuning experiments. For flat Q learning: initial Q values of 0.123 in all states, learning rate 0.25, and Boltzmann exploration with an initial temperature of 50 and a cooling rate of 0.9879. (We use initial values that end in .123 as a "signature" during debugging to detect when a weight has been modified.)

For MAXQ-Q learning without state abstraction, we used initial values of 0.123, a learning rate of 0.50, and Boltzmann exploration with an initial temperature of 50 and cooling rates of 0.9996 at MaxRoot and MaxPut, 0.9939 at MaxGet, and 0.9879 at MaxNavigate.

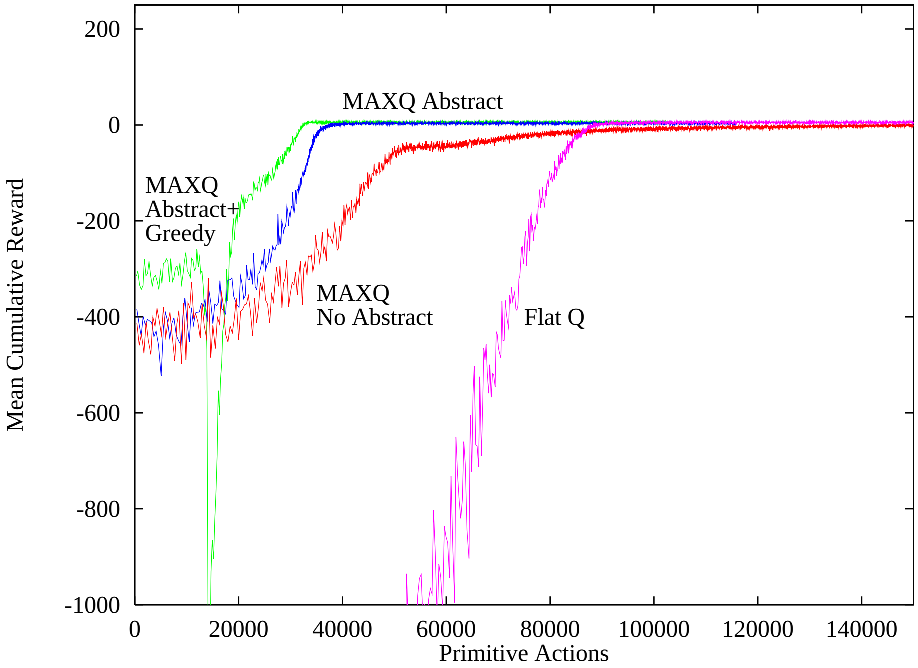

Figure 8: Comparison of performance of hierarchical MAXQ-Q learning (without state abstractions, with state abstractions, and with state abstractions combined with hierarchical greedy evaluation) to flat Q learning.

For MAXQ-Q learning with state abstraction, we used initial values of 0.123, a learning rate of 0.25, and Boltzmann exploration with an initial temperature of 50 and cooling rates of 0.9074 at MaxRoot, 0.9526 at MaxPut, 0.9526 at MaxGet, and 0.9879 at MaxNavigate.

For MAXQ-Q learning with non-hierarchical execution, we used the same settings as with state abstraction. In addition, we initialized L to 500 and decreased it by 10 with each trial until it reached 1. So after 50 trials, execution was completely greedy.

Figure 8 shows the averaged results of 100 training runs. Each training run involves performing repeated trials until convergence. Because the different trials execute different numbers of primitive actions, we have just plotted the number of primitive actions on the horizontal axis rather than the number of trials.

The first thing to note is that all forms of MAXQ learning have better initial performance than flat Q learning. This is because of the constraints introduced by the MAXQ hierarchy. For example, while the agent is executing a Navigate subtask, it will never attempt to pickup or putdown the passenger, because those actions are not available to Navigate. Similarly, the agent will never attempt to putdown the passenger until it has first picked up the passenger (and vice versa) because of the termination conditions of the Get and Put subtasks.

The second thing to notice is that without state abstractions, MAXQ-Q learning actually takes longer to converge, so that the Flat Q curve crosses the MAXQ/no abstraction curve. This shows that without state abstraction, the cost of learning the huge number of parameters in the MAXQ representation is not really worth the benefits. We suspect this is a consequence of the model-free nature of the MAXQ-Q algorithm. The MAXQ decomposition represents some information redundantly. For example, the cost of performing a Put subtask is computed both as $C(\mathsf{Root}, s, \mathsf{Get})$ and also as $V(\mathsf{Put}, s)$. A model-based algorithm could compute both of these from a learned model, but MAXQ-Q must learn each of them separately from experience.

The third thing to notice is that with state abstractions, MAXQ-Q converges very quickly to a hierarchically optimal policy. This can be seen more clearly in Figure 9, which focuses on the range of reward values in the neighborhood of the optimal policy. Here we can see that MAXQ with abstractions attains the hierarchically optimal policy after approximately 40,000 steps, whereas flat Q learning requires roughly twice as long to reach the same level. However, flat Q learning, of course, can continue onward and reach optimal performance, whereas with the MAXQ hierarchy, the best hierarchical policy is slow to respond to the "fickle" behavior of the passenger when he/she changes the destination.

The last thing to notice is that with greedy execution, the MAXQ policy is also able to attain optimal performance. But as the execution becomes "more greedy", there is a temporary drop in performance, because MAXQ-Q must learn C values in new regions of the state space that were not visited by the recursively optimal policy. Despite this drop in performance, greedy MAXQ-Q recovers rapidly and reaches hierarchically optimal performance faster than purely-hierarchical MAXQ-Q learning. Hence, there is no added cost—in terms of exploration—for introducing greedy execution.

This experiment presents evidence in favor of three claims: first, that hierarchical reinforcement learning can be much faster than flat Q learning; second, that state abstraction is required by MAXQ-Q learning for good performance; and third, that non-hierarchical execution can produce significant improvements in performance with little or no added exploration cost.

7.2 Kaelbling's HDG Method

The second task that we will consider is a simple maze task introduced by Leslie Kaelbling (1993) and shown in Figure 11. In each trial of this task, the agent starts in a randomly-chosen state and must move to a randomly-chosen goal state using the usual North, South, East, and West operators (we employed deterministic operators). There is a small cost for each move, and the agent must minimize the undiscounted sum of these costs.

Because the goal state can be in any of 100 different locations, there are actually 100 different MDPs. Kaelbling's HDG method starts by choosing an arbitrary set of landmark states and defining a Voronoi partition of the state space based on the Manhattan distances to these landmarks (i.e., two states belong to the same Voronoi cell iff they have the same nearest landmark). The method then defines one subtask for each landmark l. The subtask is to move from any state in the current Voronoi cell *or in any neighboring Voronoi cell* to the landmark l. Optimal policies for these subtasks are then computed.

Once HDG has the policies for these subtasks, it can solve the abstract Markov Decision Problem of moving from each landmark state to any other landmark state using the subtask solutions as macro actions (subroutines). So it computes a value function for this MDP.

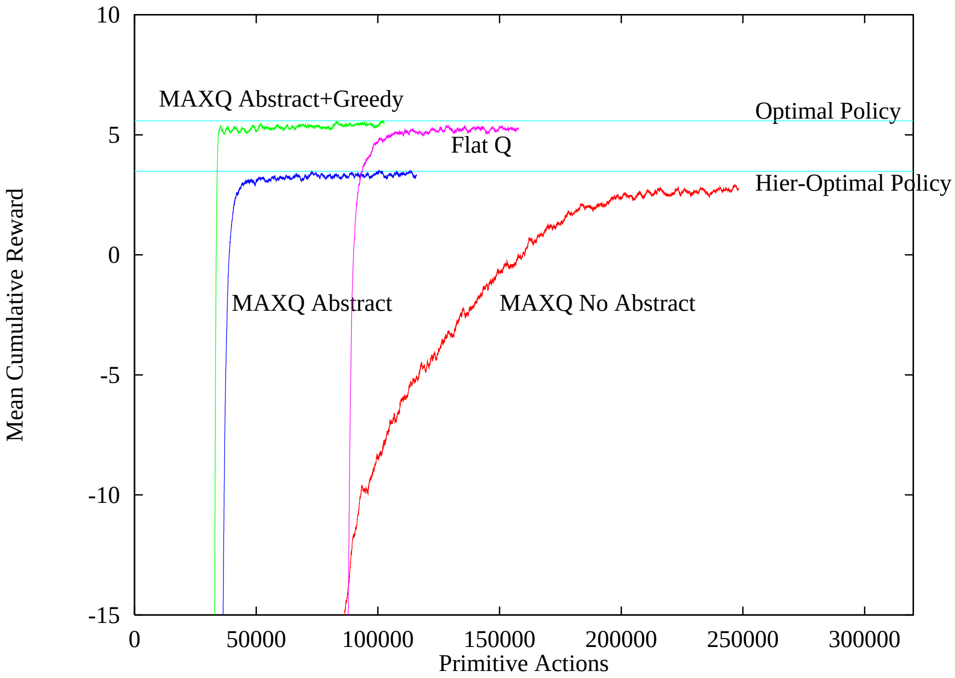

Figure 9: Close-up view of the previous figure. This figure also shows two horizontal lines indicating optimal performance and hierarchically optimal performance in this domain. To make this figure more readable, we have applied a 100-step moving average to the data points (which are themselves the average of 100 runs).

Finally, for each possible destination location g within a Voronoi cell for landmark l, the HDG method computes the optimal policy of getting from l to g.

By combining these subtasks, the HDG method can construct a good approximation to the optimal policy as follows. In addition to the value functions discussed above, the agent maintains two other functions: $NL(s)$, the name of the landmark nearest to state s, and $N(l)$, a list of the landmarks of the cells that are immediate neighbors of cell l. By combining these, the agent can build a list for each state s of the current landmark and the landmarks of the neighboring cells. For each such landmark, the agent computes the sum of three terms:

(t1) the expected cost of reaching that landmark,

(t2) the expected cost of moving from that landmark to the landmark in the goal cell, and

(t3) the expected cost of moving from the goal-cell landmark to the goal state.

Note that while terms (t1) and (t3) can be exact estimates, term (t2) is computed using the landmark subtasks as subroutines. This means that the corresponding path must pass through the intermediate landmark states rather than going directly to the goal landmark.

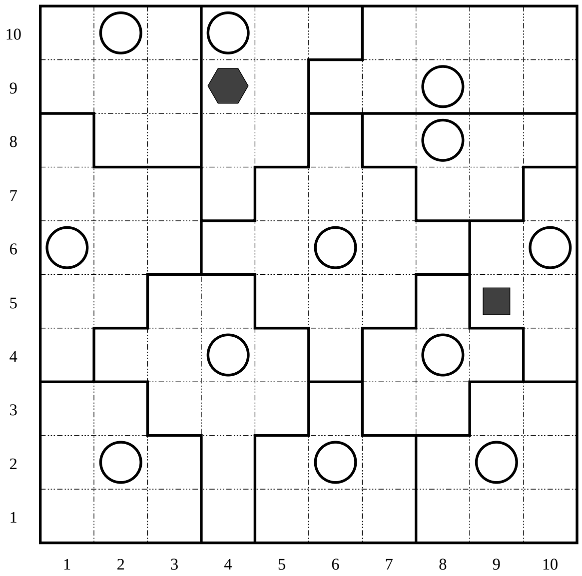

Figure 10: Kaelbling's 10-by-10 navigation task. Each circled state is a landmark state, and the heavy lines show the boundaries of the Voronoi cells. In each episode, a start state and a goal state are chosen at random. In this figure, the start state is shown by the black square, and the goal state is shown by the black hexagon.

Hence, term (t2) is typically an overestimate of the required distance. (Also note that (t3) is the same for all choices of the intermediate landmarks, so it does not need to be explicitly included in the computation of the best action until the agent enters the cell containing the goal.)

Given this information, the agent then chooses to move toward the best of the landmarks (unless the agent is already in the goal Voronoi cell, in which case the agent moves toward the goal state). For example, in Figure 10, term (t1) is the cost of reaching the landmark in row 6, column 6, which is 4. Term (t2) is the cost of getting from row 6, column 6 to the landmark at row 1 column 4 (by going from one landmark to another). In this case, the best landmark-to-landmark path is to go directly from row 6 column 6 to row 1 column 4. Hence, term (t2) is 6. Term (t3) is the cost of getting from row 1 column 4 to the goal, which is 1. The sum of these is $4 + 6 + 1 = 11$. For comparison, the optimal path has length 9.

In Kaelbling's experiments, she employed a variation of Q learning to learn terms (t1) and (t3), and she computed (t2) at regular intervals via the Floyd-Warshall all-sources shortest paths algorithm.

Figure 11 shows a MAXQ approach to solving this problem. The overall task Root, takes one argument g, which specifies the goal cell. There are three subtasks:

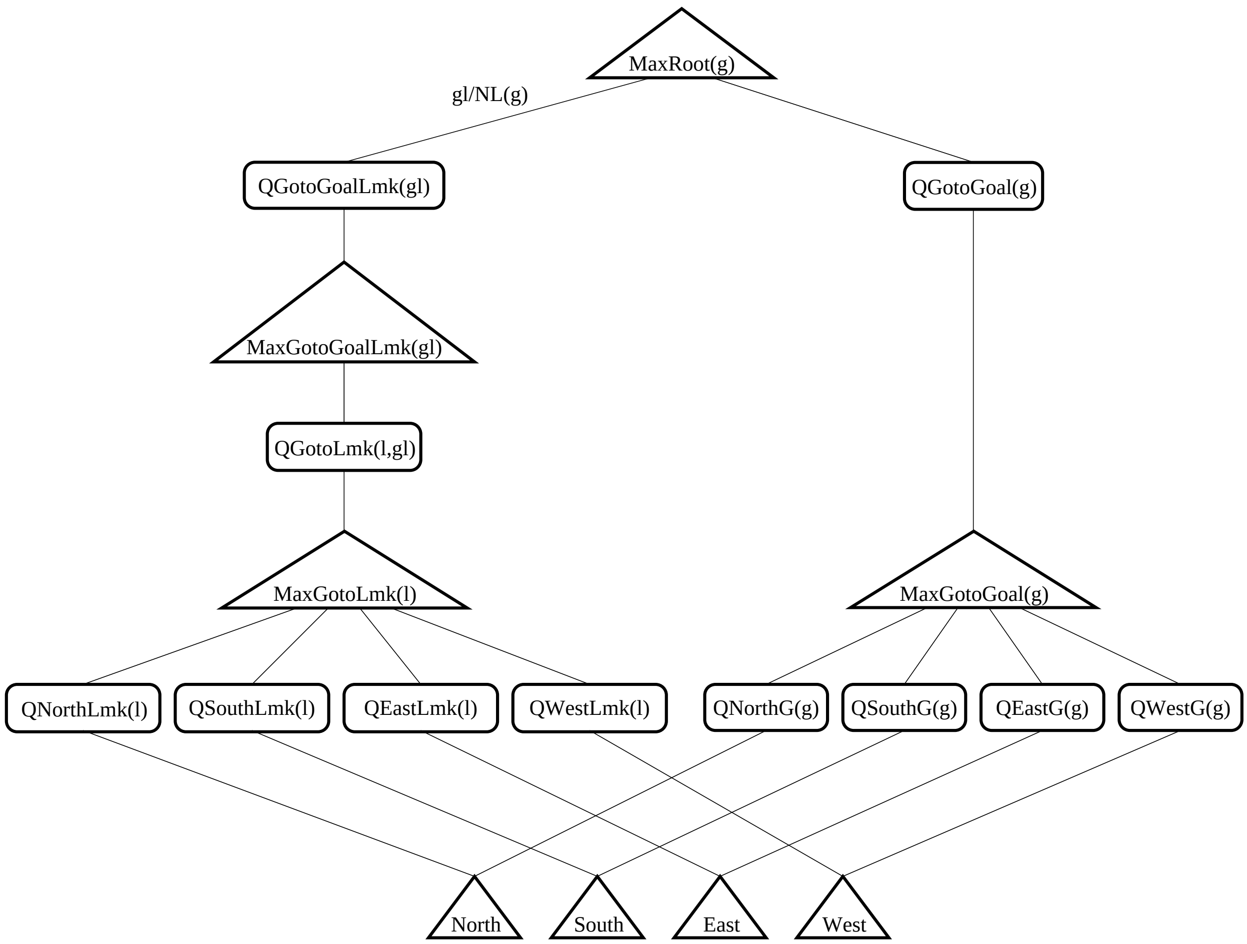

Figure 11: A MAXQ graph for the HDG navigation task.

- GotoGoalLmk, go to the landmark nearest to the goal location. The termination predicate for this subtask is true if the agent reaches the landmark nearest to the goal. The goal predicate is the same as the termination predicate.

- GotoLmk(l), go to landmark l. The termination predicate for this is true if either (a) the agent reaches landmark l or (b) the agent is outside of the region defined by the Voronoi cell for l and the neighboring Voronoi cells, $N(l)$. The goal predicate for this subtask is true only for condition (a).

- GotoGoal(g), go to the goal location g. The termination predicate for this subtask is true if either the agent is in the goal location or the agent is outside of the Voronoi cell $NL(g)$ that contains g. The goal predicate for this subtask is true if the agent is in the goal location.

The MAXQ decomposition is essentially the same as Kaelbling's method, but somewhat redundant. Consider a state where the agent is not inside the same Voronoi cell as the goal g. In such states, HDG decomposes the value function into three terms (t1), (t2), and (t3). Similarly, MAXQ also decomposes it into these same three terms:

- $V(\mathsf{GotoLmk}(l), s, a)$ the cost of getting to landmark l. This is represented as the sum of $V(a, s)$ and $C(\mathsf{GotoLmk}(l), s, a)$.
- $C(\mathsf{GotoGoalLmk}(gl), s, \mathsf{MaxGotoLmk}(l))$ the cost of getting from landmark l to the landmark gl nearest the goal.
- $C(\mathsf{Root}, s, \mathsf{GotoGoalLmk}(gl))$ the cost of getting to the goal location after reaching gl (i.e., the cost of completing the Root task after reaching gl).

When the agent is inside the goal Voronoi cell, then again HDG and MAXQ store essentially the same information. HDG stores $Q(\mathsf{GotoGoal}(g), s, a)$, while MAXQ breaks this into two terms: $C(\mathsf{GotoGoal}(g), s, a)$ and $V(a, s)$ and then sums these two quantities to compute the Q value.

Note that this MAXQ decomposition stores some information twice—specifically, the cost of getting from the goal landmark gl to the goal is stored both as $C(\mathsf{Root}, s, \mathsf{GotoGoalLmk}(gl))$ and as $C(\mathsf{GotoGoal}(g), s, a) + V(a, s)$.

Let us compare the amount of memory required by flat Q learning, HDG, and MAXQ. There are 100 locations, 4 possible actions, and 100 possible goal states, so flat Q learning must store 40,000 values.

To compute quantity (t1), HDG must store 4 Q values (for the four actions) for each state s with respect to its own landmark and the landmarks in $N(NL(s))$. This gives a total of 2,028 values that must be stored.

To compute quantity (t2), HDG must store, for each landmark, information on the shortest path to every other landmark. There are 12 landmarks. Consider the landmark at row 6, column 1. It has 5 neighboring landmarks which constitute the five macro actions that the agent can perform to move to another landmark. The nearest landmark to the goal cell could be any of the other 11 landmarks, so this gives a total of 55 Q values that must be stored. Similar computations for all 12 landmarks give a total of 506 values that must be stored.

Finally, to compute quantity (t3), HDG must store information, for each square inside each Voronoi cell, about how to get to each of the other squares inside the same Voronoi cell. This requires 3,536 values.

Hence, the grand total for HDG is 6,070, which is a huge savings over flat Q learning.

Now let's consider the MAXQ hierarchy with and without state abstractions.

- $V(a, s)$: This is the expected reward of each primitive action in each state. There are 100 states and 4 primitive actions, so this requires 400 values. However, because the reward is constant (-1), we can apply Leaf Irrelevance to store only a single value.
- $C(\mathsf{GotoLmk}(l), s, a)$, where a is one of the four primitive actions. This requires the same amount of space as (t1) in Kaelbling's representation—indeed, combined with $V(a, s)$, this represents exactly the same information as (t1). It requires 2,028 values. No state abstractions can be applied.

- $C(\mathsf{GotoGoalLmk}(gl), s, \mathsf{GotoLmk}(l))$: This is the cost of completing the GotoGoalLmk task after going to landmark l. If the primitive actions are deterministic, then $\mathsf{GotoLmk}(l)$ will always terminate at location l, and hence, we only need to store this for each pair of l and gl. This is exactly the same as Kaelbling's quantity (t2), which requires 506 values. However, if the primitive actions are stochastic—as they were in Kaelbling's original paper—then we must store this value for each possible terminal state of each GotoLmk action. Each of these actions could terminate at its target landmark l or in one of the states bordering the set of Voronoi cells that are the neighbors of the cell for l. This requires 6,600 values. When Kaelbling stores values only for (t2), she is effectively making the assumption that $\mathsf{GotoLmk}(l)$ will never fail to reach landmark l. This is an approximation which we can introduce into the MAXQ representation by our choice of state abstraction at this node.

- $C(\mathsf{GotoGoal}, s, a)$: This is the cost of completing the GotoGoal task after executing one of the primitive actions a. This is the same as quantity (t3) in the HDG representation, and it requires the same amount of space: 3,536 values.

- $C(\mathsf{Root}, s, \mathsf{GotoGoalLmk})$: This is the cost of reaching the goal once we have reached the landmark nearest the goal. MAXQ must represent this for all combinations of goal landmarks and goals. This requires 100 values. Note that these values are the same as the values of $C(\mathsf{GotoGoal}(g), s, a) + V(a, s)$ for each of the primitive actions. This means that the MAXQ representation stores this information twice, whereas the HDG representation only stores it once (as term (t3)).

- $C(\mathsf{Root}, s, \mathsf{GotoGoal})$. This is the cost of completing the Root task after we have executed the GotoGoal task. If the primitive action are deterministic, this is always zero, because GotoGoal will have reached the goal. Hence, we can apply the Termination condition and not store any values at all. However, if the primitive actions are stochastic, then we must store this value for each possible state that borders the Voronoi cell that contains the goal. This requires 96 different values. Again, in Kaelbling's HDG representation of the value function, she is ignoring the probability that GotoGoal will terminate in a non-goal state. Because MAXQ is an exact representation of the value function, it does not ignore this possibility. If we (incorrectly) apply the Termination condition in this case, the MAXQ representation becomes a function approximation.

In the stochastic case, without state abstractions, the MAXQ representation requires 12,760 values. With safe state abstractions, it requires 12,361 values. With the approximations employed by Kaelbling (or equivalently, if the primitive actions are deterministic), the MAXQ representation with state abstractions requires 6,171 values. These numbers are summarized in Table 6. We can see that, with the unsafe state abstractions, the MAXQ representation requires only slightly more space than the HDG representation (because of the redundancy in storing $C(\mathsf{Root}, s, \mathsf{GotoGoalLmk})$.

This example shows that for the HDG task, we can start with the fully-general formulation provided by MAXQ and impose assumptions to obtain a method that is similar to HDG. The MAXQ formulation guarantees that the value function of the hierarchical policy will be represented exactly. The assumptions will introduce approximations into the

Table 6: Comparison of the number of values that must be stored to represent the value function using the HDG and MAXQ methods.

HDG item	MAXQ item	HDG values	MAXQ no abs	MAXQ safe abs	MAXQ unsafe abs
	$V(a, s)$	0	400	1	1
(t1)	$C(\mathsf{GotoLmk}(l), s, a)$	2,028	2,028	2,028	2,028
(t2)	$C(\mathsf{GotoGoalLmk}, s, \mathsf{GotoLmk}(l))$	506	6,600	6,600	506
(t3)	$C(\mathsf{GotoGoal}(g), s, a)$	3,536	3,536	3,536	3,536
	$C(\mathsf{Root}, s, \mathsf{GotoGoalLmk})$	0	100	100	100
	$C(\mathsf{Root}, s, \mathsf{GotoGoal})$	0	96	96	0
Total Number of Values Required		6,070	12,760	12,361	6,171

value function representation. This might be useful as a general design methodology for building application-specific hierarchical representations. Our long-term goal is to develop such methods so that each new application does not require inventing a new set of techniques. Instead, off-the-shelf tools (e.g., based on MAXQ) could be specialized by imposing assumptions and state abstractions to produce more efficient special-purpose systems.

One of the most important contributions of the HDG method was that it introduced a form of non-hierarchical execution. As soon as the agent crosses from one Voronoi cell into another, the current subtask of reaching the landmark in that cell is "interrupted", and the agent recomputes the "current target landmark". The effect of this is that (until it reaches the goal Voronoi cell), the agent is always aiming for a landmark outside of its current Voronoi cell. Hence, although the agent "aims for" a sequence of landmark states, it typically does not visit many of these states on its way to the goal. The states just provide a convenient set of intermediate targets. By taking these "shortcuts", HDG compensates for the fact that, in general, it has overestimated the cost of getting to the goal, because its computed value function is based on a policy where the agent goes from one landmark to another.

The same effect is obtained by hierarchical greedy execution of the MAXQ graph (which was directly inspired by the HDG method). Note that by storing the NL (nearest landmark) function, Kaelbing's HDG method can detect very efficiently when the current subtask should be interrupted. This technique only works for navigation problems in a space with a distance metric. In contrast, ExecuteHGPolicy performs a kind of "polling", because it checks after each primitive action whether it should interrupt the current subroutine and invoke a new one. An important goal for future research on MAXQ is to find a general purpose mechanism for avoiding unnecessary "polling"—that is, a mechanism that can discover efficiently-evaluable interrupt conditions.

Figure 12 shows the results of our experiments with HDG using the MAXQ-Q learning algorithm. We employed the following parameters: for Flat Q learning, initial values of 0.123, a learning rate of 1.0, initial temperature of 50, and cooling rate of 0.9074; for MAXQ-Q without state abstractions: initial values of -25.123, learning rate of 1.0, initial

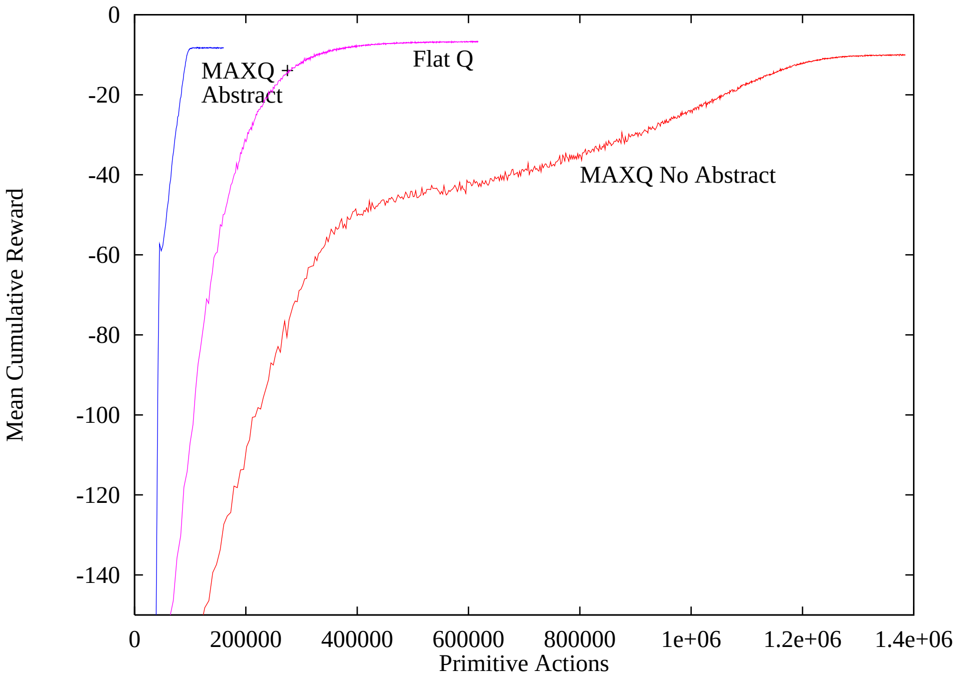

Figure 12: Comparison of Flat Q learning with MAXQ-Q learning with and without state abstraction. (Average of 100 runs.)

temperature of 50, and cooling rates of 0.9074 for MaxRoot, 0.9999 for MaxGotoGoalLmk, 0.9074 for MaxGotoGoal, and 0.9526 for MaxGotoLmk; for MAXQ-Q with state abstractions: initial values of -20.123, learning rate of 1.0, initial temperature of 50, and cooling rates of 0.9760 for MaxRoot, 0.9969 for MaxGotoGoal, 0.9984 for MaxGotoGoalLmk, and 0.9969 for MaxGotoLmk. Hierarchical greedy execution was introduced by starting with 3000 primitive actions per trial, and reducing this every trial by 2 actions, so that after 1500 trials, execution is completely greedy.

The figure confirms the observations made in our experiments with the Fickle Taxi task. Without state abstractions, MAXQ-Q converges much more slowly than flat Q learning. With state abstractions, it converges roughly three times as fast. Figure 13 shows a close-up view of Figure 12 that allows us to compare the differences in the final levels of performance of the methods. Here, we can see that MAXQ-Q with no state abstractions was not able to reach the quality of our hand-coded hierarchical policy—presumably even more exploration would be required to achieve this, whereas with state abstractions, MAXQ-Q is able to do slightly better than our hand-coded policy. With hierarchical greedy execution, MAXQ-Q is able to reach the goal using one fewer action, on the average—so that it approaches the performance of the best hierarchical greedy policy (as computed by value iteration). Notice however, that the best performance that can be obtained by hierarchical greedy execution of the best recursively-optimal policy cannot match optimal performance. Hence, Flat Q

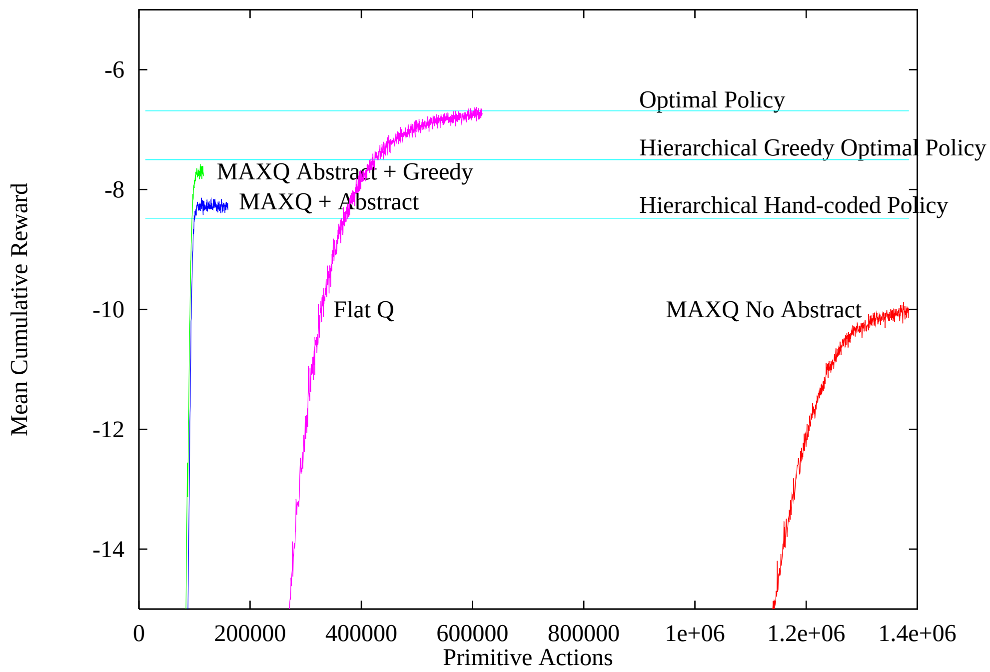

Figure 13: Expanded view comparing Flat Q learning with MAXQ-Q learning with and without state abstraction and with and without hierarchical greedy execution. (Average of 100 runs.)

learning achieves a policy that reaches the goal state, on the average, with about one fewer primitive action. Finally notice that as in the taxi domain, there was no added exploration cost for shifting to greedy execution.

Kaelbling's HDG work has recently been extended and generalized by Moore, Baird and Kaelbling (1999) to any sparse MDP where the overall task is to get from any given start state to any desired goal state. The key to the success of their approach is that each landmark subtask is guaranteed to terminate in a single resulting state. This makes it possible to identify a *sequence* of good intermediate landmark states and then assemble a policy that visits them in sequence. Moore, Baird and Kaelbling show how to construct a hierarchy of landmarks (the "airport" hierarchy) that makes this planning process efficient. Note that if each subtask did not terminate in a single state (as in general MDPs), then the airport method would not work, because there would be a combinatorial explosion of potential intermediate states that would need to be considered.

7.3 Parr and Russell: Hierarchies of Abstract Machines

In his (1998b) dissertation work, Ron Parr considered an approach to hierarchical reinforcement learning in which the programmer encodes prior knowledge in the form of a hierarchy of finite-state controllers called a HAM (Hierarchy of Abstract Machines). The hierarchy

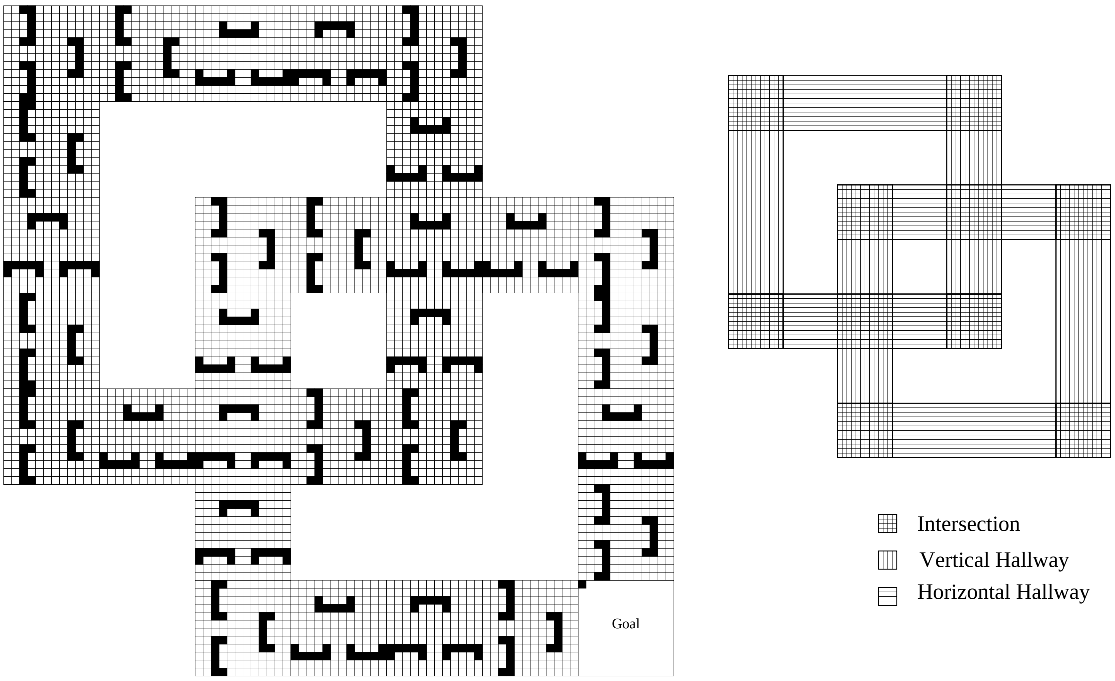

Figure 14: Parr's maze problem (on left). The start state is in the upper left corner, and all states in the lower right-hand room are terminal states. The smaller diagram on the right shows the hallway and intersection structure of the maze.

is executed using a procedure-call-and-return discipline, and it provides a *partial policy* for the task. The policy is partial because each machine can include non-deterministic "choice" machine states, in which the machine lists several options for action but does not specify which one should be chosen. The programmer puts "choice" states at any point where he/she does not know what action should be performed. Given this partial policy, Parr's goal is to find the best policy for making choices in the choice states. In other words, his goal is to learn a hierarchical value function $V(\langle s, m \rangle)$, where s is a state (of the external environment) and m contains all of the internal state of the hierarchy (i.e., the contents of the procedure call stack and the values of the current machine states for all machines appearing in the stack). A key observation is that it is only necessary to learn this value function at choice states $\langle s, m \rangle$. Parr's algorithm does not learn a decomposition of the value function. Instead, it "flattens" the hierarchy to create a new Markov decision problem over the choice states $\langle s, m \rangle$. Hence, it is hierarchical primarily in the sense that the programmer structures the prior knowledge hierarchically. An advantage of this is that Parr's method can find the optimal hierarchical policy subject to constraints provided by the programmer. A disadvantage is that the method cannot be executed "non-hierarchically" to produce a better policy.

Parr illustrated his work using the maze shown in Figure 14. This maze has a large-scale structure (as a series of hallways and intersections), and a small-scale structure (a series of obstacles that must be avoided in order to move through the hallways and intersections).

In each trial, the agent starts in the top left corner, and it must move to any state in the bottom right corner room. The agent has the usual four primitive actions, North, South, East, and West. The actions are stochastic: with probability 0.8, they succeed, but with probability 0.1 the action will move to the "left" and with probability 0.1 the action will move to the "right" instead (e.g., a North action will move east with probability 0.1 and west with probability 0.1). If an action would collide with a wall or an obstacle, it has no effect.

The maze is structured as a series of "rooms", each containing a 12-by-12 block of states (and various obstacles). Some rooms are parts of "hallways", because they are connected to two other rooms on opposite sides. Other rooms are "intersections", where two or more hallways meet.

To test the representational power of the MAXQ hierarchy, we want to see how well it can represent the prior knowledge that Parr is able to represent using the HAM. We begin by describing Parr's HAM for his maze task, and then we will present a MAXQ hierarchy that captures much of the same prior knowledge.[3]

Parr's top level machine, MRoot, consists of a loop with a single choice state that chooses among four possible child machines: MGo($East$), MGo($South$), MGo($West$), and MGo($North$). The loop terminates when the agent reaches a goal state. MRoot will only invoke a particular machine if there is a hallway in the specified direction. Hence, in the start state, it will only consider MGo($South$) and MGo($East$).

The MGo(d) machine begins executing when the agent is in an intersection. So the first thing it tries to do is to exit the intersection into a hallway in the specified direction d. Then it attempts to traverse the hallway until it reaches another intersection. It does this by first invoking an MExitIntersection(d) machine. When that machine returns, it then invokes an MExitHallway(d) machine. When that machine returns, MGo also returns.

The MExitIntersection and MExitHallway machines are identical except for their termination conditions. Both machines consist of a loop with one choice state that chooses among four possible subroutines. To simplify their description, suppose that MGo($East$) has chosen MExitIntersection($East$). Then the four possible subroutines are MSniff($East, North$), MSniff($East, South$), MBack($East, North$), and MBack($East, South$).

The MSniff(d, p) machine always moves in direction d until it encounters a wall (either part of an obstacle or part of the walls of the maze). Then it moves in perpendicular direction p until it reaches the end of the wall. A wall can "end" in two ways: either the agent is now trapped in a corner with walls in both directions d and p or else there is no longer a wall in direction d. In the first case, the MSniff machine terminates; in the second case, it resumes moving in direction d.

The MBack(d, p) machine moves one step backwards (in the direction opposite from d) and then moves five steps in direction p. These moves may or may not succeed, because the actions are stochastic and there may be walls blocking the way. But the actions are carried out in any case, and then the MBack machine returns.

The MSniff and MBack machines also terminate if they reach the end of a hall or the end of an intersection.

3. The author thanks Ron Parr for providing the details of the HAM for this task.

These finite-state controllers define a highly constrained partial policy. The MBack, MSniff, and MGo machines contain no choice states at all. The only choice points are in MRoot, which must choose the direction in which to move, and in MExitIntersection and MExitHall, which must decide when to call MSniff, when to call MBack, and which "perpendicular" direction to tell these machines to try when they cannot move forward.

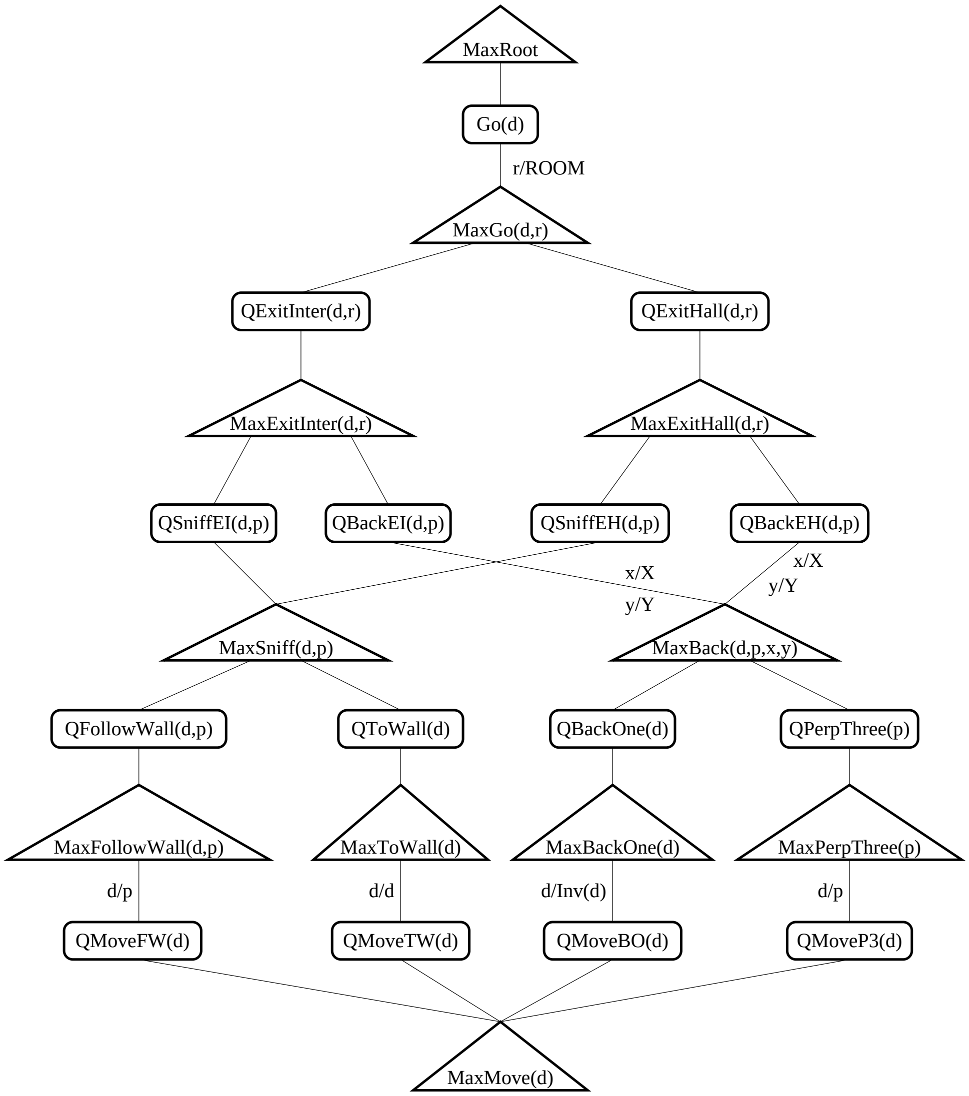

Figure 15: MAXQ graph for Parr's maze task.

Figure 15 shows a MAXQ graph that encodes a similar set of constraints on the policy. The subtasks are defined as follows:

- Root. This is exactly the same as the MRoot machine. It must choose a direction d and invoke Go. It terminates when the agent enters a terminal state. This is also its goal condition (of course).

- Go(d, r). (Go in direction d leaving room r.) The parameter r is bound to an identification number corresponding to the current 12-by-12 "room" in which the agent is located. Go terminates when the agent enters the room at the end of the hallway in direction d or when it leaves the desired hallway (e.g., in the wrong direction). The goal condition for Go is satisfied only if the agent reaches the desired intersection.

- ExitInter(d, r). This terminates when the agent has exited room r. The goal condition is that the agent exit room r in direction d.

- ExitHall(d, r). This terminates when the agent has exited the current hall (into some intersection). The goal condition is that the agent has entered the desired intersection in direction d.

- Sniff(d, r). This encodes a subtask that is equivalent to the MSniff machine. However, Sniff must have two child subtasks, ToWall and FollowWall, that were simply internal states of MSniff. This is necessary, because a subtask in the MAXQ framework cannot contain any internal state, whereas a finite-state controller in the HAM representation can contain as many internal states as necessary. In particular, it can have one state for when it is moving forward and another state for when it is following a wall sideways.

- ToWall(d). This is equivalent to one part of MSniff. It terminates when there is a wall in "front" of the agent in direction d. The goal condition is the same as the termination condition.

- FollowWall(d, p). This is equivalent to the other part of MSniff. It moves in direction p until the wall in direction d ends (or until it is stuck in a corner with walls in both directions d and p). The goal condition is the same as the termination condition.

- Back(d, p, x, y). This attempts to encode the same information as the MBack machine, but this is a case where the MAXQ hierarchy cannot capture the same information. MBack simply executes a sequence of 6 primitive actions (one step back, five steps in direction p). But to do this, MBack must have 6 internal states, which MAXQ does not allow. Instead, the Back subtask has the subgoal of moving the agent at least one square backwards and at least 3 squares in the direction p. In order to determine whether it has achieved this subgoal, it must remember the x and y position where it started to execute, so these are bound as parameters to Back. Back terminates if it achieves the desired change in position or if it runs into walls that prevent it from achieving the subgoal. The goal condition is the same as the termination condition.

- BackOne(d, x, y). This moves the agent one step backwards (in the direction opposite to d. It needs the starting x and y position in order to tell when it has succeeded. It terminates if it has moved at least one unit in direction d or if there is a wall in this direction. Its goal condition is the same as its termination condition.

- PerpThree(p, x, y). This moves the agent three steps in the direction p. It needs the starting x and y positions in order to tell when it has succeeded. It terminates when it has moved at least three units in the direction p or if there is a wall in that direction. The goal condition is the same as the termination condition.

- Move(d). This is a "parameterized primitive" action. It executes one primitive move in direction d and terminates immediately.

From this, we can see that there are three major differences between the MAXQ representation and the HAM representation. First, a HAM finite-state controller can contain internal states. To convert them into a MAXQ subtask graph, we must make a separate subtask for each internal state in the HAM. Second, a HAM can terminate based on an "amount of effort" (e.g., performing 5 actions), whereas a MAXQ subtask must terminate based on some change in the state of the world. It is impossible to define a MAXQ subtask that performs k steps and then terminate regardless of the effects of those steps (i.e., without adding some kind of "counter" to the state of the MDP). Third, it is more difficult to formulate the termination conditions for MAXQ subtasks than for HAM machines. For example, in the HAM, it was not necessary to specify that the MExitHallway machine terminates when it has entered a *different* intersection than the one where the MGo was executed. However, this is important for the MAXQ method, because in MAXQ, each subtask learns its own value function and policy—independent of its parent tasks. For example, without the requirement to enter a *different* intersection, the learning algorithms for MAXQ will always prefer to have MaxExitHall take one step backward and return to the room in which the Go action was started (because that is a much easier terminal state to reach). This problem does not arise in the HAM approach, because the policy learned for a subtask depends on the whole "flattened" hierarchy of machines, and returning to the state where the Go action was started does not help solve the overall problem of reaching the goal state in the lower right corner.

To construct the MAXQ graph for this problem, we have introduced three programming tricks: (a) binding parameters to aspects of the current state (in order to serve as a kind of "local memory" for where the subtask began executing), (b) having a parameterized primitive action (in order to be able to pass a parameter value that specifies which primitive action to perform), and (c) employing "inheritance of termination conditions"—that is, each subtask in this MAXQ graph (but not the others in this paper) inherits the termination conditions of all its ancestor tasks. Hence, if the agent is in the middle of executing a ToWall action when it leaves an intersection, the ToWall subroutine terminates because the ExitInter termination condition is satisfied. This behavior is very similar to the standard behavior of MAXQ. Ordinarily, when an ancestor task terminates, all of its descendent tasks are forced to return *without updating their C values*. With inheritance of termination conditions, on the other hand, the descendent tasks are forced to terminate, but *after updating their C values*. In other words, the termination condition of each child task is the logical disjuntion of all of the termination conditions of its ancestors (plus its own termination condition). This inheritance made it easier to write the MAXQ graph, because the parents did not need to pass down to their children all of the information necessary for the children to define the complete termination and goal predicates.

There are essentially no opportunities for state abstraction in this task, because there are no irrelevant features of the state. There are some opportunities to apply the Shielding and Termination properties, however. In particular, ExitHall(d) is guaranteed to cause its parent task, MaxGo(d), to terminate, so it does not require any stored C values. There are many states where some subtasks are terminated (e.g., Go($East$) in any state where there is a wall on the east side of the room), and so no C values need to be stored.

Nonetheless, even after applying the state elimination conditions, the MAXQ representation for this task requires much more space than a flat representation. An exact computation is difficult, but after applying MAXQ-Q learning, the MAXQ representation required 52,043 values, whereas flat Q learning requires fewer than 16,704 values. Parr states that his method requires only 4,300 values.

To test the relative effectiveness of the MAXQ representation, we compare MAXQ-Q learning with flat Q learning. Because of the very large negative values that some states acquire (particularly during the early phases of learning), we were unable to get Boltzmann exploration to work well—one very bad experience would cause an action to receive such a low Q value, that it would never be tried again. Hence, we experimented with both ϵ-greedy exploration and counter-based exploration. The ϵ-greedy exploration policy is an ordered, abstract GLIE policy in which a random action is chosen with probability ϵ, and ϵ is gradually decreased over time. The counter-based exploration policy keeps track of how many times each action a has been executed in each state s. To choose an action in state s, it selects the action that has been executed the fewest times until all actions have been executed T times. Then it switches to greedy execution. Hence, it is not a genuine GLIE policy. Parr employed counter-based exploration policies in his experiments with this task.

As in the other domains, we conducted several experimental runs (e.g., testing Boltzmann, ϵ-greedy, and counter-based exploration) to determine the best parameters for each algorithm. For Flat Q learning, we chose the following parameters: learning rate 0.50, ϵ-greedy exploration with initial value for ϵ of 1.0, ϵ decreased by 0.001 after each successful execution of a Max node, and initial Q values of -200.123. For MAXQ-Q learning, we chose the following parameters: counter-based exploration with $T = 10$, learning rate equal to the reciprocal of the number of times an action had been performed, and initial values for the C values selected carefully to provide underestimates of the true C values. For example, the initial values for QExitInter were -40.123, because in the worst case, after completing an ExitInter task, it takes about 40 steps to complete the subsequent ExitHall task and hence, complete the Go parent task. Performance was quite sensitive to these initial C values, which is a potential drawback of the MAXQ approach.

Figure 16 plots the results. We can see that MAXQ-Q learning converges about 10 times faster than Flat Q learning. We do not know whether MAXQ-Q has converged to a recursively optimal policy. For comparison, we also show the performance of a hierarchical policy that we coded by hand, but in our hand-coded policy, we used knowledge of contextual information to choose operators, so this policy is surely better than the best recursively optimal policy. HAMQ learning should converge to a policy equal to or slightly better than our hand-coded policy.

This experiment demonstrates that the MAXQ representation can capture most—but not all—of the prior knowledge that can be represented by the HAMQ hierarchy. It also

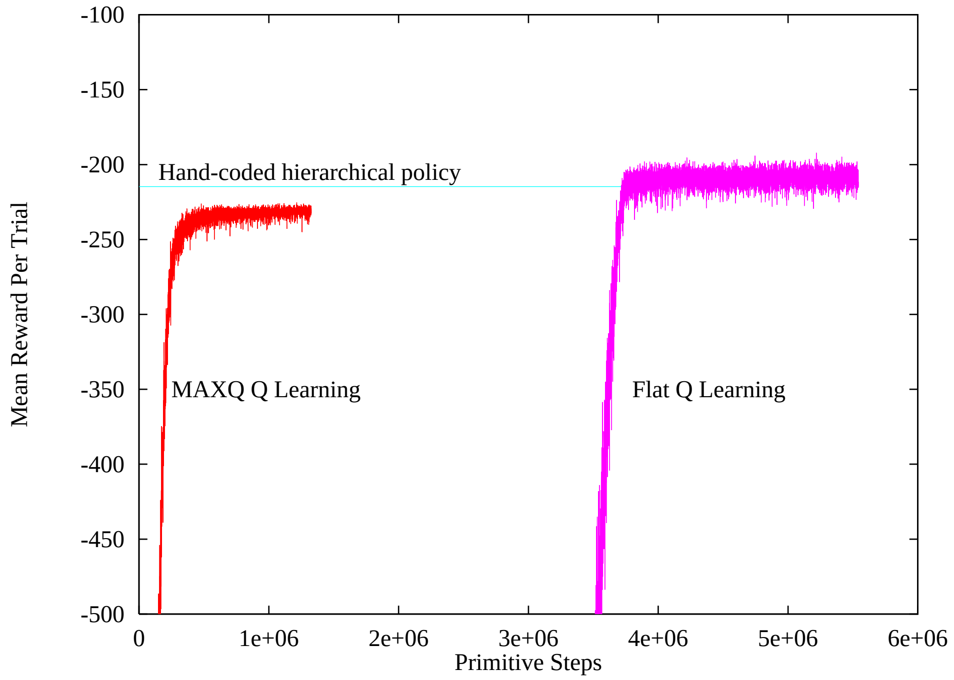

Figure 16: Comparison of Flat Q learning and MAXQ-Q learning in the Parr maze task.

shows that the MAXQ representation requires much more care in the design of the goal conditions for the subtasks.

7.4 Other Domains

In addition to the three domains discussed above, we have developed MAXQ graphs for Singh's (1992) "flag task", the treasure hunter task described by Tadepalli and Dietterich (1997), and Dayan and Hinton's (1993) Feudal-Q learning task. All of these tasks can be easily and naturally placed into the MAXQ framework—indeed, all of them fit more easily than the Parr and Russell maze task.

MAXQ is able to exactly duplicate Singh's work and his decomposition of the value function—while using exactly the same amount of space to represent the value function. MAXQ can also duplicate the results from Tadepalli and Dietterich—however, because MAXQ is not an explanation-based method, it is considerably slower and requires substantially more space to represent the value function.

In the Feudal-Q task, MAXQ is able to give better performance than Feudal-Q learning. The reason is that in Feudal-Q learning, each subroutine makes decisions using only a Q function learned at its own level of the hierarchy—that is, without information about the estimated costs of the actions of its descendents. In contrast, the MAXQ value function decomposition permits each Max node to make decisions based on the sum of its completion function, $C(i,s,j)$, and the costs estimated by its descendents, $V(j,s)$. Of course, MAXQ also supports non-hierarchical execution, which is not possible for Feudal-Q, because it does not learn a value function decomposition.

8. Discussion

Before concluding this paper, we wish to discuss two issues: (a) design tradeoffs in hierarchical reinforcement learning and (b) methods for automatically learning (or at least improving) MAXQ hierarchies.

8.1 Design Tradeoffs in Hierarchical Reinforcement Learning

In the introduction to this paper, we discussed four issues concerning the design of hierarchical reinforcement learning architectures: (a) the method for defining subtasks, (b) the use of state abstraction, (c) non-hierarchical execution, and (d) the design of learning algorithms. In this subsection, we want to highlight a tradeoff between the first two of these issues.

MAXQ defines subtasks using a termination predicate T_i and a pseudo-reward function $\tilde{R}$. There are at least two drawbacks of this method. First, it can be hard for the programmer to define T_i and $\tilde{R}$ correctly, since this essentially requires guessing the value function of the optimal policy for the MDP at all states where the subtask terminates. Second, it leads us to seek a recursively optimal policy rather than a hierarchically optimal policy. Recursively optimal policies may be much worse than hierarchically optimal ones, so we may be giving up substantial performance.

However, in return for these two drawbacks, MAXQ obtains a very important benefit: the policies and value functions for subtasks become *context-free*. In other words, they do not depend on their parent tasks or the larger context in which they are invoked. To understand this point, consider again the MDP shown in Figure 6. It is clear that the *optimal* policy for exiting the left-hand room (the Exit subtask) depends on the location of the goal. If it is at the top of the right-hand room, then the agent should prefer to exit via the upper door, whereas if it is at the bottom of the right-hand room, the agent should prefer to exit by the lower door. However, if we define the subtask of exiting the left-hand room using a pseudo-reward of zero for both doors, then we obtain a policy that is not optimal in either case, but a policy that we can re-use in both cases. Furthermore, this policy *does not depend on the location of the goal.* Hence, we can apply Max node irrelevance to solve the Exit subtask using only the location of the robot and ignore the location of the goal.

This example shows that we obtain the benefits of subtask reuse and state abstraction because we define the subtask using a termination predicate and a pseudo-reward function. The termination predicate and pseudo-reward function provide a barrier that prevents "communication" of value information between the Exit subtask and its context.

Compare this to Parr's HAM method. The HAMQ algorithm finds the best policy consistent with the hierarchy. To achieve this, it must permit information to propagate "into" the Exit subtask (i.e., the Exit finite-state controller) from its environment. But this means that if any state that is reached after leaving the Exit subtask has different values depending on the location of the goal, then these different values will propagate back into the Exit subtask. To represent these different values, the Exit subtask must know the location of the goal. In short, to achieve a hierarchically optimal policy within the Exit subtask, we must (in general) represent its value function using the *entire* state space. State abstractions cannot be employed without losing hierarchical optimality.

We can see, therefore, that there is a direct tradeoff between achieving hierarchical optimality and employing state abstractions. Methods for hierarchical optimality have more freedom in defining subtasks (e.g., using partial policies, as in the HAM approach). But they cannot (safely) employ state abstractions within subtasks, and in general, they cannot reuse the solution of one subtask in multiple contexts. Methods for recursive optimality, on the other hand, must define subtasks using some method (such as pseudo-reward functions for MAXQ or fixed policies for the options framework) that isolates the subtask from its context. But in return, they can apply state abstraction and the learned policy can be reused in many contexts (where it will be more or less optimal).

It is interesting that the iterative method described by Dean and Lin (1995) can be viewed as a method for moving along this tradeoff. In the Dean and Lin method, the programmer makes an initial guess for the values of the terminal states of each subtask (i.e., the doorways in Figure 6). Based on this initial guess, the locally optimal policies for the subtasks are computed. Then the locally optimal policy for the parent task is computed—while holding the subtask policies fixed (i.e., treating them as options). At this point, their algorithm has computed the recursively optimal solution to the original problem, given the initial guesses. Instead of solving the various subproblems sequentially via an offline algorithm as Dean and Lin suggested, we could use the MAXQ-Q learning algorithm.

But the method of Dean and Lin does not stop here. Instead, it computes new values of the terminal states of each subtask based on the learned value function for the entire problem. This allows it to update its "guesses" for the values of the terminal states. The entire solution process can now be repeated to obtain a new recursively optimal solution, based on the new guesses. They prove that if this process is iterated indefinitely, it will converge to the hierarchically optimal policy (provided, of course, that no state abstractions are used within the subtasks).

This suggests an extension to MAXQ-Q learning that adapts the $\tilde{R}$ values online. Each time a subtask terminates, we could update the $\tilde{R}$ function based on the computed value of the terminated state. To be precise, if j is a subtask of i, then when j terminates in state s', we should update $\tilde{R}_j(s')$ to be equal to $\tilde{V}(i,s') = \max_{a'} \tilde{Q}(i,s',a')$. However, this will only work if $\tilde{R}_j(s')$ is represented using the full state s'. If subtask j is employing state abstractions, $x = \chi(s)$, then $\tilde{R}_j(x')$ will need to be the average value of $\tilde{V}(i,s')$, where the average is taken over all states s' such that $x' = \chi(s')$ (weighted by the probability of visiting those states). This is easily accomplished by performing a stochastic approximation update of the form

$$\tilde{R}_j(x') = (1-\alpha_t)\tilde{R}_j(x') + \alpha_t \tilde{V}(i,s')$$

each time subtask j terminates. Such an algorithm could be expected to converge to the best hierarchical policy consistent with the given state abstractions.

This also suggests that in some problems, it may be worthwhile to first learn a recursively optimal policy using very aggressive state abstractions and then use the learned value function to initialize a MAXQ representation with a more detailed representation of the states. These progressive refinements of the state space could be guided by monitoring the degree to which the values of $\tilde{V}(i, x')$ vary for each abstract state x'. If they have a large variance, this means that the state abstractions are failing to make important distinctions in the values of the states, and they should be refined.

Both of these kinds of adaptive algorithms will take longer to converge than the basic MAXQ method described in this paper. But for tasks that an agent must solve many times in its lifetime, it is worthwhile to have learning algorithms that provide an initial useful solution but then gradually improve that solution until it is optimal. An important goal for future research is to find methods for diagnosing and repairing errors (or sub-optimalities) in the initial hierarchy so that ultimately the optimal policy will be discovered.

8.2 Automated Discovery of Abstractions

The approach taken in this paper has been to rely upon the programmer to design the MAXQ hierarchy including the termination conditions, pseudo-reward functions, and state abstractions. But the results of this paper, particularly concerning state abstraction, suggest ways in which we might be able to automate the construction of the hierarchy.

The main purpose of the hierarchy is to create opportunities for subtask sharing and state abstraction. These are actually very closely related. In order for a subtask to be shared in two different regions of the state space, it must be the case that the value function in those two different regions is identical except for an additive offset. In the MAXQ framework, that additive offset would be the difference in the C values of the parent task. So one way to find reusable subtasks would be to look for regions of state space where the value function exhibits these additive offsets.

A second way would be to search for structure in the one-step probability transition function $P(s'|s, a)$. A subtask will be useful if it enables state abstractions such as Max Node Irrelevance. We can formulate this as the problem of identifying some region of state space such that, conditioned on being in that region, $P(s'|s, a)$ factors according to Equation 17. A top-down divide-and-conquer algorithm similar to decision-tree algorithms might be able to do this.

A third way would be to search for funnel actions by looking for bottlenecks in the state space through which all policies must travel. This would be useful for discovering cases of Result Distribution Irrelevance.

In some ways, the most difficult kinds of state abstractions to discover are those in which arbitrary subgoals are introduced to constrain the policy (and sacrifice optimality). For example, how could an algorithm automatically decide to impose landmarks onto the HDG task? Perhaps by detecting a large region of state space without bottlenecks or variations in the reward function?

The problem of discovering hierarchies is an important challenge for the future, but at least this paper has provided some guidelines for what constitute good state abstractions, and these can serve as objective functions for guiding the automated search for abstractions.

9. Concluding Remarks

This paper has introduced a new representation for the value function in hierarchical reinforcement learning—the MAXQ value function decomposition. We have proved that the MAXQ decomposition can represent the value function of any hierarchical policy under both the finite-horizon undiscounted, cumulative reward criterion and the infinite-horizon discounted reward criterion. This representation supports subtask sharing and re-use, because the overall value function is decomposed into value functions for individual subtasks.

The paper introduced a learning algorithm, MAXQ-Q learning, and proved that it converges with probability 1 to a recursively optimal policy. The paper argued that although recursive optimality is weaker than either hierarchical optimality or global optimality, it is an important form of optimality because it permits each subtask to learn a locally optimal policy while ignoring the behavior of its ancestors in the MAXQ graph. This increases the opportunities for subtask sharing and state abstraction.

We have shown that the MAXQ decomposition creates opportunities for state abstraction, and we identified a set of five properties (Max Node Irrelevance, Leaf Irrelevance, Result Distribution Irrelevance, Shielding, and Termination) that allow us to ignore large parts of the state space within subtasks. We proved that MAXQ-Q still converges in the presence of these forms of state abstraction, and we showed experimentally that state abstraction is important in practice for the successful application of MAXQ-Q learning—at least in the Taxi and HDG tasks.

The paper presented two different methods for deriving improved non-hierarchical policies from the MAXQ value function representation, and it has formalized the conditions under which these methods can improve over the hierarchical policy. The paper verified experimentally that non-hierarchical execution gives improved performance in the Fickle Taxi Task (where it achieves optimal performance) and in the HDG task (where it gives a substantial improvement).

Finally, the paper has argued that there is a tradeoff governing the design of hierarchical reinforcement learning methods. At one end of the design spectrum are "context free" methods such as MAXQ-Q learning. They provide good support for state abstraction and subtask sharing but they can only learn recursively optimal policies. At the other end of the spectrum are "context-sensitive" methods such as HAMQ, the options framework, and the early work of Dean and Lin. These methods can discover hierarchically optimal policies (or, in some cases, globally optimal policies), but their drawback is that they cannot easily exploit state abstractions or share subtasks. Because of the great speedups that are enabled by state abstraction, this paper has argued that the context-free approach is to be preferred—and that it can be relaxed as needed to obtain improved policies.

Acknowledgements

The author gratefully acknowledges the support of the National Science Foundation under grant number IRI-9626584, the Office of Naval Research under grant number N00014-95-1-0557, the Air Force Office of Scientific Research under grant number F49620-98-1-0375, and the Spanish government under their program of Estancias de Investigadores Extranjeros en Regimen de Año Sabatico en España. In addition, the author is indebted to many colleagues for helping develop and clarify the ideas in this paper including Valentina Zubek, Leslie Kaelbling, Bill Langford, Wes Pinchot, Rich Sutton, Prasad Tadepalli, and Sebastian Thrun. I particularly want to thank Eric Chown for encouraging me to study Feudal reinforcement learning, Ron Parr for providing the details of his HAM machines, and Sebastian Thrun for encouraging me to write a single comprehensive paper. I also thank Andrew Moore (the action editor), Valentina Zubek, and the two sets of anonymous reviewers of previous drafts of this paper for their suggestions and careful reading, which have improved the paper immeasurably.

References

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

Bertsekas, D. P., & Tsitsiklis, J. N. (1996). *Neuro-Dynamic Programming.* Athena Scientific, Belmont, MA.

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

Currie, K., & Tate, A. (1991). O-plan: The open planning architecture. *Artificial Intelligence*, *52*(1), 49–86.

Dayan, P., & Hinton, G. (1993). Feudal reinforcement learning. In *Advances in Neural Information Processing Systems, 5*, pp. 271–278. Morgan Kaufmann, San Francisco, CA.

Dean, T., & Lin, S.-H. (1995). Decomposition techniques for planning in stochastic domains. Tech. rep. CS-95-10, Department of Computer Science, Brown University, Providence, Rhode Island.

Dietterich, T. G. (1998). The MAXQ method for hierarchical reinforcement learning. In *Fifteenth International Conference on Machine Learning*, pp. 118–126. Morgan Kaufmann.

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

Forgy, C. L. (1982). Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, *19*(1), 17–37.

Hauskrecht, M., Meuleau, N., Kaelbling, L. P., Dean, T., & Boutilier, C. (1998). Hierarchical solution of Markov decision processes using macro-actions. In *Proceedings of the Fourteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI–98)*, pp. 220–229 San Francisco, CA. Morgan Kaufmann Publishers.

Howard, R. A. (1960). *Dynamic Programming and Markov Processes.* MIT Press, Cambridge, MA.

Jaakkola, T., Jordan, M. I., & Singh, S. P. (1994). On the convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, *6*(6), 1185–1201.

Kaelbling, L. P. (1993). Hierarchical reinforcement learning: Preliminary results. In *Proceedings of the Tenth International Conference on Machine Learning*, pp. 167–173 San Francisco, CA. Morgan Kaufmann.

Kalmár, Z., Szepesvári, C., & Lörincz, A. (1998). Module based reinforcement learning for a real robot. *Machine Learning*, *31*, 55–85.

Knoblock, C. A. (1990). Learning abstraction hierarchies for problem solving. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pp. 923–928 Boston, MA. AAAI Press.

Korf, R. E. (1985). Macro-operators: A weak method for learning. *Artificial Intelligence*, *26*(1), 35–77.

Lin, L.-J. (1993). *Reinforcement learning for robots using neural networks.* Ph.D. thesis, Carnegie Mellon University, Department of Computer Science, Pittsburgh, PA.

Moore, A. W., Baird, L., & Kaelbling, L. P. (1999). Multi-value-functions: Efficient automatic action hierarchies for multiple goal MDPs. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 1316–1323 San Francisco. Morgan Kaufmann.

Parr, R. (1998a). Flexible decomposition algorithms for weakly coupled Markov decision problems. In *Proceedings of the Fourteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI–98)*, pp. 422–430 San Francisco, CA. Morgan Kaufmann Publishers.

Parr, R. (1998b). *Hierarchical control and learning for Markov decision processes.* Ph.D. thesis, University of California, Berkeley, California.

Parr, R., & Russell, S. (1998). Reinforcement learning with hierarchies of machines. In *Advances in Neural Information Processing Systems*, Vol. 10, pp. 1043–1049 Cambridge, MA. MIT Press.

Pearl, J. (1988). *Probabilistic Inference in Intelligent Systems. Networks of Plausible Inference.* Morgan Kaufmann, San Mateo, CA.

Rummery, G. A., & Niranjan, M. (1994). Online Q-learning using connectionist systems. Tech. rep. CUED/FINFENG/TR 166, Cambridge University Engineering Department, Cambridge, England.

Sacerdoti, E. D. (1974). Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, *5*(2), 115–135.

Singh, S., Jaakkola, T., Littman, M. L., & Szepesvári, C. (1998). Convergence results for single-step on-policy reinforcement-learning algorithms. Tech. rep., University of Colorado, Department of Computer Science, Boulder, CO. To appear in *Machine Learning.*

Singh, S. P. (1992). Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning*, *8*, 323–339.

Sutton, R. S., Singh, S., Precup, D., & Ravindran, B. (1999). Improved switching among temporally abstract actions. In *Advances in Neural Information Processing Systems*, Vol. 11, pp. 1066–1072. MIT Press.

Sutton, R., & Barto, A. G. (1998). *Introduction to Reinforcement Learning.* MIT Press, Cambridge, MA.

Sutton, R. S., Precup, D., & Singh, S. (1998). Between MDPs and Semi-MDPs: Learning, planning, and representing knowledge at multiple temporal scales. Tech. rep., University of Massachusetts, Department of Computer and Information Sciences, Amherst, MA. To appear in *Artificial Intelligence.*

Tadepalli, P., & Dietterich, T. G. (1997). Hierarchical explanation-based reinforcement learning. In *Proceedings of the Fourteenth International Conference on Machine Learning*, pp. 358–366 San Francisco, CA. Morgan Kaufmann.

Tambe, M., & Rosenbloom, P. S. (1994). Investigating production system representations for non-combinatorial match. *Artificial Intelligence*, *68*(1), 155–199.

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards.* Ph.D. thesis, King's College, Oxford. (To be reprinted by MIT Press.).

Watkins, C. J., & Dayan, P. (1992). Technical note Q-Learning. *Machine Learning*, *8*, 279.