Recent Advances in Hierarchical Reinforcement Learning

Andrew G. Barto
Sridhar Mahadevan

Autonomous Learning Laboratory
Department of Computer Science
University of Massachusetts, Amherst MA 01003

Abstract

Reinforcement learning is bedeviled by the curse of dimensionality: the number of parameters to be learned grows exponentially with the size of any compact encoding of a state. Recent attempts to combat the curse of dimensionality have turned to principled ways of exploiting temporal abstraction, where decisions are not required at each step, but rather invoke the execution of temporally-extended activities which follow their own policies until termination. This leads naturally to hierarchical control architectures and associated learning algorithms. We review several approaches to temporal abstraction and hierarchical organization that machine learning researchers have recently developed. Common to these approaches is a reliance on the theory of semi-Markov decision processes, which we emphasize in our review. We then discuss extensions of these ideas to concurrent activities, multiagent coordination, and hierarchical memory for addressing partial observability. Concluding remarks address open challenges facing the further development of reinforcement learning in a hierarchical setting.

1 Introduction

Reinforcement learning (RL) [5, 72] is an active area of machine learning research that is also receiving attention from the fields of decision theory, operations research, and control engineering. RL algorithms address the problem of how a behaving agent can learn to approximate an optimal behavioral strategy while interacting directly with its environment. In control terms, this involves approximating solutions to stochastic optimal control problems, usually under conditions of incomplete knowledge of the system being controlled. Most RL algorithms adapt standard methods of stochastic dynamic programming (DP) so that they can be used on-line for problems with large state spaces. By focusing computational effort along behavioral trajectories and by using function approximation methods for accumulating value function information, RL algorithms have produced good results on problems that pose significant challenges for standard methods (e.g., refs. [11, 75]). However, current RL methods by no means completely circumvent the curse of dimensionality: the exponential growth of the number of parameters to be learned with the size of any compact encoding of system state. Recent attempts to combat the curse of dimensionality have turned to principled ways of exploiting temporal abstraction, where decisions are not required at each step, but rather invoke the execution of temporally-extended activities which follow their own policies until termination. This leads naturally to hierarchical control architectures and learning algorithms.

In this article we review several related approaches to temporal abstraction and hierarchical control that have been developed by machine learning researchers, and we then discuss several extensions to these ideas and the open challenges facing the further development of RL in a hierarchical setting. Despite the fact that research in this area began only recently within the machine learning community, we cannot provide a survey of all the related literature, which is already quite extensive. We do attempt, however, to provide enough information so that interested readers can probe the relevant topics more deeply. Another important goal that we do not attempt is to thoroughly relate machine learning research on these topics to the extensive literature in systems and control engineering on hierarchical and multilayer control, hybrid systems, and other closely related topics. The most obvious parallels with some of these approaches have not escaped us, but a careful rapprochement is beyond the scope of this article. We largely adhere to notation and terminology typical of that used in the machine learning community. There is much to be gained from integrating perspectives from these different groups of resesarchers, and we hope that this article will contribute to the required dialog.

After brief introductions to Markov and semi-Markov decision processes, we introduce the basic ideas of RL, and then we review three approaches to hierarchical RL: the *options* formalism of Sutton, Precup, and Singh [73], the *hierarchies of abstract machines* (HAMs) approach of Parr and Russell [48, 49], and the MAXQ framework of Dietterich [14]. Although these approaches were developed relatively independently, they have many elements in common. In particular, they all rely on the theory of semi-Markov decision processes to provide a formal basis. Although we take advantage of these commonalities in our exposition, a thorough integration of these approaches awaits a future paper. We also cannot do full justice to these approaches, although we do attempt to provide enough detail to make the approaches concrete; the reader is directed to the original papers for detailed treatments. In the final sections, we briefly present extensions of these ideas that focus on work conducted in our laboratory on concurrent activities, multiagent coordination, and hierarchical memory for addressing partial observability. Concluding remarks address open challenges facing the further development of reinforcement learning in a hierarchical setting.

2 Markov and Semi-Markov Decision Processes

Most RL research is based on the formalism of Markov decision processes (MDPs). Although RL is by no means restricted to MDPs, this discrete-time, countable (in fact, usually finite) state and action formalism provides the simplest framework in which to study basic algorithms and their properties. Here we briefly describe this well-known framework, with a few twists characteristic of how it is used in RL research; additional details can be found in many references (e.g., refs. [4, 5, 55, 58, 72]). A finite MDP models the following type of problem. At each stage in a sequence of stages, an agent (the controller) observes a system's state s, contained in a finite set $\mathcal{S}$, and executes an action (control) a selected from a finite set, $\mathcal{A}_s$, of admissible actions. The agent receives an immediate reward having expected value $R(s, a)$, and the state at the next stage is s' with probability $P(s'|s, a)$. The expected immediate rewards, $R(s, a)$, and the state transition probabilities, $P(s'|s, a)$, $s, s' \in \mathcal{S}$, together comprise what RL researchers often call the *one-step* *model* of action a.

A (stationary, stochastic) policy $\pi : \mathcal{S} \times \cup_{s \in \mathcal{S}} \mathcal{A}_s \rightarrow [0, 1]$, with $\pi(s, a) = 0$ for $a \notin \mathcal{A}_s$, specifies that the agent executes action $a \in \mathcal{A}_s$ with probability $\pi(s, a)$ whenever it observes state s. For any policy π and $s \in \mathcal{S}$, $V^\pi(s)$ denotes the *expected infinite-horizon discounted return* from s given that the agent uses policy π. Letting s_t and r_{t+1} denote the state at stage t and the immediate reward for acting at stage t,[1] this is defined as:

$$V^\pi(s) = E\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots | s_t = s, \pi\},$$

where γ, $0 \leq \gamma < 1$, is a discount factor. $V^\pi(s)$ is the *value* of s under π, and is V^π the *value function* corresponding to π.

This is a finite, infinite-horizon, discounted MDP. The objective is to find an optimal policy, i.e., a policy, π^*, that maximizes the value of each state. The unique *optimal value function*, V^*, is the value function corresponding to any of the optimal policies. Most RL research addresses discounted MDPs since they comprise the simplest class of MDPs, and here we restrict attention to discounted problems. However, RL algorithms have also been developed for MDPs with other definitions of return, such as average reward MDPs [39, 62].

Playing important roles in many RL algortihms are *action-value functions*, which assign values to admissible state-action pairs. Given a policy π, the value of (s, a), $a \in \mathcal{A}_s$, denoted $Q^\pi(s, a)$, is the expected infinite-horizon discounted return for executing a in state s and thereafter following π:

$$Q^\pi(s, a) = E\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots | s_t = s, a_t = a, \pi\}. \tag{1}$$

The *optimal action-value function*, Q^*, assigns to each admissible state-action pair (s, a) the expected infinite-horizon discounted return for executing a in s and thereafter following an optimal policy. Action-value functions for other definitions of return are defined analogously.

Dynamic Programming (DP) algorithms exploit the fact that value functions satisfy various *Bellman equations*, such as:

$$V^\pi(s) = \sum_{a \in \mathcal{A}_s} \pi(s, a)[R(s, a) + \gamma \sum_{s'} P(s'|s, a) V^\pi(s')],$$

and

$$V^*(s) = \max_{a \in \mathcal{A}_s} [R(s, a) + \gamma \sum_{s'} P(s'|s, a) V^*(s')], \tag{2}$$

for all $s \in \mathcal{S}$. Analogous equations exist for Q^π and Q^*. For example, the Bellman equation for Q^* is:

$$Q^*(s, a) = R(s, a) + \gamma \sum_{s'} P(s'|s, a) \max_{a' \in \mathcal{A}_{s'}} Q^*(s', a'), \tag{3}$$

for all $s \in \mathcal{S}$, $a \in \mathcal{A}_s$.

As an example DP algorithm, consider value iteration, which successively approximates V^* as follows. At each iteration k, it updates an approximation V_k of V^* by applying the following operation for each state s:

$$V_{k+1}(s) = \max_{a \in \mathcal{A}_s} [R(s, a) + \gamma \sum_{s'} P(s'|s, a) V_k(s')]. \tag{4}$$

We call this operation a *backup* because it updates a state's value by transferring to it information about the approximate values of its possible successor states. Applying this backup operation once to each state is often called a *sweep*. Starting with an arbitrary initial function V_0, the sequence $\{V_k\}$ produced by repeated sweeps converges to V^*. A similar algorithm exists for successively approximating Q^* using the following backup:

$$Q_{k+1}(s, a) = R(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) \max_{a' \in \mathcal{A}_{s'}} Q_k(s', a'). \tag{5}$$

Given V^*, an optimal policy is any policy that for each s assigns non-zero probability only to those actions that realize the maximum on the right-hand side of (4). Similarly, given Q^*, an optimal policy is

[1]We follow Sutton and Barto [72] in denoting the reward for the action at stage t by r_{t+1} instead of the more usual r_t.

any policy that for each s assigns non-zero probability only to the actions that maximize $Q^*(s,\cdot)$. These maximizing actions are often called *greedy* actions, and a policy defined in this way is a (stochastic) *greedy policy.* Given sufficiently close approximations of V^* and Q^* obtained by value iteration, then, optimal policies are taken to be the corresponding greedy policies. Note that finding greedy actions via Q^* does not require access to the one-step action models (the $R(s,a)$ and $P(s'|s,a)$) as it does when only V^* is available, where the right-hand side of (4) has to be evaluated. This is one of the reasons that action-value functions play a significant role in RL.

In an MDP, only the sequential nature of the decision process is relevant, not the amount of time that passes between decision stages. A generalization of this is the semi-Markov decision process (SMDP) in which the amount of time between one decision and the next is a random variable, either real- or integer-valued. In the real-valued case, SMDPs model continuous-time discrete-event systems (e.g., refs. [40, 55]). In a *discrete-time SMDP* [26] decisions can be made only at (positive) integer multiples of an underlying time step. In either case, it is usual to treat the system as remaining in each state for a random waiting time [26], at the end of which an instantaneous transition occurs to the next state. Due to its relative simplicity, the discrete-time SMDP formulation underlies most approaches to hierarchical RL, but there are no significant obstacles to extending these approaches to the continuous-time case.

Let the random variable τ denote the (positive) waiting time for state s when action a is executed. The transition probabilities generalize to give the joint probability that a transition from state s to state s' occurs after τ time steps when action a is executed. We write this joint probability as $P(s',\tau|s,a)$. The expected immediate rewards, $R(s,a)$, (which must be bounded) now give the amount of discounted reward expected to accumulate over the waiting time in s given action a. The Bellman equations for V^* and Q^* become

$$V^*(s) = \max_{a\in\mathcal{A}_s}[R(s,a) + \sum_{s',\tau} \gamma^\tau P(s',\tau|s,a)V^*(s')], \tag{6}$$

for all $s \in \mathcal{S}$; and

$$Q^*(s,a) = R(s,a) + \sum_{s',\tau} \gamma^\tau P(s',\tau|s,a) \max_{a'\in\mathcal{A}_{s'}} Q^*(s',a'), \tag{7}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}_s$. DP algorithms correspondingly extend to SMDPs (e.g., refs. [26, 55]).

3 Reinforcement Learning

DP algorithms have complexity polynomial in the number of states, but they still require prohibitive amounts of computation for large state sets, such as those that result from discretizing multi-dimensional continuous spaces or representing state sets consisting of all possible configurations of a finite set of structural elements (e.g., possible configurations of a backgammon board [75]). Many methods have been proposed for approximating MDP solutions with less effort than required by conventional DP, but RL methods are novel in their use of Monte Carlo, stochastic approximation, and function approximation techniques. Specifically, RL algorithms combine some, or all, of the following features:

1. Avoid the exhaustive sweeps of DP by restricting computation to states on, or in the neighborhood of, multiple sample trajectories, either real or simulated. Because computation is guided along sample trajectories, this approach can exploit situations in which many states have low probabilities of occurring in actual experience.

2. Simplify the basic DP backup by sampling. Instead of generating and evaluating all of a state's possible immediate successors, estimate a backup's effect by sampling from the appropriate distribution.

3. Represent value functions and/or policies more compactly than lookup-table representations by using function approximation methods, such as linear combinations of basis functions, neural networks, or other methods.

Features 1 and 2 reflect the nature of the approximations usually sought when RL is used. Instead of policies that are close to optimal uniformly over the entire state space, RL methods arrive at non-uniform approximations that reflect the behavior of the agent. The agent's policy does not need high precision in states that are rarely visited. Feature 3 is the least understood aspect of RL, but results exist for the linear case (notably ref. [81]) and numerous examples illustrate how function approximation schemes that are nonlinear in the adjustable parameters (e.g., multilayer nerual networks) can be effective for difficult problems (e.g., refs. [11, 40, 64, 75]).

Of the many RL algorithms, perhaps the most widely used are Q-learning [82, 83] and Sarsa [59, 70]. Q-learning is based on the DP backup (5) but with the expected immediate reward and the expected maximum action-value of the successor state on the right-hand side of (5) respectively replaced by a sample reward and the maximum action-value for a sample next state. The most common way to obtain these samples is to generate sample trajectories by simulation or by observing the actual decision process over time. Suppose the agent observes a current state s, executes action a, receives immediate reward r, and then observes a next state s'. The Q-learning algorithm updates the current estimate, $Q_k(s,a)$, of $Q^*(s,a)$ using the following update:

$$Q_{k+1}(s,a) = (1-\alpha_k)Q_k(s,a) + \alpha_k[r + \gamma \max_{a' \in \mathcal{A}_{s'}} Q_k(s',a')], \tag{8}$$

where α_k is a time-varying learning-rate parameter. The values of all the other state-action pairs remain unchanged at this update. If in the limit the action-values of all admissible state-action pairs are updated infinitely often, and α_k decays with increasing k while obeying the usual stochastic approximation conditions, then $\{Q_k\}$ converges to Q^* with probability 1 [29, 5]. As long as these conditions are satisfied, the policy followed by the agent during learning is irrelevant. Of course, when Q-learning is being used, the agent's policy does matter since one is usually interested in the agent's performance throughout the learning process, not just asymptotically. It is usual practice to let the agent select actions using a policy that is greedy with respect to the current estimate of Q^*, while also introducing non-greedy "exploratory actions" in an attempt to widely sample state-action pairs.

Sarsa is similar to Q-learning except that the maximum action-value for the next state on the right-hand side of (8) is replaced by the action-value of the actual next state-action pair:

$$Q_{k+1}(s,a) = (1-\alpha_k)Q_k(s,a) + \alpha_k[r + \gamma Q_k(s',a')], \tag{9}$$

where a' is the action executed in state s'. (Sutton [70] called this algorithm Sarsa due to its dependence on s, a, r, s', and a'. Eq. (9) is actually a special case called Sarsa(0).) Unlike Q-learning, here the agent's policy does matter. Singh et al. [65] show that if the policy has the property that each action is executed infinitely often in every state that is visited infinitely often, and it is greedy with respect to the current action-value function in the limit, which Singh et al. [65] call a GLIE (*Greedy in the Limit with Infinite Exploration*) policy, then with appropriately decaying α_k, the sequence $\{Q_k\}$ generated by Sarsa converges to Q^* with probability 1.

Both the Q-learning and Sarsa learning algorithms also apply to SMDPs, both continuous- and discrete-time, if one interprets the immediate reward, r, as the return accumulated during the waiting time in state s and appropriately adjusts the discounting to reflect the waiting time. For example, in the discrete-time case, if a is executed in state s at time step t and the transition to s' follows after τ time steps, then

$$Q_{k+1}(s,a) = (1-\alpha_k)Q_k(s,a) + \alpha_k[r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{\tau-1} r_{t+\tau} + \gamma^{\tau} \max_{a' \in \mathcal{A}_{s'}} Q_k(s',a')], \tag{10}$$

where r_{t+i} is the immediate reward received at time step $t+i$. The return accumulated during the waiting time must be bounded, and it can be computed recursively during the waiting time. Bradtke and Duff [7] showed how to do this for continuous-time SMDPs, Parr [48] proved that it converges under essentially the same conditions required for Q-learning convergence, and Das et al. [12] developed the average reward case.

Crites [10, 11] used SMDP Q-learning in a continuous-time discrete-event formulation of an elevator dispatching problem, an application that illustrates two useful features of RL methods for discrete-event systems. First, Q-learning and Sarsa do not require explicit knowledge of the expected immediate rewards or the state-transition probilities. Instead, they use samples from the respective distributions, which can come from a stochastic simulation or from the real world itself. This can be a significant advantage since in many problems it is often much easier to produce a simulation than it is to make the expected rewards and transition probabilities explicit. In Crites' elevator dispatcher, for example, SMDP Q-learning was applied along trajectories generated by a simulation of a 10-story, 4-elevator system. An explicit SMDP model of this system would have been difficult to make explicit. This illustrates advantages of so-called *model-free* RL algorithms such as Q-learning and Sarsa, meaning that they do not need access to explicit representations of the expected immediate reward function and the state-transition probabilities. Importantly, however, RL is not restricted to such model-free methods [72, 5].

RL algorithms that estimate action-values, such as Q-learning and Sarsa, have a second advantage when applied to discrete-event systems. As mentioned in Section 2, finding optimal actions via Q^* does not require access to the one-step action models (the $R(s, a)$ and $P(s'|s, a)$) as it does when only V^* is available. That is, a one-step ahead search is not needed to determine optimal actions. In many problems involving discrete-event systems, such as the elevator dispatching problem, it is not clear how to conduct a one-step ahead search since the next event can occur at any of an infinite number of times in the future. The use of action-values eliminates this difficulty.

Finally, we point out that in our brief presentation of RL algorithms we assumed that it was possible to explicitly store values for every state, or action-values for every state-action pair. This is obviously not feasible for large-scale problems, and extensions of these algorithms need to be considered that adjust the parameters of parametric representations of value functions. It is relatively easy to produce such extensions, although, as we mentioned above, the theory of their behavior still contains many open questions which are beyond the scope of this article.

The view of DP-based RL just outlined, which has been assembled by many researchers over roughly the last ten years, represents our current state of understanding rather than the intuition underlying the origination of these methods. Indeed, DP-based learning originated at least as far back as Samuel's famous checkers player of the 1950s [61, 60], which, however, made no reference to the DP literature existing at that time. Other early RL research was explicitly motivated by animal behavior and its neural basis [45, 33, 34, 71]. Much of the current interest is attributable to Werbos [85, 86, 87], Watkins [82], and Tesauro's backgammon-playing system TD-Gammon [75, 76]. Additional information about RL can be found in several references (e.g., [2, 5, 32, 72]). Despite the utility of RL methods in many applications, the amount of time they can take to form acceptable approximate solutions can still be unacceptable. As a result, RL researchers are investigating various means for introducing abstraction and hierarchical structure into RL algorithms. In the following sections we review several of the proposed approaches.

4 Approaches to Hierarchical Reinforcement Learning

Artificial intelligence researchers have addressed the need for large-scale planning and problem solving by introducing various forms of abstraction into problem solving and planning systems, e.g., refs. [18, 37]. Abstraction allows a system to ignore details that are irrelevant for the task at hand. One of the simplest types of abstraction is the idea of a "macro-operator," or just a "macro," which is a sequence of operators or actions that can be invoked by name as if it were a primitive operator or action. Macros form the basis of hierarchical specifications of operator or action sequences because macros can include other macros in their definitions: a macro can "call" other macros. Also familiar is the idea of a subroutine that can call other subroutines as well as execute primitive commands. Most of the current research on hierarchical RL focuses on action hierarchies that follow roughly the same semantics as hierarchies of macros or subroutines.

From a control perpsective, a macro is an open-loop control policy and, as such, is inappropriate for most interesting control purposes, especially the control of a stochastic system. Hierarchical approaches to RL generalize the macro idea to closed-loop policies, or more precisely, closed-loop *partial* policies because they are generally defined for a subset of the state set. The partial policies must also have well-defined termination conditions. These partial policies are sometimes called *temporally-extended actions*, *options* [73], *skills* [80], *behaviors* [9, 27], or the more control-theoretic *modes* [22]. When not discussing a specific formalism, we will use the term *activity*, as suggested by Harel [23].

For MDPs, this extension adds to the sets of admissible actions, $\mathcal{A}_s$, $s \in \mathcal{S}$, sets of activities, each of which can itself invoke other activities, thus allowing a hierarchical specification of an overall policy. The original one-step actions, now called the "primitive actions," may or may not remain admissible. Extensions along these general lines result in decision processes modeled as SMDPs, where the waiting time in a state now corresponds to the duration of the selected activity. If τ is the waiting time in state s upon execution of activity a, then a takes τ steps to complete when initiated in s, where the distribution of the random variable τ now depends on the policies and termination conditions of all of the lower-level activities that comprise a.

To the best of our knowledge (and as pointed out by Parr [48]), the approach most closely related to this in the control literature is that of Forestier and Varaiya [20], which we discuss briefly in Section 4.2.

4.1 Options

Sutton, Precup, and Singh [73] formalize this approach to including activities in RL with their notion of an *option.* Starting from a finite MDP, which we call the *core* MDP, the simplest kind of option consists of a (stationary, stochastic) policy $\pi : \mathcal{S} \times \cup_{s \in \mathcal{S}} A_s \rightarrow [0,1]$, a termination condition $\beta : \mathcal{S} \rightarrow [0,1]$, and an input set $\mathcal{I} \subseteq \mathcal{S}$. The option $\langle \mathcal{I}, \pi, \beta \rangle$ is available in state s if and only if $s \in \mathcal{I}$. If the option is executed, then actions are selected according to π until the option terminates stochastically according to β. For example, if the current state is s, the next action is a with probability $\pi(s,a)$, the environment makes a transition to state s', where the option either terminates with probability $\beta(s')$ or else continues, determining the next action a' with probability $\pi(s',a')$, and so on. When the option terminates, the agent can select another option.

It is usual to assume that for any state in which an option can continue, it can also be initiated, that is, $\{s : \beta(s) < 1\} \subseteq \mathcal{I}$. This implies that an option's policy only needs to be defined over its input set $\mathcal{I}$. Note that any action of the core MDP, a *primitive action* $a \in \cup_{s \in \mathcal{S}} \mathcal{A}_s$, is also an option, called a *one-step option*, with $\mathcal{I} = \{s : a \in \mathcal{A}_s\}$ and $\beta(s) = 1$ for all $s \in \mathcal{S}$. Sutton et al. [73] give the example of an option named `open-the-door` for a hypothetical robot control system. This option consists of a policy for reaching, grasping and turning the door knob, a termination condition for recognizing that the door has been opened, and an input set restricting execution of `open-the-door` to states in which a door is within reach.

An option of the type just defined is called a *Markov option* because its policy is Markov, that is, it sets action probabilities based solely on the current state of the core MDP. To allow more flexibility, especially with respect to hierarchical architectures, one must include *semi-Markov options* whose policies can set action probabilities based on the entire history of states, actions, and rewards since the option was initiated [73]. Semi-Markov options include options that terminate after a pre-specified number of time steps, and most importantly, they are needed when policies over options are considered, i.e., policies $\mu : \mathcal{S} \times \cup_{s \in \mathcal{S}} \mathcal{O}_s \rightarrow [0,1]$, where $\mathcal{O}_s$ is the set of admissible options for state s (which can include all the one-step options corresponding to the admissible primitive actions in $\mathcal{A}_s$).

A policy μ over options selects option o in state s with probability $\mu(s,o)$; o's policy in turn selects other options until o terminates. The policy of each of these selected options selects other options, and so on. Expanding each option down to primitive actions, we see that any policy over options, μ, determines a conventional policy of the core MDP, which Sutton et al. [73] call the *flat* (i.e, non-hierarchical) policy corresponding to μ, denoted $flat(\mu)$. Flat policies corresponding to policies over options are generally not Markov even if all the options are Markov. The probability of a primitive action at any time step depends on the current core state plus the policies of all the options currently involved in the hierarchical specification. This dependence is made more explicit in the work of Parr [48] and Dietterich [14], which we discuss below. Using this machinery (made precise by Precup [52]), one can define *hierarchical options* as triples $\langle \mathcal{I}, \mu, \beta \rangle$, where $\mathcal{I}$ and β are the same as for Markov options but μ is a semi-Markov policy over options.

Value functions for option policies can be defined in terms of value functions of semi-Markov flat policies. For a semi-Markov flat policy π:

$$V^{\pi}(s) = E\{r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{\tau-1} r_{t+\tau} + \cdots | \mathcal{E}(\pi, s, t)\},$$

where $\mathcal{E}(\pi, s, t)$ is the event of π being initiated at time t in s. Note that this value can depend on the complete history from t onwards, but not on events earlier than t since π is semi-Markov. Given this definition for flat policies, $V^{\mu}(s)$, the value of s for a policy μ over options, is defined to be $V^{flat(\mu)}(s)$. Similarly, one can define the option-value function for μ as follows:

$$Q^{\mu}(s, o) = E\{r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{\tau-1} r_{t+\tau} + \cdots | \mathcal{E}(o\mu, s, t)\},$$

where $o\mu$ is the semi-Markov policy that follows o until it terminates after τ time steps and then continues according to μ.

Adding any set of semi-Markov options to a core finite MDP yields a well-defined discrete-time SMDP whose actions are the options and whose rewards are the returns delivered over the course of an option's execution. Since the policy of each option is semi-Markov, the distributions defining the next state (the state at an option's termination), waiting time, and rewards depend only on the option executed and the state in which its execution was initiated. Thus, all of the theory and algorithms applicable to SMDPs can be appropriated for decision making with options.

In their effort to treat options as much as possible as if they were conventional single-step actions, Sutton et al. [73] introduced the interesting concept of a *multi-time model* of an option that generalizes the single-step model consisting of $R(s,a)$ and $P(s'|s,a)$, $s, s' \in \mathcal{S}$, of a conventional action a. For any option o, let $\mathcal{E}(o,s,t)$ denote the event of o being initiated in state s at time t. Then the reward part of the multi-time model of o for any $s \in S$ is:

$$R(s,o) = E\{r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{\tau-1} r_{t+\tau} | \mathcal{E}(o,s,t)\},$$

where $t+\tau$ is the random time at which o terminates. The state-prediction part of the model of o for s is:

$$P(s'|s,o) = \sum_{\tau=1}^{\infty} p(s',\tau)\gamma^{\tau},$$

for all $s \in \mathcal{S}$, where $p(s',\tau)$ is the probability that the o terminates in s' after τ steps when initiated in state s. Though not itself a probability, $P(s'|s,o)$ is a combination of the probability that s' is the state in which o terminates together with a measure of how delayed that outcome is in terms of γ.

The quantities $R(s,o)$ and $P(s'|s,o)$ respectively generalize the reward and transition probabilities, $R(s,a)$ and $P(s'|s,a)$, of the usual MDP in such a way that one can write a generalized form of the Bellman optimality equation. If $V^*_{\mathcal{O}}$ denotes the optimal value function over an option set $\mathcal{O}$, then

$$V^*_{\mathcal{O}}(s) = \max_{o \in \mathcal{O}_s} [R(s,o) + \sum_{s'} P(s'|s,o) V^*_{\mathcal{O}}(s')], \tag{11}$$

which reduces to the usual Bellman optimality equation (2) if all the options are one-step options ($\beta(s) = 1$, $s \in \mathcal{S}$). A Bellman equation analogous to (3) can be written for $Q^*_{\mathcal{O}}$:

$$Q^*_{\mathcal{O}}(s,o) = R(s,o) + \sum_{s'} P(s'|s,o) \max_{o' \in \mathcal{O}_{s'}} Q^*_{\mathcal{O}}(s',o'), \tag{12}$$

for all $s \in \mathcal{S}$ and $o \in \mathcal{O}_s$.

The system of equations (11) and (12) can be solved respectively for $V^*_{\mathcal{O}}$ and $Q^*_{\mathcal{O}}$, exactly or approximately, using methods that generalize the usual DP and RL algorithms [54]. For example, the DP backup analogous to (5) for computing option-values is:

$$Q_{k+1}(s,o) = R(s,o) + \sum_{s' \in \mathcal{S}} P(s'|s,o) \max_{o' \in \mathcal{O}_{s'}} Q_k(s',o'),$$

and the corresponding Q-learning update analogous to (8) is:

$$Q_{k+1}(s,o) = (1-\alpha_k) Q_k(s,o) + \alpha_k [r + \gamma^{\tau} \max_{o' \in \mathcal{O}_{s'}} Q_k(s',o')]. \tag{13}$$

This update is applied upon the termination of o at state s' after executing for τ time steps, and r is the return accumulated during o's execution. This is the specialization of SMDP Q-learning given by (10) to the SMDP that results from adding options to an MDP. It reduces to conventional Q-learning when all the options are one-step options. In addition to refs. [52, 73], see refs. [54, 53, 48] for discussion of this formulation and its relation to conventional SMDP theory.

As in the case of conventional MDPs, given $V^*_{\mathcal{O}}$ or $Q^*_{\mathcal{O}}$, optimal policies over options can be determined as (stochastic) greedy policies. If each set of admissible options, $\mathcal{O}_s$, $s \in \mathcal{S}$, contains one-step options corresponding to all of the primitive actions $a \in \mathcal{A}_s$ of the core MDP, then it is clear that the optimal policies over options are the same as the optimal policies for the core MDP (since primitive actions give the most refined degree of control). On the other hand, if some of the primitive actions are not available as one-step options, then optimal policies over the set of available options are in general suboptimal policies of the core MDP.

The theory described above all falls within conventional SMDP theory, which is specialized by the focus on semi-Markov options. A shortcoming of this is that the internal structure of an option is not readily exploited. Precup [52] writes:

> SMDP methods apply to options, but only when they are treated as opaque indivisible units. Once an option has been selected, such methods require that its policy be followed until the option terminates. More interesting and potentially more powerful methods are possible by looking inside options and by altering their internal structure. —Precup [52] p. 58.

For example, the Q-learning update (13) for options does nothing until an option terminates (and so does not apply to non-terminating options), and it only applies to one option at a time. This is the motivation for *intra-option learning methods* which allow learning useful information before an option terminates and can be used for multiple options simultaneously. For example, a variant of Q-learning, called *one-step intra-option Q-learning* [52], works as follows. Suppose (primitive) action a_t is taken in s_t, and the next state and immediate reward are respectively s_{t+1} and r_{t+1}. Then for every Markov option $o = \langle \mathcal{I}, \pi, \beta \rangle$ whose policy could have selected a_t according to the same distribution $\pi(s_t, \cdot)$, this update is applied:

$$Q_{k+1}(s_t, o) = (1 - \alpha_k)Q_k(s_t, o) + \alpha_k[r_{t+1} + \gamma U_k(s_t, o)], \tag{14}$$

where

$$U_k(s, o) = (1 - \beta(s))Q_k(s, o) + \beta(s) \max_{o' \in \mathcal{O}} Q_k(s, o'),$$

which is an estimate of the value of state-option pair (s, o) upon arrival in state s. In the case of deterministic option policies, for example, this update is applied to all options whose policies select a_t in s_t *whether that option was executing or not.* If all the options in $\mathcal{O}$ are deterministic and Markov, then for every option in $\mathcal{O}$ this converges to $Q^*_{\mathcal{O}}$ with probability 1, provided the α_k decay appropriately and that in the limit every primitive action is executed infinitely often in every state [52].

Although its utility for hierarchical RL is limited due to its restriction to Markov options, one-step intra-option Q-learning is one example of a class of methods that take advantage of the structure of the core MDP. Related methods have been developed for estimating multi-time models of many options simultaneously by exploiting Bellman-like equations relating the components $R(s, a)$ and $P(s'|s, a)$ of multi-time models for successive states. Results also exist on interrupting an option's execution in favor of an option more highly valued for the current state, and for adjusting an option's termination condition to allow the longest expected execution without sacrificing performance. See Sutton et al. [73] and Precup [52] for details on these and other option-related algorithms and illustrations of their performance.

The primary motivation for the options framework is to permit one to add temporally-extended activities to the repertoire of choices available to an RL agent, while at the same time not precluding planning and learning at the finer grain of the core MDP. The emphasis is therefore on augmentation rather than simplification of the core MDP. If all the primitive actions remain in the option set as one-step options, then clearly the space of realizable policies is unrestricted so that the optimal policies over options are the same as the optimal policies for the core MDP. But since finding optimal policies in this case takes more computation via conventional DP than does just solving the core MDP, one is tempted to ask what one gains from this augmentation of the core MDP. One answer is to be found in the use of RL methods. For RL, the availability of temporally-extended activities can dramatically improve the agent's performance while it is learning, especially in the initial stages of learning. Invoking multi-step options provides one way to prevent the prolonged period of "flailing" that one often sees in RL systems. Options also can facilitate transfer of learning to related tasks. Of course, only some options can facilitate learning in this way, and a key question is how does a system designer decide on what options to provide. On the other hand, if the set of options does not include the one-step options corresponding to all of the primitive actions, then the space of policies over options is a proper subset of the set of all policies of the core MDP. In this case, though, the resulting SMDP can be much easier to solve than the core MDP: the options simplify rather than augment. This is the primary motivation of two other apporaches to abstraction in RL that we consider in the next two sections.

In the current state-of-the-art, the designer of an RL system typically uses prior knowledge about the task to add a specific set of options to the set of primitive actions available to the agent. In some cases, complete option policies can be provided; in other cases, option policies can be learned using, for example, intra-option learning methods together with option-specific reward functions that are provided by the designer. Providing options and their policies a priori is an opportunity to use background knowledge about the task to try to accelerate learning and/or provide guarantees about system performance during learning. Perkins and Barto [51, 50], for example, consider collections of options each of which descends on a Lyapunov function. Not only is learning accelerated, but the goal state is reached on every learning trial while the agent learns to reach the goal more quickly by approximating a minimum-time policy over these options.

When option policies are learned, they usually are policies for efficiently achieving *subgoals*, where a subgoal is often a state, or a region of the state space, such that reaching that state or region is assumed to facilitate achieving the overall goal of the task. The canonical example of a subgoal is a doorway in a robot navigation scenario. Given a collection of subgoals, one can define subgoal-specific reward functions that positively reward the agent for achieving the subgoal (while possibly penalizing it until the subgoal is achieved). Options are then defined which terminate upon achieving a subgoal, and their policies can be learned using the subgoal-specific reward function and standard RL methods. Precup [52] discusses one way to do this by introducing *subgoal values*, and Dietterich [14], whose approach we discuss in Section 4.3, proposes a similar scheme using *pseudo-reward* functions.

A natural question, then, is how are useful subgoals determined? McGovern [43, 44] developed a method for automatically identifying potentially useful subgoals by detecting regions that the agent visits frequently on successful trajectories but not on unsuccessful trajectories. An agent using this method selects such regions that appear early in learning and persist throughout learning, creates options for achieving them and learns their policies, and at the same time learns a higher-level policy that invokes these options appropriately to solve the overall task. Experiments with this method suggest that it can be useful for accelerating learning on single tasks, and that it can facilitate knowledge transfer as previously-discovered options are reused in related tasks. This approach builds on previous work in artificial intelligence that addresses abstraction, particularly that of Iba [28], who proposed a method for discovering macro-operators in problem solving. Related ideas have been studied by Digney [15, 16].

4.2 Hierarchies of Abstract Machines

Parr [48, 49] developed an approach to hierarchically structuring MDP policies called *Hierarchies of Abstract Machines* or HAMs. Like the options formalism, HAMs exploit the theory of SMDPs, but the emphasis is on simplifying complex MDPs by restricting the class of realizable policies rather than expanding the action choices. In this respect, as pointed out by Parr [48], it has much in common with the multilayer approach for controlling large Markov chains described by Forestier and Varaiya [20] who considered a two-layer structure in which the lower level controls the plant via one of a set of pre-defined regulators. The higher level, the supervisor, monitors the behavior of the plant and intervenes when its state enters a set of boundary states. Intervention takes the form of switching to a new low-level regulator. This is not unlike many hybrid control methods [8] except that the low-level process is formalized as a finite MDP and the supervisor's task as a finite SMDP. The supervisor's decisions occur whenever the plant reaches a boundary state, which effectively "erases" the intervening states from the supervisor's decision problem, thereby reducing its complexity [20]. In the options framework, each option corresponds to a low-level regulator, and when the option set does not contain the one-step options corresponding to all primitive actions, the same simplification results. HAMs extend this idea by allowing policies to be specified as hierarchies of stochastic finite-state machines.

The idea of the HAM approach is that policies of a core MDP are defined as programs which execute based on their own states as well as the current states of the core MDP. Departing somewhat from Parr's [48] notation, let $\mathcal{M}$ be a finite MDP with state set $\mathcal{S}$ and action sets $\mathcal{A}_s$, $s \in \mathcal{S}$. A HAM policy is defined by a collection of stochastic finite-state machines, $\{\mathcal{H}_i\}$, with state sets $\mathcal{S}_i$, stochastic transition functions δ_i, and input sets all equal to $\mathcal{M}$'s state set, $\mathcal{S}$. Each machine i also has a stochastic function $\mathcal{I}_i : \mathcal{S} \rightarrow \mathcal{S}_i$ that sets the initial state of $\mathcal{M}$ in the manner described below. Each $\mathcal{H}_i$ has four types of states: *action*, *call*, *choice*, and *stop*. An action state generates an action of the core MDP, $\mathcal{M}$, based on the current state of $\mathcal{M}$ and the current state of the currently executing machine, say $\mathcal{H}_i$. That is, at time step t, the action $a_t = \pi(m_t^i, s_t) \in \mathcal{A}_{s_t}$, where m_t^i is the current state of $\mathcal{H}_i$ and s_t is the current state of $\mathcal{M}$. A call state suspends execution of the currently executing $\mathcal{H}_i$ and initiates execution of another machine, say $\mathcal{H}_j$, where j is a function of $\mathcal{H}_i$'s state m_t^i. Upon being called, the state of $\mathcal{H}_j$ is set to $\mathcal{I}_j(s_t)$. A choice state nondeterministically selects a next state of $\mathcal{H}_i$. Finally, a stop state terminates execution of $\mathcal{H}_i$ and returns control to the machine that called it (whose execution commences where it was suspended). Meanwhile, the core MDP, upon receiving an action, makes a transition to a next state according to its transition probabilities and generates an immediate reward.[2] If no action is generated at step t, then $\mathcal{M}$ remains in its current state. Parr defines a HAM $\mathcal{H}$ to be the initial machine together with the closure of all machine states in all machines reachable from the possible initial states of the initial machine. Let us call this state set $\mathcal{S}_{\mathcal{H}}$. For convenience, he also assumes that the initial machine does not have a stop state and that there are no infinite, probability 1, loops that do not contain action states. This ensures that the core MDP continues to receive primitive actions.

Figure 1 shows a simple HAM state-transition diagram similar to an example given by Parr and Russell [49] for control simple simulated mobile robot. This HAM runs until the robot reaches an intersection. Whenever an obstacle is encountered, a choice state is entered that allows the robot to decide to back away from the obstacle by calling the machine `back-off` or to try to get around the obstacle by calling the machine `follow-wall`. Each of these machines has its own state-transition structure, possibly containing additional choice and call states. When this HAM is selected, it deterministically starts by calling the `follow-wall` machine.

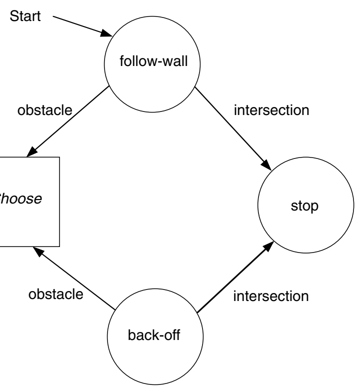


Figure 1: State-Transition Structure of a Simple HAM (after Parr [49]).

The composition of a HAM $\mathcal{H}$ and an MDP $\mathcal{M}$, as described above, yields a discrete-time SMDP denoted $\mathcal{H} \circ \mathcal{M}$. The state set of $\mathcal{H} \circ \mathcal{M}$ is $\mathcal{S} \times \mathcal{S}_{\mathcal{H}}$, and its transitions are determined by the parallel action of the transition functions of $\mathcal{H}$ and $\mathcal{M}$. The only actions of $\mathcal{H} \circ \mathcal{M}$ are the choices allowed in the *choice points* of $\mathcal{H} \circ \mathcal{M}$, which are the states whose $\mathcal{H}$ components are choice states. These actions change only the HAM component of each state. This is an SMDP because after a choice is made, the system—the composition of $\mathcal{H}$ and $\mathcal{M}$—runs autonomously until another choice point is reached. All the primitive actions to $\mathcal{M}$ during this period are fully determined by the action states of $\mathcal{H}$. The expected immediate rewards of $\mathcal{H} \circ \mathcal{M}$ are the expected returns accumulated during these periods between choice points, and they are determined by the immediate rewards of $\mathcal{M}$ together with rewards of zero for the time steps in which $\mathcal{M}$'s state does not change. Thus, one can think of a HAM as a method for delineating a possibly drastically restricted set of policies for $\mathcal{M}$. This restriction is determined by the prior knowledge that the HAM's designer, or programmer, has about what might be good ways to control $\mathcal{M}$.

The next step, which corresponds to the main observation of Forestier and Varaiya [20], is to note that in determining an optimal policy for $\mathcal{H} \circ \mathcal{M}$, the only relevant states are the choice points; the rest can be "erased." Therefore, there is an SMDP, called *reduce*$(\mathcal{H} \circ \mathcal{M})$, that is equivalent to $\mathcal{H} \circ \mathcal{M}$ but whose states are just the choice points of $\mathcal{H} \circ \mathcal{M}$. Optimal policies of *reduce*$(\mathcal{H} \circ \mathcal{M})$ are the same as the optimal policies of $\mathcal{H} \circ \mathcal{M}$. Of course, how close these policies will be to optimal policies of $\mathcal{M}$ will depend on the programmer's knowledge and skill.

How does RL enter into the HAM framework? It is easy to see that SMDP Q-learning given by (10)

[2]Parr [48] restricts the HAM call graph to be a tree so that call stack contents do not need to be treated as part of the program state, a point we gloss over in our discussion. This kind of machine hierarchy is an instance of a Recurisve Transition Network as discussed by Woods [88].

can be applied to *reduce*$(\mathcal{H} \circ \mathcal{M})$ to approximate optimal policies for $\mathcal{H} \circ \mathcal{M}$. The important strength of an RL method like SMDP Q-learning in this context is that it can be applied to *reduce*$(\mathcal{H} \circ \mathcal{M})$ without performing any explicit reduction of $\mathcal{H} \circ \mathcal{M}$. Trajectories of $\mathcal{M}$ under control of HAM $\mathcal{H}$, i.e., trajectories of $\mathcal{H} \circ \mathcal{M}$, are generated, either through simulation or observed from the real system. The update (10) is applied from choice point to choice point. In more detail, the learning system maintains a Q-table with entries $Q([s, m], a)$ for each state, s, of $\mathcal{M}$, each choice state, m, of $\mathcal{H}$, and each action, a, that can be taken from the corresponding choice point. It also has to store the previous choice point, $[s_c, m_c]$, and the action selected at that choice point, a_c. Suppose choice point $[s_c, m_c]$ is encountered at time step t, and the next choice point, $[s'_c, m'_c]$, is encountered at $t + \tau$. Then the SMDP Q-learning update (10) appears as follows:

$$Q_{k+1}([s_c, m_c], a_c) = (1 - \alpha_k)Q_k([s_c, m_c], a_c) + \alpha_k[r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{\tau-1} r_{t+\tau} + \gamma^\tau \max_{a'} Q_k([s'_c, m'_c], a')].$$

Here, the max is taken over all actions available in choice point $[s'_c, m'_c]$. Clearly the between-choice return, $r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{\tau-1} r_{t+\tau}$, can be accumulated iteratively between the visits to these choice points. If the SMDP Q-learning convergence conditions are in force (Section 3), then the sequence of action-values functions generated by this algorithm converges to Q^* of *reduce*$(\mathcal{H} \circ \mathcal{M})$ with probability 1. Consequently, any sequence of policies that are greedy with respect to these successive action-value functions converges with probability 1 to an optimal policy.

We know of no large-scale applications of HAMs at the present time, but Parr [48] and Parr and Russell [49] illustrate the advantages that HAMs can provide in several simulated robot navigation tasks. Andre and Russell [1] increased the expressive power of HAMs by introducing Programmable HAMs (PHAMs), which add interrupts, aborts, local state variables, and the ability to pass parameters. PHAMs, and other extenstions of HAMs that may occur in the future, point the way toward methods theoretically grounded in stochastic optimal control that use expressive programming languages to provide a knowledge-rich context for RL.

4.3 MAXQ Value Function Decomposition

Dietterich [14] developed another approach to hierarchical RL called the MAXQ Value Function Decomposition, which we call simply MAXQ. Like options and HAMs, this approach relies on the theory of SMDPs. Unlike options and HAMs, however, the MAXQ approach does not rely directly on reducing the entire problem to a single SMDP. Instead, a hiearchy of SMDPs is created whose solutions can be learned simultaneously. The MAXQ approach starts with a decomposition of a core MDP $\mathcal{M}$ into a set of subtasks $\{\mathcal{M}_0, \mathcal{M}_1, \ldots, \mathcal{M}_n\}$. The subtasks form a hierarchy with $\mathcal{M}_0$ being the root subtask, which means that solving $\mathcal{M}_0$ solves $\mathcal{M}$. Actions taken in solving $\mathcal{M}_0$ consist of either executing primitive actions or policies that solve other subtasks, which can in turn invoke primitive actions or policies of other subtasks, etc.

The structure of the hierarchy is summarized in a *task graph*, an example of which is given in Figure 2 for a Taxi problem that Dietterich uses as an illustration. Each episode of the overall task consists of picking up, transporting, and dropping off a passenger. The overall problem, corresponding to the root node of the graph, is decomposed into the subtask `Get`, which is the subtask of going to the passenger's location and picking them up, and the subtask `Put`, which is the subtask of going to the passenger's destination and dropping them off. These subtasks, in turn, are respectively decomposed into the primitive actions `Pickup` or `Dropoff`, which respectively pick up and drop off a passenger, and the subtask `Navigate`(t), which consists of navigating to one of the locations indicated by the parameter t. (A subtask parameterized like this is shorthand for multiple copies of the subtask, one for each value of the parameter.) This subtask `Navigate`(t) is decomposed into the primitive actions that are moves `North`, `South`, `East`, or `West`. The subtasks and primitive actions into which a subtask $\mathcal{M}_i$ is decomposed are called the "children" of $\mathcal{M}_i$. An important aspect of a task graph is that the order in which a subtask's children are shown is arbitrary. Which choice the higher level controller makes depends on its policy. The graph just restricts the action choices that can be made at each level. See Dietterich [14] for details.

Each subtask, $\mathcal{M}_i$, consists of three components. First, it has a subtask policy, π_i, that can select other subtasks from the set of $\mathcal{M}_i$'s children. Here, as with options, primitive actions are special cases of subtasks. Second, each subtask has a termination predicate that partitions the state set, $\mathcal{S}$, of the core MDP into $\mathcal{S}_i$, the set of *active states*, in which M_i's policy can execute, and $\mathcal{T}_i$, the set of *termination states*, which when entered causes the policy to terminate. Third, each subtask $\mathcal{M}_i$ has a *pseudo-reward function* that assigns

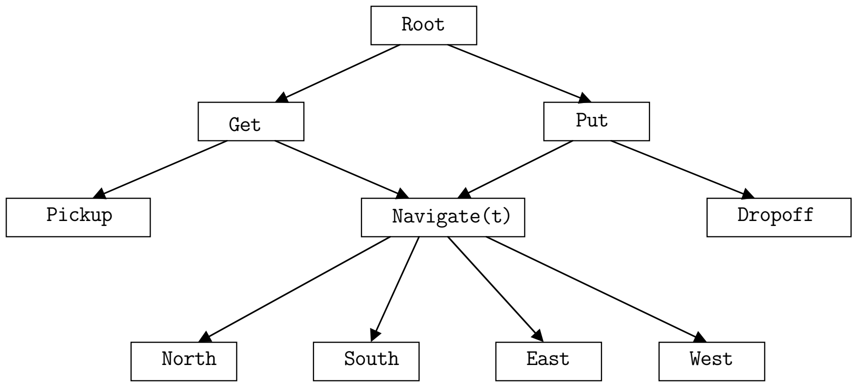


Figure 2: A Task Graph for the Taxi Problem (after Dietterich [14]).

reward values to the states in $\mathcal{T}_i$. The pseudo-reward function is only used during learning, which we discuss after first describing how the task graph hierarchy allows value functions to be decomposed.

A subtask is very much like a hierarchical option, $\langle \mathcal{I}_i, \mu_i, \beta_i \rangle$, as defined in Section 4.1, with the addition of a pseudo-reward function. The policy over options, μ_i, corresponds to the subtask's π_i; the termination condition, β_i, in this case assigns to states termination probabilities of only 1 or 0; and the option's input set $\mathcal{I}_i$ corresponds to $\mathcal{S}_i$. Unlike the option formalism, however, which treats semi-Markov options, MAXQ explicitly adds a component to each state that gives the current contents, K, of a pushdown stack containing the names and parameter values of the hierarchy of calling subtasks, as in subroutine handling of ordinary programming languages. At any time step, the top of the stack contains the name of the subtask currently being executed. Thus, while a subtask's policy is non-Markov with respect to the state set of the core MDP, it is Markov with respect to this augmented state set. As a consequence, each subtask policy has to assign actions to every combination, $[s, K]$, of core state s and stack contents K.

Given a hierarchical decomposition of $\mathcal{M}$ into n subtasks as given by a task graph, a *hierarchical policy* is defined to be $\pi = \{\pi_0, \dots, \pi_n\}$, where π_i is the policy of $\mathcal{M}_i$. The *hierarchical value function* for π gives the value, i.e., the expected return, for each state-stack pair, $[s, K]$, given that π is followed from s when the stack contents are K. This value is denoted $V^\pi([s, K])$. The "top level" value of a state s is $V^\pi([s, nil])$, indicating that the stack is empty. This is the value of s under the flat policy induced by the hierarchy of subtask calls starting from the root of the hierarchy. The *projected value function* of hierarchical policy π on subtask $\mathcal{M}_i$ gives the expected return of each state s under the assumption that π_i is executed until it terminates. This projected value is denoted $V^\pi(i, s)$. This is the same as the value function of the hierarchical option corresponding to $\mathcal{M}_i$ as defined in Section 4.1.

Given a hierarchical decomposition of $\mathcal{M}$ and a hierarchical policy, π, each subtask $\mathcal{M}_i$ defines a discrete-time SMDP with state set is S_i. Its actions are its child subtasks, $\mathcal{M}_a$, and its transition probabilities, $P_i(s', \tau | s, a)$ (cf. Eqs. 6 and 7), are well-defined given the policies of the lower-level subtasks. A key observation, which follows that of Singh [66, 67], is that this SMDP's expected immediate reward, $R_i(s, a)$, for executing action (subtask) a is the projected value of π on subtask $\mathcal{M}_a$. That is, for all $s \in \mathcal{S}_i$ and all child subtasks $\mathcal{M}_a$ of $\mathcal{M}_i$, $R_i(s, a) = V^\pi(s, a)$. To see why this is true, suppose the core MDP is in state s when subtask $\mathcal{M}_i$ selects one of its child subtasks, $\mathcal{M}_a$, for execution. Expanding $\mathcal{M}_a$'s policy down to primitive actions results in a flat policy that accumulates rewards according to the core MDP until $\mathcal{M}_a$ terminates at a state $s' \in \mathcal{T}_a$. The waiting time for s when this action is chosen is the execution time of $\mathcal{M}_a$'s policy. The reward accumulated during this waiting time is $V^\pi(s, a)$. Given this, one can write a Bellman equation for the SMDP corresponding to subtask $\mathcal{M}_i$:

$$V^\pi(i, s) = V^\pi(\pi_i(s), s) + \sum_{s', \tau} P_i^\pi(s', \tau | s, \pi_i(s)) \gamma^\tau V^\pi(i, s'), \tag{15}$$

where $V^\pi(i, s')$ is the expected return for completing subtask $\mathcal{M}_i$ starting in state s' (cf. Eq. 6).

The action-value function, Q, defined by (1) can be extended to apply to subtasks: for hierarchical policy π, $Q^\pi(i, s, a)$ is the expected return for action a (a primitive action or a child subtask) being executed in subtask $\mathcal{M}_i$ and then π being followed until $\mathcal{M}_i$ terminates. In terms of this subtask action-value function, the observation expressed in (15) takes the form:

$$Q^\pi(i, s, a) = V^\pi(a, s) + \sum_{s', \tau} P_i^\pi(s', \tau | s, a)\gamma^\tau Q^\pi(i, s', \pi(s')). \tag{16}$$

Dietterich [14] calls the second term on the right of this equation the *completion function*:

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

This gives expected return for completing subtask $\mathcal{M}_i$ after subtask $\mathcal{M}_a$ terminates. Rewriting (16) using the completion function, we have

$$Q^\pi(i, s, a) = V^\pi(a, s) + C^\pi(i, s, a). \tag{17}$$

Now it is possible to describe the MAXQ hierarchical value function decomposition. Given a hierarchical policy π and a state s of the core MDP, suppose the policy of the top-level subtask, $\mathcal{M}_0$, selects subtask $\mathcal{M}_{a_1}$, and that this subtask's policy selects subtask $\mathcal{M}_{a_2}$, whose policy in turn selects subtask $\mathcal{M}_{a_3}$, etc. until finally subtask $\mathcal{M}_{a_{n-1}}$'s policy selects a_n, a primitive action that is executed in the core MDP. Then the projected value of s for the root subtask, $V^\pi(0, s)$, which is the value of s in the core MDP, can be written as follows:

$$V^\pi(0, s) = V^\pi(a_n, s) + C^\pi(a_{n-1}, s, a_n) + \cdots + C^\pi(a_1, s, a_2) + C^\pi(0, s, a_1), \tag{18}$$

where $V^\pi(a_n, s) = \sum_{s'} P(s'|s, a_n)R(s'|s, a_n)$.

The decomposition (18) is the basis of a learning algorithm that is able to learn hierarchical policies from sample trajectories. The details of this algorithm are somewhat complex and beyond the scope of this review, but we describe the basic idea. This algorithm is a recursively applied form of SMDP Q-learning that takes advantage of the MAXQ value function decomposition to update estimates of subtask completion functions. The simplest version of this algorithm applies when the pseudo-reward functions of all the subtasks are identically zero. The key component of the algorithm is a function—called MAXQ-0 for this special case—that calls itself recursively to descend through the hierarchy to finally execute primitive actions. When it returns from each call, it updates the completion function corresponding to the appropriate subtask, with discounting determined by the returned count of the number of primitive actions executed. For details see Dietterich [14].

If the agent follows a policy that is GLIE (Section 3) and also further constrained to always break ties in the same order, and the step-size parameter converges to zero according to the usual stochastic approximation conditions, then the algorithm sketched above converges with probability 1 to the unique *recursively optimal* policy for $\mathcal{M}$ that is consistent with the task graph. A recursively optimal policy is a hierarchical policy, $\pi = \{\pi_0, \ldots, \pi_n\}$, such that for each subtask $\mathcal{M}_i$, $i = 0, \ldots, n$, π_i is optimal for the SMDP corresponding to $\mathcal{M}_i$, given the policies of $\mathcal{M}_i$'s children subtasks. This form of optimal policy stands in contrast to a *hierarchically optimal* policy, which is a hierarchical policy that is optimal among all the policies that can be expressed within the constraints imposed by the given hierarchical structure.

Examples of policies that are recursively optimal but not hierarchically optimal are easy to construct. This is because the hierarchical optimality of a subtask generally depends not only on that subtask's children, but also on how the subtask participates in higher-level subtasks. For example, the hierarchically optimal way to travel to a given destination may depend on what you intend to do after arriving there in addition to properties of the trip itself. SMDP DP or RL methods applied to HAMs, or to MDPs with options whose policies are fixed a priori, yield hierarchically optimal policies, and it is relatively easy to define a learning algorithm for MAXQ hierarchies that also produces hierarchically optimal policies. Dietterich's interest in the weaker recursive optimality stems from the fact that this form of optimality can be determined without considering the context of a subtask. This facilitates using subtask policies a building blocks for other tasks and has implications for state abstraction, although we do not touch on the latter aspect in this review.

What about the pseudo-reward functions? These functions allow a system designer within the MAXQ framework to specify subtasks by defining subgoals that they must achieve but without specifying policies for achieving them. In this respect, they play the same role as do the auxiliary reward functions that can be used in the options framework for learning option policies. The MAXQ-0 learning algorithm sketched above can be extended to an algorithm, called MAXQ-Q, that learns a policy that is recursively optimal hierarchical with respect to the sum of the original reward function and the pseudo-reward functions.

We have not been able to do justice to the MAXQ approach and associated algorithms in this short review. However, we have presented enough to provide a basis for seeing how SMDP theory again plays a central role in an approach to hierarical RL. Like HAMs, MAXQ is an excellent example of how concepts from programming languages can be fruitfully integrated with a stochastic optimal control framework.

5 Recent Advances in Hierarchical RL

Thus far we have surveyed different approaches to hierarchical RL, all of which are based on the underlying framework of SMDPs. These approaches collectively suffer from some key limitations: policies are restricted to *sequential* combinations of activities; agents are assumed to act alone in the environment; and finally, states are considered to be fully accessible. In this section, we address these assumptions and describe how the SMDP framework can be extended to *concurrent activities*, *multiagent domains*, and *partially observable* states.

5.1 Concurrent Activities

Here we summarize recent work by Rohanimanesh and Mahadevan [57] towards a general framework for modeling concurrent activities. This framework is motivated by situations in which a single agent can execute multiple parallel processes, as well as by the multiagent case in which many agents act in parallel (addressed in Section 5.2). Managing concurrent activities clearly is a central component of our everyday behavior: in making breakfast, we interleave making toast and coffee with other activities such as getting milk; in driving, we search for road signs while controlling the wheel, accelerator and brakes.

Building on the SMDP framework described above, this approach focuses on modeling concurrent activities, where each component activity is temporally extended. One immediate question that arises in modeling concurrent activities is that of termination: unlike the purely sequential case, when multiple activities are executed simultaneously, concurrently executing activities do not generally terminate at the same time. How does one define the termination of a set of concurrent activities? Among the termination schemes studied by Rohanimanesh and Mahadevan [57] are the following two. The *any* scheme terminates all other activities when the first activity terminates, whereas the *all* scheme waits until all existing activities terminate before choosing a new concurrent set of activities. These researchers have also studied other schemes, such as *continue*, which replaces the terminated activity with a set of new activities that includes the activities already executing. For simplicity, we restrict our discussion to the first two schemes.

For concreteness, we describe the concurrent activity model using the options formalism described in Section 4.1. The treatment here is restricted to options over discrete-time SMDPs and having deterministic policies, but the main ideas extend readily to the other variants (HAMs, MAXQ), as well as to continuous-time SMDPs. (See ref. [21] for a treatment of hierarchical RL for continuous-time SMDPs.) The sequential option model is generalised to a *multi-option*, which is a set of options that can be executed in parallel. Here we discuss the simple case in which the options comprising a multi-option influence different sets of state variables that do not interact. (This assumption can be generalized, but for simplicity, we restrict our treatment here.) For example, turning the radio off and pressing the brake can always be executed in parallel since they affect different non-interacting state variables.

We now define a multi-option (denoted by $\vec{o}$) more precisely. Let $o =< I, \pi, \beta >$ be an option as defined in Section 4.1 on a core MDP with state set $\mathcal{S} \subseteq \Pi_1^n \mathcal{S}_i$, where $\mathcal{S}_i$ is the range of state variable s_i, $i = 1, \ldots, n$. Suppose that while option o is executing, it influences only a subset of the state variables $\mathcal{S}_o \subseteq \{s_1, s_2, ..., s_n\}$. Assume also that while o is executing, these variables do not influence, and are not influenced by, any variables not in $\mathcal{S}_o$. A set of options $\{o_1, \ldots, o_m\}$, each defined on the same core MDP, is *coherent* if $\cap_1^m \mathcal{S}_{o_i} = \emptyset$. This ensures that each option will affect different components of the state set so that any subset of them can be executed in parallel without interfering with one another. In other words, while a coherent set of options is executing, the system admits a parallel decomposition with each component corresponding to the state variables influenced by one of the options. In the driving example, the `turn-radio-on` and `brake` options comprise a coherent set, but the `turn-right` and `accelerate` options do not since the state variable `position` is influenced by both. A multi-option over an MDP $\mathcal{M}$ is a coherent set of options over $\mathcal{M}$.

When multi-option $\vec{o}$ is executed in state s, several options $o_i \in \vec{o}$ are initiated. Each option o_i will terminate at some random time t_{o_i}. We can define the termination of a multi-option based on either of the following events: (1) $T_{all} = \max_i(t_{o_i})$: when all the options $o_i \in \vec{o}$ terminate according to $\beta_i(s)$, multi-option $\vec{o}$ is declared terminated, or (2) $T_{any} = \min_i(t_{o_i})$: when any (i.e., the first) of the options terminate, the options that are still executing at that time are interrupted.

We then have the following result. Given a finite MDP and a collection of multi-options defined on it, where the underlying options are Markov, the decision process that selects only among multi-options, and executes each one until its termination according to the T_{all} or T_{any} termination conditions, is a discrete-time SMDP. The proof requires showing that the state transition probabilities and the rewards corresponding to any concurrent option $\vec{o}$ defines an SMDP [57]. The significance of this result is that SMDP Q-learning methods can be extended to learn policies over concurrent options under this model.

The extended SMDP Q-learning algorithm for learning policies over multi-options updates the multi-option value function $Q(s, \vec{o})$ after each decision epoch in which the multi-option $\vec{o}$ is taken in state s and terminates in state s' (under either termination condition):

$$Q_{k+1}(s, \vec{o}) = (1 - \alpha_k) Q_k(s, \vec{o}) + \alpha \left[r + \gamma^\tau \max_{\vec{o'} \in \mathcal{O}_{s'}} Q_k(s', \vec{o'}) \right], \tag{19}$$

where τ denotes the number of time steps between initiation of the multi-option $\vec{o}$ in state s and its termination in state s', and r denotes the cumulative discounted reward over this period. This learning rule generalizes (13) to multi-options. It is straightforward to also generalize other learning algorithms, such as the intra-option Q-learning algorithm (14) to multi-options.

As a simple illustrative example of using this concurrent SMDP Q-learning algorithm, Rohanimanesh and Mahadevan [57] considered an environment consisting of four "rooms", each of which is divided into a discrete set of cells (Figure 3). Each room contains two doors, each of which can be "opened" by two keys. The agent is given options for getting to each door from any interior room state, and for opening a locked door. It has to learn the shortest path to the goal by concurrently combining these options. The agent can reach the goal more quickly if it learns to parallelize the option for retrieving the key before it reaches a locked door. However, retrieving the key too early is counterproductive since it can be dropped with some probability. The process of retrieving a key is modeled as an SMDP consisting of a linear chain of states to model the waiting process before the agent is holding the key.

Figure 4 compares the policies learned with strictly sequential options, as described in Section 4.1, using the Q-learning algorithm (13), with policies over multi-options learned using the extended Q-learning rule (19). The vertical axis plots the median steps to the goal in each trial. Note that the sequential solution is the slowest, taking the agent about 50 steps to reach the goal. The termination condition T_{all} is faster than the sequential case, taking about 45 steps to reach the goal. Both of these, however, are significantly slower than the T_{any} and $T_{continue}$ policies, which reach the goal in half the time. T_{any} converges the slowest, since it terminates multi-options fairly aggressively resulting in more decision epochs. $T_{continue}$ provides the best tradeoff between speed of convergence and optimality of the learned policy.

The multi-option formalism can be extended to allow cases in which options executing in parallel modify the same shared variables at the same time. Details of this extension is beyond the scope of the present article.

5.2 Multiagent Coordination

Concurrency is the basis for modeling coordination among multiple *simultaneously behaving* agents. From a theoretical point of view, it matters little if concurrent activities are being executed by a single agent or by multiple cooperating agents. However, the multiagent problem usually involves other complexities as well, such as the fact that one agent cannot usually observe the actions or the states of other agents in the environment. We address the issue of hidden state in the next section; here we focus on the problem of learning a policy across *joint states* and *joint actions*.

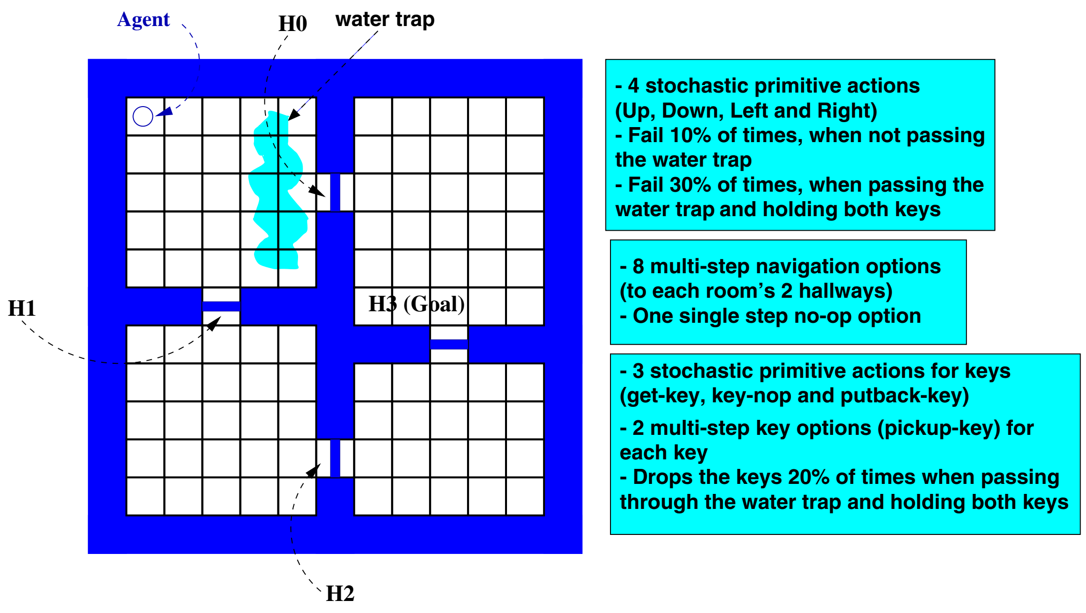

Agent
H0
water trap
H1
H3 (Goal)
H2
- 4 stochastic primitive actions (Up, Down, Left and Right)
- Fail 10% of times, when not passing the water trap
- Fail 30% of times, when passing the water trap and holding both keys
- 8 multi-step navigation options (to each room's 2 hallways)
- One single step no-op option
- 3 stochastic primitive actions for keys (get-key, key-nop and putback-key)
- 2 multi-step key options (pickup-key) for each key
- Drops the keys 20% of times when passing through the water trap and holding both keys

Figure 3: An Illustrative Problem for Concurrent Options.

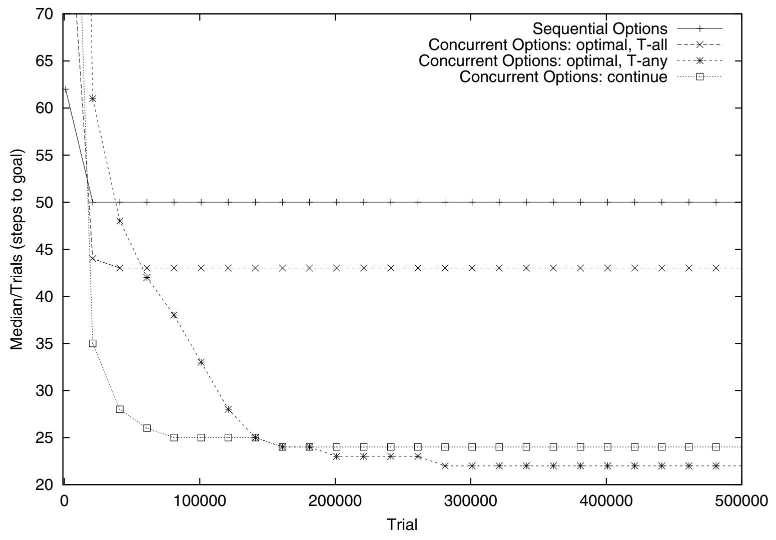


Figure 4: Multi-Option Comparison. This graph compares a multi-option SMDP Q-learning system (under different termination schemes) with one that learns sequential policies. Policies over multi-options easily outperform sequential policies, and termination makes a large difference in of convergence rate and quality of the learned policy.

In general, problems involving multiagent coordination can be modeled by assuming that states now represent the *joint state* of n agents, where each agent i may only have access to a partial view s_i. Also, the joint action is represented as $(a_1, \ldots, a_n)$, where again, each agent may not have knowledge of the other agents' actions. Typically, the assumption made in multiagent studies is that the set of joint actions defines an SMDP (or MDP) over the joint state set. Of course, in most practical problems, the joint state and action sets are exponential in the number of agents, and the aim is to find a *distributed* solution that does not require combinatorial enumeration over joint states and actions.

One general approach to learning task-level coordination is to extend the above concurrency model to the joint state and action spaces, where each action is a fixed option with a pre-specified policy. This approach requires very minimal modification of the approach described in the previous section. In contrast, we now describe an extension of this approach due to Makar and Mahadevan [41] in which agents learn both coordination skills and the base-level policies using a multiagent MAXQ-like task graph. However, convergence to (hierarchically) optimal policies is no longer assured since lower-level subtask policies are varying at the same time when learning higher-level politices. The ideas can be extended to other formalisms also, but for the sake of clarity, we focus on the MAXQ value function decomposition approach described in Section 4.3.

It is necessary to generalize the MAXQ decomposition from its original sequential single-agent setting to the multiagent coordination problem. Let $\vec{o} = (o_1, ..., o_n)$ denote a multi-option, where o_i is the option executed by agent i.[3] Let $s = (s_1, ..., s_n)$ denote a joint state. The joint action-value of a multi-option $\vec{o}$ in a joint state s, and in the context of doing parent task p, is denoted $Q(p, s, \vec{o})$. The MAXQ decomposition of the Q-function given by (17) can be extended to joint action-values as follows. The *joint completion function* for agent j assigns values $C^j(p, s_j, \vec{o})$ giving the discounted return for agent j completing a multi-option in in the context of doing parent task p, when the other agents are performing multi-options o_k, for all $k \in \{1, \dots, n\}$, $k \neq j$. The joint mulit-option value $Q(p, \vec{s}, \vec{o})$ is now approximated by each agent j (given only its local state s_j) as:

$$Q^j(p, s_j, \vec{o}) \approx V^j(\vec{o_j}, s_j) + C^j(p, s_j, \vec{o}),$$

where

$$V^j(p, s_j) = \begin{cases} \max_{\vec{o_k}} Q^j(p, s_j, \vec{o_k}) & \text{if parent task p is non-primitive} \\ \sum_{s'_j} P(s'_j \mid s_j, p) R(s'_j \mid s_j, p) & \text{if p is a primitive action.} \end{cases}$$

The first term in the $Q(p, s_j, \vec{o})$ expansion above refers to the discounted sum of rewards received by agent j for doing concurrent action $\vec{o_j}$ in state s_j. The second term "completes" the sum by accounting for rewards earned for completing the parent task p after finishing $\vec{o_j}$. The completion function is updated from sample values using an SMDP learning rule. Note that the correct action value is approximated by only considering local state s_j and also by ignoring the effect of concurrent actions $\vec{o_k}$, $k \neq j$ by other agents when agent j is performing $\vec{o_j}$. In practice, a human designer can configure the task graph to store joint concurrent action values at the highest level(s) of the hierarchy as needed.

Figure 5 illustrates a robot trash collection task, where the two agents, $A1$ and $A2$, will maximize their performance at the task if they learn to coordinate with each other. Here, we want to design learning algorithms for *cooperative* multiagent tasks [84], where the agents learn the coordination skills by trial and error. The key idea here is that coordination skills are learned more efficiently if agents learn to synchronize using a hierarchical representation of the task structure [69]. In particular, rather than each robot learning its response to low-level primitive actions of the other robots (for instance, if A1 goes forward, what should A2 do), they learn high-level coordination knowledge (what is the utility of A2 picking up trash from T1 if A1 is also picking up from the same bin, and so on). The proposed approach differs significantly from previous work in multiagent reinforcement learning [38, 74] in using hierarchical task structure to accelerate learning, and as well in its use of concurrent activities.

To illustrate the use of this decomposition in learning multiagent coordination, for the two-robot trash collection task, if the joint action-values are restricted to only the highest level of the task graph under the root, we get the following value function decomposition for agent $A1$:

$$Q^1(\mathit{Root}, s_1, (NavT1, NavT2)) \approx V^1_t((NavT1), s_1) + C^1_t(\mathit{Root}, s_1, (NavT1, NavT2)),$$

which represents the value of agent $A1$ doing task $NavT1$ in the context of the overall *Root* task, when agent $A2$ is doing task $NavT2$. Note that this value is decomposed into the value of agent $A1$ doing $NavT1$ subtask itself and the completion sum of the remainder of the overall task done by both agents. In this example, the multiagent MAXQ decomposition embodies the constraint that the value of $A1$ navigating to trash bin $T1$ is independent of whatever $A2$ is doing.

[3]For multiagent problems, we treat a multi-option as a tuple rather than a set since its elements are associated with specific agents. It is also possible to generalize this so that each o_i is itself a multi-option.

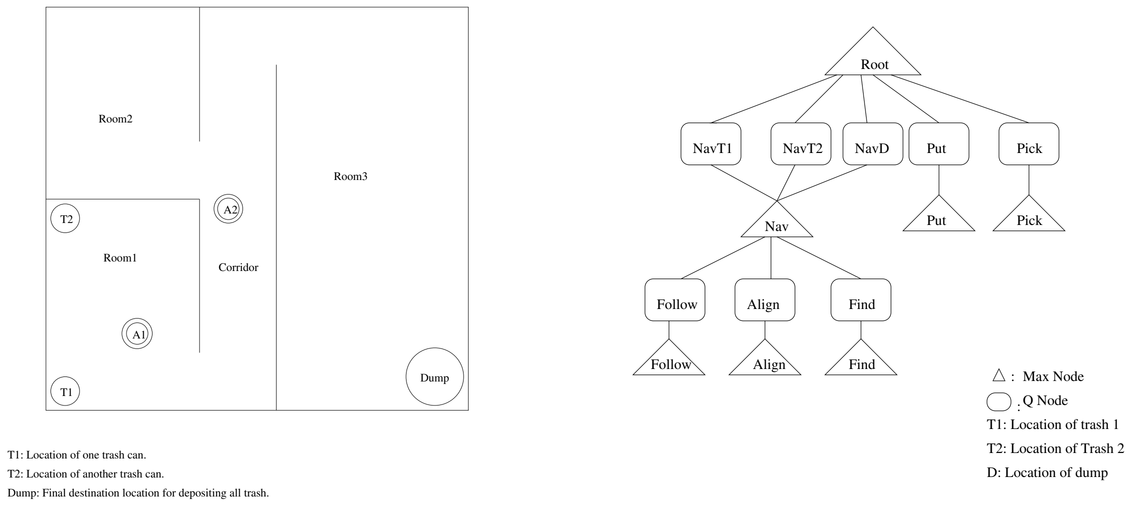


Figure 5: A Two-Robot ($A1$ and $A2$) Trash Collection Task. The robots can learn to coordinate much more rapidly using the task structure than if they attempted to coordinate at the level of primitive movements. On the right is shown a MAXQ graph, which is MAXQ task graph with two kinds of nodes representing subtasks and the actions those subtasks can select [14].

5.3 Hierarchical Memory

In multiagent environments, agents cannot observe joint states and joint actions, but must act based on estimates of these hidden variables (the problem of hidden state occurs in many single-agent tasks as well, such as a robot navigating in an indoor office environment). One approach is formalized in terms of Partially observable Markov decision processes (POMDPs), where agents learn policies over *belief states*, i.e., probability distributions over the underlying state set [31]. It can be shown that belief states satisfy the Markov property and consequently yield a new (and more complex) MDP over information states. Belief states can be recursively updated using the transition model, and an observation model $O(y \mid s, a)$ specifying the likelihood of observing y if action a was performed and resulted in state s. However, mapping belief states to optimal actions is known to be intractable, particularly in the decentralized multiagent formulation [3]. Also, learning a perfect model of the underlying POMDP is a challenging task. An empirically more effective (but theoretically less powerful) approach is to use finite memory models as linear chains or nonlinear trees over histories [42]. However, such finite memory structures can be defeated by long sequences of mostly irrelevant observations and actions that conceal a critical past observation.

We briefly summarize three multiscale memory models that have been explored recently by Hernandez and Mahadevan [25], Theocharous and Mahadevan [79], and Jonsson and Barto [30]. These models combine temporal abstraction with previous methods for dealing with hidden state. *Hierarchical Suffix Memory* (HSM) [25] generalizes the suffix tree model [42], to SMDP-based temporally-extended activities. Suffix memory constructs state estimators from finite chains of observation-action-reward triples. In addition to extending suffix models to temporally-extended activities, HSM also uses multiple layers of temporal abstraction to form longer-term memories at more abstract levels. Figure 6 illustrates this idea for robot navigation for the simpler case of a linear chain, although the tree-based model has also been investigated. An important side-effect is that the agent can look back many steps back in time while ignoring the exact sequence of low-level observations and actions that transpired. Tests in a robot navigation domain showed that HSM outperformed "flat" suffix tree methods, as well as hierarchical methods that used no memory [25].

POMDPs are theoretically more powerful than finite memory models, but past work on POMDPs has mostly studied "flat" models for which learning and planning algorithms scale poorly with model size. Theocharous et al. [79] developed a *hierarchical POMDP* formalism, termed H-POMDPs (Figure 7), by extending the hierarchical hidden Markov model (HHMM) [19] to include rewards and temporally-extended

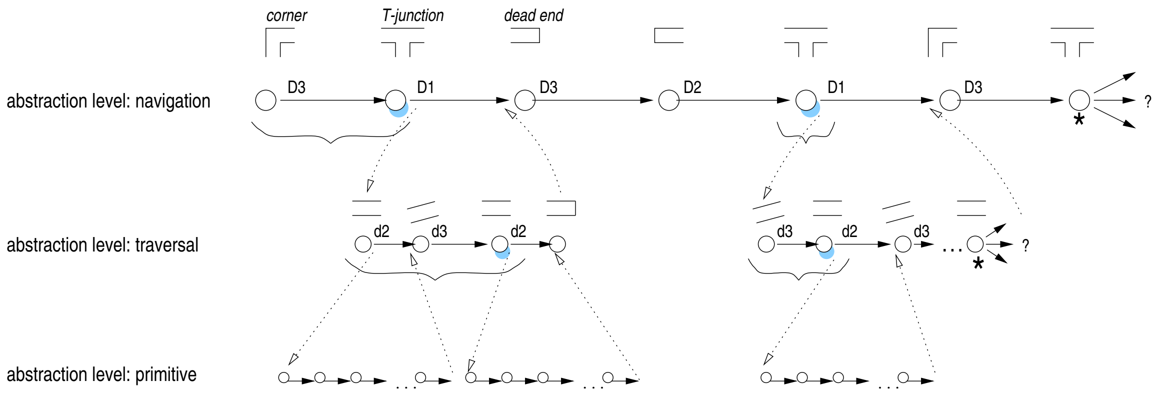

corner
T-junction
dead end
abstraction level: navigation
D3
D1
D3
D2
D1
D3
?
*
abstraction level: traversal
d2
d3
d2
d3
d2
d3
?
*
abstraction level: primitive

Figure 6: Hierarchical Suffix Memory State Estimator for a Robot Navigation Task. At the abstract (navigation) level, observations and decisions occur at intersections. At the lower (corridor-traversal) level, observations and decisions occur within the corridor. At each level, each agent constructs state representations from its past experience with similar history (shown with shadows).

activities. They also developed a hierarchical EM algorithm for learning the parameters of an H-POMDP model from sequences of observations and actions. Extensive tests on a robot navigation domain show learning and planning performance is much improved over flat POMDP models [79, 78]. The hierarchical EM-based parameter estimation algorithm scales more gracefully to large models because previously learned sub-models can be reused when learning at higher levels. In addition, the effect of temporally-extended activities in H-POMDPs (e.g., exit the corridor) can be modeled at abstract and product-level states, which supports planning at multiple levels of abstraction.

H-POMDPs have an inherent advantage in allowing belief states to be computed at different levels of the tree. In addition, there is often less uncertainty at higher levels (e.g., a robot is more sure of which corridor it is in than exactly where it is in the corridor). A number of heuristics for mapping belief states to actions provide good performance in robot navigation (e.g, the most-likely-state (MLS) heuristic assumes the agent is in the state corresponding to the "peak" of the belief state distribution) [35, 63, 47]. Such heuristics work much better in H-POMDPs because they can be applied at multiple levels, and belief states over abstract states usually have lower entropy (Figure 8). For a detailed study of the H-POMDP model, as well as its application to robot navigation, see [77].

Jonsson and Barto [30] also addressed partial observability by adapting suffix tree methods to hierarchical RL systems. Their approach focused on automating the process of constructing activity-specific state representations by applying McCallum's U-Tree algorithm [42] to individual options. The U-Tree algorithm employs the concept of a suffix tree to automatically construct a state representation starting from one that makes no distinctions between different observation vectors. Thus, no specification of state-feature dependencies is necessary prior to learning. With a separate U-Tree assigned to each option, it is possible to perform state abstraction separately for each option.

The U-Tree algorithm retains a history of transition instances $T_t =< T_{t-1}, a_{t-1}, r_t, s_t >$ composed of the observation vector, s_t, at time step t, the previous action, a_{t-1}, the reward, r_t, received during the transition into s_t, and the previous instance, T_{t-1}. A decision tree—the U-Tree—sorts a new instance T_t based on its components and assigns it to a unique leaf $L(T_t)$ of the tree. The distinctions associated with a leaf are determined by the root-to-leaf path. For each leaf-action pair (L_j, a), the algorithm keeps an action-value $Q(L_j, a)$ estimating the future discounted reward associated with being in L_j and executing a. These action-values can be updated in a variety of ways, such as via a DP algorithm if a system model is available (or can be learned) or RL algorithms such as Q-learning.

The U-Tree algorithm periodically adds new distinctions to the tree in the form of temporary nodes, called fringe nodes, and performs statistical tests to see whether the added distinctions increase the predictive power of the U-Tree. The tree is extended with new distinctions when these are estimated to increase the tree's predictive power. Whenever the tree is extended, the action-values of the previous leaf node are passed on to

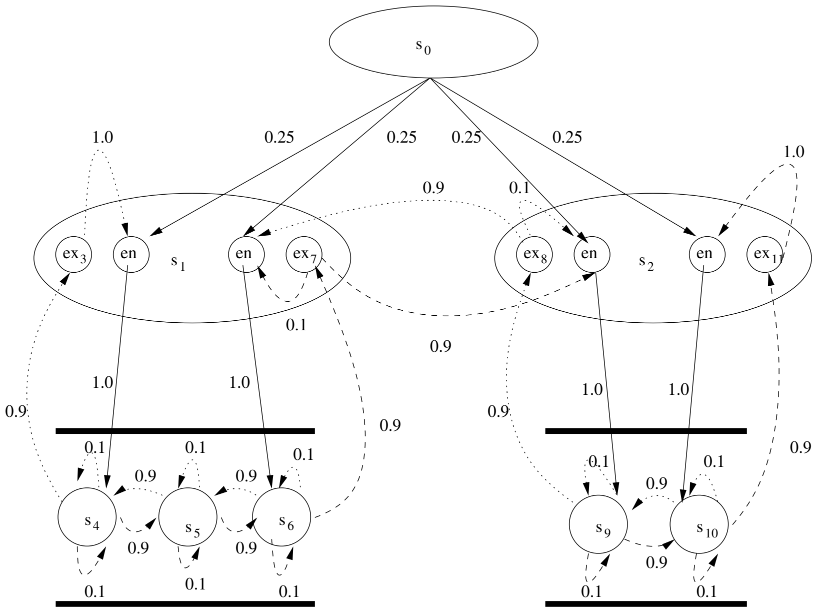
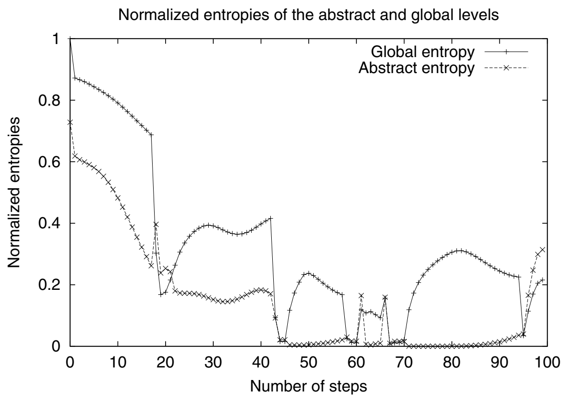


Figure 7: Hierarchical POMDP Model. This hierarchical POMDP model represents two adjacent corridors in a robot navigation task. The model has two primitive actions, "go-left" indicated with the dotted arrows and "go-right" indicated with the dashed arrows. This HPOMDP has two (unobservable) abstract states s_1 and s_2, and each abstract state has two entry and two exit states. The (hidden) product states s_4, s_5, s_6, s_9, and s_{10} have associated observation models.



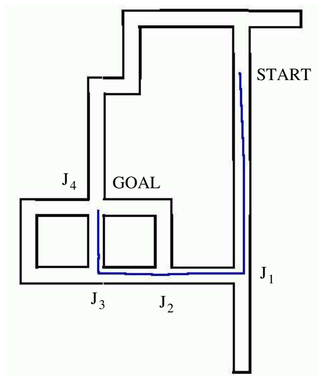


Figure 8: Entropy of a Sample Robot Navigation Run. This graph shows a sample robot navigation run whose trace is on the right, where positional uncertainty (measured by belief state entropy) at the abstract (corridor) level is less than at the product state level. Spatiotemporal abstraction reduces the uncertainty and requires less frequent decision-making, allowing the robot to get to goals without inital positional information.

the new leaf nodes. Each distinction is based on (1) a perceptual dimension, which is either an observation or a previous action, and (2) a history index, indicating how far back in the current history the dimension will be examined.

Jonsson and Barto [30] adapted the U-Tree algorithm for use with options and hierarchical learning architectures. Given a finite MDP and a set of options whose policies were not yet defined, they assigned each option a separate U-Tree, which was updated using the U-Tree algorithm (with some modifications) based on each option's local history. What makes the U-Tree algorithm suitable for performing option-specific state abstraction is that a U-Tree simultaneously defines a state representation and a policy over this representation. By assigning one U-Tree to each option, the algorithm is able to perform state abstraction separately for each option while also modifying its policy. It is also possible to use intra-option learning methods (Section 4.1) so that something about an option can be learned from the behavior of other options.

This version of the U-Tree algorithm was illustrated using Dietterich's [14] Taxi task that we briefly described in Section 4.3. Results show that starting with no distinctions being made in the state representation (i.e., all states were represented as a single block), the algorithm was able to solve this task by introducing distinctions that partitioned the state-action set into subsets requiring different action-values, at a significant savings over an approach that initially distinguished between all possible states. Although this example is merely illustrative, it suggests that automated option-specific state abstraction is an attractive approach to making hierarchical learning systems more powerful.

6 Topics For Future Research

6.1 Compact Representations

In most interesting real-world tasks, states have significant internal structure. For example, states are very often represented as vectors of state variables (usually called *factored states* by machine learning researchers), or even possess richer relational structure [17]. Much work in artificial intelligence has focused on exploiting this structure to develop compact representations of single-step actions (e.g., the Dynamic Bayes Net representation [13]). A natural question to consider is how to extend these single-step compact models into compact models of temporally-extended activities, such as options. The problem is a bit subtle since even if all actions have limited single-step influence on state variables, this property generally does not hold over an extended activity. One approach that Rohanimanesh and Mahadevan [56] have been studying is how to exploit results from approximation of structured stochastic processes [6] to develop structured ways of approximating the next-state predictions of temporally-extended activities. The key idea is that by clustering the state variables into disjoint subsets, and keeping track of a next-state distribution for each local cluster, it is possible to efficiently approximate the underlying next-state distribution for a temporally-extended activity. Preliminary analysis of this approach appears promising, and further theoretical and experimental study is under way.

6.2 Learning Task Hierarchies

In the approaches discussed above, the components of the hierarchy, their places in the hierarchy, and the abstractions that are used are decided upon in advance. A key open question is how to form task hierarchies automatically, such as those used in the MAXQ framework. We briefly discussed in Section 4.1 automated methods for identifying useful subtoals [15, 16, 43, 44] which address some aspects of this problem. Another approach called HEXQ was recently proposed by Hengst [24]. It exploits a factored state representation and sorts state variables into an ordered list, beginning with the variable that changes most rapidly. HEXQ builds a task hierarchy, consisting of one level for each state variable, where each level contains a simpler MDP that is connected to other smaller MDPs through a set of "bottleneck" states (for example, if the environment is a set of rooms, as in the navigation example shown in Figure 3 above, each room would coresond to such a smaller MDP). The obvious limitation of HEXQ is that it is limited to considering each state variable in isolation, an approach that fails for more complex problems. Further work is required for understanding how to build task hierarchies in such cases, and how to integrate this approach to related systems approaches such as singular perturbation methods [36, 46].

6.3 Dynamic Abstraction

Systems such as those outlined in this article naturally provide opportunities for using different state representations depending on the activity that is currently executing. There is a crucial distinction between static abstractions, which remain fixed throughout all phases of a sequential decision task, and what we call *dynamic abstractions* that are conditional on the execution of particular temporally-extended activities. In other words, the variables that a dynamic abstraction renders relevant or irrelevant are afforded that status only for a temporally-confined segment of time. For example, during the course of driving, the steering wheel angle, the gear position, or the radio status may all be relevant, but become irrelevant during other activities. Dietterich [14] introduced activity-specific state abstraction in the MAXQ framework, and Jonsson and Barto [30] explored automatic methods for constructing such representation from experience, as described in Section 5.3. The ability to use dynamic abstraction in RL is one of the key reasons that the hierarchical RL approaches discussed in this article appear so attractive to machine learning researchers. This is an area in which future research can have a significant impact.

6.4 Large Applications

Although the proposed ideas for hierarchical RL described above appear promising, to date there has been insufficient experience in experimentally testing the effectiveness of these ideas on large applications. Makar, Mahadevan, and Ghavamzadeh [41] extended the MAXQ approach to multiagent domains, and applied it to a large multi-vehicle autonomous guided vehicle (AGV) routing problem. They demonstrated that the policies learned for this problem were better than standard heuristics used in industry, such as the "go to the nearest free machine" heuristic. Stone and Sutton [68] applied the framework of options to a "keep away" task in Robot soccer. This task involves a set of players from one team passing the ball between them and keeping the ball in their possession against the defending opponents. While these initial studies are promising, much further work is necessary to establish the effectiveness of hierarchical RL, particularly on large complex continuous control tasks.

7 Conclusion

It has been our goal in the article to review several closely related approaches to temporal abstraction and hierarchical control that have been developed by machine learning researchers: the *options* formalism of Sutton, Precup, and Singh, the *hierarchies of abstract machines* (HAMs) approach of Parr and Russell, and Dietterich's MAXQ framework. We also discussed extensions of these ideas addressing on concurrent activities, multiagent coordination, and hierarchical memory for partial observability. Although many of these ideas are closely related to similar concepts in systems and control engineering on hierarchical, hybrid, and multilayer control, we have not attempted to provide a careful rapprochement between these areas and what machine learning, and other researchers associated with artificial intelligence, have been developing. However, we strongly believe that there is much to be gained—on both sides—from such a rapproachement, and it is our hope that this article will prove to be useful in stimulating the needed dialog.

Acknowledgments
The authors would like to thank Anders Jonsson, Khashayar Rohanimanesh, and Mohammad Ghavamzadeh for their crtical reading of an early draft of this article. This work was funded by the National Science Foundation under Grant No. ECS-9980062. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] D. Andre and S. J. Russell. Programmable reinforcement learning agents. In *Advances in Neural Information Processing Systems: Proceedings of the 2000 Conference*, pages 1019–1025, Cambridge, MA, 2001. MIT Press.

[2] A. G. Barto, S. J. Bradtke, and S. P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72:81–138, 1995.

[3] D. Bernstein, S. Zilberstein, and N. Immerman. The complexity of decentralized control of markov decision processes. In *16th Conference on Uncertainty in Artificial Intelligence*, 2000.

[4] D. P. Bertsekas. *Dynamic Programming: Deterministic and Stochastic Models.* Prentice-Hall, Englewood Cliffs, NJ, 1987.

[5] D. P. Bertsekas and J. N. Tsitsiklis. *Neuro-Dynamic Programming.* Athena Scientific, Belmont, MA, 1996.

[6] X. Boyen and D. Koller. Tractable inference for complex stochastic processes. In G. F. Cooper and S. Moral, editors, *Proceedings of the Fourteenth Conference on Uncertainty in AI*, pages 33–42, San Francisco, CA, 1998. Morgan Kaufmann.

[7] S. J. Bradtke and M. O. Duff. Reinforcement learning methods for continuous-time markov decision problems. In G. Tesauro, D. S. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems: Proceedings of the 1994 Conference*, pages 393–400, Cambridge, MA, 1995. MIT Press.

[8] M. S. Branicky, V. S. Borkar, and S. K. Mitter. A unified framework for hybrid control: Model and optimal control theory. *IEEE Transactions on Automatic Control*, 43:31=45, 1998.

[9] R. A. Brooks. Achieving artificial intelligence through building robots. Technical Report A.I. Memo 899, Massachusetts Institute of Technology Artificial Intelligence Laboratory, Cambridge, MA, 1986.

[10] R. H. Crites. *Large-Scale Dynamic Optimization Using Teams of Reinforcement Learning Agents.* PhD thesis, University of Massachusetts, Amherst, MA, 1996.

[11] R. H. Crites and A. G. Barto. Elevator group control using multiple reinforcement learning agents. *Machine Learning*, 33:235–262, 1998.

[12] T. K. Das, A. Gosavi, S. Mahadevan, and N. Marchalleck. Solving semi-markov decision problems using average reward reinforcement learning. *Management Science*, 45:560–574, 1999.

[13] T. L. Dean and K. Kanazawa. A model for reasoning about persistence and causation. *Computational Intelligence*, 5:142–150, 1989.

[14] T. G. Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.

[15] B. Digney. Emergent hierarchical control structures: Learning reactive/hierarchical relationships in reinforcement environments. In P. Meas and M. Mataric, editors, *From Animals to Animats 4: The Fourth Conference on Simulation of Adaptive Behavior.* MIT Press, 1996.

[16] B. Digney. Learning hierarchical control structure from multiple tasks and changing environments. In *From Animals to Animats 5: The Fifth Conference on Simulation of Adaptive Behavior.* MIT Press, 1998.

[17] K. Driessens and S. Dzeroski. Integrating experimentation and guidance in relational reinforcement learning. In *Maching Learning: Proceedings of the Nineteenth International Conference on Machine Learning*, 2002.

[18] R. E. Fikes, P. E. Hart, and N. J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3:251–288, 1972.

[19] S. Fine, Y. Singer, and N. Tishby. The Hierarchical Hidden Markov Model: Analysis and Applications. *Machine Learning*, 32(1), July 1998.

[20] J.-P. Forestier and P. Varaiya. Multilayer control of large markov chains. *IEEE Transactions on Automatic Control*, AC-23:298–304, 1978.

[21] M. Ghavamzadeh and S. Mahadevan. Continuous-time hierarchical reinforcement learning. In *Proceedings of the Eighteenth International Conference on Machine Learning*, 2001.

[22] G. Z. Grudic and L. H. Ungar. Localizing search in reinforcement learning. In *Proceedings of the 18th National Conference on Artificial Intelligence, (AAAI-00)*, pages 590–595, 2000.

[23] D. Harel. Statecharts: A visual formalixm for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[24] B. Hengst. Discovering hierarchy in reinforcement learning with hexq. In *Maching Learning: Proceedings of the Nineteenth International Conference on Machine Learning*, 2002.

[25] N. Hernandez and S. Mahadevan. Hierarchical memory-based reinforcement learning. *Proceedings of Neural Information Processing Systems*, 2001.

[26] R. A. Howard. *Dynamic Probabilistic Systems: Semi-Markov and Decision Processes.* Wiley, NY, 1971.

[27] M. Huber and R. A. Grupen. A feedback control structure for on-line learning tasks. *Robotics and Autonomous Systems*, 22:303–315, 1997.

[28] G. A. Iba. A heuristic approach to the discovery of macro-operators. *Machine Learning*, 3:285–317, 1989.

[29] T. Jaakkola, M. I. Jordan, and S. P. Singh. On the convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, 6:1185–1201, 1994.

[30] A. Jonsson and A. G. Barto. Automated state abstraction for options using the U-tree algorithm. In *Advances in Neural Information Processing Systems: Proceedings of the 2000 Conference*, pages 1054–1060, Cambridge, MA, 2001. MIT Press.

[31] L. Kaelbling, M. Littman, and A. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101, 1998.

[32] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[33] A. H. Klopf. Brain function and adaptive systems—A heterostatic theory. Technical Report AFCRL-72-0164, Air Force Cambridge Research Laboratories, Bedford, MA, 1972. A summary appears in *Proceedings of the International Conference on Systems, Man, and Cybernetics*, 1974, IEEE Systems, Man, and Cybernetics Society, Dallas, TX.

[34] A. H. Klopf. *The Hedonistic Neuron: A Theory of Memory, Learning, and Intelligence.* Hemisphere, Washington, D.C., 1982.

[35] S. Koenig and R. Simmons. Xavier: A robot navigation architecture based on partially observable markov decision process models. In D. Kortenkamp, P. Bonasso, and Murphy. R., editors, *AI-based Mobile Robots: Case-studies of Successful Robot Systems.* MIT Press, 1997.

[36] P. V. Kokotovic, H. K. Khalil, and J. O'Reilly. *Singular Perturbation Methods in Control: Analysis and Design.* Academic Press, London, 1986.

[37] R. E. Korf. *Learning to Solve Problems by Searching for Macro-Operators.* Pitman, Boston, MA, 1985.

[38] M. Littman. Markov games as a framework for multi-agent reinforcement learning. In *Proceedings of the Eleventh International Conference on Machine Learning*, pages 157–163, 1994.

[39] S. Mahadevan. Average reward reinforcement learning: Foundations, algorithms, and empirical results. *Machine Learning*, 22:159–196, 1996.

[40] S. Mahadevan, N. Marchalleck, T. Das, and A. Gosavi. Self-improving factory simulation using continuous-time average-reward reinforcement learning. In *Machine Learning: Proceedings of the Fourteenth International Conference*, 1997.

[41] R. Makar, S. Mahadevan, and M. Ghavamzadeh. Hierarchical multi-agent reinforcement learning. In J. P. Müller, E. Andre, S. Sen, and C. Frasson, editors, *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 246–253, 2001.

[42] A. K. McCallum. *Reinforcement Learning with Selective Perceptioin and Hidden State*. PhD thesis, University of Rochester, 1996.

[43] A. McGovern. *Autonomous Discovery of Temporal Abstractions from Interaction with An Environment*. PhD thesis, University of Massachusetts, 2002.

[44] A. McGovern and A. Barto. Automatic discovery of subgoals in reinforcement learning using diverse density. In C. Brodley and A. Danyluk, editors, *Proceedings of the Eighteenth International Conference on Machine Learning*, pages 361–368, San Francisco, CA, 2001. Morgan Kaufmann.

[45] M. L. Minsky. *Theory of Neural-Analog Reinforcement Systems and its Application to the Brain-Model Problem*. PhD thesis, Princeton University, 1954.

[46] D. S. Naidu. *Singular Perturbation Methodology in Control Systems*. Peter Peregrinus Ltd., London, 1988.

[47] I. Nourbakhsh, R. Powers, and S. Birchfield. Dervish: An office-navigation robot. *AI Magazine*, 16(2):53–60, 1995.

[48] R. Parr. *Hierarchical Control and Learning for Markov Decision Processes*. PhD thesis, University of California, Berkeley, CA, 1998.

[49] R. Parr and S. Russell. Reinforcement learning with hierarchies of machines. In *Advances in Neural Information Processing Systems: Proceedings of the 1997 Conference*, Cambridge, MA, 1998. MIT Press.

[50] T. J. Perkins and A. G. Barto. Lyapunov design for safe reinforcement learning. *Journal of Machine Learning Research*. To appear.

[51] T. J. Perkins and A. G. Barto. Lyapunov-constrained action sets for reinforcement learning. In C. Brodley and A. Danyluk, editors, *Proceedings of the Eighteenth International Conference on Machine Learning*, pages 409–416, San Francisco, CA, 2001. Morgan Kaufmann.

[52] D. Precup. *Temporal Abstraction in Reinforcement Learning*. PhD thesis, University of Massachusetts, Amherst, MA, 2000.

[53] D. Precup and R. S. Sutton. Multi-time models for temporally abstract planning. In *Advances in Neural Information Processing Systems: Proceedings of the 1997 Conference*, pages 1050–1056. MIT Press, Cambridge MA, 1998.

[54] D. Precup, R. S. Sutton, and S. Singh. Theoretical results on reinforcement learning with temporally abstract options. In *Proceedings of the 10th European Conference on Machine Learning, ECML-98*, pages 382–393. Springer Verlag, 1998.

[55] M. L. Puterman. *Markov Decision Problems*. Wiley, NY, 1994.

[56] K. Rohanimanesh and S. Mahadevan. Structured approximation of stochastic temporally extended actions. In preparation.

[57] K. Rohanimanesh and S. Mahadevan. Decision-theoretic planning with concurrent temporally extended actions. In *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence*, 2001.

[58] S. Ross. *Introduction to Stochastic Dynamic Programming*. Academic Press, New York, 1983.

[59] G. A. Rummery and M. Niranjan. On-line q-learning using connectionist systems. Technical Report CUED/F-INFENG/TR 166, Cambridge University Engineering Department, 1994.

[60] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal on Research and Development*, 3:211–229, 1959. Reprinted in E. A. Feigenbaum and J. Feldman, editors, *Computers and Thought*, pp. 71-105, McGraw-Hill, New York, 1963.

[61] A. L. Samuel. Some studies in machine learning using the game of checkers. II—Recent progress. *IBM Journal on Research and Development*, 11:601–617, 1967.

[62] A. Schwartz. A reinforcement learning method for maximizing undiscounted rewards. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 298–305. Morgan Kaufmann, 1993.

[63] Hagit Shatkay and Leslie Pack Kaelbling. Learning topological maps with weak local odometric information. In *IJCAI (2)*, pages 920–929, 1997.

[64] S. Singh and D. Bertsekas. Reinforcement learning for dynamic channel allocation in cellular telephone systems. In *Advances in Neural Information Processing Systems: Proceedings of the 1996 Conference.* MIT Press, Cambridge, MA, 1997.

[65] S. Singh, T. Jaakkola, M. L.. Littman, and C. Szepesvári. Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine Learning*, 38:287–308, 2000.

[66] S.P. Singh. Reinforcement learning with a hierarchy of abstract models. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 202–207, Menlo Park, CA, 1992. AAAI Press/MIT Press.

[67] S.P. Singh. Scaling reinforcement learning algorithms by learning variable temporal resolution models. In *Proceedings of the Ninth International Machine Learning Conference*, pages 406–415, San Mateo, CA, 1992. Morgan Kaufmann.

[68] P. Stone and R. S. Sutton. Scaling reinforcement learning toward RoboCup soccer. In C. Brodley and A. Danyluk, editors, *Proceedings of the Eighteenth International Conference on Machine Learning*, pages 537–544, San Francisco, CA, 2001. Morgan Kaufmann.

[69] T. Sugawara and V. Lesser. Learning to improve coordinated actions in cooperative distributed problem-solving environments. *Machine Learning*, 33:129–154, 1998.

[70] R. S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, editors, *Advances in Neural Information Processing Systems: Proceedings of the 1995 Conference*, pages 1038–1044, Cambridge, MA, 1996. MIT Press.

[71] R. S. Sutton and A. G. Barto. Toward a modern theory of adaptive networks: Expectation and prediction. *Psychological Review*, 88:135–170, 1981.

[72] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, 1998.

[73] R. S. Sutton, D. Precup, and S. Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181–211, 1999.

[74] M. Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 330–337, 1993.

[75] G. J. Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8:257–277, 1992.

[76] G. J. Tesauro. TD–gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(2):215–219, 1994.

[77] G. Theocharous. *Hierarchical Learning and Planning in Partially Observable Markov Decision Processes.* PhD Thesis, Michigan State University, 2002.

[78] G. Theocharous and S. Mahadevan. Approximate planning with hierarchical partially observable markov decision processs for robot navigation. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2002.

[79] G. Theocharous, K. Rohanimanesh, and S. Mahadevan. Learning hierarchical partially observable markov decision processs for robot navigation. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2001.

[80] S. B. Thrun and A. Schwartz. Finding structure in reinforcement learning. In G. Tesauro, D. S. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems: Proceedings of the 1994 Conference*, pages 385–392, Cambridge, MA, 1995. MIT Press.

[81] J. N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42:674–690, 1997.

[82] C. J. C. H. Watkins. *Learning from Delayed Rewards.* PhD thesis, Cambridge University, Cambridge, England, 1989.

[83] C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992.

[84] G. Weiss. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence.* MIT Press, Cambridge, MA., 1999.

[85] P. J. Werbos. Advanced forecasting methods for global crisis warning and models of intelligence. *General Systems Yearbook*, 22:25–38, 1977.

[86] P. J. Werbos. Building and understanding adaptive systems: A statistical/numerical approach to factory automation and brain research. *IEEE Transactions on Systems, Man, and Cybernetics*, 17:7–20, 1987.

[87] P.J. Werbos. Approximate dynamic programming for real-time control and neural modeling. In D. A. White and D. A. Sofge, editors, *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches*, pages 493–525. Van Nostrand Reinhold, New York, 1992.

[88] W. A. Woods. Transition network grammars for natural language analysis. *Communications of the ACM*, 13:591–606, 1970.