Generalizing Adversarial 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 has been used for a number of years in single agent environments. This article reports on our investigation of Reinforcement Learning techniques in a multi-agent and adversarial environment with continuous observable state information. Our framework for evaluating algorithms is two-player hexagonal grid soccer. We introduce an extension to Prioritized Sweeping that allows generalization of learnt knowledge over neighboring states in the domain and we introduce an extension to the U Tree generalizing algorithm that allows the handling of continuous state spaces.

Introduction

Multi-agent adversarial environments have traditionally been addressed as game playing situations. Indeed, one of the first areas studied in Artificial Intelligence was game playing. For example, the pioneering checkers playing algorithm by Samuel (1959) used both search and machine learning strategies. Interestingly, his approach is similar to modern Reinforcement Learning techniques (summarized in Kaelbling, Littman, & Moore 1996) in that both learn an evaluation function based on experience and a Markov assumption about the world.

In the Reinforcement Learning paradigm an agent is placed in a situation without knowledge of any goals or other information about the environment. As the agent acts in the environment it is given feedback: a reinforcement value or reward that defines the utility of being in the current state. Over time the agent is supposed to customize its actions to the environment so as to maximize the sum of this reward. By only giving the agent reward when a goal is reached, the agent learns to achieve its goals.

Since Samuel's work however, Reinforcement Learning techniques were not used again in an adversarial setting until quite recently. In an adversarial setting there are multiple (at least two) agents in the world. In a game with two players, when an agent wins a game it is given a positive reinforcement and its opponent is given negative reinforcement. Maximizing reward corresponds directly to winning games. Over time the agent is learning to act so that it wins the game.

Traditional Reinforcement Learning algorithms rely on a prior discretization and do not generalize over multiple states in that generalization. In this paper we introduce an extension to Prioritized Sweeping (Moore & Atkeson 1993), Fitted Prioritized Sweeping, and a modification of the U Tree algorithm (McCallum 1995), Continuous U Tree, as examples of algorithms that generalize over multiple states.

Tesauro (1995) and Thrun (1995) have both used neural nets in a Reinforcement Learning paradigm. Tesauro's work in the game of backgammon was successful, but required hand tuned features being fed to the algorithm for high quality play. Thrun was moderately successful in using similar techniques in chess, but these techniques were not as successful as they had been in the backgammon domain. This work has been repeated in other domains, but again, without the same success as in the backgammon domain (Kaelbling, Littman, & Moore 1996).

We use a similar environment to the one used by Littman (1994) to investigate Markov games. Our environment is larger, both in number of states and number of actions per state. This increases our ability to test the generalization capabilities of our algorithms. Neither of the algorithms we test perform any internal search or lookahead when deciding actions; they use just the current state and their learnt evaluation for that state. While search would improve performance, we considered it orthogonal to learning the evaluation function. We intend to explore methods of combining search with these techniques in future work.

Hexcer: The Adversarial Learning Environment

As a substrate to our investigation, we introduce a hexagonal grid based soccer simulation, Hexcer, which is similar to the game framework used by Littman (1994) to test Minimax Q Learning. Hexcer consists of a board with a hexagonal grid with 53 cells, two players and a ball (see Figure 1). Each player can be in

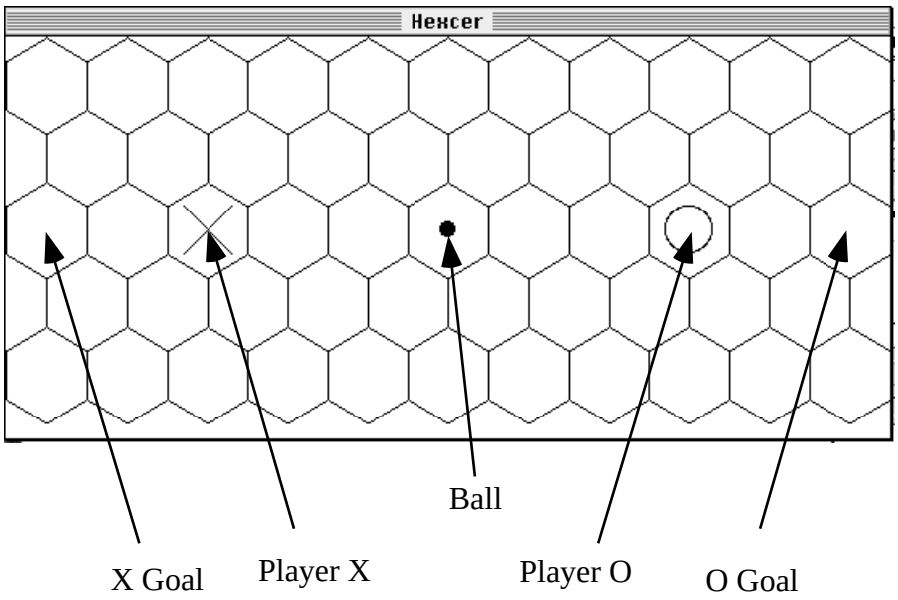

Figure 1: The Hexcer board

any cell not already occupied by the other player. The ball is either in the center of the board, or is held by 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 the two players try to move onto the same cell one of them succeeds and the other fails, remaining in its original position. The choice of who moves into the contested cell is usually made randomly with each player having an equal chance. The single exception is when one of the players is already in that cell (for example, if it tried to move into a wall and so didn't move). In this case the player already occupying the cell is considered to win and remains in that cell. The ball, if it was in the control of one of the players, moves into the contested cell regardless of which player wins and so may change hands.

Players score by taking the ball into their opponents goal. When the ball arrives in a goal the game ends. The player who owns the goal loses the game and gets a negative reward. Their opponent, usually the player that took the ball into the goal, receives an equal magnitude, but positive, reward. It does not matter which player took the ball into the goal; that is, if I take the ball into my own goal my opponent receives positive reward and I receive negative reward.

The hexcer game provides an interesting environment for studying the use of Reinforcement Learning. Each player observes its own position, the position of the ball and the position of its adversary. Initially, it does not know that the objective of the game is to take the ball to a goal position. Reinforcement learning seems the appropriate technique to acquire the necessary action selection information for each state.

Reinforcement Learning Revisited

In the machine learning formulation of Reinforcement Learning (Kaelbling, Littman, & Moore 1996) 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, $R(s, a, s')$. If action a by the agent when it is in state s leads to state s', the agent receives the reward associated with that state/action/resulting state triple, $R(s, a, s')$.

The world is assumed to be Markov. That is, the current state defines all relevant information about the world. There is no predictive power gained by knowing the agent's history in arriving at the current state. This does not mean the world must be deterministic. It is quite possible that the mapping from state/action pairs to next states is probabilistic. That is, 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', from the state s when action a is performed in that state. Because the opponent is not necessarily deterministic, this is the case in Hexcer.

As stated above, the goal is for the agent "to maximize its reward over time." To keep this sum of future reward bounded, we discount future rewards. A discount factor, γ, $0 < \gamma < 1$, is added to the sum giving: $G = \sum_{t=1}^{\infty} \gamma^t R_t$. At each step the agent is supposed to behave in a way that maximizes this sum of expected future discounted reward. γ can be interpreted in different ways. It corresponds to there being a chance, probability $(1 - \gamma)$, that the world ends between each move. It can also be seen as an interest rate, or just as a trick to make the sum bounded. In our experiments, $\gamma = 0.95$.

The foundations of Reinforcement Learning are the Bellman Equations:

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

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

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). The result is the expected discounted reward for executing that action in that state then behaving optimally from then on. The value function is a function from states to sums of discounted reward. It is the expected sum of discounted reward for behaving optimally in that state as well as from then on.

Prioritized Sweeping (Moore & Atkeson 1993) builds a model of the world by recording state transition probabilities and the reward associated with each state and action. Using this model and the Bellman equations Prioritized Sweeping can calculate Q values directly.

Re-solving the Bellman equations entirely at every timestep is too expensive even if all the probabilities are known. Prioritized Sweeping proceeds by updating first those entries those inputs have changed most.

As the agent moves through the environment a state transition graph is built (see Figure 2). Each time the agent makes a transition from state s to state s' by executing action a, the transition count on the link from s to s' labelled a is incremented. At the same time the reward for that transition is recorded if not already known. $Q(s,a)$ can then be calculated directly for the state s using the Bellman equations. The transition probabilities from state s, $P_{(s,a)}(s')$ can be calculated from the transition counts when we update $Q(s,a)$.

We performed two state value updates per simulation step. One update was performed after each step to reflect the change in transition probabilities after that step. The second update was performed on the state whose input has changed most since it was last updated to propagate the change in Q values.

Each time a Q value changes, say $Q(s',a)$, its state/action pair, (s',a), is entered into a priority queue with its priority being the magnitude of the change. Each timestep the agent can pull the top pair from the queue and update the value of the state, $V(s')$. This change in value for that state leads to a change in the Q value for transitions into that state, in our example $Q(s,a)$ and $Q(s,a')$. If those Q values change, their state/action pairs are entered into the priority queue and the wave of updates continues.

Generalizing Algorithms

Many previous Reinforcement Learning algorithms have a major weakness: they cannot generalize experience over multiple states. In a single agent environment this is not good, but in a multi-agent environment it is disastrous. Imagine a game of Hexcer where our agent has the ball in the center of the field and the opposing agent is well behind us. By moving directly towards the goal we can score every time regardless of the opponent's exact position.

Unfortunately, changing the opponent's position at all, even by the smallest amount, will place our agent in a new state. If it has never been in this particular state before then it will not know how to behave. If the opponent's position is irrelevant then the agent should realise this and generalize over all those states that only differ in opponent position, learning the concept of "opponent behind".

One naive generalization method is to take a simple model-free algorithm like Q Learning and replace the table of values with a generalizing function approximator (e.g. a Neural Net). This has been shown to be unstable in some cases (Boyan & Moore 1995) never converging to a solution, although some good results have been obtained with this method (Tesauro 1995). Stable Methods exist for using either Neural Nets (Baird 1995) or Decision Trees (McCallum 1995). First we introduce a stable method which uses both a table and a generalizing function approximator. Then McCallum's (1995) method using Decision Trees is described and we extend it to handle continuous state values. Finally we compare the methods.

Fitted Prioritized Sweeping

As pointed out by Boyan & Moore (1996), generalizing function approximators are not a problem if the approximation is not iterated, i.e., if an approximation is not used to update itself. Fitted Prioritized Sweeping makes use of this result by using standard Prioritized Sweeping as a base, and then doing the generalization afterwards.

At each timestep during the game, the agent updates a standard Prioritized Sweeping Q-function over a standard discretization of the state space. To choose its actions however, it reads the Q-values from a generalizing function approximator that was learnt from the previous game. At the end of each game the set of Q values generated by the Prioritized Sweeping algorithm for all visited state/action pairs is used as the input to update the generalizing function approximator.

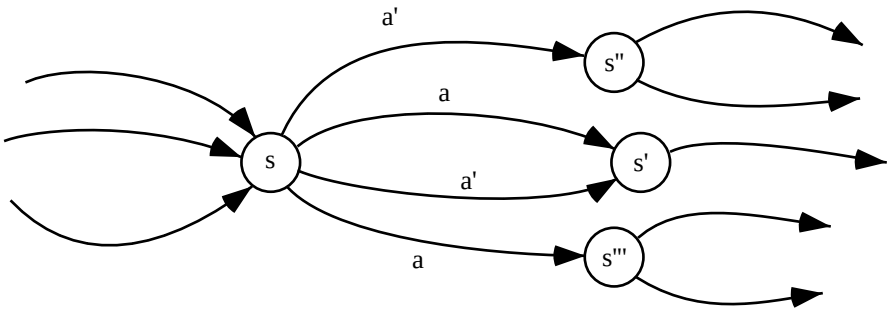

Figure 2: The Graph stored by Prioritized Sweeping

Here the approximator is used to decide the agent's actions, but all data that is used as input to the approximator is data gathered from the real world. The approximator decides where to sample, but the value of the sample itself is not approximate. No approximate values are recycled as input to the approximator directly.

In our implementation a piecewise linear function approximator was used, although any generalizing function approximator could be used. The piecewise linear approximation was built up using a recursive splitting technique similar to a decision tree. At each stage in the recursion the best way of splitting the data is found. If this split gives a better fit than having no split then the split is accepted. Each part of the data is then fit recursively in a similar manner forming a tree of splits. In the final tree, the leaves contain least squares linear fits of the data that fall in that leaf. Initially there is a single leaf in the tree with a constant value of 0. Each time the tree is updated, the leaves of the tree are further divided. Early splits are never reconsidered. Figure 3 contains the Fitted Prioritized Sweeping Algorithm.

During Game:
- Update Prioritized Sweeping table entry
- Choose move using linear approximation tree for Q values

After Game:
- Update Prioritized Sweeping table till changes fall below threshold
- Grow linear approximation tree using Prioritized Sweeping Q table entries

Figure 3: The Fitted Prioritized Sweeping Algorithm

Each state was represented by a vector of attributes. For Hexcer there were 7 attributes. These were the x and y coordinates of each player, the difference in x and y coordinates between the two players, and the location of the ball (On X, on O or in the middle of the board). The values in the Q table were sorted by each attribute in turn. For each attribute, the halfway point between every adjacent pair of values was considered as a possible split point. A linear fit was made of the data on each side of this split point and the χ^2 statistic of the combined fits was compared to the χ^2 statistic of a single linear fit for all the data. If the split with the best combined χ^2 statistic is better than no split then that split is recorded and the splitting proceeds recursively.

Continuous U Tree

The U Tree algorithm (McCallum 1995) is a method for using a decision or regression tree (Breiman *et al.* 1984; Quinlan 1986) instead of a table of values in the Prioritized Sweeping algorithm. In the original U Tree algorithm the state space is described in terms of discrete attributes. If a continuous value is needed it is split over multiple attributes in a manner analogous to using one attribute for each bit of the continuous value. We developed a "Continuous U Tree" algorithm, similar to U Tree, but which handles continuous state spaces directly.

Like U Tree, Continuous U Tree can handle moderately high dimensional state spaces. The original U Tree work uses this capability to 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.

Continuous U Tree is different from the Prioritized Sweeping algorithms previously mentioned in that it does not require a prior discretization of the world into separate states. The algorithm can take a continuous state vector and automatically form its own discretization upon which any Reinforcement Learning algorithm can be used. We used Prioritized Sweeping over this new discretization.

In the Continuous U Tree algorithm there are two distinct concepts of state. The first type of state is the full continuous state of the world the agent is moving through; or at least what the agent can see of it. 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 sec-

ond sense. Each state is an area in the sensory input space, and each sensory input falls within a state. In previous algorithms these two concepts were identical. Sensory input was discrete and each different sensory input corresponded to a state.

In order to form the discretization, data must be saved from the agent's experience. Unfortunately, as the agent is forming its own discretization, there is no prior discretization that can be used to aggregate the data. The datapoints must be saved individually with full sensor accuracy.

Each datapoint saved is a vector of the sensory input at the start I, the action performed a, the resulting sensory input I' and the reward obtained for that transition r, (I, a, I', r). As in Fitted Prioritized Sweeping the sensory input is itself a vector of values, one for each attribute of the sensory input. Unlike Fitted Prioritized Sweeping these values can be fully continuous. In Hexcer we used the same sensory input vector for Continuous U Tree and Fitted Prioritized Sweeping.

The discretization formed is a tree-structure found by recursive partitioning of the datapoints. Initially the world is considered to be a single state with an expected reward, $V(s)$, of 0. At each stage in the recursive partitioning we re-calculate expected reward values of all the datapoints and if a significant difference between datapoint values is found within a state that state is split in two.

To calculate the value of a datapoint, $q(I, a)$ we rely on the Bellman equations. Each datapoint ends in a state. That is, the resulting sensory input, I', of a transition, (I, a, I', r), can be passed down the tree structure till it reaches a leaf. That leaf is a state in the discretization, say s'. That state is considered to be the final state for the transition. Using the expected reward for that state, $V(s')$, and the recorded reward for the transition, r, we can assign a value to the initial sensory input/action pair: $q(I, a) = r + \gamma V(s')$.

For each state in the current discretization we can then loop through all possible single splits for that state and choose the split that maximizes the difference between the datapoint values, $q(I, a)$, on either side of the split. This difference is measured using the Kolmogorov-Smirnov statistical test[1]. If the most statistically significant split point has a probability of being random less than some threshold (we used $p < 0.01$) then that split is added - forming two new states to replace the old one.

Having calculated values for the datapoints and used that to discretize the world we fall back into a standard, discrete, Reinforcement Learning problem: we need to find Q values for our new states, $Q(s, a)$. This is done by calculating state transition probabilities from the given data and then using Prioritized Sweeping. If the initial sensory input and the resulting sensory input for a datapoint are in the same state then that is considered a self-transition. If the initial and final sensory inputs are in two different states then that is an example of the action resulting in a state change. Figure 4 contains the Continuous U Tree algorithm.

During Game:
- Pass the current sensory input, I, down the state tree to find the current state in the discretization, s
- Use Q values for the state s to choose an action, a
- Store transition datapoint (I, a, I', r)

After Game:
- For each leaf:
  - Update datapoint values, $q(I, a)$, for each datapoint in that leaf
  - Find best split point
  - If split point is statistically significant then split leaf into two states
- Add all states to priority queue
- Run Prioritized Sweeping over new states until all changes are below threshold

Figure 4: The Continuous U Tree Algorithm

Results

We performed empirical comparisons along two different dimensions. Firstly, how fast does each algorithm learn to play? Secondly, what level of expertise is reached, discounting a "reasonable" initial learning period?

In each experiment a 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 allow us to measure learning speed as all agents start out with no knowledge of the game. The second 500 games give an indication of the level of ability of the algorithm after an initial learning period.

This 1000 game test was then repeated 20 times for each pair of algorithms. The results shown in each table are the number of games (mean $\pm$ standard deviation) that the first player listed wins. A check mark ($\surd$) indicates statistical significance, i.e., the probability random variation would produce a difference this great is less than 1%.

As well as being tested against each other, we also compare these algorithms to Prioritized Sweeping which we showed in Uther & Veloso (1997) was one of the better non-generalizing algorithms.

In Table 1 Fitted Prioritized Sweeping is compared to standard Prioritized Sweeping. It learns significantly faster than standard Prioritized Sweeping, and keeps performing better even after the initial games. Standard Prioritized Sweeping hasn't even finished

[1] see Press *et al.* (1992) for more information on these sorts of tests.

	Prioritized Sweeping		Fitted Prioritized Sweeping		
First 500 games	75 ± 74	$15\% \pm 15\%$	425 ± 74	$85\% \pm 15\%$	$\surd$
Second 500 games	133 ± 81	$27\% \pm 16\%$	367 ± 81	$73\% \pm 16\%$	$\surd$
Total	208 ± 135	$21\% \pm 14\%$	792 ± 135	$79\% \pm 14\%$	$\surd$

Table 1: Prioritized Sweeping vs. Fitted Prioritized Sweeping

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

Table 2: Prioritized Sweeping vs. Continuous U Tree

learning after this short trial period. Fitted Prioritized Sweeping has learnt to be an effective player with much less data. While effective, the Fitted Prioritized Sweeping algorithm still requires a prior discretization of the world. It cannot handle continuous state spaces.

Table 2 shows that Continuous U Tree performs better than standard Prioritized Sweeping, although the differences for the second 500 games are only significant at the 5% level. Table 3 shows that Continuous U Tree also performs comparably to Fitted Prioritized Sweeping. These differences are not statistically significant at the 1% level. However, the results for the total are significant at the 5% level.

Continuing Work

We are currently extending this work along 4 different dimensions.

Firstly, we expect the value function to be in an exponential form. Trees with constant value leaves do not represent continuous functions very efficiently. To solve this we are looking at embedding exponential functions in the leaves of the tree. Preliminary results show that this system grows smaller trees than using constant valued leaves.

We are also extending U Tree to handle continuous action spaces. A game similar to hexcer, but using a fully continuous state and action space where each player sets a pair of polar coordinates for each move has been developed. An *action tree* is placed in each leaf of the state tree. Each action tree discriminates between regions of the action space in that state that have different expected rewards. Each of these regions can then be treated as a separate action for the discrete Reinforcement Learning algorithm used.

We are exploring the use of constructive induction in the state tree to build abstractions over the state-space and so improve both generalization and planning.

We are intending to move towards teams of agents on a larger board to test the generalization capabilities of U Tree.

Conclusion

Introducing extra agents into a domain increases the state space exponentially. Not only does this affect learning speed, but often the exact location of the other agents is not relevant. We show effective generalizing Reinforcement Learning algorithms to help reduce this state space size explosion. Both our generalizing algorithms perform significantly better than non-generalizing algorithms. Unlike naive use of generalization, both algorithms are guaranteed to converge. The second algorithm, Continuous U Tree, does not require a prior discretization of the state space. It performs comparably to the Fitted Prioritized Sweeping algorithm which uses a discretization.

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.

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.

Boyan, J. A., and Moore, A. W. 1996. Learning evaluation functions for large acyclic domains. In Saitta, L., ed., *Machine Learning: Proceedings of the Thirteenth International Conference (ICML96)*. San Mateo, CA: Morgan Kaufmann.

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.

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

	Fitted Prioritized Sweeping		Continuous U Tree	
First 500 games	246 ± 101	$49\% \pm 20\%$	254 ± 101	$51\% \pm 20\%$
Second 500 games	223 ± 73	$45\% \pm 15\%$	277 ± 73	$55\% \pm 15\%$
Total	461 ± 91	$46\% \pm 9\%$	539 ± 91	$54\% \pm 9\%$

Table 3: Fitted Prioritized Sweeping vs. Continuous U Tree

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.

Press, W. H.; Teukolsky, S. A.; Vetterling, W. T.; and Flannery, B. P. 1992. *Numerical Recipies 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.

Samuel, A. L. 1959. Some studies in machine learning using the game of checkers. *IBM Research Journal* 3(3).

Tesauro, G. 1992. Practical issues in temporal difference learning. *Machine Learning* 8:257–277.

Tesauro, G. 1995. Temporal difference learning and td-gammon. *Communications of the ACM* 38(3):58–67.

Thrun, S. 1995. Learning to play the game of chess. In Tesauro, G., and Touretzky, D. S., eds., *Advances in Neural Information Processing Systems*, volume 7. Cambridge, MA: The MIT Press.

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.