The MAXQ Method for Hierarchical Reinforcement Learning

Thomas G. Dietterich
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331
tgd@cs.orst.edu

Abstract

This paper presents a new approach to hierarchical reinforcement learning based on the MAXQ decomposition of the value function. 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. Conditions under which the MAXQ decomposition can represent the optimal value function are derived. The paper defines a hierarchical Q learning algorithm, proves its convergence, and shows experimentally that it can learn much faster than ordinary "flat" Q learning. Finally, the paper discusses some interesting issues that arise in hierarchical reinforcement learning including the hierarchical credit assignment problem and non-hierarchical execution of the MAXQ hierarchy.

1 Introduction

Hierarchical approaches to reinforcement learning (RL) problems promise many benefits: (a) improved exploration (because exploration can take "big steps" at high levels of abstraction), (b) learning from fewer trials (because fewer parameters must be learned and because subtasks can ignore irrelevant features of the full state) and (c) faster learning for new problems (because subtasks learned on previous problems can be re-used).

Recent research has explored three general approaches to reaching these goals. The first approach, introduced by Dean and Lin (1995), exploits a hierarchical decomposition primarily as a computational device to accelerate the computation of the optimal policy. The second approach, introduced by Parr and Russell (1998) relies on a programmer to design a hierarchy of abstract machines that constrains the possible policies to be considered. Their method computes the policy that is optimal subject to these hierarchical constraints by effectively flattening the hierarchy. We will call this kind of policy *hierarchically optimal*, because it is the best policy consistent with the imposed hierarchy. The third approach, pioneered by Singh (1992), Kaelbling (1993), and Dayan and Hinton (1993), also relies on a programmer-designed hierarchy. In this hierarchy, each subtask is defined in terms of goal states or termination conditions. Each subtask in the hierarchy corresponds to its own Markov Decision Problem (MDP), and the methods seek to compute a policy that is locally optimal for each subtask. We will call such policies *recursively optimal*. Recent work by Precup, Sutton, and Singh (1998) studies aspects of both the first and third approaches.

In this paper, we extend the research on recursively optimal policies by introducing the MAXQ method for hierarchical reinforcement learning. The methods introduced by Singh, Kaelbling, and Dayan and Hinton are all specific to particular tasks. The Feudal Q learning method of Dayan and Hinton suffers from the problem that at all non-primitive levels of a Feudal-Q hierarchy, the learning task can become non-Markovian, and therefore difficult to solve. In contrast, the MAXQ method is general purpose. At each level of the hierarchy, the task is Markovian and can be solved by standard RL methods. In many cases, state abstractions can be introduced without destroying the optimality of the learned policy. Like Kaelbling's work, MAXQ supports non-hierarchical execution of the learned policy, which permits it to behave well even when the optimal policy violates the structure of the hierarchy.

This paper is organized as follows. First, we introduce the MAXQ hierarchy using an example and define its procedural and declarative semantics. Then we introduce two theo-

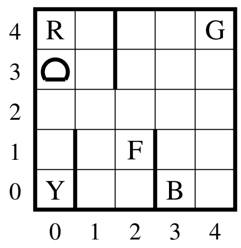

Figure 1: The Taxi Domain

rems that describe the conditions under which the MAXQ hierarchy can successfully represent the value function of a fixed hierarchical policy. Section 4 introduces a learning algorithm for training a MAXQ hierarchy and shows experimentally and theoretically that it works well. Finally, the paper shows how a non-hierarchical policy can be computed and executed using the MAXQ hierarchy.

2 The MAXQ Hierarchy

We will introduce the MAXQ method using the simple Taxi Problem shown in Figure 1. A taxi inhabits a 5-by-5 grid world. 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 state and with a randomly-chosen amount of fuel (ranging from 5 to 12 units). 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 seven primitive actions in this domain: (a) four navigation actions that move the taxi one square North, South, East, or West (each of these consumes one unit of fuel), (b) a Pickup action, (c) a Putdown action, and (d) a Fillup action (which can only be executed when the taxi is at location F(uel)). Each action is deterministic. 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. Finally, the episode also ends (with a reward of -20) if the fuel level falls below zero.

We seek a policy that maximizes the average reward per step. In this domain, this is equivalent to maximizing the total reward per episode. The optimal policy—which is non-trivial to implement by hand—attains an average reward per step of 0.92 (computed over 5000 trials). There are 8,750 possible states: 25 squares, 5 locations for the passenger (counting the four starting locations and the taxi), 5 destinations, and 14 fuel levels.

This task has a simple hierarchical structure in which there are three sub-tasks: Get the passenger, Refuel the taxi, and Deliver the passenger. Each subtask involves navigating to one of the five locations and then performing a Pickup, Fillup, or Putdown action. While the taxi is navigating to a location, only that location is relevant. We would like to capture this hierarchical structure and take advantage of it during learning and performance.

Figure 2 shows a MAXQ graph for this problem. This graph contains two kinds of nodes: Max nodes (indicated by triangles) and Q nodes (indicated by ovals). Max nodes with no children denote primitive actions in the domain; Max nodes with children represent subtasks. In this simple problem, there are five such subtasks: (a) Navigate(t) (move the taxi to target location t), (b) Get (move to the passenger's location and pick up the passenger), (c) Put (move to the passenger's destination and put down the passenger), (d) Refuel (move to F and Fillup), and (e) Root (perform the overall task of picking up and delivering the passenger). Notice that the Navigate task is shared by the Get, Put, and Refuel tasks.

The immediate children of each Max node are Q nodes. Each Q node represents an action that can be performed to achieve its parent's subtask. For example, the MaxGet node has a child QNavigateForGet which represents the action of navigating from the current state to the passenger's location. The distinction between Max nodes and Q nodes is critical to ensuring that subtasks can be shared and reused. Each Max node will learn the *context independent* expected cumulative reward of performing its subtask. For example, MaxNavigate(t) will estimate the expected cumulative reward of navigating from any state to one of the five target locations t. Each Q node will learn the *context dependent* expected cumulative reward of performing its subtask. For example, QNavigateForGet(t) will learn the expected cumulative reward of navigating to location t *and then completing the* Get *task*. On the other hand, QNavigateForPut(t) will learn the expected cumulative reward of navigating to location t *and then completing the* Put *task*. Both of these Q nodes will "ask" MaxNavigate(t) how much it will cost to get to location t, and they will use this to help them compute their Q values. The value function computed by MaxNavigate is context independent and

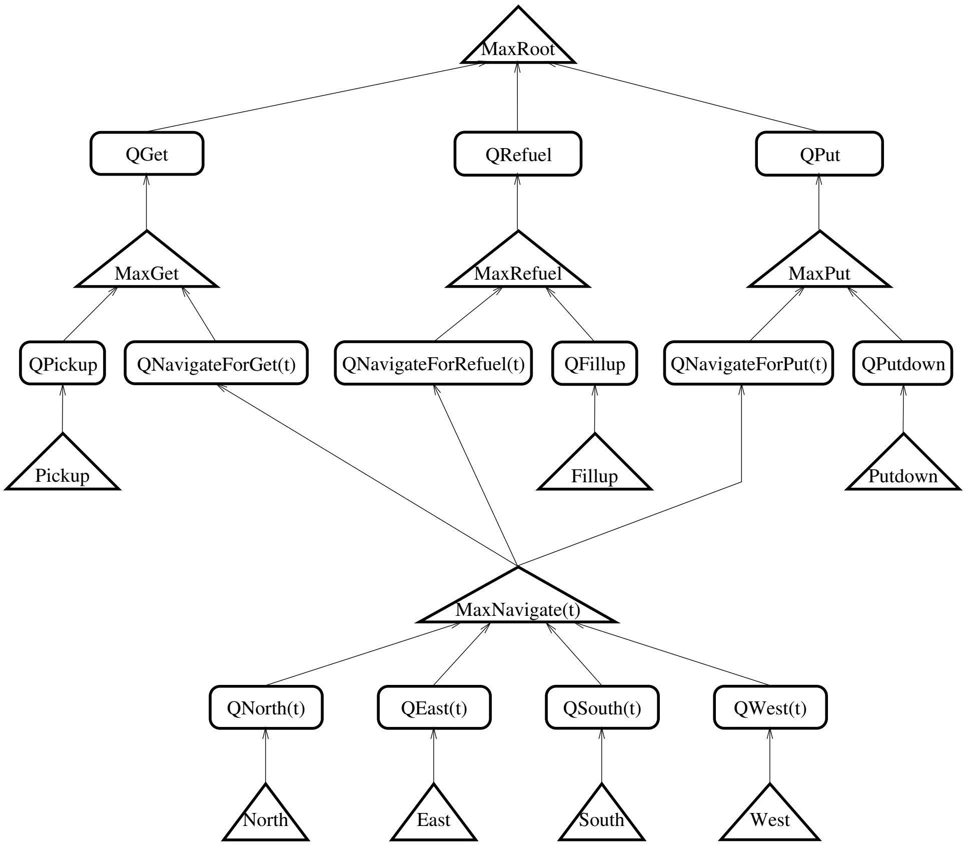

Figure 2: A MAXQ graph for the Taxi Domain

can be shared by all three of its parent Q nodes.

In the rest of the paper, we will say that Max node a is the child of Max node i if there is a Q node whose parent is i and whose child is a.

To define the semantics of the MAXQ graph more formally, let us suppose that the overall task is to solve a Markov Decision Problem (MDP) M defined over a set of states S and actions A with reward function $R(s'|s,a)$ (the reward received upon entering state s' after performing action a in state s) and transition probability function $P(s'|s,a)$ (the probability of entering state s' as a result of performing a in s). In this paper, we will assume that the MDP M defines an undiscounted stochastic shortest path problem. All of the results can be extended to the infinite-horizon discounted case.

Each Max node i corresponds to a separate subtask M_i. The children of Max node i are the actions of M_i. Each subtask M_i divides the set S of all states into two disjoint subsets: S_i and T_i. The set T_i is the set of *terminal states* for M_i. Subtask M_i will terminate whenever the environment enters one of the states in T_i. A subset $G_i \subseteq T_i$ of the terminal states are the *goal states* of M_i. Below, we will discuss the details of defining a reward function that will encourage M_i to terminate in one of these goal states. Let us define π_i to be some (arbitrary) policy for subtask i. This policy "attempts" to get from any state in S_i to one of the goal states in G_i.

A *hierarchical policy* for a MAXQ graph is a set of policies $\pi = \{\pi_0, \ldots, \pi_n\}$, one for each Max node, that indicate how each Max node should choose its actions. The hierarchical policy is executed the same way that subroutines are executed in ordinary programming languages. The Root policy chooses one of its child actions to perform, say, Get. The Get policy then chooses one of its child actions, say, Pickup. Then the Pickup action is executed, since it is a primitive. A Max node's policy is executed until that Max node enters a terminating state, at which point, "control" returns to its parent Max node.

Therefore, we can view the MAXQ graph as a subroutine call graph. Like subroutines, Max nodes can be parameterized. In this graph, MaxNavigate takes one parameter, t, which specifies which of the five locations (R, B, G, Y, F) is the target of the MaxNavigate. One way in which the graph is different from an ordinary program is that the children of each Max node are *unordered*. They can be called in any order, and a Max node can execute each of its children multiple times before it completes its subtask. The MAXQ graph is therefore a kind of incompletely-specified non-deterministic program. One result of learning will be to determine a policy for each Max node that tells how and when to invoke its children. This will make the MAXQ graph a completely-specified deterministic program (interacting with a non-deterministic environment).

Thus far, our formulation of the MAXQ method is essentially the same as the Feudal Q learning method of Dayan and Hinton (1993). However, an important improvement over Feudal Q learning is the ability to interpret the MAXQ graph as a representation of the value function for a hierarchical policy. Consider Max node i, and define $V_i^\pi(s)$ to be the expected cumulative reward for following the hierarchical policy π starting in state s until we enter some state in T_i. For a fixed hierarchical policy π, subtask M_i has a well-defined transition probability function $P_i^\pi(s'|s,a)$, which is the probability that the environment will move from state s to state s' when M_i executes action a. This probability is well defined, because the child M_a is executing a fixed policy π_a (as are all of its descendants). Hence, node i can treat action a as an atomic action. The immediate reward for node i of executing a will be the expected reward for node a of moving from the current state s to a terminal state in T_a according to policy π_a. This is denoted $V_a^\pi(s)$. Hence, we can write

$$V_i^\pi(s) = V_a^\pi(s) + \sum_{s'} P_i(s'|s,a) V_i^\pi(s'), \tag{1}$$

where $a = \pi_i(s)$. This gives us a recursive decomposition of the value function so that the value function of the root node is the value function of the entire MDP M and each subtask M_i is a separate MDP.

This recursive expression becomes more useful when we switch to the action-value (or "Q") representation of the value function. Define $Q_i^\pi(s,a)$ to be the expected cumulative reward for MDP M_i of performing action a in state s and then following the hierarchical policy π thereafter. Define the second term on the right-hand side of Eq. (1) to be $C_i^\pi(s,a)$, which we will call the *completion function*. This is the expected cumulative reward of *completing* MDP M_i following policy π *after* executing action a in state s. With these definitions, we can rewrite Eq. (1) as

$$Q_i^\pi(s,a) = V_a^\pi(s) + C_i^\pi(s,a) \tag{2}$$

where

$$V_i^\pi(s) = \begin{cases} Q_i^\pi(s,\pi_i(s)) & i \text{ composite} \\ \sum_{s'} P(s'|s,i) R(s'|s,i) & i \text{ primitive} \end{cases} \tag{3}$$

$$C_i^\pi(s,a) = \sum_{s'} P(s'|s,a) V_i^\pi(s') \tag{4}$$

These completely define the value-function semantics of the MAXQ hierarchy. Each Q node with parent i and child a stores the information $C_i^\pi(s,a)$ for each state s in S_i. Each Max node i returns the Q value of the child chosen by π_i.

To compute the value of a hierarchical policy π in state s, we begin at MaxRoot (node 0) and compute $Q_0^\pi(s,\pi_0(s))$. This requires that we ask our child node $a_1 = \pi_0(s)$ for its value $V_{a_1}^\pi(s)$. Our child recursively asks its child $a_2 = \pi_{a_1}(s)$ for its value, and so on until a leaf node a_n is reached. Let $(a_1, a_2, \ldots, a_n)$ be the path that was traversed through the MAXQ graph. Now leaf node a_n returns $V_{a_n}^\pi(s)$, to which its parent adds $C_{a_{n-1}}^\pi(s,a_n)$ and so on recursively. The value returned by MaxRoot is

$$V_0^\pi(s) = V_{a_n}^\pi(s) + C_{a_{n-1}}^\pi(s,a_n) + \ldots + C_{a_1}^\pi(s,a_2) + C_0^\pi(s,a_1) \tag{5}$$

Figure 3 shows how the sequence of rewards $r_1, r_2, \ldots$ received from the primitive actions is decomposed hierarchically into the sum of the C terms.

3 Representation Theorems

Under what conditions can this hierarchy represent the value function of a fixed, hierarchical policy? We will say that a MAXQ graph is a *full-state* graph if separate $C_i^\pi(s,a)$ values are stored for each state $s \in S_i$. In most applications, including Figure 1, it will be desirable to introduce an abstraction function $X_i(s)$ that will provide a set of features that abstract essential information from the state. Each Q node will then store the function $C_i^\pi(X_i(s),a)$, with one value for each distinct abstract state $X_i(s)$.

For full-state graphs, it is easy to prove the following theorem by expanding Equations (2–4):

Theorem 1 *Let $\pi = \{\pi_i; i = 0, \ldots, n\}$ be a hierarchical policy defined over a full-state MAXQ graph, and let $i = 0$ be the root node of the graph. Then there exist values for C_i (for internal Max nodes) and V_i (for primitive, leaf Max nodes) such that $V_0(s)$ is the expected cumulative reward of following policy π in state s.*

A more important and difficult question is to understand the conditions under which an abstract-state MAXQ graph can exactly represent the value function of a hierarchical policy. The following theorem establishes one condition:

Theorem 2 *For all Max nodes i and actions a, let $Result_i^\pi(s,a) = \{s' \mid P_i^\pi(s'|s,a) > 0\}$ be the set of states that can result from applying abstract action a in state s at node i while following hierarchical policy π. If the following*

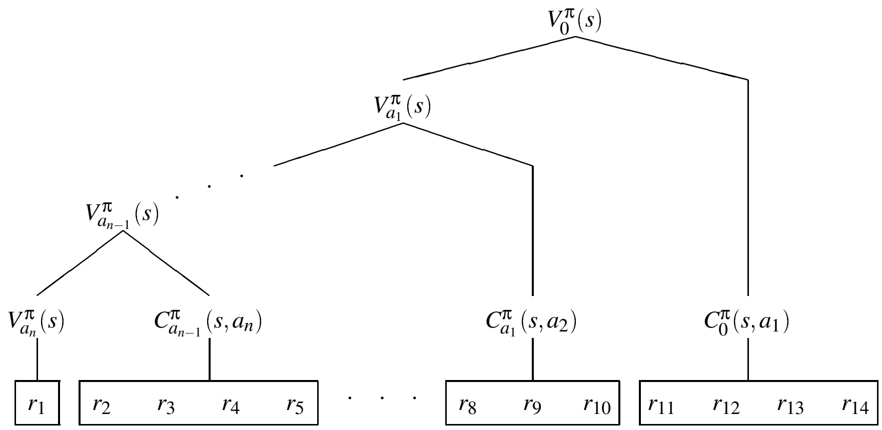

Figure 3: The MAXQ decomposition; $r_1,\ldots,r_{14}$ denote the sequence of rewards received from primitive actions at times $1,\ldots,14$.

condition holds, then the MAXQ graph with abstraction functions $X_i(s,a)$ can represent the value function of any policy π whose value function can be represented by the MAXQ graph with no abstraction functions:

> *For all Max nodes i, actions a, states $s \in S_i$, and distinct states $s_1, s_2 \in Result_i^\pi(s,a)$ whenever $C_i^\pi(s_1,a) \neq C_i^\pi(s_2,a)$ it is the case that $X_i(s_1,a) \neq X_i(s_2,a)$*

In other words, if an abstraction function X_i treats a pair of result states s_1 and s_2 as identical, then their un-abstracted values must be equal. Otherwise, the value function cannot be properly represented. The four children of MaxNavigate all satisfy this condition. The expected reward of completing the MaxNavigate action depends only on the current location of the taxi, the target location, and the amount of fuel remaining. If we are navigating to F (for refueling), for example, the expected reward does not depend on the source or destination locations.

The introduction of abstractions can create a *hierarchical credit assignment problem*. For example, in our implementation, we used only the taxi location and the target location to represent the C functions for QNorth, QSouth, QEast, and QWest. We wanted these nodes to learn a navigation policy that was independent of how much fuel remained. But this means that when the fuel is exhausted and a -20 penalty is received, these Q nodes cannot represent the reason for this penalty! This is the hierarchical credit assignment problem: to determine which node is responsible for a reward that is received. Our solution is for the designer of the MAXQ hierarchy to also *decompose the reward function*. When each reward is generated, a marker is attached that indicates which Q nodes are potentially responsible for this reward. For the -20 empty fuel penalty, the QGet, QPut, and QRefuel nodes are held responsible, because their parent, MaxRoot, must compare their Q values to decide when to refuel to avoid the penalty. Their C functions must therefore be able to represent the rewards.

This requires a change to the decomposition equations. Let $R_i(s'|s,a)$ be the portion of the reward that is assigned to node i. Then we write the following:

$$C_i^\pi(s,a) = \sum_{s'} P(s'|s,a)[R_i(s'|s,a) + V_i^\pi(s')] \qquad (6)$$

$$V_i^\pi(s) = \begin{cases} Q_i^\pi(s,\pi_i(s)) & i \text{ composite} \\ \sum_{s'} P(s'|s,i)R_i(s'|s,i) & i \text{ primitive} \end{cases} \qquad (7)$$

In many domains, we believe it will be easy for the designer of the hierarchy to also decompose the reward function. However, an interesting problem for future research is to develop algorithms for autonomously solving the hierarchical credit assignment problem.

4 A Learning Algorithm

The preceding section has shown that the hierarchy can correctly represent the value function of any hierarchical policy if the full state is employed to represent the C_i function in each node i. Hence, we could apply Parr and Russell's HAM-Q algorithm to learn the best hierarchical policy. However, because we are committed to employing state abstractions, we have chosen instead to develop a reinforcement learning algorithm for finding a recursively optimal policy.

It turns out that in general there can be many different recursively optimal policies, and that some of them achieve better expected rewards than others. The problem is that a subtask may have many policies that are locally optimal, but some of them are more useful than others for the overall task. For example, suppose we changed the taxi domain so that if the taxi hits a wall, the trial is terminated with a reward of -5. Then for MaxNavigate(t), if the target location t is more than 5 steps away, the locally optimal policy would be to hit a wall. This would not be part of any hierarchically optimal policy, however! Dayan and Hinton faced this same problem, and they solved it by providing a penalty of 10 points to subtask i for entering an undesired terminal state (i.e., a state in T_i but not in G_i). This has the proper effect, but in the MAXQ hierarchy, it causes the value function computed by the entire hierarchy to be incorrect, because it incorporates the (often non-zero) probability of receiving these terminal state penalties.

A better method is to define, for each Max node MDP M_i, a parallel Markov decision problem $\tilde{M}_i$ with the same states, actions, and transition probabilities as M_i but with a second reward function $\tilde{R}_i$ that is zero except for undesired terminal states, where it provides a large penalty. (We used a penalty of -100 points). Our learning algorithm will seek a locally optimal policy $\tilde{\pi}_i^*$ for $\tilde{M}_i$. However, it will also compute the value function for executing $\tilde{\pi}_i^*$ in the original MDP M_i, and this is the value that will be passed "up" the MAXQ hierarchy.

Specifically, our learning algorithm MAXQ-Q is a variant of Q learning that performs the following. At each composite Max node, we maintain two tables $C_i(s,a)$ and $\tilde{C}_i(s,a)$. The algorithm chooses an action a to perform according to its current exploration policy. It executes a, observes the resulting state s' and reward $R_i(s'|s,a)$, and computes the following:

$$a^* := \operatorname*{argmax}_{a'} [\tilde{C}_i(s',a') + V_{a'}(s')] \quad (8)$$

$$\begin{aligned}\tilde{C}_i(s,a) := & (1-\alpha_t(i))\tilde{C}_i(s,a) + \alpha_t(i) \cdot \\ & [\tilde{R}_i(s') + R_i(s'|s,a) + \tilde{C}_i(s',a^*) + V_{a^*}(s')]\end{aligned} \quad (9)$$

$$\begin{aligned}C_i(s,a) := & (1-\alpha_t(i))C_i(s,a) + \alpha_t(i) \cdot \\ & [R_i(s'|s,a) + C_i(s',a^*) + V_{a^*}(s')]\end{aligned} \quad (10)$$

Here a^* is the best action in s' according to the current $\tilde{C}$ and V values. Both $\tilde{C}$ and C are updated using a^*. At each leaf node i, the update is slightly different:

$$V_i(s) := (1-\alpha_t(i))V_i(s) + \alpha_t(i)R_i(s'|s,i). \quad (11)$$

The quantity $\alpha_t(i)$ is the learning rate for node i at time step t.

In order to prove convergence of this algorithm, we must make several assumptions. First, we must assume that all deterministic policies in MDP M are proper (i.e., they all terminate with probability 1). Second, we must assume that all locally optimal policies, $\tilde{\pi}_a^*$, give the same transition probability distribution $P_i^{\tilde{\pi}_a^*}(s'|s,a)$. This ensures that all locally optimal policies at node a give rise to the same MDP at any node i that is a parent of a. (A consequence of this assumption is that all recursively optimal policies will have the same value function.) Third, we must assume that $|V_i|$, $|C_i|$, and $|\tilde{C}_i|$ are bounded at all times (this is easy to enforce). Fourth, the exploration policy executed at each node i during learning must be a GLIE (greedy in the limit with infinite exploration) policy—that is, a policy that executes each action infinitely often in every state that is visited infinitely often, and that is greedy with respect to $\tilde{Q}_i$ with probability 1. Finally, the learning rates $\alpha_t(i)$ must satisfy the usual conditions:

$$\lim_{T\to\infty} \sum_{t=1}^{T} \alpha_t(i) = \infty \quad \text{and} \quad \lim_{T\to\infty} \sum_{t=1}^{T} \alpha_t^2(i) < \infty \quad (12)$$

Theorem 3 *Under the assumptions listed above, with probability 1, MAXQ-Q will converge to a recursively optimal policy for MDP M consistent with MAXQ hierarchy H.*

Proof Sketch: The proof employs a stochastic approximation argument similar to those introduced to prove the convergence of Q learning and $SARSA(0)$ (Jaakkola, Jordan, & Singh, 1994; Bertsekas & Tsitsiklis, 1996; Singh, Jaakkola, Littman, & Szpesvari, 1998). The proof is by induction on the levels of the tree, starting at the Max nodes all of whose children are primitive leaf nodes. At these "first-level" Max nodes, the standard results for Q learning can be applied to prove that the C_i values will converge with probability 1 to the optimal value function. Furthermore, because each node i is executing a GLIE exploration policy, the policy at these nodes will also converge with probability 1 to a locally optimal policy.

Now consider a Max node j all of whose children are either primitive nodes or "first-level" Max nodes. Define $P_j^t(s'|s,i)$ to be the transition probabilities observed by parent node j when it invokes child node i in state s at time t in the learning process. Because the first-level Max nodes are executing GLIE policies, $P_j^t(s'|s,i)$ will converge (with probability 1) to the state transitions $P_j^*(s'|s,i)$ that will be produced by any of the locally optimal policies for node i (by assumption, all of these locally optimally policies give the same state transition probabilities). This enables us to prove that node j also converges with probability 1 to the optimal C_j values and a locally-optimal policy. The key is to decompose the error in any particular C_j backup into two terms. One term—corresponding to the difference between a sample backup (using the observed state transition) and a full Bellman backup (using $P_j^t(s'|s,i)$)—has expected value of zero. The other term—corresponding to the difference between doing a full Bellman backup using the current transition probabilities, $P^t_j(s'|s,i)$ and doing a full Bellman backup using the final transition probabilities $P^*_j(s'|s,i)$—converges to zero with probability 1. By applying a stochastic approximation result (Proposition 4.5 from Bertsekas and Tsitsiklis, 1996), we can prove that node j will converge to a locally optimal policy. Hence, by induction, we can prove that the entire hierarchy converges to a recursively optimal policy. **End of Proof Sketch.**

There is one interesting method that can be employed to accelerate learning in the higher nodes of the graph. When an action a is chosen for Max node i in state s, the execution of a will move the environment through a series of states $s_1, \ldots, s_k, s_{k+1} = s'$. If a was indeed the best action to choose in s_1, then it should also be the best action to choose (at node i) in states s_2 through s_k. Hence, equations (9) and (10) can be applied in all of these states. This reflects an important difference between standard subroutine calls and the MAXQ hierarchy. In standard subroutines, there is a set of preconditions that must be true at the start of the subroutine. A partially-executed subroutine can often make these preconditions false, so that it is not possible to interrupt a subroutine and then call it again without first re-establishing the preconditions. In the MAXQ hierarchy, however, a Max node i can be invoked in any state $s \in S_i$, and it must "complete" execution of the task from that state onward. This means that the execution of the Max node can be interrupted and restarted with no change to the hierarchy.

We applied algorithm MAXQ-Q to the Taxi task using a tabular representation of the C functions. We employed state abstraction as follows. For the QNorth, QSouth, QEast, and QWest nodes, the C function ignores the passenger source and destination locations and the amount of fuel. The C function of QPickup ignores the passenger destination and fuel, but it must know the source location and taxi location in order to predict the effects of illegal Pickup actions. Similarly, QPutdown ignores the passenger source location and the fuel, and QFillup ignores the source and destination locations and the fuel. QNavigateForGet can represent its C function by a single value, because after a successful Navigate, only a Pickup remains to complete the Get action. The same is true for QNavigateForPut and QNavigateForRefuel. Because of the hierarchical credit assignment, QGet and QRefuel need to see the entire state, but QPut can ignore all of the state information, because once it succeeds, the task is completed. All of these abstractions mean that instead of a set of seven 8,750-element Q functions (61,250 values) for flat Q learning, the MAXQ hierarchy requires only 18,253 values to represent the C functions.

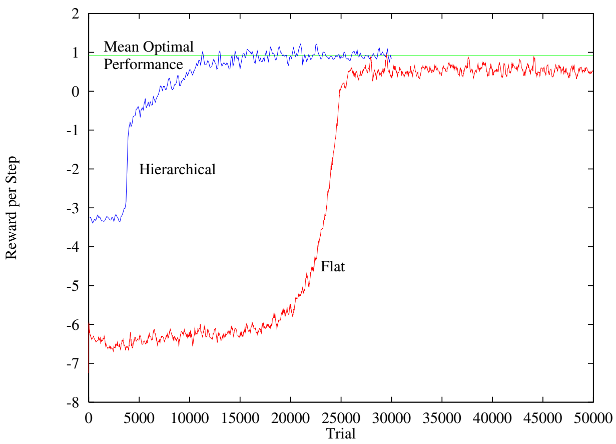

Figure 4: Online performance of flat and hierarchical Q learning on the Taxi task. Each curve is smoothed using a 200-trial moving average. The horizontal line shows the average performance of the optimal policy.

Figure 4 compares the online performance of flat and hierarchical Q learning. For flat Q learning, we employed Boltzmann exploration with an initial temperature of 50. This was decreased by a factor of 0.997 after each successful trial. We experimented with many different cooling schedules, but we were unable to get flat Q learning to converge to the optimal policy within 50,000 trials. This was the fastest cooling schedule that was able to attain (at least briefly) the optimal expected reward. For hierarchical Q learning, we employed a separate temperature for each Max node. The starting temperature for all nodes was 50 except MaxRoot, which used 100. Each node decreased its temperature when it successfully reached a goal terminal state. MaxRoot was cooled by a factor of 0.9986, the second level Max nodes at 0.997, and MaxNavigate at 0.995. In all cases, a learning rate of $\alpha = 1$ was employed, since all actions and rewards are deterministic.

These cooling rates were chosen so that the lower Max nodes in the graph can become reasonably competent at their subtasks before the nodes higher in the graph try to learn. If care is not taken, a Max node i may conclude that a subtask a is very expensive (because the subtask has not yet learned a good policy), and therefore, it sets the C value for a very low. When this is combined with Boltzmann exploration, the result is that the subtask may never be tried again. Hence, we only performed an update for a Q node if that node completed its subtask with an average absolute Bellman error per step of less than 0.2. (This parameter was not tuned at all.)

Figure 4 shows that the hierarchical method is able to learn the task much faster and achieve a higher level of performance than flat Q learning. Of course, both methods could be improved by employing techniques for accelerating Q learning, such as eligibility traces (e.g., Peng & Williams, 1996).

5 Non-Hierarchical Execution

We have shown that the MAXQ hierarchy can learn an optimal policy for an MDP if that policy is a recursively optimal hierarchical. However, there are situations in which the optimal policy is almost—but not quite—hierarchical. For example, consider a modified Taxi task (the "fickle Taxi problem") in which as soon as the taxi picks up the passenger and moves one square, the passenger can randomly change the destination with probability 0.3. This change comes after the hierarchical policy has committed to executing QNavigateForPut(t) for the original destination. As a result, the MaxNavigate subtask will take the taxi to the old destination. Then control will return to MaxPut, which will invoke QNavigateForPut to move the taxi to the new destination.

Such "almost hierarchical" MDP's raise the question of whether there is a way to convert a recursively-optimal hierarchical policy into an optimal non-hierarchical policy.

To answer this question, we implemented the Fickle Taxi domain. We removed all aspects of fuel from the domain so that we could figure out the optimal policy and hand-code it. Figure 5 compares the performance of flat Q learning and hierarchical Q learning on this modified task. The optimal policy can achieve an average reward per step of 1.172; but the best hierarchical policy (compatible with the MAXQ graph of Figure 2) can only achieve 1.002. Hierarchical learning with MAXQ-Q is able to attain this level rapidly. Flat Q learning approaches the optimum, but does not reach it within 10,000 trials. We tuned each algorithm to optimize its performance. We employed a learning rate of 0.35 and decayed the initial temperature of 50.0 by a factor of .460 (for flat Q) and .211 (for hierarchical Q) whenever a goal terminal state was reached.

An alternative to hierarchical execution of the MAXQ graph is *polling execution*, as first suggested by Kaelbling in her (1993) Hierarchical Distance to Goal method. In the polling approach to MAXQ, each action is chosen by starting at MaxRoot and computing the path (from root to leaf) with the highest Q value. The primitive action at the end of this path is then executed, and the process is repeated. This is equivalent to computing the one-step greedy lookahead policy given the current value function. If the hierarchical policy is not optimal, then this one-step greedy policy will be closer to an optimal policy, because it corresponds to one step of policy improvement in the policy iteration algorithm (Bertsekas, 1995). This informally proves the following:

Theorem 4 *For all states s, the value of the policy computed by polling execution of the MAXQ hierarchy is $\geq$ the value of the policy computed by hierarchical execution.*

Hence, polling execution of a MAXQ graph can produce a non-hierarchical policy that is better than the hierarchical policy represented by the graph.

We tested this on the Fickle Taxi task by first training the MAXQ hierarchy by MAXQ-Q for 1000 trials and then continuing the training with polling execution. Figure 6 shows that there is an initial loss of performance when we switch to polling execution. This is because during hierarchical training, the more abstract Q nodes in the graph have only learned their C values well in states where they were frequently executed. Under polling, they are now executed in other states as well, and they rapidly learn the correct values so that performance is able to reach the level of the optimal non-hierarchical policy. In this domain, polling execution of the best hierarchical policy can produce the optimal policy.

6 Concluding Remarks

This paper has defined the MAXQ value function decomposition for hierarchical reinforcement learning. The paper has shown that the MAXQ graph can represent the value function of any hierarchical policy implemented by the graph. A learning algorithm based on Q learning was introduced, proved to converge, and shown experimentally to perform much better than ordinary, non-hierarchical Q learning.

The most important aspect of the MAXQ method is the separation between the context-independent policy and value function (represented by the Max nodes) and the context-dependent value function (represented by the Q) nodes. This permits the value functions of subtasks to be learned independent of their context, and this enhances the reusability of the subtasks and makes it easier to employ state abstraction within the subtasks. However, optimality of the learned policy is lost in general, and hierarchical credit-assignment problems may be introduced. Fortunately, the ability of the MAXQ hierarchy to represent the value function of the hierarchical policy permits the non-hierarchical execution of a one-step greedy policy that is better than the hierarchical policy.

Acknowledgements. The author thanks Eric Chown for many helpful discussions of this work and Valentina Bayer,

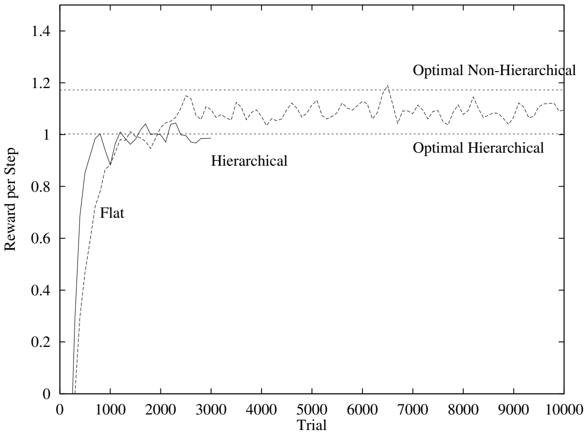

Figure 5: Online performance of flat and hierarchical Q learning on the Fickle Taxi task. Each curve is the average of 10 runs; the returns from each run were smoothed by a 200-trial moving average.

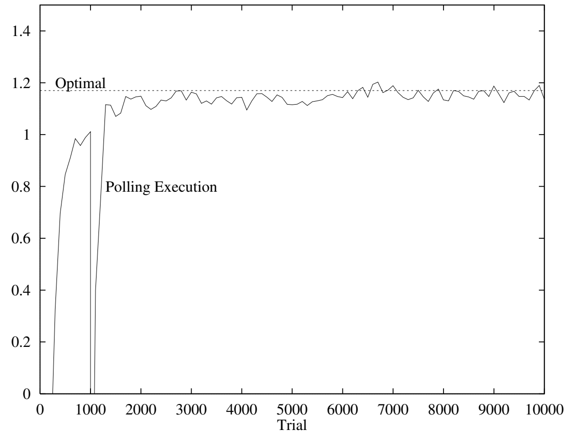

Figure 6: Online performance on the Fickle Taxi task. The first 1000 trials are trained hierarchically. The remaining trials are trained while polling.

William Langford, and Wesley Pinchot for helpful comments on an earlier draft. The support of ONR grant N00014-95-1-0557 and of NSF grant IRI-9626584 is gratefully acknowledged.

References

Bertsekas, D. P. (1995). *Dynamic programming and optimal control*. Athena Scientific, Belmont, MA.

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

Dayan, P., & Hinton, G. (1993). Feudal reinforcement learning. *NIPS, 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, Dept. of Computer Science, Brown University, Providence, Rhode Island.

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

Kaelbling, L. P. (1993). Hierarchical reinforcement learning: Preliminary results. *ICML-93*, pp. 167–173 San Francisco, CA. Morgan Kaufmann.

Parr, R., & Russell, S. (1998). Reinforcement learning with hierarchies of machines. *NIPS*, Vol. 10 Cambridge, MA. MIT Press.

Peng, J., & Williams, R. J. (1996). Incremental multi-step Q-learning. *Mach. Learn.*, *22*, 283–290.

Singh, S., Jaakkola, T., Littman, M. L., & Szpesvari, C. (1998). Convergence results for single-step on-policy reinforcement-learning algorithms. Tech. rep., University of Colorado, Dept. Comp. Sci.

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

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 Mass., Dept. Comp. Inf. Sci., Amherst, MA.