Applicability of Reinforcement Learning

Paul E. Utgoff and **Paul R. Cohen**
Department of Computer Science
University of Massachusetts
Amherst, MA 01002
{utgoff|cohen}@cs.umass.edu

Abstract

We describe our experiences in trying to implement a hierarchical reinforcement learning system, and follow with conclusions that we have drawn from the difficulties that we encountered. We present our objectives before we started, the problems we encountered along the way, the solutions we devised for some of these problems, and our conclusions afterward about the class of problems for which reinforcement learning may be suitable. Our experience has made it clearer to us when and when not to select a reinforcement learning method.

Introduction

We describe our experiences in trying to implement a hierarchical reinforcement learning system. This includes our design objectives, the problems we encountered, the solutions we devised for some of these problems, and our conclusions afterward. These conclusions are somewhat lengthy, making the paper fall into two larger parts: the technical aspects of what we tried to do, and the thinking that our difficulties led us to do.

Before proceeding, it is important to circumscribe what we mean by 'reinforcement learning'. To us, this is the class of techniques such as Q-learning and V-learning (temporal difference learning) that modify a Q or V function based on estimates of future discounted reward. This is a narrower definition than is often assumed. For example, Sutton and Barto (in press) say 'Reinforcement learning is defined not by characterizing learning algorithms, but by characterizing a learning problem. Any algorithm that is well suited to solving that problem we consider to be a reinforcement learning algorithm.' Q-learning and V-learning are very useful techniques that can be used in conjunction with other components to build useful learning systems. We prefer to talk about a configuration of components, instead of calling the entire system reinforcement learning.

Related Work

People have long recognized that an acquired skill may depend on previously acquired skills. For example, to be able ride a bicycle, one needs to be able to balance, steer via handlebars, and pedal with the feet. Researchers have begun to study how skills learned through reinforcement learning can be composed, controlled, and learned effectively. This work is still in its infancy, and represents one of the open problems in reinforcement learning (Dietterich, 1997).

Kaelbling (1993) discusses her HDG learning algorithm, which uses landmarks as subgoals in its state space. This helps to organize the state space hierarchically, so that the agent can consider travel in larger chunks by travelling from one landmark to the next.

Thrun and Schwartz (1995) present the SKILLS algorithm, which constructs new operators from useful sequences of existing operators. The usefulness of a sequence is related to how often it is used in a collection of related tasks. Policy learning and operator creation are interleaved.

Tadepalli and Dieterrich (1997) describe their EBRL algorithm for hierarchical explanation-based reinforcement learning. Their system has a two-level hierarchy. The lower level uses a planner to achieve a selected subgoal, and their higher level uses their own version of Q-learning to learn a policy for selecting (and implicity ordering) which subgoal to achieve next.

Design Objectives

The context of our work is that we are attempting to build an autonomous agent (as are a great many other researchers). Part of the system's design is that it will construct simple activities in a recursive manner, producing a logical hierarchy of activities. For each activity, the agent has available the global state vector, a set of subactivities that can be employed while executing the activity, and a goal for the activity. A subactivity is either another activity or a base-level operator that switches an effector. For each activity, the system needs to learn how to achieve the goal for that activity. Said another way, the system needs to learn a policy for each activity, using its available subactivities as its operators, and the stated goal as its objective.

Q-learning (Watkins & Dayan, 1992) seemed to be a natural choice for policy learning in such a scenario, and this is the choice that we made. Based on our experiences as reported below, we would not make this same choice again. One pragmatic reason is that much tinkering is required to get Q-learning to work, and we cannot envision an agent doing this specification and tinkering autonomously. However, there are more fundamental reasons that suggest a more limited scope of useful applicability for reinforcement learning techniques, as discussed below in Section .

When building an autonomous agent, it is essential to give it an embodiment in the real world. We have available a Pioneer 1 robot for this purpose, but we decided to do our initial experiments in a simulated environment, for two reasons. First, a simulated world can be deterministic. Second, a simulated world can run at a speed many orders of magnitude faster than real time, allowing one to test and debug prototypes much more rapidly. We assume that if an algorithm does not work in a simulated environment, then it will not work in a real environment. Hence, our plan was to develop and test the algorithms in a simulated environment, and, when seemingly adequate, transfer the subsequent development process to the robot. Enough problems arose in the simulated world that we have yet to make such a transfer. A related project which tried to implement hierarchical reinforcement learning on the Pioneer, ran into the same difficulties as this one.

We adopted Q-learning because it does not require an operator model. In a robotics application, the successor state that will result from the application of an operator is not entirely predictable. This choice is also in keeping with our longer term design goal that the agent not start with any operator model. One could envision learning an operator model, but we did not do this. Setting up a Q-learning task, and refining the setup so that it works well are known to require considerable manual tinkering. One must specify and refine:

1. the state space
2. the operators
3. the function approximator, possibly with its own parameters
4. the immediate reward function
5. the penultimate reward function, implicitly depending on the goal
6. the stepsize parameter α
7. the discount rate γ
8. the exploration strategy, possibly with its own parameters

This is the state of the art, but from the outset we know that it will be undesirable to adjust these choices manually, both because it violates the autonomy of the agent, and because it will be impractical in a system such as ours with multiple Q-learning tasks. We require the ability to spawn an activity automatically, including the components on which Q-learning is based. How can we formulate a Q-learning task in such a way that manual tuning of the Q-learning specification is not needed?

Attempting An Implementation

We consider first how to specify a Q-learning task in an activity-independent manner, given the world in which the agent will exist, and given the effectors and sensors that are available to the agent in its embodiment. We then define the hierarchical flow of control. The section concludes with an experiment in a simulated world. Although this attempt at an implementation represents a solution of sorts, we encountered several problems that we did not expect, many of which remain unsolved. Some of these problems are indigenous to reinforcement learning approaches.

Activity-Independent Specification

Let us consider the eight components of a Q-learning specification in turn. The state space is the state vector of real values of the various sensors and effector settings. One might implement a mechanism for including additional state variables that are functions of the present or past state values, or higher level interpretations of them. We would rather not do that because ultimately we will want the agent to be able to construct such variables. Instead, we make just one transformation of the state vector, based on the fundamental assumption that the goal associated with an activity is defined as a particular value for each of those state variables that are also designated as goal variable for the activity.

In this view, one can state a single acceptable value for a state variable, not a multitude of values. For a given activity, the state vector $\mathbf{s}$ is transformed to a mapped state vector $\mathbf{s}'$ as follows. For each state variable s_i that is not a goal variable, define s'_i to have the same value as s_i. For each state variable s_i that is a goal variable, construct two mapped state variables s'_{ia} and s'_{ib}. Define s'_{ia} to have the value $s_i - g_i$ if $s_i > g_i$ or 0 otherwise. Also define s'_{ib} to have the value $g_i - s_i$ if $g_i > s_i$ or 0 otherwise. Here, g_i is the goal value for state variable s_i. These mapped variables measure the amount of overshoot and undershoot with respect to a goal value, making the state vector a function of the present state and the goal state.

The operators available for the activity are specified once when the activity is created, and held fixed thereafter. In principle, it is possible that the activity's goal may not be achievable from one or more states, but this is not an issue in the work reported here. In the broader design of the agent, constructing a fatally flawed activity, in this sense, might occur. We envision useless activities becoming unattractive and purged. In any case, this is immaterial here because we guarantee in our setup that this does not occur.

The function approximator is a homegrown algorithm ATI (Adeline Tree Inducer) that fits the observed point estimates with a tree-structured piece-wise linear fit. The approximation starts with a single linear fit, and then breaks any single linear fit that has too much error into two separate linear fits. This is done recursively as necessary to produce finer precision where needed. The split of one block of the partition into two is done obliquely. We dispense with further explanation of the algorithm because in the application here, the tree structured piecewise linear fit always remained a leaf, i.e. a single linear fit. One might want to use a lookup table, with one cell per state, but that is infeasible except for the simplest of domains, so we did not.

The immediate reward function is the amount of the reduction in the distance to the goal state. Another choice might be -1 for the cost of the step, but some steps are better than others, so we preferred to use distance reduction. A potential problem with using distance reduction is that it can be misleading when apparently retrogressive steps are needed in order to achieve a goal, for example in moving around an obstacle, or unstacking a block.

The penultimate reward function is zero. This is seemingly anticlimactic, but it corresponds to distance from the goal. With these choices for immediate and penultimate reward, the Q values can be interpreted as minimum distance to the goal.

The stepsize parameter α is fixed at 0.3. This controls the rate at which old Q values are phased out as new Q-values are phased in. The discount rate γ is fixed at 0.0, which means that the cost of future steps is ignored (fully discounted). This is extreme, and seems to disable much of what Q-learning offers. Non-zero values produced divergence in the Q function values, as described below.

The exploration strategy consists of selecting the action with highest value 98% of the time, and a different action selected at random the remaining 2% of the time. Several versions of softmax action selection were tried, but were not as good in the application.

Simulated World

The simple agent has three degrees of freedom. First, it can move one unit of distance per time step forward or backward along a one-dimensional line. Second, the agent can widen or narrow its gripper aperture by one unit of width per time step. Finally, the agent can raise or lower its arm by one unit of elevation per time step. The gripper is attached to the arm.

There is a block of a fixed width at one end of the travel line. At the beginning of a trial, the agent is placed at a random location on the travel line at least one unit of distance from the block. The gripper is initialized to a random width at least as wide as the block. The arm is initialized to a random elevation. The top-level task is for the agent to raise the block as high as it can. This requires that the agent move to the block, grasp it, and raise its arm. One can easily devise more interesting worlds, but this one is sufficient for a simple test of the hierarchical reinforcement learning system that has been described.

The state variables and their value sets are shown in Table 1. The size of the state space is 11x2x21x26x21=252252, though some states are impossible. For example, the block cannot have a nonzero elevation without being gripped. With a state space this small, one might be tempted to use a lookup table for the function approximator but as discussed above, we did not. The Q function is represented as a piecewise linear fit (tree-structured piece-wise linear fit, but everyone of them remained a single leaf, i.e. linear fit) over its mapped state space, one for each subactivity available to the activity as one of its operators. The resulting Q function is much like a ϕ-machine (Nilsson, 1965) because it partitions the mapped state space into convex regions in which one of the operator (action) values is greater the others.

Table 1: State Variables

Variable Name	Value Set	Cardinality
Gripper Aperture	{0,...,10}	11
Gripping Block	{-1,1}	2
Arm Height	{0,...,20}	21
Distance to Block	{0,...,25}	26
Elevation of Block	{0,...,20}	21

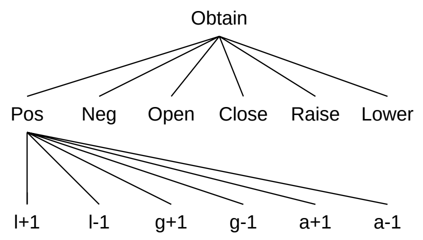

Figure 1: Activity Hierarchy

The activity hierarchy is shown in Figure 1. The lowest level activities are the six primitive operators. For example, there is an effector 'loc+1' that moves the agent one unit of distance in the positive direction. The agent needs to learn that to move in the positive direction, this is the effector to use. This may seem like a superfluous level in the hierarchy, but it represents the connection between effectors and intended action. In this way, the agent learns which effectors have which effects on which state variables. Although only connections from Pos to each of the six effectors are shown, the same connections exist for all the siblings of Pos.

Hierarchical Control

We assume that there is a distinguished root activity that is executed repeatedly. In the larger context, the hierarchy of activities is changing as new activities are created and old activities are deleted. However, for our initial adventure, we elected to construct an activity hierarchy once, and hold it fixed. In Figure 1, one sees that Obtain is the distinguished root activity. The purpose of this setup was to provide a testbed for

$$s_0 \; op_a \; s_1 \; op_a \; s_2 \; op_b \; s_3 \; op_b \; s_4 \; op_b \; s_5 \; op_a \; s_6$$

Figure 2: Discontinuity in an Abstract Trajectory

the Q-learning tasks, given a static hierarchy. Given this hierarchy, and given that the root activity is executed repeatedly, when are the other activities in the hierarchy invoked?

In our initial design, an action was selected at the root, and the subactivity was executed recursively, until a primitive operator was selected, resulting in a primitive step and a state change. Thus, at each time step, a path of activities is executed simultaneously. Each has its own mapped state, Q function, and reward. When an activity finishes executing, control returns to the calling activity. This parent activity then decides what to do next. It may itself return to its caller, or it may continue by selecting and executing one of its subactivities. This looks reasonable because the system considers what to do at each time step, based on the current state, the current Q function for each invoked activity, and stochastic exploration.

However, the system did not learn well at all. One reason is that this strategy for hierarchical control provides little continuity in the state trajectory with respect to the operators (activities). Consider the hypothetical trajectory depicted in Figure 2. There is discontinuity in the trajectory with respect to activity a. It is as though there is an unexplained jump from state s_2 to state s_5. This is very bad for Q learning because Q values need to be grounded in truth, which occurs only when an activity's goal is achieved. Updating Q values in sequences that are not grounded leads to faulty updating.

One needs to remain within an activity, preferably until its goal is achieved, but certainly for long enough that reaching its goal is a possibility. To this end, we revised the control structure so that when an activity is selected for execution, it continues to execute either until its goal is achieved, or until four consecutive operator (subactivity) applications have failed to achieve a new minimum distance from the activity's goal. Distance is measured by the sum of the distances of the current mapped state values to the goal values for the goal variables. Distance reduction is a form of progress estimator (Mataric, 1994). Tsitsiklis and Van Roy (1996) discuss the importance of online sampling more generally.

A Run of the System

The simulated system as described above was run for 2,000 trials. For each trial, the system was initialized to a quasi-random state, and then allowed to run until the goal was achieved. The efficiency of each trial was measured as the ratio of the number of primitive steps used to the minimum number of primitive steps that were needed for that starting state. Optimal efficiency would be 1.0. One would expect to see the efficiency of

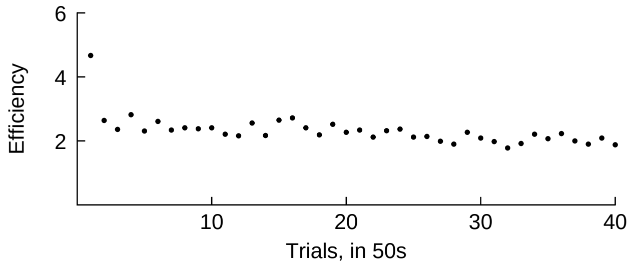

Figure 3: Efficiency Improvement

the agent improve over time, but never achieve optimality due to exploration. Note that a single misstep at a high level can cost many primitive steps. For example, deciding to open the gripper instead of closing it will cost the number of steps to open it plus the number of steps to bring it back to its previous aperture.

Figure 3 shows the learning curve for the 2,000 trials. Each block of 50 trials is shown as a single mean value. The system settles in fairly quickly to an efficiency of approximately 2.0. A more sophisticated exploration strategy might bring further improvement. At the lower level of this hierarchy, the system has learned which effectors to use to succeed in these simpler activities. The system has also learned how to use these middle level activities to succeed at its top level activity. The top level activity is quite simple.

Discussion

Our effort in trying to implement a hierarchical policy learner led us to confront a variety of issues, which led us to think more carefully about the merits of the approach. The following discussion was provoked by our experience, but the thinking and the conclusions that we draw are in a large sense independent. The conclusions follow more from our thinking about Reinforcement Learning than from our experience with our implementation.

Our goal is *activity-independent* learning of activities, and we have concluded that we are expecting too much from a formulation based on a reinforcement learning method such as Q-learning. Indeed, a great many activity-specific specifications must be provided for Q-learning to produce desired behaviors, and one must tinker with these at length to produce a Q-learner that finds a policy to achieve the activity's goal. Above in Section we list eight specifications, and while some might have activity-independent values, e.g. the step-size parameter or the discount rate, others cannot be specified without biasing Q-learning to learn some activities in preference to others. We expand on this point in Section .

Why is it necessary to specify so much, or conversely, why should we not expect to achieve activity-independent Q-learning? The answer we give in Section is that Q-learning is a weak search algorithm, so its search space must be tightly constrained or its search made more directed or both. We argue that strengthening the search in these ways makes Q-learning look more like a planning algorithm. Unfortunately, policy learning lacks some desirable characteristics of planning, notably flexibility in novel situations and explicit representations of action sequences, as we discuss below in Section .

Two Specification Problems

After one has specified a goal, a state space, a set of operators, a function approximator, immediate and penultimate reward functions, discount and stepsize parameters, and an exploration strategy, one has specified tightly an activity for Q-learning to learn. Unfortunately, one generally does not know what policy will be learned given these choices, and more importantly whether the policy will achieve the goal of the task. Tinkering with these specifications to make Q-learning learn any *particular* task is an art. Changing a specification often has unpredictable effects on what will be learned, in part because the specifications are not independent. Some dependencies, such as those between goals and reward functions, are relatively clear, but others become apparent only when a system fails to master the activity that the implementer had in mind. One can think of getting Q-learning to work for a specific task as programming in specification space for the eight (groups of) specifications outlined above.

To illustrate the dependencies between specifications, consider the problem of subgoal ordering. For example, the goal for the Obtain activity is to be gripping the block, and be holding it at as high an elevation as possible. The elevation of the block will be the same as that of the arm when it is being gripped, and zero otherwise. One could just as easily state the goal this way, that the arm should be as high as possible while gripping the block. However, were the goal to be stated this way, the agent would detect progress when raising the arm, whether or not the block was gripped. One cares about the elevation of the arm if it is gripping the block, and not otherwise. This can be modeled by stating the goal in terms of the elevation of the block. This is disconcerting however because now one needs to tinker with the statement of the goal, and potentially the state variables. One might say that the poorly specified subgoal can be used anyway, and that the learned Q function will iron this out. However, this supposes that the goal can be achieved through a random walk, and that the function approximator can represent a function with this aberration.

As another example, suppose that the goal is to fetch the block, i.e. go get it and return to the point of origin. If the goal states that the location should be the point of origin, then moving away from it constitutes negative progress. One can get around this by creating subactivities with useful intermediate goals, but this is another kind of tinkering. In the reinforcement learning community, subgoals are sometimes built into the reward function, but this too is just pushing the problem into a different realm. (It also nullifies the often-heard characterization that one does not need to give any indication of how to solve a task when using reinforcement learning.)

In these examples goals, operators, reward functions, the state space and function approximator, and the exploration strategy must all be considered simultaneously to enable Q-learning to learn a policy for achieving the goal of a particular activity (learning the activity). However, no mapping from these specifications to the policy that will be learned when using them is known, and not all specifications work. To learn a new activity, one must change the configuration of the learning algorithm. To be autonomous, an agent therefore be able to search this specification space. Since no mapping of specifications to policies is known, it will be difficult to search such a space. If we are going to burden the agent with a large search like this, then one must then ask whether that search effort could be spent more effectively elsewhere.

The Weak Search Problem

Much of what is specified for Q-learning is intended to make learning more efficient. For example, function approximators can provide values for unvisited parts of the state space, operator models permit improvement over random exploration by enabling lookahead, the state space itself can be made smaller by recoding continuous state variables such as ‘orientation’ to have discrete, task-specific, meaningful values such as ‘left-of-goal.’ Without such specifications, Q-learning is a weak search algorithm, hence slow.

It is striking to us that for activity learning, at least, these specifications require the same knowledge that a state-space planner needs to formulate plans. Means-ends analysis, for example, requires an operator-difference table, which relates operators to their effects on state variables. In reinforcement-learning parlance, an operator difference table is an *action model.* If Q-learning needs such knowledge to be efficient, might it not be more efficient to learn activities by planning, executing, evaluating and storing the plans that work and their generalizations? Some proponents of reinforcement-learning will argue that this is what they do: use operator models to guide reinforcement-learning, value functions to “store the plans that work,” and function approximators to represent their generalizations. For example, Sutton and Barto (in press) say (ch 9), “(1) all state space planning methods involve computing value functions as a key intermediate step toward improving the policy; and (2) they compute their value functions by backup operations applied to simulated experience.” This is either a very general characterization of planning that encompasses conventional AI methods such as means-ends analysis, or it is a recommendation that planning be implemented by one or another reinforcement learning method. The former interpretation offers no guidance for building planners, and the latter requires a manual search of specification space. A conventional state-space planner can change its plans quickly when the environment changes, whereas Q-learning updates value functions slowly; planners can reason about what to do, whereas policy-followers do not (they react); and plans are explicit representations of activities, whereas activities are implicit in policies.

Our argument to this point is that if one is going to specify a state space, function approximator, and action model, then one is better off building a planner and learning which plans work than relying on Q-learning to learn policies. But there is a deeper argument, related to the specification problems of the previous section: Each specification of an Q-learning algorithm leads to *one* policy, whereas the knowledge inherent in — the state space, action model, etc. — can be used to build a planner that can generate *many* plans, which suggests a better approach to activity-independent learning than Q-learning. To have Q-learning learn a policy to produce a new activity, we have to tinker with the specification, but to have a planner construct a plan to achieve the goal of that activity, we need only change the goal. For instance, suppose we want an agent to learn to approach and pick up a block. This required considerable tinkering with the state space and operator definitions in the experiment we reported above, but a plan to approach and then pick up a block is very easy to generate.

Compilation and Reasoning

Learning a Q function or a value function, V, represents a compilation of experience, possibly with generalization from the function approximator that represents Q or V. Consider the shape of an optimal V^* over the state space. Every goal state represents a peak cost of 0. Every other state has a negative value. There are no local maxima for the nongoal states. With this function, or a good approximation $\hat{V}$, one could hillclimb from any state to a goal. Possession of such functions obviates the need for search of any greater complexity. Given Q, V^*, or $\hat{V}$, who needs planning? Who needs reasoning of any kind? Compilation plus hillclimbing constitutes conditioning, and for some tasks, conditioning is enough.

When is compilation not enough? It takes time to learn a policy for an activity, so if an agent must learn many activities or if the state space or reward function for an activity changes, then planning is a more agile way to produce rewarding actions. In our work here, we tried to tackle the first problem — many activities — by parameterizing the agent's goal so that Q is compiled with respect to a parameterized goal instead of the customary fixed goal. However, this is really just another compilation. This adds to the complexity of the Q or V function, which places an increased burden on the function approximator. In retrospect, this effort has been misguided because it is an attempt to compile even more information, attempting to be free of the need to search and plan in real time.

The state vector for Q-learning comprises a fixed set of components, corresponding to the available sensors and effector settings. This defines statically the state space in which the agent operates. Imagine for a moment an activity in which the agent moves from one (x, y) location to another, without obstruction. Through experience, the agent learns how to move to a goal location when unobstructed. Now suppose that an obstruction is placed between the agent and its goal location. How does this affect the state space?

There are two basic alternatives. In the first, the obstruction is not part of the state space, and the agent simply compiles a Q or V in terms of extra cost when trying to move through the obstruction. To those who equate learning a policy with planning, we say this is at best *lethargic replanning* because the agent must slowly revise its compiled policy by revising its Q or V function through experience and learning. We note that a planner would also be powerless if it could not represent the obstruction. In the second, the obstruction is part of the state space, and the agent learns its Q or V with the location of the obstruction as part of the state space. The agent needs to recondition itself to avoid the obstacle, and *reconditioning is not reasoning*. In contrast, a planner could reason about the obstruction, searching for a way to achieve its goal given the obstruction.

Including the object that constitutes the obstruction in the state vector makes the approximation of Q or V still more complex, but more importantly, it assumes the state space has a static structure. What if there are no obstructions, or two obstructions? There is a strong mismatch in attempting to use a statically structured representation for a space that is dynamically structured. One might choose a representation with slots for objects that may or may not be present, but this is really just a doomed attempt to get beyond a more fundamental mismatch. It is not practical to assume a state space with a fixed constituency, at least for autonomous agents in dynamic spaces. It can be practical for fixed-constituency spaces such as backgammon (Tesauro, 1992). Of course, the present theory of reinforcement learning is not tied to a vector representation of state, though most function approximators are. One does however need constancy at some level of abstraction in order to compile, and our doubts about the desirability of compiling as a method of action selection remain.

Finally, activities represented by plans generally have explicit, declarative structures, so an agent can reason about what it is doing, whereas activities are only implicit in policies and are not available to reasoning. Three kinds of reasoning about activities are particularly important to agents in dynamic environments. We have already discussed dynamic plan modification or replanning, and noted that policies are not easily modified, whereas plans are. A second kind of reasoning is *reasoning about roles*. We want agents to learn activities because we have an *interactionist* theory of category learning; briefly, categories of objects correspond to the roles the objects play in activities. Thus the Pioneer 1 robot might be expected to learn categories of things that move when pushed, things that stall the wheels, and so on. To learn that an object belongs to one category rather than another, one must observe the role of the object in activities. Policies are not explicit representations of activities and state vectors do not necessarily represent objects. It is easier to identify roles of things in explicit plans.

With respect to learning activities, our experience has convinced us that Q learning will do the job if we devote sufficient resources to tinkering with the specifications, if the algorithm is given sufficient time, and if the state space has a static structure. The result will be compromised when the state space changes in any way not anticipated in our state vector and function approximator. Also, the activity will be implicit in the Q function, and not available to reasoning such as replanning. We started this project with considerable optimism about reinforcement learning. Our troubles in applying it as broadly as we had intended seem to be rooted in a fundamental mismatch between our problem and the technology we selected. If it were a matter of our needing to work more diligently to find that elusive activity-independent specification, we would continue in that direction, but the fundamental problems we have encountered tell us that we made a bad initial choice. We now believe agents will learn activities more efficiently if the activities are generated by planning, evaluated in execution, modified by replanning, and generalized by well-understood inductive methods.

We are reworking the control architecture for our autonomous agent. Wherever a reinforcement learning method is applicable, we shall use it. However, the class of problems for which reinforcement learning would seem to be the method of choice is smaller than we thought when we selected it initially. We expect that a simple planner will form the backbone of the system, rather than a hierarchy of activities in which each policy is learned to the point of compilation. Reasoning via planning is good, and there is no need to drive decision making by compilation alone. It was our own mistake to anticipate that this would work well. Our state representation will not be a state vector, but will instead be a list of state elements that grows and shrinks depending on the complexity and constitution of the present environment, and the extent of the abstractions that can be applied dynamically.

Acknowledgements This research is supported by DARPA/Air Force Research laboratory under contract No. F30602-97-1-0289. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements either expressed or implied, of DARPA/AFRL or the U.S. Government.

References

Dietterich, T. G. (1997). Machine learning research. *AI Magazine* (pp. 97-136).

Kaelbling, L. P. (1993). Hierarchical learning in stochastic domains: Preliminary results. *Proceedings of the Tenth International Conference on Machine Learning* (pp. 167-173). Morgan Kaufmann.

Mataric, M. J. (1994). Reward functions for accelerated learning. *Machine Learning: Proceedings of the Eleventh International Conference* (pp. 181-189). New Brunswick, NJ: Morgan Kaufmann.

Nilsson, N. J. (1965). *Learning machines.* New York: McGraw-Hill.

Sutton, R. S., & Barto, A. G. (in press). *Reinforcement learning: An introduction.* MIT Press.

Tadepalli, P., & Dietterich, T. G. (1997). Hierarchical explanation-based reinforcement learning. *Machine Learning: Proceedings of the Fourteenth International Conference* (pp. 358-366). Nashville, TN: Morgan Kaufmann.

Tesauro, G. (1992). Practical issues in temporal difference learning. *Machine Learning, 8*, 257-277.

Thrun, S., & Schwartz, A. (1995). Finding structure in reinforcement learning. In Tesauro, Touretzky & Leen (Eds.), *Advances in Neural Information Processing Systems.* San Mateo, CA: Morgan Kaufman.

Tsitsiklis, J. N. , & Van Roy, B. (1996). *An analysis of temporal-difference learning with function approximation*, (Technical report LIDS-P-2322), Cambridge, MA: MIT.

Watkins, C.J.C.H., & Dayan, P. (1992). Q-Learning. *Machine Learning, 8*, 279-292.