Tree Based Discretization for Continuous State Space Reinforcement Learning

William T. B. Uther and **Manuela M. Veloso**
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213
{uther,veloso}@cs.cmu.edu

Abstract

Reinforcement learning is an effective technique for learning action policies in discrete stochastic environments, but its efficiency can decay exponentially with the size of the state space. In many situations significant portions of a large state space may be irrelevant to a specific goal and can be aggregated into a few, relevant, states. The U Tree algorithm generates a tree based state discretization that efficiently finds the relevant state chunks of large propositional domains. In this paper, we extend the U Tree algorithm to challenging domains with a continuous state space for which there is no initial discretization. This Continuous U Tree algorithm transfers traditional regression tree techniques to reinforcement learning. We have performed experiments in a variety of domains that show that Continuous U Tree effectively handles large continuous state spaces. In this paper, we report on results in two different domains, one gives a clear visualization of the algorithm and another empirically demonstrates an effective state discretization in a simple multi-agent environment.

Introduction

Reinforcement learning is a technique for learning a control policy for an agent as it moves through a world and receives a series of rewards for its actions (Kaelbling, Littman, & Moore 1996).

The number of states that need to be considered by a traditional reinforcement learning algorithm increases exponentially with the dimensionality of the state space. The dimensionality of the state space in turn increases linearly with, for example, the number of agents in the world.

However, there are parts of the state space where the exact position in state space is irrelevant to the agent. All the states in a region where this is true are equivalent and can be replaced by a single state, reducing the state space explosion. In many environments this reduction is significant. The concept of starting with many states and joining them together when there is too much resolution is known as state aggregation. Techniques using a non-uniform discretization are referred to as variable resolution techniques (Moore 1991).

The Parti-game algorithm (Moore 1994) is an algorithm for automatically generating a variable resolution discretization of a continuous, deterministic domain based on observed data. This algorithm uses a greedy local controller to move within a state or between adjacent states in the discretization. When the greedy controller fails the resolution of the discretization is increased in that state.

The G algorithm (Chapman & Kaelbling 1991) and the U Tree algorithm (McCallum 1995) are similar algorithms that automatically generate a variable resolution discretization by re-discretizing propositional state spaces. A policy can then be found for the new discretization using traditional techniques. Like Parti-game, they both start with the world as a single state and recursively split it where necessary. The Continuous U Tree algorithm described in this paper extends these algorithms to work with continuous state spaces rather than propositional state spaces.

To scale up reinforcement learning techniques, the standard approach is to substitute a function approximator for the table of states in a standard algorithm. If done naively this can lead to convergence problems (Boyan & Moore 1995). Gordon (1995) showed how to use an averaging system to sub-sample the state space. This system could only interpolate between sub-sampled points, not extrapolate outside its sampled range. Baird (1995) describes a technique for using any gradient descent function approximator instead of a table of values.

The Continuous U Tree algorithm described below can be viewed as using a regression tree (Breiman *et al.* 1984; Quinlan 1986) to store the state values. Markov Decision Problems (MDPs) are similar to reinforcement learning problems except that they assume a complete and correct prior model of the world. MDPs can also be solved using state aggregation techniques (Dean & Lin 1995).

In this paper we describe Continuous U Tree, a generalizing, variable-resolution, reinforcement learning algorithm that works with continuous state spaces. To evaluate Continuous U Tree we used two domains, a small two dimensional domain and a larger seven dimensional domain. The small domain, a robot travelling down a corridor, shows some of the main features of the algorithm. The larger domain, hexagonal grid soccer, is similar to the one used by Littman (1994) to investigate Markov games. It is ordered discrete rather than strictly continuous. We increased the size of this domain, both in number of states and number of actions per state, to test the generalization capabilities of

our algorithm.

In the machine learning formulation of reinforcement learning there are a discrete set of states, s, and actions, a. The agent can detect its current state, and in each state can choose to perform an action which will in turn move it to its next state. For each state/action/resulting state triple, (s, a, s'), there is a reinforcement signal, $R(s, a, s')$.

The world is assumed to be Markovian. For each state/action pair, (s, a), there is a probability distribution, $P_{(s,a)}(s')$, giving the probability of reaching a particular successor state, s'.

The foundations of reinforcement learning are the Bellman Equations (Bellman 1957):

$$Q(s,a) = \sum_{s'} P_{(s,a)}(s') \left(R(s,a,s') + \gamma V(s')\right) \quad (1)$$

$$V(s) = \max_a Q(s,a) \quad (2)$$

These equations define a Q function, $Q(s, a)$, and a value function, $V(s)$. The Q function is a function from state/action pairs to an expected sum of discounted reward (Watkins & Dayan 1992).

Continuous U Tree

U Tree (McCallum 1995) includes a method to apply propositional decision tree techniques to reinforcement learning. We introduce an extension to U Tree that is capable of directly handling both propositional and continuous-valued domains. Our resulting algorithm, "Continuous U Tree", uses decision tree learning techniques (Breiman *et al.* 1984; Quinlan 1986) to find a discretization of the continuous state space.

Continuous U Tree is different from U Tree and traditional reinforcement learning algorithms in that it does not require a prior discretization of the world into separate states. The algorithm takes a continuous, or ordered discrete, state space and automatically splits it to form a discretization. Any discrete, state-based, reinforcement learning algorithm can then be used on this discretization.

In both U Tree and Continuous U Tree there are two distinct concepts of state. In Continuous U Tree the first type of state is the fully continuous state of the world that the agent is moving through. We term this *sensory input*. The second type of state is the position in the discretization being formed by the algorithm. We will use the term *state* in this second sense. Each state is an area in the sensory input space, and each sensory input falls within a state. In many previous algorithms these two concepts were identical. Sensory input was discrete and each different sensory input corresponded to a state.

The translation between these two concepts of state is in terms of a *state tree* that describes the discretization. Each node in this binary tree corresponds to an area of sensory input space. Each internal node has a *decision* attached to it that describes a way to divide its area of sensor space in two. These decisions are described in terms of an attribute of the sensory input to split on, and a value of that attribute. Any sensory input where the relevant attribute falls above or below the stored value is passed down the left or right side of the tree respectively. Leaf nodes in the tree correspond to states. A state tree enables generalization – states become areas of sensory input space.

We refer to each step the agent takes in the world as a *transition*. Each saved transition is a vector of the starting sensory input I, the action performed a, the resulting sensory input I' and the reward obtained for that transition r, (I, a, I', r). The sensory input is itself a vector of values. These values can be fully continuous.

As the agent is forming its own discretization, there is no prior discretization that can be used to aggregate the data. (If a partial prior discretization exists, this can be utilized by the algorithm by initializing the state tree with that discretization but this is not required.) Only a subset of the transitions need to be saved, but they need to be saved with full sensor accuracy. Deciding which subset of the transitions is saved will be discussed later.

Each transition (I, a, I', r) gives one *datapoint* $(I, a, q(I, a))$ – a triple of the sensory input I, the action a, and a value $q(I, a)$. The value of a datapoint is its expected reward for performing that transition and behaving optimally from then on, $q(I, a)$ (defined below).

Continuous U Tree forms a discretization by recursively growing the state tree. Table 1 summarizes the algorithm. In a nutshell, the algorithm is as follows: Initially the world is considered to be a single state with an expected reward, $V(s) = 0$. The state tree is a single leaf node describing the entire sensory input space. The algorithm then loops through a two phase process: Datapoints are accumulated for learning in a data gathering phase, then a processing phase updates the discretization. During the data gathering phase the algorithm behaves as a standard reinforcement learning algorithm, with the added step of using the state tree to translate sensory input to a state. It also remembers the transitions it sees. During the processing phase, the values of all the datapoints, $q(I, a)$, are re-calculated. If a "significant" difference (defined below) in the distribution of datapoint values is found within a state, then that state is split in two. A split adds two new leaves to the state tree representing the two new states formed from the old state. The MDP defined over the new discretization is solved to find the state values. Initializing the state values to the values in the previous discretization and using an incremental algorithm makes updating these values relatively fast. Deciding when to execute a pass through the processing phase is a parameter to the algorithm.

This is similar to the recursive partitioning of a decision or regression tree, except that the values of the datapoints can change when more data is gathered and the MDP is re-solved after each processing phase. Splitting is also breadth-first rather than depth-first.

The value of a datapoint $q(I, a)$ is calculated using a modification of the Bellman equations. Using the value for the resulting state of a transition $V(s')$ and the recorded reward for the transition r, we can assign a value to the corresponding datapoint: $q(I, a) = r + \gamma V(s')$.

- Data Gathering Phase:
  - Pass the current sensory input I, down the state tree to find the current state s in the discretization.
  - Use Q values for the state s to choose an action $a = \arg\max_{a'} Q(s, a')$
  - Store transition datapoint (I, a, I', r).
  - (optional) Update the value function using a standard, discrete Reinforcement Learning technique.
- Processing Phase:
  - For each leaf:
    - Update the values of datapoints in that leaf: $q(I, a) = r + \gamma V(s')$.
    - Find the best split point using the splitting criterion.
    - If the split point satisfies the stopping criterion, then split the leaf into two states.
  - Calculate the transition probabilities $P_{(s,a)}(s')$, and expected rewards $R(s, a, s')$, using the recorded transitions (I, a, I', r).
  - Solve the MDP so specified to find $V(s)$ and $Q(s, a)$.

Table 1: The Continuous U Tree algorithm

Having calculated these datapoint values, Continuous U Tree then tests if they vary systematically within any state, and, if so, finds the best single decision to use to divide that state in two. This is done using decision tree techniques. Each attribute in the sensory input is considered in turn. The datapoints are sorted according to that attribute. The algorithm loops through this sorted list and a trial split is added between each consecutive pair of datapoints. This split divides the datapoints into two sets. These two sets are compared using a *splitting criterion* (see below) which returns a single number describing how different the two distributions are. The trial split that leads to the largest difference between the two distributions is remembered. The "best" split is then evaluated by a fixed *stopping criterion* (see below) to check whether the difference is significant.

Having found a discretization, the problem has been reduced to a standard, discrete, reinforcement learning problem: finding Q values for the new states, $Q(s, a)$. This is done by calculating state transition probabilities, $P_{(s,a)}(s')$, and expected rewards, $R(s, a, s')$, from the recorded transitions and then solving the MDP so specified. We have used both Prioritized Sweeping (Moore & Atkeson 1993) and conjugate gradient descent on the sum-squared Bellman residuals (Baird 1995) to solve the MDP defined over this new discretization.

Splitting Criteria: Testing for a difference between data distributions

We tried two different splitting criteria. The first is a non-parametric statistical test – the Kolmogorov-Smirnov test. The second is based on sum-squared error.

The first splitting criterion is that used by McCallum (1995) in the original U Tree algorithm, which looked for violations of the Markov assumption. If the distribution of datapoint values in each of the two new states was different, then there was a violation of the Markov assumption – more information about the world is available by knowing where the agent is within the old state. The splitting criterion was then a test of the statistical similarity of the distributions of the datapoints on either side of the split. We used the Kolmogorov-Smirnov non-parametric test. This test is based on the difference in the cumulative distributions of the two datasets.

Figure 1 illustrates the cumulative distributions of two datasets. The arrow between the two lines marks the maximum difference D between the cumulative probability distributions, C_1 and C_2. This distance has a distribution that can be approximately calculated for two independent, but identically distributed, sets of datapoints regardless of what distribution the sets are drawn from. Assuming the cumulative distributions are C_1 and C_2, with dataset sizes of N_1 and N_2 respectively, the probability that the observed difference could be generated by noise is calculated using the equations in Table 2 (Press *et al.* 1992). The smaller this probability, the more evidence there is of a true difference. If it is small enough to pass the stopping criterion, then a split is introduced.

$$D = \max_{-\infty < x < \infty} |C_1(x) - C_2(x)| \quad (3)$$

$$N_e = \frac{N_1 N_2}{N_1 + N_2} \quad (4)$$

$$\lambda = \left[\sqrt{N_e} + 0.12 + 0.11/\sqrt{N_e}\right] D \quad (5)$$

$$P_{Noise} = 2\sum_{j=1}^{\infty} (-1)^{j-1} e^{-2j^2\lambda^2} \quad (6)$$

Table 2: The Kolmogorov-Smirnov test equations

We investigated a second squared-error based splitting criterion (Breiman *et al.* 1984). The aim is to approximate

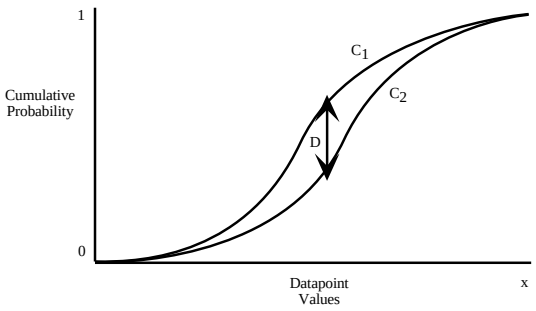


Figure 1: Two cumulative probability distributions

the Q values of the transitions, $q(I, a)$, using only the Q values of the states, $Q(s, a)$. The error measure for this fit is the sum-squared error.

Assuming the other Q values do not change, the Q values of the two new states will equal the mean of the datapoints in those states. The mean-squared error equals to the expected squared deviation from the mean, i.e. the variance. The splitting criterion is the weighted sum of the variances of the Q values of the transitions on either side of the split. A second justification for this splitting criterion is the finding of Williams & Baird (1993) that reducing the Bellman residual of a value function increases performance.

Both these splitting criteria can be made more sensitive by performing the tests separately for each action and then combining the results. For the mean-squared error test a weighted sum was used. For the Kolmogorov-Smirnov test we multiply the probabilities calculated.

Importantly from an efficiency viewpoint, both of these tests can be done at least partly incrementally. For the sum-squared error, we can keep track of $\sum_n x$ and $\sum_n x^2$ and use these to estimate the variance, $V = \frac{\sum_n x^2 - \frac{\left(\sum_n x\right)^2}{n}}{n-1}$. The time complexity of a series of tests on the data in a leaf is then $O(n)$.

For the Kolmogorov-Smirnov test, the datasets need to be sorted to find the cumulative probability distributions. This sorting can be performed once by trading $O(n)$ space, but we still need to loop over the distributions to find D for each test making the total time $O(n^2)$.

Stopping Criteria: When have we gone far enough?

When learning a regression tree from data, the standard technique is to grow the tree by recursively splitting the data until a stopping criterion is met. Obviously, if all the datapoints in a leaf have the same value, the algorithm can stop. Similarly, it can stop if there is no way to separate the data – for example all datapoints fall at the same point in the sensory input space.

More generally, the algorithm should stop when it cannot detect a "significant" difference between the datasets on either side of the best split in a leaf. Here the word significant can have different meanings. For the Kolmogorov-Smirnov test we use significant to mean statistically significant at the $P = 0.05$ level.

The stopping criterion for the squared-error based test is the reduction in mean squared-error. If the difference between the variance of the entire dataset and the weighted mean variance of the datasets induced by the split is below a threshold, then there is not a significant difference. This test is less theoretically based than the other test, but with careful adjustment of the threshold produced reasonable results.

In the tree based learning literature, it is well known that stopping criteria often have to be weak to find good splits hidden low in the tree. To stop overly large trees being produced, the trees are often pruned before being used. We are assuming that little error is introduced by over-discretizing.

Datapoint sampling: What do we save?

In the description of the algorithm above, we simply noted that transitions are saved. Notably, Continuous U Tree does not need to save all of the transitions. Its generalization capabilities allow it to learn from non-continuous trajectories through state space. Recording all of the transitions is expensive, both in terms of memory to record the data and in processing time when testing for splits. However, the fewer transitions recorded the harder it is to detect differences within a leaf.

When limiting the number of saved datapoints, we simply recorded a fixed number of datapoints per state. Even once the number of transitions to store is set, it is still unclear which transitions to store. In the experiments reported we remembered all transitions until the preset per-state limit was reached. We then made room for new transitions by discarding a random transition from that state. This has the advantage that no bias, which might mislead our splitting or stopping criterion, is introduced.

A Corridor Domain

To qualitatively test our algorithm we introduce a simple 2-D domain simulating a robot moving in a corridor. The domain has two state dimensions: location and temperature. Location is an ordered, discrete attribute where the agent needs to distinguish every value to represent the correct value function. Temperature is continuous, but can be divided into three qualitatively different regions - cold, normal and hot. The robot is not given this division. As we will see, Continuous U Tree correctly and autonomously finds the three qualitatively different temperature regions. It also finds that the distance down the corridor is needed at full accuracy.

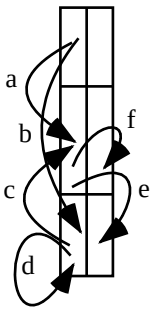
In this domain the length of the corridor is divided into three sections: sections A, B and C. These three sections of corridor each have a different base temperature. A is cooled, B is normal temperature and C is heated. The robot starts at a random position in the corridor and is rewarded when it reaches the right-hand end of the corridor. The robot is temperature sensitive, but also has limited temperature control on board. The robot must learn to move towards the correct end of the corridor, to turn its heater on in section A and to turn its cooler on in section C.

The robot's actions make it move and change its temperature control settings. The robot has 6 actions it can perform: move forward or backward 1 unit, each with either heater on, cooler on, or neither on. The robot moves nondeterministically as a function of its temperature. While the robot's temperature is in the $(30°, 70°)$ range, it successfully moves with 90% probability, otherwise it only moves with a 10% probability.

The robot's temperature is the base temperature of that section of corridor adjusted by the robot's temperature control unit. In our experiments, the corridor was 10 units long. Sections A , B and C were 3, 4 and 3 units in length respectively, with base temperature ranges of $(5°, 35°)$, $(25°, 75°)$ and $(65°, 95°)$ respectively. The robot's temperature control unit changed the temperature of the robot to differ from

the corridor section base temperature by $(25°, 45°)$ if the heater was on, and by $(-25°, -45°)$ when the cooler was on. All temperatures and temperature adjustments were independently sampled from a uniform distribution in the range given for each time-step.

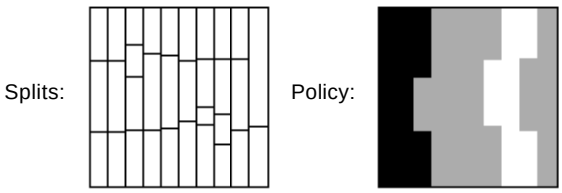
Figure 2 shows part of a state transition diagram for this domain. The robot moves with high probability if and only if it is in the $(30°, 70°)$ temperature range. Its temperature for the next timestep is then determined by its new location and the temperature control unit settings from its previous move. This means that if it is likely to move it needs to set its temperature unit correctly for the next unit of corridor, not the current unit. If the section of corridor shown was the cooled section of corridor then transitions a, c and f correspond to moving right with the heater on, and transitions b, d and e correspond to moving right with the heater off.



The y axis is robot's temperature. The x axis represents two consecutive locations in the cool section of the corridor. High probability transitions are shown.

Figure 2: Sample transitions for part of the corridor domain

Figure 3 shows the discretization and policy for this task after 3000 steps in the world. The y axis is the temperature of the robot on a scale of $0 - 100$. The x axis is the location of the robot in the corridor, on a scale of $1 - 10$.



The y axis is the robot's temperature. The x axis is the location along the corridor. The goal is on the right hand edge. The policy was to move right at every point. Black areas indicate the heater was to be active. White areas indicate the cooler was to be active.

Figure 3: The learnt discretization and policy for the corridor task after 3000 steps

In the tests run in the corridor domain, both splitting criterion behaved similarly. The sum-squared error testing is significantly faster however.

A Hexagonal Soccer Domain

As a more complex domain for empirical evaluation of Continuous U Tree we introduce a hexagonal grid based soccer (Hexcer) simulation. Hexcer is similar to, but larger than, the game framework used by Littman (1994) to test Minimax Q Learning. We test Continuous U Tree by having it learn to play Hexcer against another reinforcement learning algorithm. Uther & Veloso (1997) compare a number of reinforcement learning algorithms in the Hexcer domain. They found Prioritized Sweeping to be the best of the tradition reinforcement learning algorithms. We compare Continuous U Tree against Prioritized Sweeping here.

Hexcer consists of a field with a hexagonal grid of 53 cells, two players and a ball. Each player can be in any cell not already occupied by the other player. The ball is either in the center of the board, or is controlled by (in the same cell as) one of the players. This gives a total of 8268 distinct states in the game.

The two players start in fixed positions on the board, as shown. The game then proceeds in rounds. During each round the players make simultaneous moves to a neighboring cell in any of the six possible directions. Players must specify a direction in which to move, but if a player attempts to move off the edge of the grid, it remains in the same cell. Once one player moves onto the ball, the ball stays with that player until stolen by the other player. When a cell is contested the winner is decided non-detministically with the winner getting the ball.

Players score by taking the ball into their opponent's goal. When the ball arrives in a goal the game ends. The player guarding the goal loses the game and gets a negative reward. The opponent receives an equal magnitude, but positive, reward. It is possible to score an own goal.

We performed empirical comparisons along two different dimensions – how fast does each algorithm learn to play, and to what level of expertise does the algorithm learn to play, discounting a "reasonable" initial learning period? Essentially we have two points on a learning curve.

In this experiment the pair of algorithms played each other at Hexcer for 1000 games. Wins were recorded for the first 500 games and the second 500 games. The first 500 games allowed us to measure learning speed as the agents started out with no knowledge of the game. The second 500 games gave an indication of the level of ability of the algorithm after the initial learning period.

This 1000 game test was then repeated 20 times. The results shown in table 3 are the number of games (mean $\pm$ standard deviation) won by each algorithm. The percentage at the end of each row is the level of statistical significance of the difference.

Table 3 shows that Continuous U Tree performs better than Prioritized Sweeping. Interestingly, looking at the (often quite large) trees generated by Continuous U Tree we can see that it has managed to learn the concept of "opponent behind."

	Prioritized Sweeping		Continuous U Tree		Significance
First 500 games	175 ± 93	$35\% \pm 19\%$	325 ± 93	$65\% \pm 19\%$	1%
Second 500 games	197 ± 99	$39\% \pm 20\%$	303 ± 99	$61\% \pm 20\%$	5%
Total	372 ± 183	$37\% \pm 18\%$	628 ± 183	$63\% \pm 18\%$	1%

Table 3: Hexcer results: Prioritized Sweeping vs. Continuous U Tree

Discussion

Continuous U Tree learns to play Hexcer with much less world experience than Prioritized Sweeping. Despite requiring less data, the Continuous U Tree algorithm is slower in real-time than the prioritized sweeping algorithm for the domains tested. For domains where data can be expensive or time-consuming to gather, trading algorithm speed for efficient use of data is important. In this area Continuous U Tree has an advantage over traditional reinforcement learning algorithms.

The corridor domain makes clear a number of details about the algorithm. Firstly, our algorithm successfully reduced a continuous space down to a discrete space with only 32 states (see figure 3). The algorithm has managed to find the $30°$ and $70°$ cutoff points with reasonable accuracy - it does not try to represent the entire temperature range. Continuous U Tree does not make perfect splits initially, but introduces further splits to refine its discretization. A possible enhancement would be to have old splits reconsidered in the light of new data and optimize them directly.

Secondly, the algorithm has divided the length of the corridor up as much as possible. Unlike the temperature axis, where large numbers of different datapoints values are grouped in one state, every different corridor location is in a different state.

The location of the agent in the corridor is directly relevant to achieving the goal. The state values in a fully discretized world change along this axis. By comparison, the values of different points below $30°$ in the temperature range, but with the same corridor location, are equal. Once the agent is below $30°$ it will only move 10% of the time – no further information is needed. In ongoing work we are attempting to generalize over the cost to move between states in addition to the value of states.

The simple form of decision recorded by Continuous U Tree in the state tree can only divide the space parallel to the axes of the input space. Other more complex types of decision exist in regression tree literature (Utgoff & Brodley 1991), but simple decisions have been shown to be remarkably effective, and much faster to find than more complex decisions. By adding redundant attributes that are linear combinations of the primary attributes we can allow the algorithm to find diagonal splits.

Like U Tree, Continuous U Tree inherits from its decision tree background the ability to handle moderately high dimensional state spaces. The original U Tree work used this capability to partially remove the Markov assumption. As we were playing a Markov game we did not implement this part of the U Tree algorithm in Continuous U Tree although there is no reason why this could not be done.

Conclusion

Large state spaces affect learning speed, but often the exact location in that space is not relevant to achieving the goal. We have described an effective generalizing reinforcement learning algorithm, Continuous U Tree, that can discretize a continuous state space while leaving equivalent areas as single states. Our generalizing algorithm performs significantly better than non-generalizing algorithms.

References

Baird, L. C. 1995. Residual algorithms: Reinforcement learning with function approximation. In Prieditis, A., and Russell, S., eds., *Machine Learning: Proceedings of the Twelfth International Conference (ICML95)*, 30–37. San Mateo, CA: Morgan Kaufmann.

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

Boyan, J. A., and Moore, A. W. 1995. Generalization in reinforcement learning: Safely approximating the value function. In Tesauro, G.; Touretzky, D. S.; and Leen, T. K., eds., *Advances in Neural Information Processing Systems*, volume 7. Cambridge, MA: The MIT Press.

Breiman, L.; Friedman, J. H.; Olshen, R. A.; and Stone, C. J. 1984. *Classification And Regression Trees*. The Wadsworth Statistics/Probability Series. Monterey, California: Wadsworth and Brooks/Cole Advanced Books and Software.

Chapman, D., and Kaelbling, L. P. 1991. Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91)*, 726–731.

Dean, T., and Lin, S.-H. 1995. Decomposition techniques for planning in stochastic domains. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*.

Gordon, G. J. 1995. Online fitted reinforcement learning. In *Value Function Approximation workshop at ML95*.

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

Littman, M. L. 1994. Markov games as a framework for multi-agent reinforcement learning. In *Machine Learning: Proceedings of the Eleventh International Conference (ICML94)*, 157–163. San Mateo, CA: Morgan Kaufmann.

McCallum, A. K. 1995. *Reinforcement Learning with Selective Perception and Hidden State*. Phd. thesis, Department of Computer Science, University of Rochester, Rochester, NY.

Moore, A., and Atkeson, C. G. 1993. Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning* 13.

Moore, A. W. 1991. Variable resolution dynamic programming: Efficiently learning action maps in multivariate real-values state-

spaces. In Birnbaum, L., and Collins, G., eds., *Machine Learning: Proceedings on the Eighth International Workshop*. Morgan Kaufmann.

Moore, A. W. 1994. The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces. In Cowan, J. D.; Tesauro, G.; and Alspector, J., eds., *Advances in Neural Information Processing Systems 6*, 711–718. Morgan Kaufmann.

Press, W. H.; Teukolsky, S. A.; Vetterling, W. T.; and Flannery, B. P. 1992. *Numerical Recipes in C: the art of scientific computing*. Cambridge: Cambridge University Press, 2nd edition.

Quinlan, J. R. 1986. Induction of decision trees. *Machine Learning* 1:81–106.

Sutton, R. S. 1984. *Temporal Credit Assignment in Reinforcement Learning*. Ph.D. Dissertation, University of Massachusetts, Amherst.

Utgoff, P. E., and Brodley, C. E. 1991. Linear machine decision trees. Technical Report 91-10, University of Massachusetts.

Uther, W. T. B., and Veloso, M. M. 1997. Adversarial reinforcement learning. *Journal of Artificial Intelligence Research*. To be submitted.

Watkins, C. J. C. H., and Dayan, P. 1992. Q-learning. *Machine Learning* 8(3):279–292.

Williams, R. J., and Baird, L. C. I. 1993. Tight performance bounds on greedy policies based on imperfect value functions. Technical report, College of Computer Science, Northeastern University.