Learning State Features from Policies to Bias Exploration in Reinforcement Learning

Bryan Singer Manuela Veloso

April, 1999
CMU-CS-99-122

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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 first author, Bryan Singer, is partly supported by a National Science Foundation Graduate Fellowship.

Keywords: machine learning, reinforcement learning

Abstract

When given several problems to solve in some domain, a standard reinforcement learner learns an optimal policy from scratch for each problem. If the domain has particular characteristics that are goal and problem independent, the learner might be able to take advantage of previously solved problems. Unfortunately, it is generally infeasible to directly apply a learned policy to new problems. This paper presents a method to bias exploration through previous problem solutions, which is shown to speed up learning on new problems. We first allow a Q-learner to learn the optimal policies for several problems. We describe each state in terms of local features, assuming that these state features together with the learned policies can be used to abstract out the domain characteristics from the specific layout of states and rewards in a particular problem. We then use a classifier to learn this abstraction by using training examples extracted from each learned Q-table. The trained classifier maps state features to the potentially goal-independent successful actions in the domain. Given a new problem, we include the output of the classifier as an exploration bias to improve the rate of convergence of the reinforcement learner. We have validated our approach empirically. In this paper, we report results within the complex domain Sokoban which we introduce.

1 Introduction

Consider learning to optimally solve many different Markov Decision Problems (MDPs) in a particular domain. The domain may be specified by 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 state transition function. 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. 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 starts 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 (e.g., [4, 14, 5, 8, 1, 11]).

The work presented in this paper contributes a technique within this line of research. It builds upon the assumption that the solutions to a series of problems contain biasing information about selecting actions to more efficiently solve new problems within the domain. The algorithm presented here aims at extracting these 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 the agent's cell position and whether each of the adjacent cell locations are blocked or open.

This work focusses on domains for which there are *local state features* that may correlate directly with the actions to explore independent of the particular problem trying to be solved.

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 a maze there may be many locations where all immediately adjacent locations to the agent are open. Therefore, these state features have the potential to represent state generalizations.

This paper contributes an *experience-biased* reinforcement learning algorithm to learn a mapping from state features to actions worth exploring so that learning can be more effective in new problems. Our approach consists of 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, our algorithm uses the learned classifier to bias exploration in reinforcement learning, reducing the number of states visited and speeding up learning.

2 Experience-Biased Reinforcement Learning

There are two assumptions required by our learning approach: (i) a domain D with local state features predefined for each state, and (ii) a set of sufficiently simple MDPs, $\{P_1, \ldots, P_n\}$.

The first step of our approach consists of building action classifiers from reinforcement learning experience. Table 1 presents the top-level view of this first step of our method. The problems $\{P_i\}, i = 1, \ldots, n$ are given to a Q-learner to be solved. The resulting entries in the Q-tables are used to generate training examples of positive and negative uses of an action as a function of local state features of each state in the Q-table. Then a set of classifiers are trained for each action to learn a mapping from local state features to actions worth exploring in states with the same features.

Let *Examples* $\leftarrow \emptyset$
For each problem P_i in $\{P_1, \ldots, P_n\}$
Run the Q-learner on P_i saving the Q-table
Let *Examples* $\leftarrow$ *Examples* $\cup$ Generate_Examples(P_i, Q-table)
For each action a available in the domain
Train a classifier c_a on the subset of *Examples* pertaining to a.

Table 1: Top-Level Algorithm for Building Classifiers

The second step of our approach consists of using the learned classifiers to provide an exploration bias to the reinforcement learner. The goal is to allow other, potentially more difficult, problems within the domain to be solved more effectively than a reinforcement learner would if starting from scratch.

This section presents the details of our approach.

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 in another problem. Instead, if we describe the states with local state features, it may be possible to abstract information from the policy that is problem and goal independent. Since many domains may have many possible features that describe states, we now briefly discuss what may constitute good local state features.

The learned policy for each problem depends on several factors, including the reward function, and what action sequences produce what state sequences. The specific action choice in each state may depend on the complete setup of the problem. Furthermore, the policy is dependent upon the constraints placed on the problem by the domain itself. The problem-independent domain constraints may make some actions always lead to failure in some situations, making the remaining actions the only reasonable ones to explore.

Thus, good local state features would be able to capture the differences between states based on problem-independent domain constraints.

In order for the features to generalize between problems, they should not require analyzing the entire problem. That is, the features should not uniquely determine the exact state, but rather some portion of the state relevant to capturing domain constraints. This is why we use the name "*local* state features". Thus, many states within a problem and between problems could be described by the same set of feature values.

So, for the domain D, our goal is 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 from the Learned Q-table

For each state in a learned Q-table, there is an action or a set of actions that has the highest Q-value, which constitutes the optimal policy. Thus, for any other state with the same set of local state feature values in another problem, this action may be one worth exploring. On the other hand, if a particular action in some state has an unchanged value (e.g. 0), then this means that the Q-learner did not find any way to reach one of the goal states after taking that action in that state, assuming that the only non-zero rewards are given for actions moving the agent into one of the goal states. For this particular problem, taking that action in that state lead to a dead-end. Thus, for any other state with the same set of feature values in another problem, this action may not be worth exploring.

Based on this insight, we introduce the algorithm in Table 2 for extracting positive and negative *training examples* of whether a particular action should or should not be explored.

Let us add a few comments about this algorithm: First, when running the Q-learner on a problem to generate the Q-table, the local state features are *not* used. The Q-learning on the simple training problems uses only a standard state numbering. The local state features are only used when generating the state-action training examples. This distinction is important in that two states' local state feature values may be the same while the particular best action to take in each of the states is different. Second, if a very small negative reward is given for *every* action except actions moving the agent into a goal state, then the above algorithm is still valid if the line "If $Q(s,a) = 0$" is changed to "If $Q(s,a) < 0$" in the Negative_Example function. That is, under this new reward structure, actions leading to dead-ends will have negative values while actions eventually leading to a goal state will have positive values. Note that these negative rewards must be smaller in absolute value than the discounted value of reaching the goal along any optimal path. Third, as a matter of practicality, a Q-learner is often not allowed to fully converge all of its Q-values, especially those that it has not found to lead to a goal state. To try to avoid generating

```
Generate_Examples(P_i, Q-table)
    Let Examples be the empty set
    For each state s in problem P_i
        For each action a available in state s
            If Negative_Example(a, s, Q-table)
                Add ⟨F_D(s, P_i), ¬a⟩ to Examples
            else if Positive_Example(a, s, Q-table)
                Add ⟨F_D(s, P_i), a⟩ to Examples

Negative_Example(a, s, Q-table)
    If Q(s, a) = 0
        return True
    else
        return False

Positive_Example(a, s, Q-table)
    If Q(s, a) = max_a' Q(s, a')
        return True
    else
        return False
```

Table 2: Generating Examples from Q-tables

many noisy examples, only states that have been visited at least some specified number of times are used.

2.3 Building a Set of Classifiers

The set *Examples* (see Table 2) contains instances with local state features as attributes and with classifications of either taking or not taking a particular action.

To make this information useful for new problems, it needs to be condensed into a form that can quickly tell for any state what actions are worth exploring and which may not be. Further, since not every possible set of local state feature values may have been seen, some generalization from the data would be desirable.

The one difficulty of using a standard machine learning classification algorithm is that for any state the desired output is whether each of the actions should or should not be taken, not a single value. One easy solution to this difficulty is to train a separate classifier for each action, with its output indicating whether its particular action is worth exploring or not.

We learn a set of classifiers, where each takes local state feature values and maps them to whether each action is worth exploring or not, based on the experiences from the data generated from the Q-tables learned in the training problems.

2.4 Biasing Exploration

The final step of our approach consists of actually using the set of classifiers to bias the reinforcement learner's exploration strategy in new problems.

When a reinforcement learner is trying to decide what action to take, it assigns a selection probability, $P_t(a, s)$, to each action a available in the current state s at time t (with $\sum_a P_t(a, s) = 1$). This probability distribution often is based on the learner's exploration strategy and the current Q-values. The reinforcement learner then randomly chooses an action to take according to this probability distribution.

We incorporate the learned classifiers to provide a new bias that alters this probability distribution.

Assume that the classifiers only return whether each action should or should not be taken according to the training data. Then we would like to *prefer*, i.e., give higher probability weight to those actions that should be taken according to the classifiers.

Formally, let us define for all actions a and states s, a weight function w:

$$w(a, s) = \begin{cases} w_{pos} \text{ if } c_a(s) = a \\ w_{neg} \text{ if } c_a(s) = \neg a \end{cases}$$

where w_{pos} and w_{neg} are predefined large and small numbers, respectively for when the classifier returns that the action should or should not be taken.

However, the classifiers can also return a *confidence value* along with their classification. The algorithm can then use this confidence to weight the different actions. So, for all actions a and states s let

$$w(a, s) = f(c_a(s)),$$

where the classifier, c_a, now returns a value directly proportional to its confidence value, and where f is any non-decreasing non-negative function (possibly the identity function).

Then we introduce a *classifier-based-only* exploration strategy which is based solely on the learned classifiers. An action a is selected with probability:

$$\frac{w(a, s)}{\sum_{a'} w(a', s)},$$

i.e., the reinforcement learner weights each action by its $w(a, s)$ value.

In our general approach, the learned classifiers are "combined" with any built-in exploration strategy (e.g. Boltzmann exploration). Let $P_t(a, s)$ capture this existing exploration bias. We introduce the *combined-classifier-built-in* exploration strategy, in which an action a is chosen randomly with probability:

$$\frac{w(a, s) \cdot P_t(a, s)}{\sum_{a'} w(a', s) \cdot P_t(a', s)},$$

i.e., the reinforcement learner weights the existing probabilities by the new $w(a, s)$ values derived from the classifier.

As a special case, if $w_{pos} = 1$ and $w_{neg} = 0$, then only actions that are to be taken according to the classifier are ever actually taken (all the rejected actions by the classifier are weighted by $w_{neg} = 0$). Potentially, this can greatly prune the search space, but it requires high accuracy in the classifiers to guarantee the optimal policy is still learned.

3 Implementation

We have fully implemented our experience-based reinforcement learning approach and tested it in several grid domains. We used classical Q-learning on the training problems.

Initially, in a simple maze world, we used a decision tree as the state-action classifier. The decision tree provided a bias for learning new mazes that significantly reduced search time.

In this paper, we report on the implementation in a complex grid world, namely the Sokoban domain. [1] We use a neural network for the state-action classifiers with a standard back-propagation algorithm.

In this section, we introduce the Sokoban game and the experimental setup that supports the empirical results.

3.1 Sokoban

Sokoban is an interesting domain of puzzles which falls in the general class of motion planning problems with movable obstacles [15]. The object of the puzzle is for an agent in a grid world to move all of the balls so that each is located on a destination grid cell. Each grid location is either open or blocked by a wall. An open grid cell may:

- contain a single ball,
- contain the agent if it does not contain a ball,
- be a destination location.

The agent:

- has at most four deterministic actions available to it: moving North, East, South, or West;
- may not move into a blocked location;
- pushes a ball if it moves to an adjacent cell that has a ball; i.e., if the agent is at position x, y and a ball is north of it at position $x, y+1$, then, when the agent moves north, the new agent's and new ball's location are $x, y+1$ and $x, y+2$;
- may not push a ball into a wall;

[1] See for example http://xsokoban.lcs.mit.edu/xsokoban.html.

Because the agent can only push balls and not pull them, there are many situations in which a ball can become stuck or can have a limited set of cells it can be moved to. For example, a ball pushed into a corner is not retrievable, and a ball pushed onto a wall can only be moved along that wall unless the wall comes to an end before reaching a corner. Thus, many rather small mazes can be surprisingly difficult to solve. Being able to avoid these stuck positions could greatly limit the search space. It should be noted that many stuck or limited range situations are not easily detectable.

3.2 Experimental Setup

A state is defined by the location of the agent and the location of each of the balls. A small negative reward is given for each move, and a large reward is given for reaching any state that has all of the balls on destination locations. Since it is possible to reach states from which the goal is not achievable and since this situation is not easily detectable, a maximum number of moves is specified at which time the problem is reset to its initial configuration.

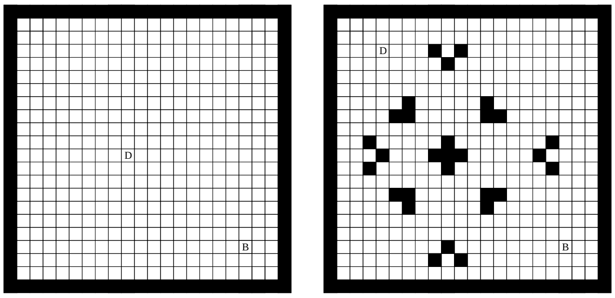
Local state features are chosen to describe all of the grid locations immediately surrounding the agent up to a radius of r cells away from the agent. Note that we must use at least $r = 3$ in order to tell if a move will push a ball next to a wall or into a corner. In the results presented below, we used $r = 4$. Each of these local state locations are described in terms of three boolean attributes: *is-blocked, has-ball, is-destination.*

Note that a grid world has inherent symmetry built in. For example, a move East in some grid is equivalent to a move North in a 90^o counter-clockwise rotated grid. Thus, through the use of rotations, any example generated that corresponds to taking or not taking an action other than North, can be transformed into an example for taking or not taking a move northward. Therefore we generate a single classifier for the North action. We use a standard back-propagation neural network. Its inputs are the local state features ($r = 4$) and its output indicates whether moving North is worth exploring or not for such generalized input state.

When solving a new problem, the state feature values for the current state and each of its rotations are presented to the classifier in turn to determine whether each of the four actions are worth exploring or not.

4 Empirical Results

For the first set of tests in the Sokoban domain, we began by running a Q-learner on a set of 12 simple puzzles. Each puzzle contained only a single ball and a single destination location and consisted of all open locations except a wall of blocked locations around the perimeter. The amount of open space changed in different puzzles, with an equal number of grids of sizes 5×5, 10×10, and 20×20 (see Figure 1(a)).



(a) (b)

Figure 1: Two Example Sokoban Grids (D=destination location, B=ball, agent not shown)

After the Q-learner solved these training problems, the neural network was trained on the resulting data. Then we tested with a set of similar mazes, of sizes 8×8 and 16×16.

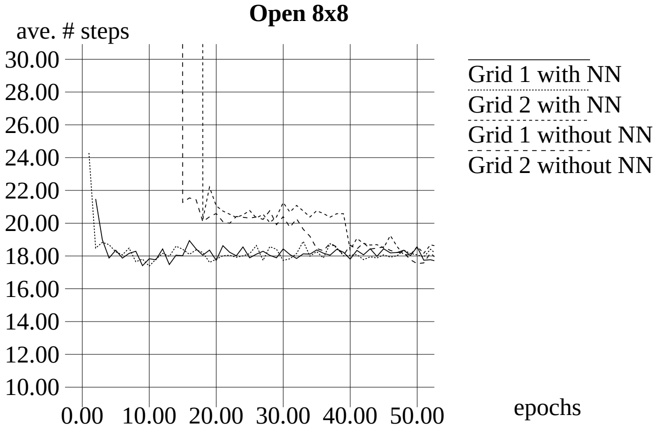
For comparison, the Q-learner was run on these testing mazes with and without using the learned neural network. The experience-based reinforcement learner used the same built-in exploration bias as the standard Q-learner and used the classifier with $w_{pos} = 1$ and $w_{neg} = 0$.

An "epoch" was defined to be one set of actions that lead to the goal state or a set of 10,000 moves, whichever came first. For every epoch that the Q-learner was allowed to run, a test was run on the Q-table that had been learned up to that point. In particular, this test consisted of fixing the Q-table and allowing the agent to follow the policy defined by the current Q-table (in the case that several actions were equally maximum valued, an action was randomly selected with uniform probability among all such actions). This was done 100 times, again allowing the agent up to 10,000 moves each time. Of the times the agent reached the goal, the average number of moves before reaching the goal was noted.

We have run a large number of empirical tests with successful results. Figure 2 shows the results for two different sized grids. Each graph shows the results for two different placements of the ball and destination location. The y-axis is the average number of steps before a goal state is reached. If upon any test, none of the hundred runs is successful in reaching the goal, then no data is plotted. The placement of the agent is random with each test, hence the average number of steps in these graphs may not be constant even after the Q-table has converged, which accounts for expected graph oscillations.

As shown in the graphs, the learner converges significantly faster when using the learned neural network. In all cases, the experience-based reinforcement learner also converged to a policy that was no worse or better than the standard learner without using the neural network as an exploration bias.

As a second test, a Q-learner was run on a set of 12 more complex grids of size 20×20 containing a number of blocked locations. We used a single ball



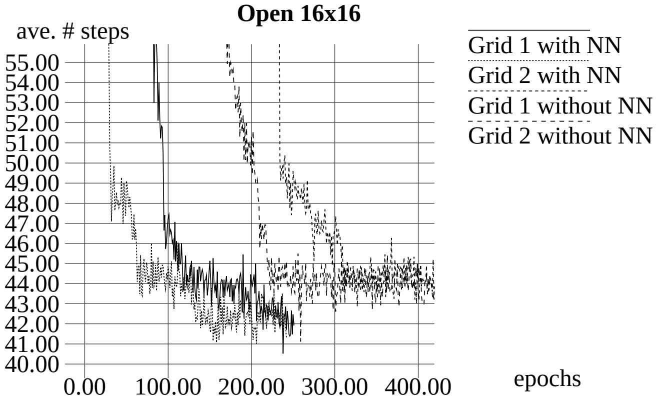


Figure 2: Tests on Open Grids

(see Figure 1(b) for an example grid).

The examples that were collected on these grids were combined with the examples generated from the open grids described above. This combined set of examples was then used to train a new neural network. Two grids of size eight by eight and containing some blocked locations were used as test problems. The results are shown in Figure 3 and are similar to the results obtained for the open grids.

Notice that in every test, the Q-learner using the learned neural network was able to find a good policy in less than half the number of epochs that the unbiased Q-learner required. This indicates that the neural network was able to provide a bias that significantly helped improve the performance on solving new problems.

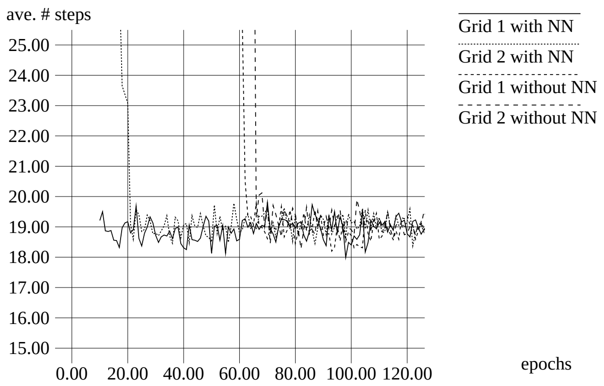


Figure 3: Tests on 8x8 Grids with Blocked Locations

5 Related Work

In other efforts in the area of life-long learning (e.g., [13]), solutions to earlier related tasks are used to make solving the next related task easier. The algorithm discussed here 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 the algorithm in this paper does not impose; in fact, it 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., [2, 4, 6, 7, 11]). In an equivalent way to state abstraction, the algorithm presented here groups states together that have the same local state feature values. However, the algorithm runs the reinforcement learner on the entire set of states instead of reducing all states with the same feature values together into one state. The local state features are used to help bias the selection of actions in each state but are not used to change the actual states the reinforcement learner sees. So, the algorithm does not gain from a reduction in the number of states that state abstraction usually provides, but it 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., [9, 14, 10]) 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, the algorithm here tries to find what actions are *not* worth exploring and thus a *set* of actions worth exploring. The 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., [3, 8, 12]) 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, the algorithm in this paper does use an extra source of knowledge in the form of a learned classifier. Unlike the often hand designed hierarchies, the algorithm automatically *learns* the classifier that it uses to guide exploration.

6 Conclusion

In this paper, we presented our work towards the improvement of the efficiency of reinforcement learning in complex problems, as a function of previously solved simple problems. The idea is that different problems in the same domain can share domain invariants, if these exist.

Our experience-based reinforcement learning contributes the following steps: first a standard reinforcement learner solves simple problems; then local state features are introduced to be used to generate state-action training examples combined with the result of the initial learning phase; classifiers are built which capture the domain invariant state-action mappings; finally, the classifiers are integrated as an exploration bias for new episodes of reinforcement learning in more complex problems.

We have reported empirical results within the Sokoban domain to validate this algorithm.

Acknowledgements

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

References

[1] Michael Bowling and Manuela Veloso. Bounding the suboptimality of reusing subproblems. In *Proceedings of the NIPS Workshop on Abstraction in Reinforcement Learning*, December 1998.

[2] 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.

[3] 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.

[4] 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.

[5] M. Hauskrecht, N. Meuleau, L. P. Kaelbling, T. Dean, and C. Boutilier. Hierarchical solution of markov decision processes using macro-actions. In *Proceedings of the Fourteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-98)*, 1998.

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

[7] 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.

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

[9] 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.

[10] Jette Randløv. Learning macro-actions in reinforcement learning. In M. S. Kearns, S. A. Solla, and D. A. Cohn, editors, *Advances in Neural Information Processing Systems*, volume 11. The MIT Press, 1999. In Press.

[11] 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.

[12] 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.

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

[14] 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.

[15] G. Wilfong. Motion planning in the presence of moving obstacles. In *Proceedings of the Fourth Annual ACM Symposium on Computational Geometry*, 1988.