Learning State Features from Policies to Bias Exploration in Reinforcement Learning

Bryan Singer and Manuela Veloso
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213
Email: {bsinger+, mmv+}@cs.cmu.edu

Abstract

When successful on a single problem, reinforcement learning converges to an optimal policy that maps each state to the best action to take in that state. Unfortunately, this policy learning does not provide the reason *why* an action should be chosen at a state. In other words, policy learning in general does not map particular *state features* to particular actions. Therefore, it is infeasible to directly apply the learned policy to new problems that may have the same state features. However, if we use reinforcement learning to solve many individual problems in a specific domain, we can accumulate many policy functions, which can constitute a valuable source of information about the common state features. In this paper, we present our approach and results exploring this research line. For a particular domain, we gather the policies learned by reinforcement learning in a variety of problems. We characterize each state with a set of features and use the policy values learned from featured-states as training examples to a classifier. The learned classifier represents a generalization over all the seen problems, mapping state features to actions. Policy learning in new complex problems is then improved by using the output of the classifier to bias exploration. We present different bias techniques and show empirical results that validate our approach. We demonstrate that state features are effectively learned from learned policies and that reinforcement learning in new problems explores fewer states by using the learned classifier than if no exploration bias is used.

Keywords: reinforcement learning, state generalization, exploration
Email address of contact author: bsinger+@cs.cmu.edu
Phone number of contact author: (412) 268-3562

1 Introduction

Consider learning to optimally solve many different problems in a particular domain. The domain may be specified by parameterized high level constraints such as the types and effects of the actions that are available (deterministic or non-deterministic). A problem specifies a set of states, initial states, a reward function, the possible actions in each state, and the outcomes of taking each possible action in each state. For example, in a maze domain, all problems might consist of a set of either open or blocked grid locations, a goal location, and actions moving in one of four directions where trying to move into a blocked grid location results in being left in the current location while all other moves succeed. A specific problem in the maze domain would then be a specific grid layout, indicating exactly which cells are open and which are blocked, and a specific goal location.

Given a specific problem in a domain, a standard reinforcement learner learns the optimal policy specifying the best action to take in each possible state for this specific problem.

For each new problem, learning will have to start from scratch because each problem may have a different set of states and rewards. This has been recognized as rather unfortunate and several approaches have been and are being investigated to find structure, abstraction, generalization, and/or policy reuse in reinforcement learning.

Our work presented in this paper contributes a technique within this line of research. We build upon the assumption that the solution to a specific problem may contain some information about solving any problem within the domain. Our algorithm aims at extracting those domain-specific invariants.

Classical reinforcement learning algorithms in their policy learning do not provide the reason *why* an action should be chosen at a particular state. In fact a state may be viewed as simply an entity. However, many domains may have a natural set of features that are attributable to each state. For example, in a maze domain where states are cells in the maze, some natural features for each state are its cell position and whether each of the adjacent cell locations are blocked or open.

Our work focusses on domains for which there are *local* features to each state that may correlate directly with the actions to explore independent of the particular problem we are trying to solve. In this way our features are "action-dependent" state features [8]. For example, in all problems in the maze domain, there is no usefulness to trying to move into a blocked location since that leaves the agent where it was. Thus, whether each of the adjacent cell locations are blocked or open is a set of local features correlating directly with the actions worth exploring.

While a plain state numbering based on location, as commonly used by a reinforcement learner, uniquely identifies each state, these local features need not uniquely determine a single state. For example, in the maze domain there may be many locations where all adjacent locations are open. These state features have the potential therefore to represent state generalizations.

The goal of our work is to learn the map from state features to actions worth exploring so that learning can be more effective in new problems. Our approach consists of directly using the learned policies from a variety of individual problems in a specific domain as training examples for a state-feature classifier. In new problems, the learned classifier then can be used to bias exploration in reinforcement learning, reducing the the number of states visited.

The paper is organized as follows. Section 2 describes how the map from state features into actions is learned. We characterize each state with a set of features. We identify and use the policy values learned from featured-states as training examples to a classifier. The learned classifier represents a generalization over all the seen problems as a map from state features to actions. Section 3 then introduces how policy learning in new unseen complex problems is then improved by using the output of the classifier to bias exploration. Section 4 shows empirical results that demonstrate that state features are effectively learned from learned policies and that reinforcement learning in new problems explores fewer states by using the learned classifier than if no exploration bias is used. Section 5 discusses our approach by comparing it to other related approaches.

2 Learning State Features

Our algorithm requires a domain for which there are local features predefined for each state, a set of sufficiently simple problems, and a reinforcement learner that solves these problems. It then uses the policies learned for these problems as training examples to learn a map from local state features to actions. The map indentifies the subset of the local state features that correlate with the actions that are worth exploring in states with the same features. This learned knowledge can then provide the reinforcement learner with an exploration bias that may allow it to solve other, potentially more difficult, problems within the domain more effectively than a reinforcement learner would if starting from scratch. Specifically, we would like to learn this knowledge by designing a classifier that maps local state features to actions worth exploring using previously learned policies as training examples. We present now the different aspects of our approach.

Let P_D be the set of all problems in a domain D, let A_D be the set of possible actions in this domain, and let S_D be the set of all possible states. Let $\{P_1, \ldots, P_n\} \subset P_D$ be some set of problems, all of which a reinforcement learner RL is able to solve. Each problem P_i contains a set of states $S_i \subset S_D$. The reinforcement learner takes as its input a problem P_i and outputs a policy $\pi_i \colon S_i \rightarrow A_D$ which maps each state within the problem to what it has found to be the best action to take in that state.

2.1 Local State Features

Since each problem may have a completely different set of states, it is impossible to *directly* use the learned policy for one problem to help solve another problem. Instead, we need some method of describing states that would allow us to recognize "similar" states between problems. The difficulty is finding an operational definition of "similar" that allows us to learn to solve new problems more efficiently.

The policy that is learned for each problem is dependent on several factors. At one level, the policy's choice of the best action to take in a particular state is dependent upon what rewards are available and what action sequences produce what state sequences. The specific action choice in each state may depend on the complete setup of the problem, making it potentially difficult to abstract this knowledge to other problems. At another level, the policy is dependent upon the constraints placed on the problem by the domain itself. For example, in the maze domain, trying to move into a blocked location always causes the agent to remain in its current state; thus, a good policy for any problem in the maze domain would not specify an action in a state that would mean trying to move into a blocked location. The problem-independent domain constraints may make some actions always poor choices in some situations and thus make the remaining actions the only reasonable ones in these situations.

It is in general infeasible to analyze why each action specified in each policy was the best action to take. Instead we focus on learning some of the domain constraints that represent good local action choices. We want our learner to capture the domain invariants that are not dependent therefore on a specific complete problem. So, our algorithm learns what actions are never worth exploring in certain situations instead of trying to infer what the best action is to take in each state of a new problem. It is conceivable that there may be some domains in which these invariants do not exist, i.e., in which there are no domain wide constraints that limit the set of reasonable actions to take in any state; however, many domains such as the simple maze domain do have a number of domain wide constraints that limit the set of reasonable actions to take at least in some states.

We aim at transferring action selection knowledge among "similar states," i.e., states that due to domain wide constraints have the same sets of actions which are reasonable or always poor. Note that by this definition there may be many similar states which have different best actions, but each of these best actions must be actions that according to the problem-independent domain constraints are reasonable (that is, actions that are not always a poor choice in the given situation). Further, there may be many similar states within the same problem.

Unfortunately, for many domains we may not have a model that describes how the domain limits the set of reasonable actions or how to recognize states that are similar by the definition above. However, we may be able to describe each state with some set of features which we believe may give some indication about what are reasonable or what are poor actions to choose. From the discussion above, we would like these features to be easily determined for each state without needing to consider the entire problem; this is why we call them "local state features." The features should be common to the domain so that they can be determined in all of the problems.

Ideally, these local state features would group all similar states together while separating all those that are not similar. Realistically, the local state features may separate similar states into several groups while grouping other states that are not similar together. As long as the features generate at least some groupings whose set of reasonable actions are not all possible actions, then we should gain from using these local state features.

So, for the domain D we wish to choose a set of features $\langle f_1, \ldots, f_m \rangle$ such that there is an easily computable function $F_D\colon S_D \times P_D \to \langle f_1, \ldots, f_m \rangle$ that given a particular state in a specific problem generates the local state features for that state.

2.2 Generating Examples

Now that we have a set of local state features that hopefully groups similar states together, we need a method for determining the set of reasonable actions for states having each of the possible feature vectors. To do this we will use the previously learned policies for problems the reinforcement learner has already solved. For each state in each problem, the policies give what the reinforcement learner found to be the best action to take in that state. This best action is thus one of the reasonable actions for states having the same local state features as the given state. So we have a method for generating training examples of reasonable actions:

$Examples \leftarrow \emptyset$
For each P_i in $\{P_1, \ldots, P_n\}$
    For each s in S_i
        Add $\langle F_D(s, P_i), \pi_i(s) \rangle$ to the set of *Examples*

As an example in the maze domain, consider the maze configuration shown in Figure 1. Suppose that a reinforcement learner converged to the policy indicated in the figure by the arrows (this is just one of several reasonable policies for this problem). If we defined our feature vector to have whether each of the neighboring locations were blocked or open in the order of <North_Loc, East_Loc, South_Loc, West_Loc>, then the examples generated would be:

```
S1: < <blocked, open,    open,    blocked>, Move_East  >
S2: < <blocked, open,    open,    open>,    Move_East  >
S3: < <blocked, open,    blocked, open>,    Move_East  >
S4: < <blocked, open,    open,    open>,    Move_South >
S5: < <blocked, blocked, open,    open>,    Move_South >
S6: < <open,    open,    blocked, blocked>, Move_East  >
S7: < <open,    blocked, blocked, open>,    Move_North >
S8: < <open,    blocked, blocked, open>,    Move_West  >
```

Note that for any given feature vector, the set of *Examples* is not guaranteed to contain examples that label the feature vector with every reasonable action. If the total number of different feature vectors is small in comparison to the total number of states in all of the problems and if the problems are sufficiently varied, we may hope to avoid this problem.

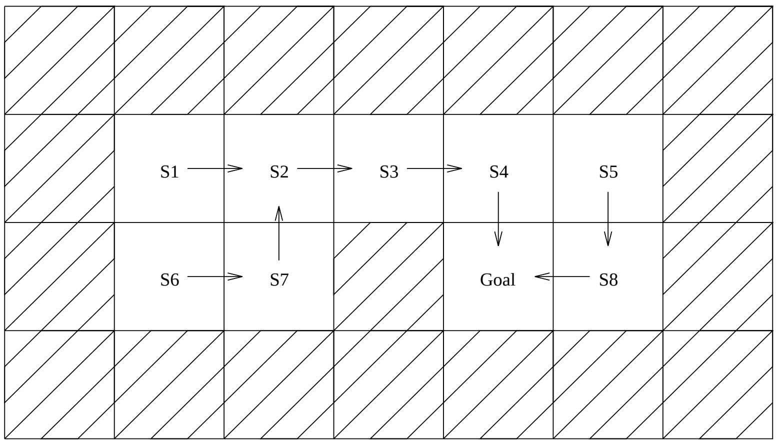

Figure 1: Example maze configuration with open and blocked cells. The arrows indicate one possible policy.

2.3 Building a Classifier

Now that we have a set of *Examples* built, the next step is to transform it into a form that allows us to easily decide what reasonable actions should be explored in states of new problems. Considering that the set of *Examples* may contain some errors, a simple approach is to use a standard machine learning classification algorithm. However, there is one additional constraint that we must place on the classification algorithm. Each example labels the feature vector with a single reasonable action, but we want the resulting classifier to return all reasonable actions for each feature vector. So, we want a classification algorithm CA that given the set of *Examples* produces a classifier $c\colon \langle f_1, \ldots, f_m \rangle \to 2^{A_D}$ where the classifier c takes any feature vector and returns the set of reasonable actions.

As an example, our current implementation uses the Iterative Tree Induction (ITI) algorithm [13] to build a decision tree. This tree induction algorithm maintains a list of all of the classifications seen at each leaf. The decision tree then provides an easy method for determining the set of reasonable actions for a state in a new problem.

Note that if the set of *Examples* is noisy or does not totally cover every feature vector, then the resulting classifier may also contain errors. The set of actions that the classifier returns may contain some actions which are always poor in states with the given state feature vector and may not contain some

$BuildClassifier($ reinforcement learner RL, problems $\{P_1, \ldots, P_n\}$,
local state feature extractor F_D, classification algorithm CA)
For each problem P_i in $\{P_1, \ldots, P_n\}$
Let $\pi_i \leftarrow RL(P_i)$
Let $Examples \leftarrow \emptyset$
For each problem P_i in $\{P_1, \ldots, P_n\}$
For each state s in S_i
Add $\langle F_D(s, P_i), \pi_i(s) \rangle$ to the set of $Examples$
Let classifier $c \leftarrow CA(Examples)$
Return c

Table 1: Algorithm for Building the Classifier

actions which are reasonable. The algorithm developed so far is outlined in Table 1.

3 Efficiently Solving New Problems

Suppose we have built a classifier c from a set of problems $\{P_1, \ldots, P_n\}$ and then are asked to solved another problem P_{n+1} from the same domain. We run the reinforcement learner RL on the problem, but this time we have some additional information to help decide which action to explore next. In particular, when we are in state s, we call $F_D(s, P_{n+1})$ to determine the local state features of state s. These local state features can be passed to the classifier c to determine the set of reasonable actions for state s. So, we have the following function:

$FindReasonableActions($ state s, problem p, classifier c,
local state feature extractor F_D)
Return $c(F_D(s, p))$

The reinforcement learner RL must have some method for determining which action out of all possible actions A_D to explore. If we are fairly confident that the classifier always returns at least all of the reasonable actions for each of the feature vectors, then we could simply have the reinforcement learner choose amongst the actions that the classifier returns instead of amongst all possible actions. If the set of actions returned by the classifier for some of the feature vectors is smaller than the total number of possible actions, then this can result in a considerable reduction of the space that needs to be explored.

On the other hand, if we are not confident that the classifier always returns at least all of the reasonable actions for each feature vector, we may not want to limit the reinforcement learner to choosing only actions returned by the classifier. If the classifier does not return all of the reasonable actions, then there may be some states in new problems from which an action not returned by the classifier must be taken in order to find the solution. If the classifier returns most of the reasonable actions most of the time, then we could simply bias the reinforcement learner to choose actions returned by the classifier over choosing all other actions. A weight W is fixed based on how likely the classifier has seen examples of every reasonable action for every feature vector. During exploration, an action is selected at random favoring actions returned by the classifier W to 1 over all other actions.

4 Empirical Results

While this is on-going research, we have gathered some very encouraging results in the simple maze domain. The maze domain we used is a rectangular grid world where each grid location is either open or blocked. There is one special goal location; a reward of 1 is associated with any action taking the agent into this goal location and all other actions have a reward of 0. There are four deterministic actions (Move_North, Move_South, Move_East, and Move_West) that take the agent one grid location in that direction if the destination grid location is open, and leave the agent in its current location if the destination grid location is blocked. Once the agent enters the goal location, the episode is completed.

The reinforcement learner we used was a standard Q learner with random exploration. For the local state features, we decided to use whether each of the four neighboring locations were blocked or open. Thus, we hoped that our algorithm would be able to learn that moving into blocked locations is always a poor choice, but our algorithm would not help decide which of several open locations were the best to take. So, we hoped to be able to take previous solutions to mazes and extract from it the rule "do not try to move into a blocked wall" which would allow the Q learner to greatly limit its search on future mazes. On new mazes, the Q learner would be able to explore the open locations without wasting time relearning for each blocked location that it is not beneficial to try to move there.

We used eight simple mazes for training. Four mazes were ten by ten completely open grids with a wall of blocked locations enclosing these open locations. A goal location was placed near one of the four corners for each of the mazes. By placing a goal near each of four corners, we hoped to prevent any bias for preferring actions of one direction over another because of the goal location. The four remaining mazes where fifteen by nine mazes with a few blocked locations placed in the middle. The blocked locations were designed to provide examples of dead ends and very short hallways. Again, the goal for each maze was placed near one of the four corners. For each episode, the starting location for the agent was chosen randomly amongst the open locations.

For the classification algorithm, we decided to use a decision tree induction algorithm. The exact tree that the ITI algorithm learned is shown in Figure 2. Consider, for example, the far right leaf in the tree; the only action not present at this leaf is Move_East since the East location is the only neighboring location that is blocked.

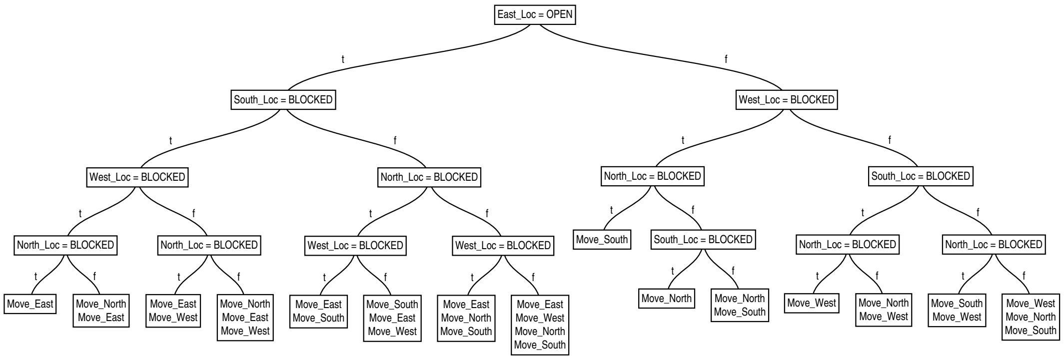

Figure 2: Decision tree *learned* in the maze domain

Finding the right balance between exploration and exploitation in reinforcement learning is often a difficult task. Since our algorithm is intended to help bias exploration, we avoided the issue of finding this balance between exploration and exploitation during testing by simply applying exploration without any exploitation. So, we tested how quickly exploration found the goal location instead of how quickly the reinforcement learner converged. In the maze domain, finding the goal more quickly will also allow the reinforcement learner to converge to a solution more quickly.

Our results are shown in Table 2. Each number in the table represents the number of moves made before finding the goal, averaged over 10,000 independent episodes. The column labeled "Random" corresponds to using just a purely random exploration strategy with every move being equally likely to be chosen at any state. The column labeled "In Tree" corresponds to biasing the exploration strategy to choose randomly, with equal likelihood, amongst the actions that the decision tree returns for the current state. The final two columns correspond to biasing the exploration strategy to favor actions that the decision tree returns for the current state either 16 to 1 or 2 to 1 over all other actions.

Grid	Random	In Tree	16:1	2:1
Open 8X8	213	189	188	197
Open 16X16	1,290	1,240	1,213	1,260
Open 32X32	7,021	6,697	6,742	7,007
Hall length 8	114	49	52	81
Hall length 16	479	226	240	354
Hall length 32	1,966	961	1,014	1,483
Hall length 64	8,052	3,904	4,178	6,206
Hall length 128	31,608	15,750	18,463	26,350
Hall length 256	130,297	64,480	77,296	105,607
Restricted 15X10	2,462	1,233	1,320	1,785
Mostly Open 20X20	10,032	7,796	7,933	8,656
Restricted 20X20	19,936	9,950	10,753	15,364
Mixed 20X40	42,276	29,866	30,776	35,827

Table 2: Average number of moves before finding the goal in the maze domain

The first three grids labeled as "Open" are grids of the specified size of all open space except for a border of blocked states around the perimeter. Not surprisingly, using one of our biasing techniques only slightly improves the performance for these grids. This is due to the fact that most states in these grids have all open neighboring locations and thus all actions are reasonable in most states. The only advantage our algorithm has is when the agent is next to the perimeter of blocked states since these are the only states where some actions are not worth exploring.

The next group of grids labeled as "Hall" are grids consisting of just one long hallway that is one location wide and the specified number of locations long. Some of the grids in this group also have a number of turns that the hallway makes. With these grids, there is a very significant improvement that all of our biasing methods make over the naive random exploration method, in some cases cutting exploration time by half. Since in a hallway there are two blocked adjacent locations, there are only two reasonable actions for each state. So, our algorithm provides a bias for exploration that reduces the number of actions tried at each state and thus greatly reduces the number of actions taken before reaching the goal.

The last group of grids give a little more of a mix between the two previous groups. The grids labeled as "Restricted" have a large number of blocked locations, but have a lot more open areas and a lot more decision points than the hallway grids. The grid labeled as "Mostly Open" consists of a lot of open grid locations with some blocked locations partitioning the open locations into a number of areas. The grid labeled as "Mixed" has an area that is mostly restricted and an area that is mostly open. Once again, our algorithm shows considerable improvement over the naive random exploration strategy.

Since the decision tree that was learned was without noise, it is not surprising that considering only actions returned by the decision tree is a better bias than simply favoring such actions over all others. Further, the stronger bias of favoring actions returned by the decision tree 16 to 1 over all other actions outperforms the weaker bias of 2 to 1. Had the training data contained some noise that was also reflected in the decision tree, then a weaker bias may have outperformed the stronger ones since in some situations it may have been necessary to take actions not returned by the decision tree.

5 Discussion

In other efforts in the area of life-long learning (e.g., [11]) solutions to earlier related tasks are used to make solving the next related task easier. Our algorithm also achieves this same goal. It uses solutions to several problems in the form of learned policies to build a classifier that helps bias exploration in new problems in such a way that avoids or reduces the selection of actions that are not worth exploring. Some of the previous life-long learning algorithms for reinforcement learning have required that at least some of the states in the previous tasks also appear in the new task. This is a restriction that our algorithm does not impose; in fact, our algorithm was designed with the idea that there are many related tasks that should benefit from previous solutions even though the state space may be totally different.

State abstraction or generalization is a common method for reducing the search space in reinforcement learning (e.g., [1, 3, 4, 5, 9, 14]). In an equivalent way to state abstraction, our algorithm groups states together with a common property which in our algorithm is having the same set of reasonable actions. However, our algorithm runs the reinforcement learner on the entire set of states instead of reducing all states grouped together into one state. The grouping is used to bias the selection of actions in each state. So, our algorithm does not gain from a reduction in the number of states that state abstraction usually provides, but our algorithm does not suffer from not being able to choose different actions in states that have been grouped together.

Action abstraction and macro-operator methods (e.g., [7, 12]) learn a sequence of actions which are applicable as a unit in a set of states. While these methods try to learn what *single* sequence of actions *should* be taken in a set of states, our algorithm tries to find what actions are *not* worth exploring and thus a *set* of states worth exploring. Our algorithm has significant flexibility as to what can be learned and when this learned knowledge can be applied, but requires the reinforcement learner to still find the best action to take in every state.

Hierarchical abstraction (e.g., [2, 6, 10]) is a method of decomposing a task into smaller and smaller subtasks. While the method is very appealing, several suggested algorithms within the context of reinforcement learning require that the hierarchy be hand designed. Although not hierarchical in nature, our algorithm does use an extra source of knowledge in the form of a learned classifier. Unlike the often hand designed hierarchies, our algorithm automatically *learns* the classifier that it uses to guide exploration.

Acknowledgements

Special thanks to Sebastian Thrun for several initial discussions that started us down this path of research.

This research is sponsored in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under agreement number F30602-97-2-0250. 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 the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory (AFRL) or the U.S. Government. The first author, Bryan Singer, is partly supported by a National Science Foundation Graduate Fellowship.

References

[1] Justin A. Boyan and Andrew W. Moore. Generalization in reinforcement learning: Safely approximating the value function. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7, pages 369–376. The MIT Press, 1995.

[2] Peter Dayan and Geoffrey E. Hinton. Feudal reinforcement learning. In Stephen José Hanson, Jack D. Cowan, and C. Lee Giles, editors, *Advances in Neural Information Processing Systems*, volume 5, pages 271–278. Morgan Kaufmann, San Mateo, CA, 1993.

[3] Thomas Dean and Shieu-Hong Lin. Decomposition techniques for planning in stochastic domains. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, 1995.

[4] Andrew Kachites McCallum. *Reinforcement Learning with Selective Perception and Hidden State*. Phd. thesis, Department of Computer Science, University of Rochester, Rochester, NY, 1995.

[5] Andrew W. Moore and Christopher G. Atkeson. The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces. *Machine Learning*, 21:199, 1995.

[6] Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. In *Advances in Neural Information Processing Systems*, volume 10. The MIT Press, 1998. In press.

[7] Doina Precup, Richard S. Sutton, and Satinder P. Singh. Planning with closed-loop macro actions. In *Working notes of the 1997 AAAI Fall Symposium on Model-directed Autonomous Systems*, 1997. In press.

[8] Peter Stone and Manuela Veloso. Using learned state features in multi-agent reinforcement learning. In *Submitted to ICML-98*, 1998.

[9] Richard S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In David S. Touretzky, Michael C. Mozer, and Michael E. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8, pages 1038–1044. The MIT Press, 1996.

[10] Prasad Tadepalli and Thomas G. Dietterich. Hierarchical explanation-based reinforcement learning. In *Proceedings of the Fourteenth International Conference on Machine Learning*, pages 358–366. Morgan Kaufmann Publishers, San Francisco, 1997.

[11] Sebastian Thrun and Lorien Pratt, editors. *Learning to Learn.* Kluwer Academic Publisher, 1997.

[12] Sebastian Thrun and Anton Schwartz. Finding structure in reinforcement learning. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7, pages 385–392. The MIT Press, 1995.

[13] Paul E. Utgoff, Neil C. Berkman, and Jeffery A. Clouse. Decision tree induction based on efficient tree restructuring. *Machine Learning*, 29:5, 1997.

[14] William T. B. Uther and Manuela M. Veloso. Tree based discretization for continuous state space reinforcement learning. In *Submitted to AAAI-98*, 1998.