Tree-Based Policy Learning in Continuous Domains through Teaching by Demonstration

Sonia Chernova and **Manuela Veloso**
Computer Science Department,
Carnegie Mellon University,
Pittsburgh, PA, USA
{soniac, veloso}@cs.cmu.edu

Abstract

This paper addresses the problem of reinforcement learning in continuous domains through teaching by demonstration. Our approach is based on the Continuous U-Tree algorithm, which generates a tree-based discretization of a continuous state space while applying general reinforcement learning techniques. We introduce a method for generating a preliminary state discretization and policy from expert demonstration in the form of a decision tree. This discretization is used to bootstrap the Continuous U-Tree algorithm and guide the autonomous learning process. In our experiments, we show how a small number of demonstration trials provided by an expert can significantly reduce the number of trials required to learn an optimal policy, resulting in a significant improvement in both learning efficiency and state space size.

Introduction

Reinforcement learning is a machine learning framework in which an agent explores its environment through a series of actions, and in return receives reward from the environment (Kaelbling, Littman, & Moore 1996). The goal of the agent is to find a policy mapping states to actions that will maximize its cumulative reward over time. This type of problem is typically formalized as a Markov Decision Process (MDP) (Howard 1960), with discrete timesteps and a finite number of states and actions.

The above formulation is widely used in the field of robotics, but its application leads to two considerable challenges. The first is that instead of discrete states, our world is more naturally represented as a continuous, multidimensional state space. The second is that the required number of learning trials can often be prohibitively large for execution on real robotic systems.

To address the problem of continuous state spaces, a number of discretization techniques have been developed that split the space into a smaller number of general states. Known as variable resolution methods, these algorithms generate non-uniform states where each discrete state region generalizes over some number of similar real-world states. Examples of such algorithms include the Parti-game algorithm (Moore & Atkeson 1995), the Continuous U-Tree algorithm (Uther & Veloso 1998), VQQL (Fernandez & Borrajo 2000), and Variable Resolution Discretization (VRD) (Munos & Moore 2002).

Additionally, a significant amount of research has focused on the challenge of reducing the number of trials required for learning. A powerful technique explored over the years has been teaching by demonstration, or apprenticeship learning (Kaiser & Dillmann 1996; Atkeson & Schaal 1997; Chen & McCarragher 2000; Smart & Kaelbling 2002; Nicolescu & Mataric 2003; Abbeel & Ng 2004). In this approach, human demonstration is used in combination with autonomous learning techniques to reduce the learning time.

In this paper, we present a general framework designed for high level, behavior-based control in continuous domains, that naturally combines variable resolution discretization techniques with teaching by demonstration. We validate our approach using the Continuous U-Tree discretization algorithm, an extension of the original U-Tree algorithm for discrete state spaces (McCallum 1996). We show how a small number of demonstrations effectively reduces the learning time of the algorithm, as well as the size of the final state space.

The Continuous U-Tree Algorithm

The Continuous U-Tree algorithm (Uther & Veloso 1998) is a variable-resolution discretization method for reinforcement learning in continuous state spaces. The algorithm can be applied to a mixture of continuous and ordered discrete state values, and allows the application of any discrete-state reinforcement learning algorithm to continuous domains.

The Continuous U-Tree algorithm relies on two distinct but related concepts of state. The first type of state is the current state of the agent in the environment, which we refer to as the *sensory input*. The sensory input is characterized by the observation o, a vector of continuous sensory attributes. The second type of state relates to the discretization generated by the algorithm that is used to form an action policy. We use the term *state* to refer specifically to these discrete states, and use $L(o)$ to represent the state associated with the sensory input o.

Each state typically generalizes over multiple sensory inputs, and each sensory input can be classified into one of the states using the *state tree*. The state tree is a binary tree where each leaf node represents a single state. Each decision node in the tree is used to classify sensory inputs by splitting on one of the sensory input attributes. At the beginning, the state tree is composed of a single leaf node which represents the entire state space. The algorithm recursively grows the tree as it iterates between two distinct phases: data gathering and expansion.

Interactions between the agent and the environment are recorded as a tuple of observations, actions and rewards. Each action a belongs to a finite set of actions A, while reward values r and observation attribute values in o can be fully continuous. Each of the agent's steps, or transitions, is recorded by the four-tuple (o, a, r, o'), where o is the starting sensory observation, a is the action performed by the agent, o' is the resulting sensory observation, and r is the reward received from the environment.

Table 1 summarizes the Continuous U-Tree learning process. The algorithm begins with a single state describing the entire state space. During the gathering phase the agent accumulates experience and records the transition tuples. The discretization is then updated by splitting the leaf nodes of the state tree during the expansion phase. The split location within a given state is determined by calculating the expected future discounted reward $q(s, a)$ of each transition tuple using equation 2. The algorithm considers each sensory attribute in turn, sorting all of the transition tuples by this attribute. The transition tuples are then repeatedly divided into two sets by performing a temporary split between each two consecutive pairs of tuples. The expected reward values $q(s, a)$ of each set are then compared using the Kolmogorov-Smirnov (KS) statistical test. The trial split that results in the largest difference between the two distributions is then tested using the *stopping criterion*.

The stopping criterion is a rule used to determine when to stop splitting the tree. The Continuous U-Tree stopping criterion states that the algorithm should only split when a statistically significant difference exists between the datasets formed by the split. In our experiments we define statistical significance at the $P = 0.01$ level for the KS test. States in which the best trial split does not pass the stopping criterion are not split.

The Continuous U-Tree algorithm also maintains an MDP over the discretization, which is updated after every split. Let $T(s, a)$ represent the set of all transition tuples, (o, a, r, o'), associated with state s and the action a. The state transition function $Pr(s, a, s')$ and the expected future discounted reward function $R(s, a)$ are estimated from the recorded transition tuples using the following formulas:

$$Pr(s, a, s') = \frac{|\forall(o, a, r, o') \in T(s, a) s.t. L(o') = s'|}{|T(s, a)|} \quad (3)$$

$$R(s, a) = \frac{\sum_{(o,a,r,o') \in T(s,a)} r}{|T(s, a)|} \quad (4)$$

Using this data, any standard discrete reinforcement learning algorithm can be applied to find the Q function $Q(s, a)$ and the value function $V(s)$ using the standard Bellman Equations (Bellman 1957):

$$Q(s, a) \leftarrow R(s, a) + \gamma(Pr(s'|s, a)V(s') \quad (5)$$

Table 1: The Continuous U-Tree Algorithm.

```
• Initialization
  – The algorithm begins with a single state representing
    the entire state space. The tree has only one node.
  – The transition history is empty.
• Data Gathering Phase
  – Determine current state $s$
  – Select action $a$ to perform:
    ∗ With probability $\varepsilon$ the agent explores by selecting a
      random action.
    ∗ Otherwise, select $a$ based on the expected reward as-
      sociated with the actions leading from state $s$.
          $a = argmax_{a' \in A} Q(s, a')$        (1)
  – Perform action $a$ and store the transition tuple ($o$,$a$,$o'$,$r$)
    in leaf $L(o)$
  – Update $Pr(s, a, s')$, $R(s, a)$ and $Q(s, a)$
• Expansion Phase
  – For every state (leaf node):
    ∗ For all datapoints in this state, update expected reward
      value:
          $q(o, a) = r + \gamma(V(L(o')))$        (2)
    ∗ Find the best split using the Kolmogorov-Smirnov test
    ∗ If the split satisfies the splitting criteria:
      · Perform split.
      · Update $Pr(s, a, s')$ and $R(s, a)$ function using the
        recorded transition tuples.
      · Solve the MDP to find $Q(s, a)$ and $V(s)$.
```

$$V(s) = max_a Q(s, a) \quad (6)$$

In our implementation we have used Prioritized Sweeping (Moore & Atkeson 1993) for learning the policy.

Combining Teaching and the Continuous U-Tree Algorithm

In this section, we present a method for generating a supplementary state discretization using expert task demonstration. We then show how this discretization can be used to bootstrap the Continuous U-Tree learning process to reduce both the number of learning trials and the size of the resulting tree.

The agent is controlled using behavioral primitives (Arkin 1998), defined as basic actions that can be combined together to perform the overall task. The goal of the system is to learn a policy over these behavioral primitives.

In our approach we use a demonstration technique called *learning by experienced demonstrations* (Nicolescu & Mataric 2003), in which the agent is fully under the expert's control while continuing to experience the task through its own sensors. We assume the expert attempts to perform the task optimally, without necessarily succeeding; our goal is not to reproduce the exact behavior of the expert, but instead to learn the task itself with the expert's guidance. Additionally, this approach can be extended to learning from observing the actions of other agents.

The Expert Demonstration

During the task demonstration, the expert fully controls the agent's action while observing its progress. The expert's perception is limited to watching the agent, and does not rely on the numerical sensor values that the agent perceives. The same set of actions is available to the expert and the agent for both the demonstration and autonomous learning phases. During the demonstration, the agent is able to perceive the sensory input, the selected action, and the reward, allowing it to record the transition tuples as if it was performing the task on its own. All experience tuples are recorded throughout the demonstration.

Upon the completion of the demonstration, the recorded transition tuples are used to generate a classification tree, called the *expert tree*, by applying the C4.5 algorithm (Quinlan 1993). The algorithm's training data uses the initial sensory observation of each tuple, o, as the input, and the action selected by the expert, a, as the associated label. As a result, the algorithm learns a mapping from the agent's observations to the actions selected by the expert. Each decision node of the expert tree encodes a binary split on one of the sensory attributes, and each leaf node specifies the action to be performed for all observations leading to this leaf.

The amount of training data required to generate the expert tree varies depending on the complexity of the domain and the level of noise in the sensory inputs. For simple domains, a small number of demonstration trials can result in a decision tree that can execute the task with nearly optimal performance. In these cases it may seem sufficient to simply use the decision tree as the control policy, but this approach would perform poorly in dynamic environments (Atkeson & Schaal 1997). For complex domains, a significant number of demonstrations may be required to obtain optimal performance, turning the demonstration task into a tedious burden for the expert. In the following sections we demonstrate that although nearly optimal expert performance leads to the most significant improvement, the performance of the algorithm degrades gracefully for inaccurate or insufficient expert data.

Bootstrapping The U-Tree Algorithm

The expert tree provides a good approximate discretization, but requires labeled data to further refine or adjust the tree in response to environment changes. The state tree of the Continuous U-Tree algorithm, on the other hand, generates its state discretization from unlabeled training data, but typically takes a large number of trials, and generates trees with a state space far larger than necessary even under a strict stopping criterion. The matching structure of the two trees enables us to combine the two discretizations to take advantage of the strengths of both approaches.

We modify the initialization phase of the Continuous U-Tree algorithm by replacing the single state tree with the expert tree structure, and initializing the transition tuple history to include all of the tuples recorded during the demonstration phase. The remainder of the learning phase remains unchanged and proceeds normally by alternating the gathering and expansion phases and recursively growing the tree starting from the expert tree base. The combined tree is referred to as the Continuous Expert U-Tree, or *EU-Tree*.

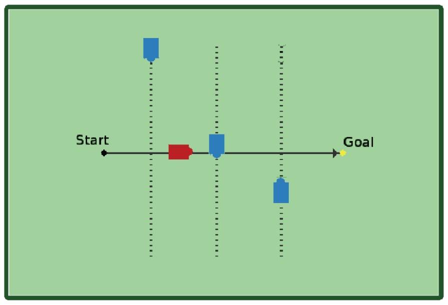

Figure 1: The 3-Evasion domain. The learning agent moves along a straight horizontal path from the Start to the Goal. Opponent agents move back and forth along their respective vertical paths marked by the dotted lines.

The benefits of this approach are twofold. The base discretization provided by the expert tree provides a good starting point for the U-Tree algorithm and helps guide further splitting, while the transition tuples recoded from the demonstration data are used to initialize the Q-table of the reinforcement learning algorithm. The demonstration experience typically highlights the positive areas of the state space, especially the goal areas, which helps guide further exploration. Since the negative areas of the state space are often avoided by the expert, the examples do not always highlight some of the main danger zones, and as a result the demonstration examples alone are not sufficient for learning a good policy.

Experimental Results

The N-Evasion Domain

To test the algorithm we introduce the N-Evasion domain. The domain environment consists of a single learning agent and N opponent agents. The learning agent's task is to move from the starting location to the goal location, using one of two available actions - *stop* and *go*, where *go* always moves the agent directly towards the goal. Each opponent agent traverses its own path, which intersects the main path of the learning agent at some location between the start and goal. The learning agent must reach the goal without colliding with any opponent agents. A small amount of noise is added to the forward movements of all of the agents to simulate uncertainty. An example of the domain for $N = 3$ can be seen in Figure 1.

The N-Evasion domain can provide a number of interesting challenges as various factors affect the difficulty of the problem. In addition to the value of N, the frequency with which an opponent agent intersects the main path also affects the difficulty. This factor can be controlled by changing the path length or speed of the opponent. The problem is simpler if the opponent only rarely crosses the path of the learning agent as collision becomes less likely. Another possible factor is the spacing between the opponent agents. In our example the opponent agents are spaced far enough apart to allow the learning agent to stop in between them in order to time its approach. The domain can also be designed without this space to force the agent to consider multiple opponents simultaneously.

The representation for the N-Evasion domain can be encoded in a variety of ways. In our implementation, it is represented using $N + 1$ state dimensions - one for the position of the learning agent along its path, and one for each of the N opponent agents' positions along their paths (direction of movement can be ignored).

In our experiments we use the 3-Evasion domain shown in Figure 1, as well as a similar 2-Evasion domain. The learning agent's path is 30 meters long, and its velocity is fixed at 2.0 m/sec. Opponent agents have a fixed velocity of 3.2 m/sec, and travel paths 22 meters in length that are directly bisected by the main path. Each trial begins with the learning agent at the start location, and ends when the goal is reached, a collision occurs, or the time allocated for the trial expires. The agent receives a reward of 100 for successfully completing the trial and reaching the goal, -50 for a collision, and -2 each time it executes the action *stop*. The performance of the agent is evaluated using the percentage of successful traversals from start to goal without any collisions.

Algorithm Performance Comparison

The performance of the proposed algorithm was tested using the 2-Evasion and 3-Evasion domains. During the demonstration phase, the agent was controlled by the human expert in the simulated environment. The demonstration phase consisted of 15 trials, which took approximately five minutes to generate. In both domain experiments the resulting expert tree policy performed perfectly in its respective domain, completing all trials without collisions.

In our analysis, we compare the performance of the following three learning methods:

- the original Continuous U-Tree algorithm (C. U-Tree)
- the EU-Tree without the demonstration transition history (EU-Tree - DT)
- the complete EU-Tree, including the demonstration transition history (EU-Tree)

Each of the above algorithms was tested in 15 experimental learning runs. During each experiment, learning was halted at 20-trial intervals for evaluation, during which the state of the algorithm was frozen and 100 evaluation trials were performed using the latest policy. Figures 2 and 3 present the results, averaged over all experimental runs, for the 2-Evasion and 3-Evasion domains respectively.

In both cases, we see a very significant improvement in performance of the teaching-based methods over the original U-Tree algorithm. While the Continuous U-Tree algorithm is able to learn the correct policy in both trial domains, it takes far longer to reach the optimal policy, one that avoids all collisions, than the other methods. In both domains, the EU-Tree methods reduce the learning time by over 50%.

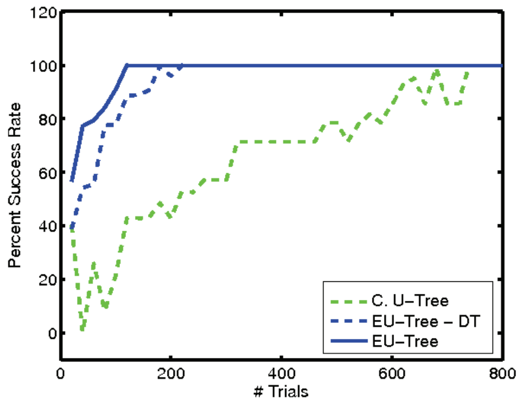

Figure 2: Performance evaluation of three learning methods in the 2-Evasion domain.

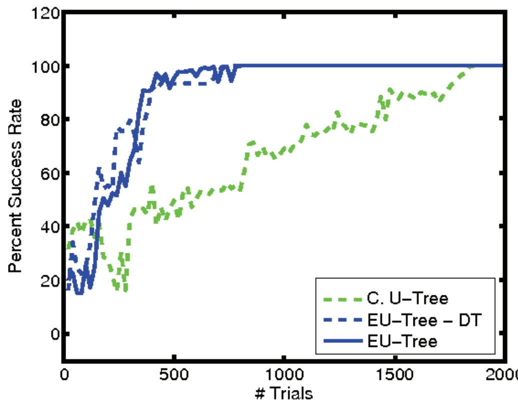

Figure 3: Performance evaluation of three learning methods in the 3-Evasion domain.

Additionally, we note that the performance of the EU-Tree methods with and without the demonstration transition history (DT) is very similar. This result supports our hypothesis that it is not the demonstration transition tuples themselves, but the expert tree discretization derived from that data, that leads to the greatest improvement in learning performance. Compared to the relatively large number of experimental trials executed by the learning agent, typically numbering in the hundreds, data from 15 extra trials contains fairly little information. The tree structure derived from this data, however, can be used to guide the learning in a much more powerful way. The expert tree base provides the U-Tree algorithm with a good initial structure which acts as a guide for further discretization splits.

State Space Size Comparison

As a result of the shorter learning time, the EU-Tree algorithm has the added effect of producing discretizations with fewer states. Although the discretization algorithm itself

Algorithm	# States	Avg. Reward Per Trial
Expert Tree	9	82
C. U-Tree	167	50
EU-Tree	30	80

Table 2: Comparison of the average number of states and average reward earned by final policies of the expert tree, C. U-Tree and EU-Tree algorithms in the 2-Evasion domain.

is the same in both cases, an algorithm running for a long time will tend to perform more splits as more data becomes available. Due to the use of the demonstration data in the EU-Tree algorithm, an optimal policy can be found more quickly, keeping the state space small as a result. This factor is important because reinforcement learning becomes increasingly more difficult as the number of states grows, affecting the overall performance of the algorithm.

Table 2 compares the average number of states and the average reward earned per trial in the final policy of the expert tree, the Continuous U-Tree and the EU-Tree algorithms in the 2-Evasion domain. This data demonstrates how the teaching based approach reduces the size of the state space compared to the Continuous U-Tree learning method.

The expert tree generated from demonstration data has the smallest number of states and the best performance by a small margin. However, the EU-Tree is able to nearly match the expert performance while maintaining a manageable number of states. The Continuous U-Tree algorithm generates a far greater number of states, resulting in poorer overall performance.

EU-Tree in Dynamic Environments

In this section, we examine the ability of the EU-Tree algorithm to cope with suboptimal demonstrations and adapt to changes in the environment. The algorithm is tested on modified versions of the 2-Evasion domain, in which opponent velocities are assigned values in the 2.2-4.2 m/sec range in 0.1 meter increments.

Figure 4(a) shows the performance of the original expert tree policy in the modified domains. The policy is able to accommodate small changes, and performs well under a variety of different conditions. As expected, reducing the speed of the opponents simplifies the problem, and the fixed policy performs well even under significantly different conditions. Increasing the speed of the opponents presents a greater challenge, and the performance of the policy quickly drops to an approximate success rate of 50%.

Figure 4(b) presents the average number of trials taken by the EU-Tree algorithm to learn the correct policy for each of the test domains. In all cases, the algorithm was initialized with the original expert tree generated by the demonstration at 3.2 m/sec. The relationship between graphs (a) and (b) shows a clear correlation between the accuracy of the expert policy and the learning time of the algorithm. Strong expert policy performance leads to short learning times in all but one case, the 2.7 m/sec domain, where we see an unusually high learning cost.

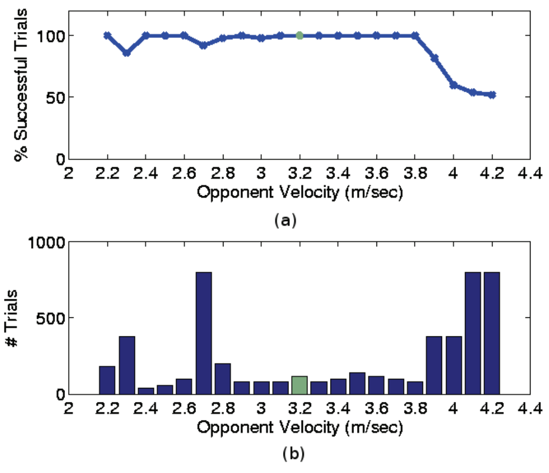

Figure 4: (a) Expert tree performance over a range of opponent velocities. (b) EU-Tree learning performance summary over a range of opponent velocities. The original expert demonstration conditions are marked in lighter color.

Figure 5 presents the learning curves of the same set of trials. Dark, solid lines are used to mark results for experiments with opponent velocities deviating by .1-.5 m/sec from the demonstrated velocity (i.e. 2.7-3.7 m/sec). Lighter, dashed lines mark experiments with velocity variation in the .6-1.0 m/sec range (i.e. 2.2-2.6 and 3.8-4.2 m/sec). For comparison, the learning curve of the Continuous U-Tree algorithm in the original domain is also provided.

The above graphs demonstrate that expert tree discretizations that perform optimally in the environment result in the best learning times, a fact that is not particularly surprising. More importantly, however, we see that the algorithm performance scales relative to the quality of the expert policy. Suboptimal policies continue to provide some boost to the learning, while in the worst case the algorithm performance is no worse than that of the Continuous U-Tree algorithm.

Further Reducing the State Discretization

Readability and ease of interpretation are key strengths of decision tree based algorithms. However, as previously mentioned, one of the drawbacks of the Continuous U-Tree algorithm is the large number of states in the final discretization, especially in complex multidimensional domains. Many of the states generated by the discretization are similar in nature, and lead to the same action, but the complexity of the tree makes it difficult to interpret the data. Ideally we would like a method that compacts the data into a smaller structure without modifying the behavior of the algorithm.

This is achieved by again making use of the standard C4.5 algorithm. The input to the algorithm is formed by the recorded transition tuples of the state tree, where each initial observation o is labeled with the learned policy action associated with its state $L(o)$. The C4.5 algorithm outputs a mapping from observations to the associated policy actions,

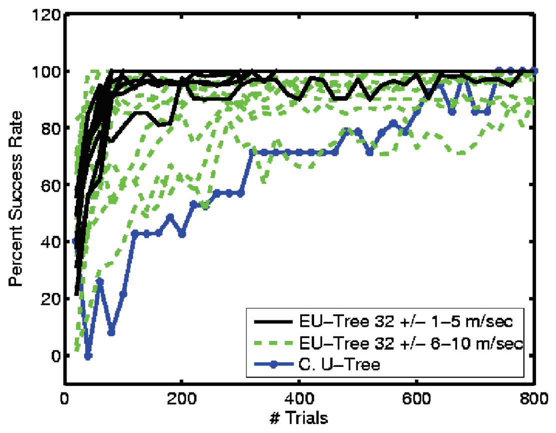

Figure 5: EU-Tree learning performance curves over the range of opponent velocities.

combining and generalizing over states where possible. This process generally results in a more compact representation, and at worst maintains the same number of states. Note that this process does not try to maintain old state boundaries and knows nothing about the structure of the original discretization. To maintain the same observation-action mapping, pruning should be turned off.

This technique can be applied at any point in the learning process to form a more compact tree, but is especially useful for the evaluation of the final result. Using this method, we have been able to systematically reduce the number of states to less than half the size of the original discretization without loss of performance.

Conclusion

In this paper, we presented a general framework for learning high level, behavior-based action policies in continuous state spaces. Our approach combines tree-based state discretization methods with teaching by demonstration to effectively reduce the overall learning time. The advantage of our approach is the use of a preliminary discretization derived from demonstration transition data, in addition to the transition data itself, to initialize the state discretization process.

Our algorithm was shown to handle sub-optimal demonstrations and adapt to dynamic environments. Additionally, the final state discretization was shown to be significantly smaller than that of the original discretization algorithm, facilitating both ease of learning and human readability. Using our state tree compression technique, the state tree can be reduced further without modifying the behavior of the algorithm.

We believe this approach can be extended to other tree-based discretization methods, forming a general framework for combining teaching by demonstration with other learning techniques. Additionally, a similar approach can be applied for an agent that learns from observing the actions of others instead of itself.

References

Abbeel, P., and Ng, A. Y. 2004. Apprenticeship learning via inverse reinforcement learning. In *International Conference on Machine learning*. New York, NY, USA: ACM Press.

Arkin, R., ed. 1998. *Behavior-based robotics*. MIT Press.

Atkeson, C. G., and Schaal, S. 1997. Robot learning from demonstration. In *International Conference on Machine Learning*, 12–20. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Bellman, R. E., ed. 1957. *Dynamic Programming*. Princeton, NJ: Princeton University Press.

Chen, J., and McCarragher, B. 2000. Programming by demonstration - constructing task level plans inhybrid dynamic framework. In *International Conference on Robotics and Automation*, 1402–1407.

Fernandez, F., and Borrajo, D. 2000. VQQL. Applying vector quantization to reinforcement learning. *Lecture Notes in Artificial Intelligence* 292–303.

Howard, R. A. 1960. *Dynamic Programming and Markov Processes*. MIT Press.

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

Kaiser, M., and Dillmann, R. 1996. Building elementary robot skills from human demonstration. In *International Conference on Robotics and Automation*.

McCallum, A. K. 1996. *Reinforcement learning with selective perception and hidden state*. Ph.D. Dissertation, University of Rochester.

Moore, A. W., and Atkeson, C. G. 1993. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning* 13(1):103–130.

Moore, A., and Atkeson, C. G. 1995. The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces. *Machine Learning* 21.

Munos, R., and Moore, A. 2002. Variable resolution discretization in optimal control. *Machine Learning* 49:291 – 323.

Nicolescu, M. N., and Mataric, M. J. 2003. Natural methods for robot task learning: instructive demonstrations, generalization and practice. In *Second International Joint Conference on Autonomous Agents and Multiagent Systems*, 241–248. New York, NY, USA: ACM Press.

Quinlan, J., ed. 1993. *C4.5: Programs for Machine Learning*. San Mateo: Morgan Kaufmann.

Smart, W. D., and Kaelbling, L. P. 2002. Effective reinforcement learning for mobile robots. In *IEEE International Conference on Robotics and Automation*.

Uther, W. T. B., and Veloso, M. M. 1998. Tree based discretization for continuous state space reinforcement learning. In *Artificial Intelligence/Innovative Applications of Artificial Intelligence*, 769–774.