An Overview of MAXQ Hierarchical Reinforcement Learning

Thomas G. Dietterich

Oregon State University, Corvallis, Oregon, USA,
tgd@cs.orst.edu,
WWW home page: http://www.cs.orst.edu/~tgd

Abstract. Reinforcement learning addresses the problem of learning optimal policies for sequential decision-making problems involving stochastic operators and numerical reward functions rather than the more traditional deterministic operators and logical goal predicates. In many ways, reinforcement learning research is recapitulating the development of classical research in planning and problem solving. After studying the problem of solving "flat" problem spaces, researchers have recently turned their attention to hierarchical methods that incorporate subroutines and state abstractions. This paper gives an overview of the MAXQ value function decomposition and its support for state abstraction and action abstraction.

1 Introduction

Reinforcement learning studies the problem of a learning agent that interacts with an unknown, stochastic, but fully-observable environment. This problem can be formalized as a Markov decision process (MDP), and reinforcement learning research has developed several new algorithms for the approximate solution of large MDPs (Sutton & Barto, 1998; Bertsekas & Tsitsiklis, 1996). These algorithms treat the state space of the MDP as a single "flat" search space. This is appropriate in many domains, such as game playing (Tesauro, 1995), elevator control (Crites & Barto, 1995), and job-shop scheduling (Zhang & Dietterich, 1995), where reinforcement learning methods have been successfully applied. But this approach does not scale to tasks such as robot soccer or air traffic control that have a complex, hierarchical structure. If reinforcement learning is to scale up to be part of a theory of human-level intelligence, we must find ways to make it hierarchical by introducing mechanisms for abstraction and sharing.

This paper describes an initial effort in this direction. We will present a method for incorporating hierarchical state and procedural abstractions into reinforcement learning systems. This method is analogous to the introduction of subroutines or parameterized macros in traditional planning and learning systems, and many of the same issues arise. But the need to address the stochastic nature of Markov decision processes (and the possibility of receiving rewards or penalties in every state) creates interesting new issues as well.

The paper begins (in Section 2) with an introduction to Markov decision processes and a toy problem that will serve as the running example for the paper. It introduces the fundamental knowledge structure of most reinforcement learning algorithms—the *value function*. The most fundamental reinforcement learning algorithm, Q learning, is introduced as well. Section 3 then introduces the problem of learning a recursively optimal policy for a programmer-supplied hierarchy of tasks and subtasks. A simple extension to Q learning, the Hierarchical Semi-Markov Q (HSMQ) learning algorithm is introduced and shown to converge to a recursively optimal policy. A drawback of HSMQ learning is that it does not provide a representational decomposition of the value function, which, among other consequences, means that it learns slowly. Section 4 introduces the MAXQ value function decomposition and its corresponding learning algorithm, MAXQ Q learning. Section 5 introduces three forms of state abstraction that can be employed in hierarchical reinforcement learning. One form can be applied to both HSMQ and MAXQ, but the other two forms depend upon the MAXQ value function decomposition. Experimental results are presented showing that with state abstractions, MAXQ Q learning is much more efficient than either flat Q learning or HSMQ learning. Section 6 discusses two tradeoffs in the design of hierarchical reinforcement learning. The first concerns the tradeoff between optimality and state abstraction, and the second concerns the tradeoff between model-based methods and state abstraction. The paper concludes with a brief discussion of omitted topics and open problems.

2 Markov Decision Processes and the Q Learning Algorithm

A Markov decision process models the situation in which an agent interacts with an external, fully-observable environment. At each time step, the agent observes the state of the environment, selects an action, performs the action, and receives a real-valued reward. The action causes the environment to make a state transition, which may be deterministic or probabilistic. The real-valued reward depends on the state of the environment, the action, and the resulting state of the environment after the action. The goal of the agent is to choose actions in such a way as to maximize the total reward that it receives until it enters a terminal state. (This kind of MDP is formally known as an undiscounted finite horizon MDP.)

Consider the example shown in Figure 1. This is a simple grid world that contains a taxi, a passenger, and four specially-designated locations labeled R, G, B, and Y. In the starting state, the taxi is in a randomly-chosen cell of the grid, and the passenger is at one of the four special locations. The passenger has a desired destination that he/she wishes to reach, and the job of the taxi is to go to the passenger, pick him/her up, go to the passenger's destination, and drop the passenger off. The taxi has six primitive actions available to it: move one square north, move one square south, move one square east, or move one square west, pickup the passenger, and putdown the passenger. For the moment,

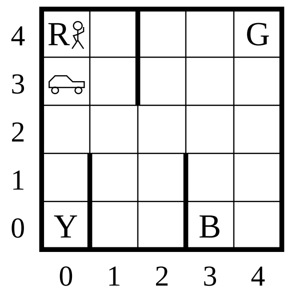

Fig. 1. The Taxi Problem: A simple Markov decision problem.

we will assume that these actions are deterministic, but we will allow them to be stochastic later.

The taxi agent receives rewards as follows. Each action receives a reward of -1. When the passenger is putdown at his/her destination, the agent receives a reward of $+20$. If the taxi attempts to pickup a non-existent passenger or putdown the passenger anywhere except one of the four special spots, it receives a reward of -10. Running into walls has no effect (but entails the usual reward of -1).

A rule for choosing actions is called a *policy*. Formally, it is a mapping π from the set of states S to the set of actions A. If an agent follows a fixed policy, then over many trials, it will receive an average total reward which is known as the *value* of the policy. In addition to computing the value of a policy averaged over all trials, we can also compute the value of a policy when it is executed starting in a particular state s. This is denoted $V^\pi(s)$, and it is the expected cumulative reward of executing policy π starting in state s. We can write it as

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

where r_t is the reward received at time t, s_t is the state of the environment at time t, and the expectation is taken over the stochastic results of actions in the environment.

For any MDP, there exist one or more optimal policies, which we will denote by π^* that maximize the expected value of the policy. All of these optimal policies share the same optimal value function, which is written V^*. The optimal value function satisfies the Bellman equation:

$$V^*(s) = \max_a \sum_{s'} P(s'|s,a)[R(s'|s,a) + V^*(s')],$$

where a denotes an action to be performed in state s, s' denotes the resulting state (which is reached according to the transition probability $P(s'|s,a)$), $R(s'|s,a)$ denotes the expected one-step reward of performing action a in state s and moving to state s', and $V^*(s')$ is the value of the resulting state. The sum on the right-hand-side is the expected value of the one step reward $R(s'|s,a)$

plus the value of the next state s', so we can think of it as the backed-up value of a one-step lookahead search, and the $\max_a$ is choosing the action with the best backed-up value.

Indeed, the sum is so important that it is given a special name, $Q^*(s,a)$:

$$Q^*(s,a) = \sum_{s'} P(s'|s,a)[R(s'|s,a) + V^*(s')].$$

This is the expected total reward that will be received when the agent performs action a in state s and then behaves optimally thereafter. By substituting this into the Bellman equation, we can see that the value function is just the maximum (over all actions) of the Q function:

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

Consequently, we can substitute this into the Q equation to obtain the Q version of the Bellman equation:

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

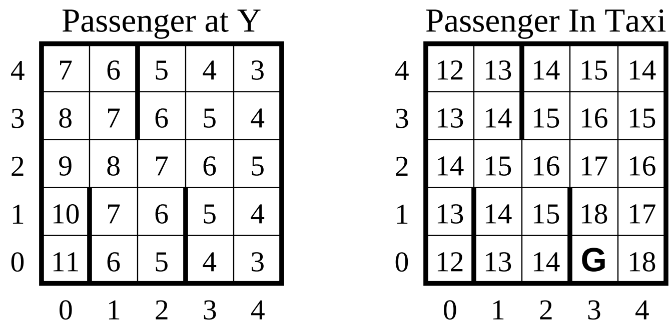

Fig. 2. Value function for the case where the passenger is at (0,0) (location Y) and wishes to get to (3,0) (location B).

Figure 2 shows the optimal value function for the taxi problem in the case where the passenger is at location Y and wishes to move to location B. To understand this figure, consider the maze on the right, and look at cell (3,1), which has the value 18. This cell corresponds to the case where the passenger is in the taxi and the taxi is at this location. If the taxi performs a south action followed by a putdown action, the passenger will arrive at his/her destination, and a reward of +20 will be received. However, each of the two actions costs -1, so the value of being in this state is $20 - 2 = 18$. Similarly, consider the cell (2,2) in the maze on the left, which contains the value 7. This corresponds to the situation where the passenger is waiting for the taxi at location (0,0), and the taxi is at (2,2). The taxi must move 4 squares (west, west, south, south), then issue a pickup. At this point, we can think of the taxi as "jumping" to the maze on the right, because now the passenger is in the taxi. Seven more moves plus a putdown are required to deliver the passenger to location G. Hence, this is a total of 13 actions (total reward -13) to deliver the passenger (reward $+20$), for a net value of 7

The problem of probabilistic planning is to compute the optimal policy π^* given complete knowledge of the MDP (i.e., the transition function $P(s'|s,a)$ and the reward function $R(s'|s,a)$). Several offline dynamic programming algorithms can perform this computation for state spaces on the order of 30,000 states. These algorithms require time that scales as the cube of the number of states. The most popular algorithm is value iteration, and it works by iteratively computing the optimal value function V^*. Given V^*, the optimal policy can be computed by performing a one-step lookahead search to compute $Q^*(s,a)$ and then choosing the action a that maximizes this:

$$\pi^*(s) = \operatorname*{argmax}_a \; Q^*(s,a).$$

The problem of reinforcement learning is to compute the optimal policy given no prior knowledge about the MDP but given instead the ability to interact online with the MDP. By interacting with the MDP, the reinforcement learning agent can observe state s, try action a, and observe the resulting state s' and the reward r. From this information (accumulated over many interactions), it can form an estimate of the probability transition function ($\hat{P}(s'|s,a)$) and of the expected one-step reward function ($\hat{R}(s'|s,a)$). Hence, one approach to reinforcement learning is simply to interact with the environment, estimate this information, and then apply offline dynamic programming algorithms.

But an alternative is to construct an estimate of V^* or Q^* directly, without learning $\hat{P}$ and $\hat{R}$ first. The Q-learning algorithm discovered by Watkins does this as follows. Let $Q^t(s,a)$ be our current estimate (at time t) of the optimal Q function. At each time step t, the agent observes the state s of the environment, chooses action a according to some *exploration policy* π_x, observes the resulting state s' and the one-step reward r, and performs the following update:

$$Q^{t+1}(s,a) := (1-\alpha)Q^t(s,a) + \alpha \left[r + \max_{a'} \; Q^t(s',a') \right].$$

The parameter α is a learning rate (typically between 0 and 1). The expression on the right-hand-side computes a moving average between the previous value of $Q(s,a)$ and a new "estimated value" resulting from the current experience. If α is gradually decreased according to certain standard conditions, and if π_x ensures that every action is executed infinitely often in every state, then with probability 1, Q^t converges to Q^*.

It is important to note that the action a can be very simple or very complex, and this algorithm will still work. Indeed, action a can be a call to a subroutine that takes many primitive actions and then exits. When that subroutine exits, it will leave the environment in some new state s'. If we define r to be the total reward that was received while the subroutine a was being executed, then the Bellman equation is still satisfied, and Q learning will still converge to Q^*. Technically, this variant of Q learning is called semi-Markov Q learning (or SMDP Q learning), because an MDP in which actions can take multiple time steps is known as a semi-Markov decision problem. In the undiscounted finite horizon case that we are considering, SMDP Q learning is identical to standard Q learning, but in other situations, the algorithm must be modified slightly (Parr, 1998).

3 Task Decompositions and Reinforcement Learning

The aim of hierarchical reinforcement learning is to discover and exploit hierarchical structure within a Markov decision problem. In this paper, we will sidestep the problem of discovering hierarchical structure and focus on the problem of exploiting a programmer-provided task hierarchy. The programmer must define a hierarchy of subroutines, but it is the reinforcement learning system that will "write the code" for each subroutine—that is, the reinforcement learning system will find a policy for choosing actions within each subroutine. When the learning has finished, the policy for each subroutine will be an optimal solution to a sub-MDP of the original MDP, and the policy of the overall MDP will be a combination of the policies of the various subroutines. An important benefit of this approach is that these sub-MDPs (and their learned optimal policies) will be re-usable in new tasks.

Given an MDP, we will rely on a programmer to design a task hierarchy. For example, Figure 3 shows a task hierarchy for the Taxi problem. This decomposes the overall task (root) into two subtasks: get the passenger (move the taxi to the passenger's location and pick up the passenger), and put the passenger (move the taxi to the passenger's destination and put down the passenger). Each of these is decomposed in turn. get decomposes into navigate(source) and the primitive pickup, while put decomposes into navigate(destination) and the primitive putdown. Finally, the parameterized subroutine navigate(t) decomposes into the four motion primitives north, south, east, and west.

To define each subtask, the programmer must specify a termination predicate and a set of child tasks. For example, the subtask for navigate(t) is terminated if and only if (iff) the taxi is at location t, the subtask for get is terminated iff the taxi contains the passenger, and the subtask for put is terminated iff the passenger is at his/her destination.

With this information, we can define the goal of our hierarchical reinforcement learning algorithm to be finding a *recursively optimal policy.* A recursively optimal policy is an assignment of policies to each individual subtask such that the policy for each subtask is optimal given the policies assigned to all of its descendants. A recursively optimal policy is a kind of local optimality. It does not guarantee anything about the quality of the resulting overall policy, but it

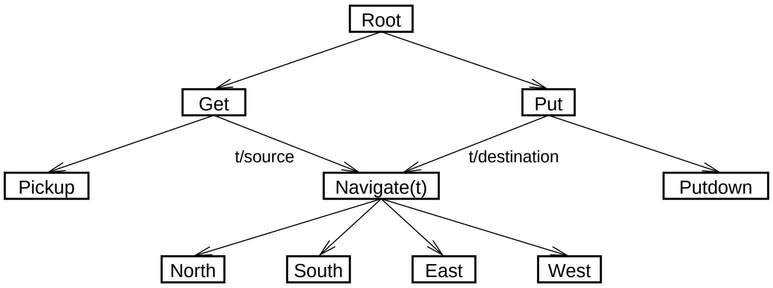

Fig. 3. A task hierarchy for the Taxi domain.

does ensure that each policy is locally the optimal solution to an MDP defined by the subtask and the policies of all of its descendants.

To learn such policies, we can apply Semi-Markov Q learning simultaneously to each task within the task hierarchy, an algorithm which we will refer to as Hierarchical Semi-Markov Q Learning (or HSMQ). Each subtask p will learn its own Q function $Q(p, s, a)$ which is the expected total reward of performing subtask p starting in state s, executing action a and then following the optimal policy thereafter. Specifically, each subtask performs the following:

function HSMQ(state s, subtask p) **returns** float
    Let $TotalReward = 0$
    while p is not terminated **do**
        Choose action $a = \pi_x(s)$ according to exploration policy π_x
        Execute a.
            if a is primitive, Observe one-step reward r
            else $r := HSMQ(s, a)$, which invokes subroutine a and
                returns the total reward received while a executed.
        $TotalReward := TotalReward + r$
        Observe resulting state s'
        Update $Q(p, s, a) := (1 - \alpha)Q(p, s, a) + \alpha \left[r + \max_{a'} Q(p, s', a')\right]$
    end // **while**
    return $TotalReward$
end

By an argument similar to Dietterich (2000), it can be proved that this algorithm will converge to a recursively optimal policy for the original MDP provided that the learning rates α decrease according to certain technical requirements and that the exploration policies π_x (i) execute every action a infinitely often in every state s that is visited infinitely often and (ii) in the limit of infinite exploration they become greedy with respect to $Q(p, s, a)$. Such exploration policies are said to be Greedy in the Limit of Infinite Exploration or GLIE (Singh, Jaakkola, Littman, & Szepesvári, 1998).

The reason that the exploration policies must be GLIE is the following. Consider a subtask whose actions are themselves subroutines. The Q learning algorithm relies on executing an action a and getting accurate samples of its expected one-step reward $R(s'|s,a)$ and its result state probabilities $P(s'|s,a)$. If a subroutine a continues executing a non-greedy exploration policy π_x forever, then the samples of its behavior obtained by the parent subtask will be samples of the behavior of this exploration policy rather than samples of the behavior of the locally optimal policy learned for subtask a.

By a similar line of thought, one might expect that simultaneous learning at all levels of the hierarchy would be pointless. That each higher level should wait until its children have converged to a fixed policy. But in practice, useful learning can take place in a parent task before its children have completely converged. And the resulting HSMQ algorithm is a fully-online incremental algorithm.

4 Value Function Decomposition and Reinforcement Learning

The HMSQ learning algorithm treats the hierarchical reinforcement learning problem as a collection of simultaneous, independent Q learning problems. Although it provides a procedural decomposition of the learned policy into policies for each subtask, it does not provide a representational decomposition of the value function: The value function of each subtask is represented and learned independently. We would like to obtain some sharing (and compactness) in the representation of the value function.

Consider, for example, the value function shown in Figure 4 and compare it to the value function in Figure 2. These are the value functions that would be represented and learned by the root task. Although the value functions of the two right-side mazes (where the passenger is in the taxi) differ, the value functions of the left-side mazes are identical except for an offset of 3. The reason is that both of these left-hand side mazes are really reflecting the same subgoal—that of moving the taxi to location (0,0) and picking up the passenger. They differ in what happens *after* the passenger is picked up. In the case of Figure 2, the passenger's destination is 7 steps away, whereas in Figure 4, the destination is only 4 steps away. The difference $7 - 4 = 3$ accounts for the difference between the left-side value functions. We would like to exploit this regularity to represent the left-side value function only once.

The MAXQ value function decomposition is a way of achieving this. The idea is to decompose the $Q(p,s,a)$ value into the sum of two components. The first component is the expected total reward received while executing action a, and the second component is the expected total reward of completing parent task p *after* a *has returned.* Clearly, the total expected reward of performing action a and then following the optimal policy thereafter is the sum of these two components. The key observation is that the first component is exactly the value function for the subtask a, which we will denote by $V(a,s)$. We will call the second component the *completion function*, and we will denote it by $C(p,s,a)$.

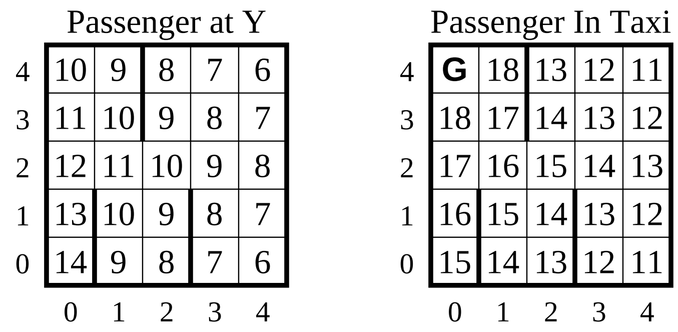

Fig. 4. Value function for the case where the passenger is at (0,0) (location Y) and wishes to get to (0,4) (location R).

This allows us to write

$$Q(p, s, a) = V(a, s) + C(p, s, a).$$

This equation shows how we can relate the Q value of a parent task to the value function of a child task. Applied recursively, it shows how we can decompose the Q function of the root task into a sum of Q values for all of its descendant tasks:

$$V(p, s) = \max_a \left[V(a, s) + C(p, s, a)\right]$$

We can terminate this recursion by defining $V(a, s)$ for primitive actions to be the expected one-step reward of performing action a in state s:

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

In the MAXQ decomposition, we can think of each non-primitive subtask p as storing $C(p, s, a)$ for each non-terminated state s and each child action a. The primitive actions store $V(a, s)$. As an example, consider again the situation in Figure 2 where the taxi is at location (2,2) and the passenger is at location Y (0,0) and wishes to get to location B (3,0). Figure 5 shows how the value of 7 for this state is decomposed into a sum of completion costs. The tree on the left side of the figure shows the values in English, and the tree on the right shows the same values using our formal notation. Each node in the tree computes the sum of its two children.

The value of 7 at the root is the sum of the reward of performing the get, which is -5, plus the reward of completing the root task, which is 12. The -5 of the get task is the sum of the reward for moving the taxi to location Y (i.e., of performing navigate(Y)), which is -4, and the reward for completing the get afterwards, which is -1. Finally, the value -4 of the navigate(Y) is the sum of the reward for performing one west action (-1) and the reward for completing the navigate, which is -3. Formally, we write this as

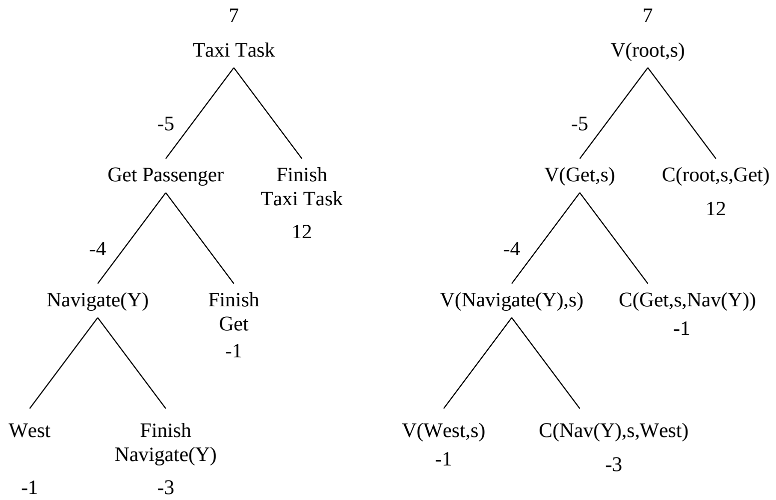

Fig. 5. An example of the MAXQ value function decomposition for the state in which the taxi is at location (2,2), the passenger is at (0,0), and wishes to get to (3,0). The left tree gives English descriptions, and the right tree uses formal notation.

$$\begin{aligned} V(\mathsf{root}, s) = V(\mathsf{west}, s) &+ C(\mathsf{navigate}(Y), s, \mathsf{west}) \\ &+ C(\mathsf{get}, s, \mathsf{navigate}(Y)) \\ &+ C(\mathsf{root}, s, \mathsf{get}). \end{aligned}$$

Dietterich (2000) proves that the MAXQ value function decomposition can represent the value function of *any* hierarchical policy (i.e., any assignment of policies to subtasks in a hierarchy), not just recursively optimal policies. This result extends to discounted, infinite-horizon MDPs and stochastic policies.

It is just as easy to learn using the MAXQ value function decomposition as it was to learn using the un-decomposed value functions and the HSMQ algorithm. We call the resulting algorithm MAXQQ learning:

function MAXQQ(state s, subtask p) **returns** float
  Let $TotalReward = 0$
  while p is not terminated **do**
    Choose action $a = \pi_x(s)$ according to exploration policy π_x
    Execute a.
      if a is primitive, Observe one-step reward r
      else $r := MAXQQ(s, a)$, which invokes subroutine a and
        returns the total reward received while a executed.
    $TotalReward := TotalReward + r$
    Observe resulting state s'
    if a is a primitive
      $V(a, s) := (1 - \alpha)V(a, s) + \alpha r$
    else a is a subroutine

$C(p, a, s) := (1 - \alpha)C(p, s, a) + \alpha \max_{a'} [V(a', s') + C(p, s', a')]$
end // **while**
return $TotalReward$
end

Under the same conditions as HSMQ, MAXQQ converges with probability 1 to a recursively optimal policy.

5 State Abstraction and Hierarchical Reinforcement Learning

In many subtasks, the value function does not depend on all of the state variables in the original MDP. For example, in the navigate(t) subtask, the value function only depends on the location of the taxi and the location of the target cell t; the location of the passenger and the destination of the passenger are irrelevant. We would like to exploit state abstraction within subtasks in order to reduce the amount of memory required to store the value function and reduce the amount of experience required to learn the value function.

There are three fundamental forms of state abstraction that can be applied within the MAXQ value function decomposition: (a) irrelevant variables, (b) funnel abstractions, and (c) structural constraints. We will see that the first of these can be applied even within the HSMQ learning algorithm, but that the second and third forms require the MAXQ value function decomposition.

A state variable is irrelevant for a subtask if the value of that state variable never affects either the values of the relevant state variables or the reward function. Formally, suppose that the state s is represented by a collection of state variables (e.g., taxi location, passenger location, passenger destination). Suppose that we partition these state variables into two subsets X and Y, and denote a state s as a pair (x, y), where x specifies the values of the state variables in X, and y specifies the values of the state variables in Y. We will say that state variables Y are irrelevant for a subtask if, for all non-terminated states (x, y), the following properties hold:

1. The probability transition function can be factored as $P(x', y'|x, y, a) = P(x'|x, a)P(y'|x, y, a)$.
2. The reward function depends only on the variables in X: $R(x', y'|x, y, a) = R(x'|x, a)$.

Figure 6 shows a dynamic belief network that captures these constraints.

It is easy to see that under these conditions, although the chosen actions a within a subtask may affect the values of the Y variables, the Y variables can have no effect on the rewards received in the subtask, and hence, they are irrelevant to the value function and can be ignored.

In the taxi domain, this form of abstraction permits us to ignore the passenger location during the navigate and put subtasks and ignore the passenger

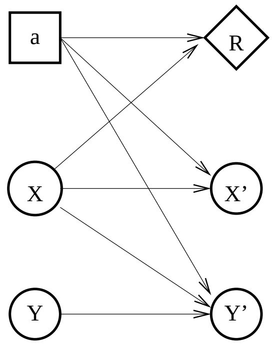

Fig. 6. A dynamic belief network illustrating the definition of irrelevance.

destination during the navigate and get subtasks. Clearly, this form of state abstraction can be applied with the HSMQ learning algorithm as well, since it does not depend on the MAXQ value function decomposition.

The second form of state abstraction is the "funnel" abstraction. A funnel action is an action that causes a larger number of initial states to be mapped into a small number of resulting states. For example, the navigate(t) action maps any state into a state where the taxi is at location t. The key thing to note is that the completion cost of an action, $C(p, s, a)$, only depends on the distribution of possible states s' that can result from performing action a. Specifically, we can write $C(p, s, a)$ as

$$C(p, s, a) = \sum_{s'} P(s'|s, a)V(p, s').$$

For the navigate(t) action, the result state s' will have the same passenger location and destination as the initial state s, but the taxi will now be located at t. This means that the completion cost is independent of the location of the taxi—it is the same for all initial locations of the taxi. This is evident in Figure 2, where the completion function for moving to location Y and picking up the passenger is 12 regardless of the starting location of the taxi.

In the Taxi task, funnel abstractions can be applied as follows. The completion costs $C(\mathsf{get}, s, \mathsf{navigate}(t))$ and $C(\mathsf{put}, s, \mathsf{navigate}(t))$ are independent of the taxi location. Similarly, the completion cost $C(\mathsf{root}, s, \mathsf{get})$ is independent of the taxi location.

The third form of state abstraction results from constraints introduced by the structure of the hierarchy. For example, if a subtask is terminated in a state s, then there is no need to represent its completion cost in that state. For example, there is no need to represent $C(\mathsf{root}, s, \mathsf{put})$ in states where the passenger is not in the taxi, because the put is terminated in such states.

Another structural constraint concerns implication relationships between a child task and its parent task. In some states, the termination predicate of the child task implies the termination predicate of the parent task. In such states, the completion cost must be zero. For example, $C(\mathsf{root}, s, \mathsf{put})$ is always zero in cases where the put is not terminated, because after the put is completed, the passenger will be at his/her destination, and therefore, the root task will be terminated as well.

The MAXQ decomposition is essential for the success of the funnel and structural abstractions. It is only because the Q value is decomposed into the completion function and the child value function that we can take advantage of state abstractions that affect only the completion function.

With state abstractions, it is possible to dramatically reduce the amount of memory required to exactly represent the Q function. In the taxi task, for example, flat Q learning requires storing 3,000 Q values. Without state abstractions, the HSMQ learning approach requires 14,000 distinct Q values. But with state abstraction and the MAXQ hierarchy, we only need to store a total of 632 values for C and V.

Interestingly, with the MAXQ decomposition, we can represent the value function for the taxi task as a sum of components such that each component only depends on a subset of the state variables. For example, in the start state, the value function usually decomposes as

$$V(\mathsf{root}, s) = V(\mathsf{navigate}(t), s) + C(\mathsf{get}, s, \mathsf{navigate}(t)) + C(\mathsf{root}, s, \mathsf{get}),$$

where $V(\mathsf{navigate}(t), s)$ depends only on taxi location and t, $C(\mathsf{get}, s, \mathsf{navigate}(t))$ depends only on the passenger's starting location, and $C(\mathsf{root}, s, \mathsf{get})$ depends only on passenger's starting location and destination. No value depends on the entire state space.

State abstraction also means that learning is faster, because learning experiences in distinct complete states become multiple learning experiences in the same abstracted state. Consequently, the amount of "training data" for a particular $C(p, s, a)$ value increases, and that value can be determined more rapidly. For example, the taxi can learn how to get to location Y both when it is going to get the passenger who is waiting there and when it is going to put the passenger who is trying to get there.

Figure 7 shows an experimental verification of this for a version of the taxi task in which the four motion actions are stochastic. With probability 0.8, each motion action succeeds, but with probability 0.2, the taxi instead moves in a direction perpendicular to the desired direction. For example, when executing a north action, with probability 0.8 the taxi moves north, but with probability 0.1 it moves east, and with probability 0.1 it moves west. In addition, the passenger is somewhat fickle. After the taxi has picked up the passenger and moved one step away from the passenger's original location, the passenger changes his/her destination location with probability 0.3.

The graph compares the online performance (in terms of cumulative reward per trial) of flat Q learning, MAXQQ learning with no state abstractions, and MAXQQ learning with state abstractions. We can see that without state abstractions, MAXQQ learning is reasonably successful, although it takes somewhat longer than flat Q learning to finally converge. But with state abstractions, MAXQQ converges to a near-optimal policy more than twice as fast as flat Q learning.

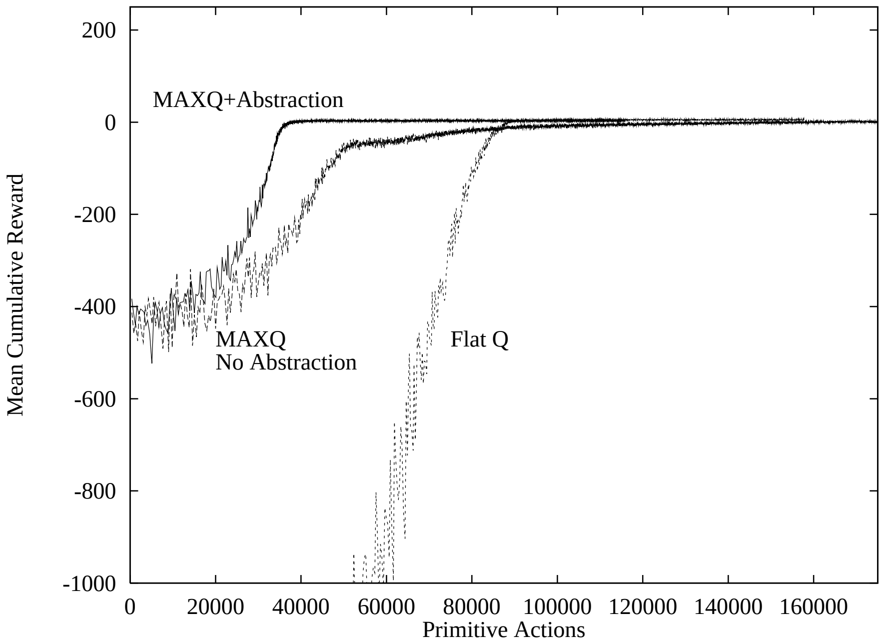

Fig. 7. Comparison of Flat Q learning, MAXQ Q learning with no state abstraction, and MAXQ Q learning with state abstraction on a noisy version of the taxi task.

The MAXQ value function decomposition builds upon previous work by Singh (1992), Kaelbling (1993), and Dayan and Hinton (1993). Singh and Kaelbling were the first researchers to seek a decomposition of the value function as well as a decomposition of the policy. Kaelbling developed the HDG method, which was suitable only for a special kind of navigation task. We have replicated her work using the MAXQ value function decomposition, and the resulting learning curves are shown in Figure 8. In this domain, we see that MAXQQ without state abstractions performs much worse than simple flat Q learning, but with state abstractions, MAXQ Q learning is approximately four times more efficient. This experiment shows even more clearly than the taxi domain how important state abstractions are for hierarchical reinforcement learning.

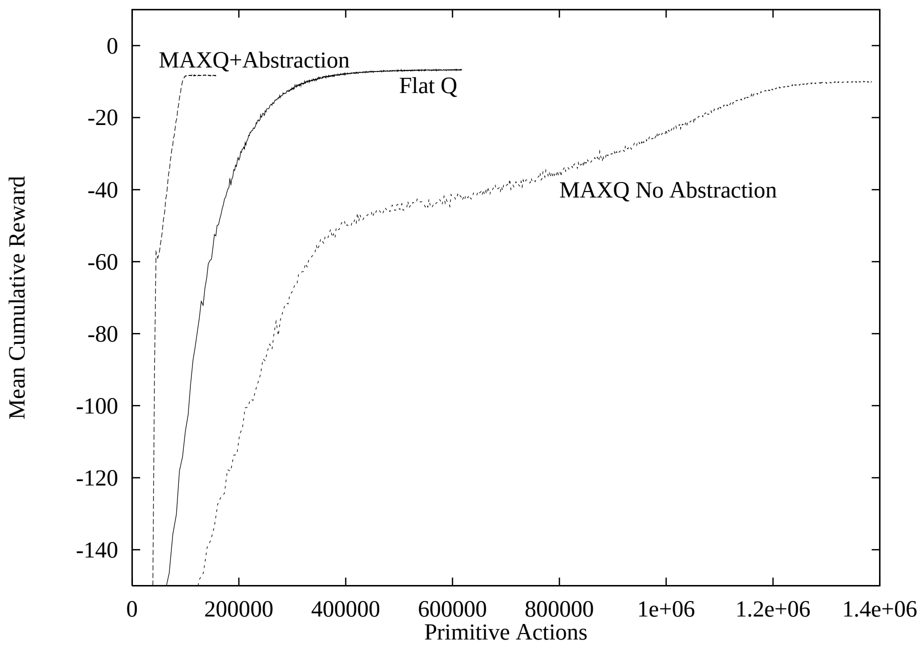

Fig. 8. Comparison of Flat Q learning, MAXQ Q learning with no state abstraction, and MAXQ Q learning with state abstraction on a noise-free version of Kaelbling's navigation task.

6 Design Tradeoffs in Hierarchical Reinforcement Learning

In both HSMQ learning and MAXQQ learning, we have focused on learning a recursively optimal policy. However, a recursively optimal policy can be very far from being optimal. Other authors, notably Parr and Russell (1998) and Dean and Lin (1995), have developed algorithms for learning *hierarchically optimal* policies—that is, policies that are the best possible given the constraints of an imposed hierarchy. In such policies, it is often the case that the policy for a subroutine is *not* optimal given the policies of its children. Consequently, hierarchically optimal policies are not necessarily recursively optimal, and vice versa.

In order to learn a hierarchically optimal policy, it is essential that information from "outside" of a subtask be able to propagate "into" the subtask. Consider the simple two-room maze problem shown in Figure 9. Suppose that there are two defined subtasks: exit from the room on the left (which terminates when the agent leaves the room by either door), and go to the goal in the room on the right. The recursively optimal policy for the left room is to leave by the nearest door. But this is not the hierarchically optimal policy for the shaded squares. For these squares, it is better to move upward and exit by the upper door. To discover this hierarchically optimal policy, information about the distance to the goal after the agent leaves the room must be propagated "into" the room (as it would be in flat Q learning, for example).

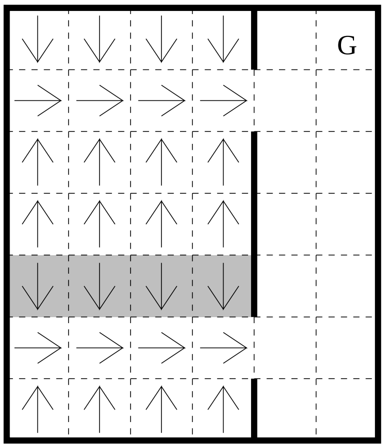

Fig. 9. A simple two-room maze problem. An agent starting in the left room must reach the goal G in the right room in the fewest number of steps. Arrows show a policy for exiting the left room in the fewest number of steps. The hierarchically optimal policy goes north in the shaded region instead of south.

It would be easy to modify MAXQQ learning to permit this. The basic idea would be to treat the completion function $Q(p, s, a)$ as a "terminal state" reward for action a, which would then influence the value function inside subtask a.

Although this would permit MAXQQ learning to discover a hierarchically optimal policy, it would also destroy many opportunities for state abstraction. Consider what happens if the goal location (in the right room) is a state variable that can change. Then the choice of hierarchically optimal policy inside the left room will depend on the location of the goal in the right room. If the goal moves to the bottom of the right room, then the optimal policy for the shaded squares is to move down. This means that the value function for the left room subtask depends on an additional state variable—the current location of the goal. However, if we are willing to settle for recursive optimality, we can abstract away this state variable.

From this argument, we can see that there is a tradeoff between state abstraction and quality of the resulting policy. Recursive optimality permits more state abstraction and more re-use of subtasks than does hierarchical optimality. But hierarchical optimality is generally better than recursive optimality.

There is another design question that we will discuss briefly. The learning algorithms presented in this paper are all model-free Q learning algorithms. Q learning is one of the most widely-applicable algorithms in reinforcement learning, but it is also one of the least efficient, particularly when measured in terms of the number of actions that the agent must execute in the environment. Model-based algorithms (that is, algorithms that try to learn $P(s'|s,a)$ and $R(s'|s,a)$) are generally much more efficient, because they remember past experience rather than having to re-experience it.

One of the best model-based algorithms is Prioritized Sweeping (Moore & Atkeson, 1993). Prioritized sweeping maintains estimates of $P(s'|s,a)$ and $R(s'|s,a)$—typically by remembering all $\langle s,a,r,s'\rangle$ tuples of experience. After each action in the real world, prioritized sweeping performs a fixed number of dynamic programming steps to update the Q values of those state-action pairs whose Q values are believed to be most inaccurate. Experimental tests show that prioritized sweeping can be dramatically more efficient than Q learning.

We have implemented a prioritized sweeping algorithm for the MAXQ hierarchy. The algorithm learns a model for each subtask within the MAXQ task hierarchy. Unfortunately, the model for subtask p must remember the values of all state variables that are relevant to subtask p or to any of its descendants. This is necessary, because in order to perform dynamic programming steps, prioritized sweeping needs to evaluate $V(p,s)$, the value function for subtask p. This in turn requires that it evaluate $C(p,s,a)$ for all children a of p (and their children, recursively). As a consequence, while the completion function for subtask p may only depend on some set of state variables X, the learned probability transition model $P(s'|s,a)$ for subtask p must depend on all of the state variables relevant to p or any descendant of p. This significantly reduces the effectiveness of state abstractions, at least for purposes of learning transition models.

One conclusion to be drawn from this discussion is that Q learning and the MAXQ hierarchy are well-suited to one another. Because Q learning only needs samples of the probability transitions and rewards, it works well with the MAXQ hierarchy, which only needs to represent the completion function of each subtask, rather than the full value function of the subtask.

7 Conclusions

This paper has attempted to provide an overview of the MAXQ value function decomposition including its representational capabilities, learning algorithms, support for state abstractions, and design tradeoffs. The experiments show that hierarchical reinforcement learning can be much faster (and more compact) than flat reinforcement learning. Recursively optimal policies can be decomposed into recursively optimal policies for individual subtasks, and these subtask policies can be re-used whereever the same subtask arises.

We have omitted two important topics in this paper. The first is the question of whether programmers will be able to design good MAXQ task hierarchies. Elsewhere (Dietterich, 2000), we have shown how the MAXQ formalism can be extended to permit the programmer to specify separate "pseudo-reward functions" for each subtask. This permits the programmer to express such things as "leaving the room by the top door is better than leaving by the bottom door". Even with this additional expressive power, we have found that the proper definition of the termination predicates for each subtask can be difficult and often requires observing the behavior of MAXQ Q learning and debugging the termination predicates to improve that behavior.

The second topic concerns how to recover from the suboptimal performance resulting from the task hierarchy. Neither recursively optimal nor hierarchically optimal policies are necessarily close to globally optimal. Fortunately, several methods have been developed for reducing the degree of suboptimality. The most interesting of these involves using the hierarchical value function to construct a non-hierarchical policy that is provably better than the hierarchical policy. See Dietterich (2000), Kaelbling (1993), and Sutton, Precup, and Singh (1998) for more details.

As we stated in the introduction, the goal of hierarchical reinforcement learning is to discover and exploit hierarchical structure within complex Markov decision problems. This paper has focused on exploiting programmer-specified hierarchical structure. The biggest open problem in hierarchical reinforcement learning is to discover hierarchical structure. One definition of a good hierarchy is that it would permit the three forms of state abstraction that we have discussed in this paper. We hope that the formalization of these abstractions (and others yet to be identified) will help guide the search for good abstractions.

Bibliography

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

Crites, R. H., & Barto, A. G. (1995). Improving elevator performance using reinforcement learning. In *Advances in Neural Information Processing Systems*, Vol. 8, pp. 1017–1023 San Francisco, CA. Morgan Kaufmann.

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. (2000). Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*. To appear.

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.

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

Parr, R. (1998). *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.

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.

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*.

Tesauro, G. (1995). Temporal difference learning and TD-Gammon. *Communications of the ACM*, *28*(3), 58–68.

Zhang, W., & Dietterich, T. G. (1995). A reinforcement learning approach to job-shop scheduling. In *1995 International Joint Conference on Artificial Intelligence*, pp. 1114–1120. Morgan Kaufmann, San Francisco, CA.