Adversarial Reinforcement Learning

William Uther Manuela Veloso

January 2003
CMU-CS-03-107

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This manualscript was originally submitted for publication in April 1997. Corrections were never completed, and so the paper was not published. However, a copy was placed on the web and a number of people referenced the work from there. It is now being published as a technical report for ease of reference. With the exception of the addition of this title page, the work is unmodified from the 1997 original.

Keywords: Reinforcement Learning, Markov Games, Adversarial Reinforcement Learning

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. We introduce a new framework, two-player hexagonal grid soccer, in which to evaluate algorithms. We then compare the performance of several single-agent Reinforcement Learning techniques in that environment. These are further compared to a previously developed adversarial Reinforcement Learning algorithm designed for Markov games. Building upon these efforts, we introduce new algorithms to handle the multi-agent, the adversarial, and the continuous-valued aspects of the domain. We introduce a technique for modelling the opponent in an adversarial game. 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. Extensive empirical evaluation is conducted in the grid soccer domain.

This page intentionally left blank.

Adversarial Reinforcement Learning

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

April 24, 1997

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. We introduce a new framework, two-player hexagonal grid soccer, in which to evaluate algorithms. We then compare the performance of several single-agent Reinforcement Learning techniques in that environment. These are further compared to a previously developed adversarial Reinforcement Learning algorithm designed for Markov games. Building upon these efforts, we introduce new algorithms to handle the multi-agent, the adversarial, and the continuous-valued aspects of the domain. We introduce a technique for modelling the opponent in an adversarial game. 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. Extensive empirical evaluation is conducted in the grid soccer domain.

1 Introduction

Multi-agent adversarial environments have traditionally been addressed as game playing situations. Indeed, one of the first areas to be 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 [Kaelbling *et al.*, 1996]. An evaluation function that guides the selection of moves is represented as a parameterized weighted sum of game features. Parameters are incrementally refined as a function of the game playing performance. This is a similar method to classical Reinforcement Learning which also provides for incremental update of an evaluation function, although in this case it is represented as a table of values.

Since Samuel's work however, Reinforcement Learning techniques were not used again in an adversarial setting until quite recently. [Tesauro, 1995, Thrun, 1995] have both used neural nets in a Reinforcement Learning paradigm. [Tesauro, 1995]'s work in the game of checkers was successful, but required hand tuned features being fed to the algorithm for high quality play. [Thrun, 1995] was moderately successful in using similar techniques in chess, but these techniques were not as successful as they had been in the checkers domain. This work has been repeated in other domains, but again, without the same success as in the checkers domain (in [Kaelbling *et al.*, 1996]).

[Littman, 1994] took standard Q Learning, [Watkins and Dayan, 1992], and modified it to work with Markov games. He replaced the simple min update used in standard Q Learning with a mixed strategy (probabilistic) $minimax$ update. He then evaluated this by playing against both standard Q Learning and random players in a simple game. The game used in [Littman, 1994] is a small two player grid soccer game designed to be able to be solved quickly by traditional Q Learning techniques. He trained 4 different players for his game. Two players used his algorithm, two used normal Q Learning. One of each was trained against a random opponent, the other against an opponent of the same type. Littman then froze those four players and trained 'challengers' against them. His results showed that his algorithms, which learned a probabilistic strategy, performed better under these conditions than Q Learning, which learned a deterministic strategy, or his hand coded, but again deterministic, strategy.

We use a similar environment to that used by [Littman, 1994] to investigate Markov games. Our environment is larger, both in number of states and number of actions per state, to more effectively test the generalization capabilities of our algorithms. We conduct tests where both players are learning as they play. This allows learning to take the place of a mixed, or probabilistic, strategy. We look at a number of standard Reinforcement Learning algorithms and compare them in a simple game. None of the algorithms we test perform any internal search or lookahead when deciding actions; they all use just the current state and their learnt evaluation for that state. While search would improve performance, we considered it orthogonal, and a future step, to learning the evaluation function.

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.

In an adversarial setting there are multiple (at least two) agents in the world. In particular, 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.

In this paper we investigate the performance of some previously published algorithms in an adversarial environment; Q Learning, Minimax Q Learning, and Prioritized Sweeping. We also introduce a new algorithm, Opponent Modelling Q Learning, to try and improve upon these algorithms. All of these techniques rely on a table of values and actions and do not generalize between similar or equivalent states. The learned tables are "state-specific." We introduce 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. Finally, we look at what can be learned by looking at the world from your opponent's point of view.

2 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, two players and a ball (See Figure 1).

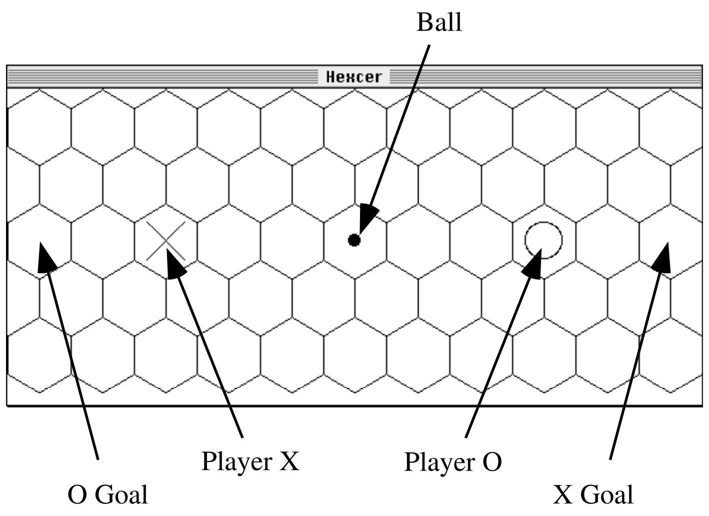

Figure 1: The Hexcer board

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 of one 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 new cell is made randomly. If one of the players had the ball when this occurs then the player that succeeds in moving to the contested cell takes possession of the ball.

Play continues until the ball arrives in either goal. At this time the player who owns the goal with the ball wins the game and receives some reward. The opposing player gets an equal, but negative, 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. The learning player observes its position, and the positions of the ball and 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.

We incrementally compare different algorithms in this domain and develop new extensions based on an analysis of their performances. We perform empirical comparisons along two different dimensions. Firstly, how fast does the algorithm learn to play? Secondly, what level of expertise is reached, discounting a "reasonable" initial learning period?

In the next sections, we introduce in detail the algorithms and the experiments. We show consistently for all the experiments two sets of results. A pair of algorithms played each other at Hexcer for 100,000 games. Wins were recorded for the first 50,000 games and the second 50,000 games. The first 50,000 games allow us to measure learning speed as all agents start out with no knowledge of the game. The second 50,000 games give an indication of the final level of ability of the algorithm.

This 100,000 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) 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%.

The generalizing Reinforcement Learning algorithms are significantly slower to update than the state-specific algorithms. Hence, it is not feasible to run 100,000 games for these algorithms. However the generalizing algorithms learn to play in fewer games than the state-specific ones. To compare the generalizing algorithms only 1,000 games were played. Similarly to the rest of the experiments, these 1,000 games were split into the first 500 games and the second 500 games to measure learning speed and final learned ability.

3 State-Specific Learning Algorithms

In the machine learning formulation of Reinforcement Learning [Kaelbling *et al.*, 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 pair, (s, a) there is a reinforcement signal, $R(s, a)$. Action a by the agent when it is in state s gives the agent the reward associated with that state/action pair, $R(s, a)$.

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." One might expect $G = \sum_{t=1}^{\infty} R_t$ - we just sum the reinforcement signal for every timestep to get some measure of how we are doing. Unfortunately, this sum diverges. The standard solution, although others have been tried (see [Kaelbling *et al.*, 1996]), is to 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, $\gamma = 0.95$.

3.1 Q Learning

Q Learning [Watkins and Dayan, 1992] is a method of building a table of values that can be used to decide how to act so as to maximize the agent's discounted reward over time. The foundations of this method are the Bellman Equations:

$$\begin{aligned} Q(s,a) &= R(s,a) + \gamma \sum_{s'} P_{(s,a)}(s')V(s') && (1) \\ V(s) &= \max_a(Q(s,a)) && (2) \end{aligned}$$

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. 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.

It is possible, if the state transition probabilities and rewards are known, to solve these equations directly for the Q function. Once this is known an agent can behave optimally simply by picking the action that has the highest Q value in the current state. If there is a tie for highest Q value, then it does not matter which option is taken; usually the choice is made randomly. Unfortunately, these state transition probabilities and rewards are usually not known in advance.

Q Learning is a technique for learning the Q function online as the agent explores. The Q function is recorded in a Q table, which stores the current estimates of $Q(s,a)$, i.e., the Q values for each state/action pair. Initially all these estimates are 0. As the agent performs each action it is able to update the Q value corresponding to that action.

$$Q(s,a) := \alpha Q(s,a) + (1-\alpha)(R(s,a) + \gamma V(s')) \quad (3)$$

This is achieved by using the Bellman equations and replacing the equality with a weighted sum assignment. The Q value of the state/action is updated to be closer to the Q value for this state/action/next-state transition. The probability distribution, $P_{(s,a)}(s')$, can be removed from the assignment, because the destination states are distributed according to that distribution. The learning rate, α, determines how fast the Q values are updated. In our experiments, we use a fixed learning rate, $\alpha = 0.5$. The Q values will converge to the correct value with probability 1, assuming that the learning rate decreases slowly towards zero and each state/action pair is sampled infinitely often. In order to continue learning, we did not decrease the learning rate. While this removes the convergence guarantee, it allows the algorithm to play against an opponent that is itself learning. In this case, there is no fixed simple, i.e. non-stochastic, optimal policy anyway. To ensure adequate exploration, i.e., every state is visited infinitely often in an infinite series of games, all players described chose a random action with 5% probability.

This formulation works well, but its direct implementation is inefficient for large problems. Exploration is usually achieved by adding a small probability that a random action will be chosen. Given a slowly decreasing learning rate, and the guarantee that every state/action pair will be visited infinitely often this algorithm is guaranteed to converge to an optimal policy. As stated above, we used a fixed learning rate. We trading guaranteed convergence for the ability to learn the non-stationary policy we need to play against an opponent that is itself learning.

3.2 Minimax Q Learning

Minimax Q Learning [Littman, 1994] is a modification of Q Learning to work with Markov games. Instead of treating the opponent as part of the environment, Minimax Q Learning records a function not just from state/action pairs to values, but from state/action/action triples to values, $Q(s, a_m, a_o)$. Both the action of the agent, a_m, and the action of its opponent, a_o, are explicitly modelled. Table 1 shows an example of a Minimax Q Learning table for a particular state. The columns correspond to our agent's actions, and the rows to the opponent's actions. The values are the expected sum of discounted reward for that action pair in that state.

		My Moves, a_m	
		a_m^0	a_m^1
Their Moves, a_o	a_o^0	500	175
	a_o^1	150	450
	a_o^2	375	200

Table 1: A sample minimax Q table, $Q(s, a_m, a_o)$, for a state s

Once this table of values exists game theory is used to work out the value of this position of the board, $V(s)$. As moves are made simultaneously, the max in the equation for the value of a state is replaced with a $minimax$ function. The backup of values is the same as Q Learning. The minimax value is calculated using linear programming (see below). This returns a probability distribution over the actions for the agent. The agent chooses its action according to this distribution.

Calculation of the value of a state where there is hidden information, such as what move your opponent is going to make, relies on game theory and linear programming. Consider Figure 2 which represents the same data as Table 1. The x-axis represents a probability distribution over our two actions, $P(a_m)$. On the left we choose move a_m^0 all the time. On the right we choose move a_m^1 all the time. As we move along the x axis, we change the probability that we will pick a particular move. We want to find the point on this axis that maximizes our expected return.

If we fix the probability distribution over our actions, $P(a_m)$, then the expected return assuming the opponent makes move a_o is going to be:

$$E(V(s|a_o)) = \sum_{a_m} P(a_m)Q(s, a_m, a_o) \tag{4}$$

As we move along the x axis, our expected return if the opponent chooses a particular move varies as a linear combination of its endpoint values. The endpoint values are the entries in the Q table, the lines represent the linear combinations; one for each opponent move.

We wish to maximize the expected reward regardless of the opponents action. This corresponds to finding the highest point in Figure 2 that is beneath all the opponent move lines. Projecting this point down onto the x axis gives the required distribution over our actions. The y coordinate of this point is the expected reward for moving with that probability distribution. Finding this point is a classical linear programming problem. (See [Press *et al.*, 1992] for more detail.)

In the example above, the maximum expected reward is at the intersection of the lines generated from opponent moves a_o^1 and a_o^2. The result is that we should choose move a_m^0 with

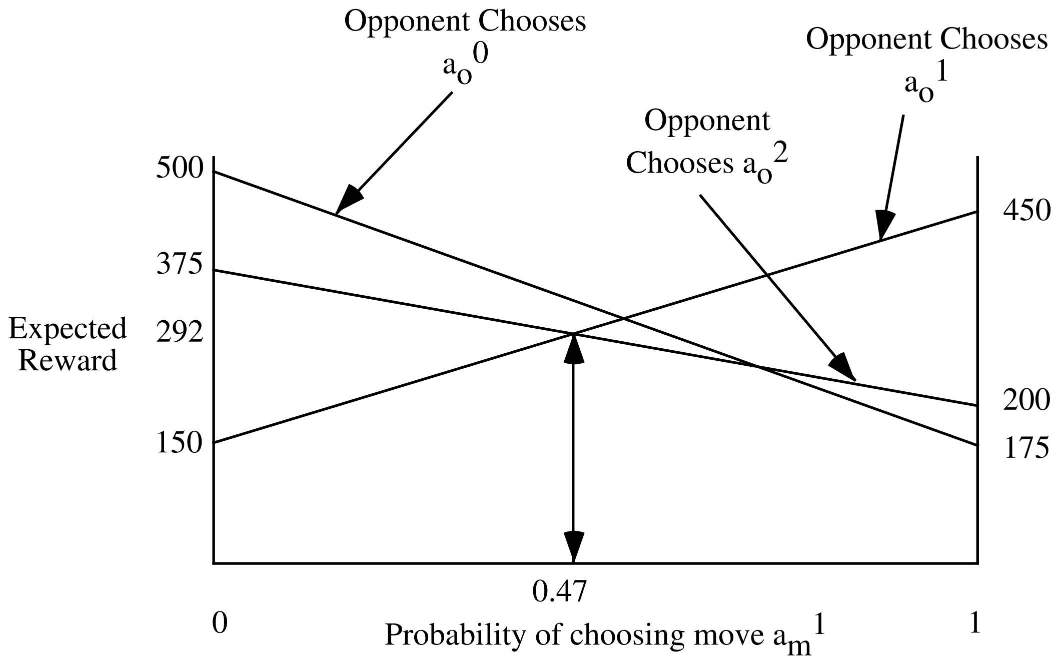

Figure 2: Linear Programming for Mixed Strategies

probability 0.47, and move a_m^1 with probability 0.53. The expected sum of discounted reward is 292. This the value, $V(s)$, of this state.

As [Littman, 1994] points out, this is a pessimistic value. Whatever the opponent actually does, the expected return value generated, and probability distribution over our moves, will be that for playing an optimal opponent. This means that the Minimax Q Learning algorithm learns a strategy for playing an optimal opponent no matter who it plays while learning.

The Minimax Q Learning algorithm has a similar constraint to that of normal Q Learning. All Q values must be updated infinitely often for this to converge with probability 1. In Q Learning this poses no real problem as the agent controls what action is taken and can insert some exploration (random move selection). In Minimax Q Learning this can be a problem as the agent requires all combinations of its move and the opponents move to visited infinitely often. It only controls its move however. This algorithm will learn better against a random opponent, who will try each move infinitely often, than against an optimal opponent if the optimal opponent never makes some moves.

Minimax Q Learning being pessimistic is also a problem at the start of the game. When a particular entry in the Q table is still zero (i.e. it hasn't been tried yet), it will force the player away from that column. Consider the very first game played. The algorithm chooses randomly because all entries are 0. It finally scores a goal through pure luck. Unfortunately, the pessimistic estimate of that state's value will still be 0 because you've only seen it with the opponent moving in one direction. Without changes, this algorithm requires that we see the opponent fail to stop us after trying all possible moves before we will back the win up even one state.

To help alleviate this, we introduced a mild form of generalization into the algorithm. If a particular Q value has never been changed, then it is considered to be equal to the lowest value in that column that has been examined. Values are never pushed below zero in this fashion.

Without this change the algorithm has limited applicability.

3.3 Q Learning and Minimax Q Learning in Hexcer

Table 2 shows the results of 100,000 games between the standard Q Learning algorithm and the Minimax Q Learning algorithm. As can be seen in the first row, Q Learning learns much faster than Minimax Q Learning. Minimax Q Learning never quite catches up, although the difference eventually becomes small.

	Q Learning vs. Minimax Q Learning		
First 50000 games	36637 ± 3412	$73\% \pm 7\%$	$\surd$
Second 50000 games	27289 ± 3115	$55\% \pm 6\%$	$\surd$
Total	63926 ± 4706	$64\% \pm 5\%$	$\surd$

Table 2: Q Learning vs. Minimax Q Learning

The reason for this is that the stochastic updates in normal Q Learning result in a distribution of moves that approximates the optimal mixed strategy. As long as these updates continue an explicit mixed strategy is not necessary. As [Littman, 1994] points out though, if you stop learning then the explicit mixed strategy is an advantage against an opponent that is learning.

3.4 Opponent Modelling Q Learning

Minimax Q Learning is a pessimistic algorithm. It assumes it will be playing an optimal opponent and so learns to play against one. We should be able to do better by modelling the opponent, taking advantage of the opponent's sub-optimal moves.

We developed a new algorithm, "Opponent Modelling Q Learning," which achieves this desired behavior. It assumes that the opponent is Markov, i.e., the history of how the opponent arrived in its current state is unimportant - it will behave the same way in this state. We record the number of times our opponent chooses each action in each state as well as the Q table recorded by Minimax Q Learning, $Q(s, a_m, a_o)$. This gives us the probability distribution of our opponents actions in each state, $P(a_o|s)$, which is then used to calculate the best action for us. While this Markov assumption may not be entirely correct, we are assuming a Markov game so the optimal player is Markov and this assumption does not seem too unreasonable. Table 3 is an example of a Q table used by the Opponent Modelling Q Learning algorithm. For each state we have the probability distribution over our opponents moves recorded as well as the minimax Q table.

	Probabilities, $P(a_o|s)$	My Moves, a_m	
Their Moves, a_o	20%	500	175
	30%	150	450
	50%	375	200
Expected Reward		332.5	270

Table 3: A sample Opponent Modelling Q table

We can find the expected value of one of our moves by simply taking the weighted average of each column:

$$E(Q(s,a_m)) \quad = \quad \sum_{a_o} P(a_o|s)Q(s,a_m,a_o) \tag{5}$$

The agent picks the action with the highest expected reward in this state, $Q(s,a_m)$. The value of this action is the value, $V(s)$, for that state. This method of picking an action is known in the game theory literature as 'solution by fictitious play' [Owen, 1995]. It has been shown that two players using this method against each other repeatedly, with a single state and known move values, will converge. The probability distribution of each players actions will converge to the same optimal distribution as found by the linear programming method above although at any particular time one action will be best.

This method of choosing an action does not suffer from most of the problems of Minimax Q Learning. If an opponent never makes a particular move, then it's associated probability will be zero and the relevant Q table entries will be ignored.

3.5 Opponent Modelling Q Learning in Hexcer

Tables 4 and 5 show that opponent modelling is an advance over both normal Q Learning and minimax Q Learning. Normal Q Learning learns a little faster than the opponent modelling, but does not learn to play as well. The extra time for Minimax Q Learning to learn to play is easily explained by its increased table size. Opponent Modelling is better than Minimax Q Learning in both dimensions.

	Q Learning vs. Opponent Modelling		
First 50000 games	19863 ± 11439	$40\% \pm 23\%$	
Second 50000 games	20293 ± 6179	$41\% \pm 12\%$	$\surd$
Total	40157 ± 11933	$40\% \pm 12\%$	$\surd$

Table 4: Q Learning vs. Opponent Modelling

	Minimax Q Learning vs. Opponent Modelling		
First 50000 games	16143 ± 5195	$32\% \pm 10\%$	$\surd$
Second 50000 games	21690 ± 4456	$43\% \pm 9\%$	$\surd$
Total	37833 ± 6150	$38\% \pm 6\%$	$\surd$

Table 5: Minimax Q Learning vs. Opponent Modelling

It should be noted that the opponent modelling strategy is not a mixed (probabilistic) strategy. If we stopped performing updates as [Littman, 1994] did then it would fail the same way Q Learning does against a learning opponent.

So, why is opponent modelling effective? On the surface it is very similar to the original Q Learning, except Q Learning used stochastic updates to model the opponent in the same way it uses stochastic updates to model transition probabilities. The answer is in the fact that changing opponent move probabilities can occur much faster than Q Learning will update its Q table. If the opponent is changing its behavior, as is happening in our experiments as the opponent learns, then opponent modelling allows more efficient credit blame assignment. The change in opponent behavior is detected directly and can cause changes in backed up value much faster than the normal learning rate would allow.

3.6 Prioritized Sweeping

Prioritized Sweeping [Moore and Atkeson, 1993] is significantly different from the preceeding methods. Instead of relying on sampling to model the state transition probability distribution, $P_{(s,a)}(s')$, implicitly, Prioritized Sweeping proceeds by building a model of the world explicitly. Using this model it can calculate Q values directly rather than just performing stochastic updates when data arrives. This allows it to propagate significant changes.

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 whose inputs have changed most.

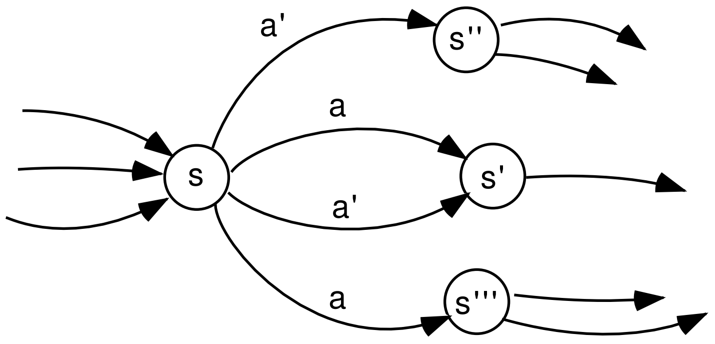

Figure 3: The Graph stored by Prioritized Sweeping

As the agent moves through the environment a state transition graph is built (See Figure 3). 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.

The number of updates that are performed per step the agent takes is a parameter that can be optimised for the task. In order to make comparisons as fair as possible between different algorithms, we only allowed Prioritized Sweeping 2 updates per move - one to insert the current node in the priority queue with the correct weight, and one from the priority queue. It was performing twice as many updates as standard Q Learning, and was also able to use it's priority queue to perform the most useful updates. The second update might not update the state it just moved from, but rather propagate an important change it made previously.

Prioritized Sweeping also remembers all information gathered, unlike Q Learning which forgets about any transitions between like-valued nodes.

3.7 Prioritized Sweeping in Hexcer

Tables 6, 7 and 8 show the results of Prioritized Sweeping playing against all the previous algorithms. It beats all of them by a significant margin. The reason for this is very simple. All the previous methods make very poor use of the data. On the first run, most of the updates have no effect as all Q values have the same, zero, value. Only the last update actually causes a table entry to change value - most of the exploration is lost.

	Q Learning vs. Prioritized Sweeping		
First 50000 games	5349 ± 970	$11\% \pm 2\%$	$\surd$
Second 50000 games	5797 ± 547	$12\% \pm 1\%$	$\surd$
Total	11145 ± 1318	$11\% \pm 1\%$	$\surd$

Table 6: Q-learning vs. Prioritized Sweeping

	Minimax Q Learning vs. Prioritized Sweeping		
First 50000 games	3069 ± 482	$6\% \pm 1\%$	$\surd$
Second 50000 games	6149 ± 530	$12\% \pm 1\%$	$\surd$
Total	9216 ± 763	$9\% \pm 1\%$	$\surd$

Table 7: Minimax Q-learning vs. Prioritized Sweeping

	Opponent Modelling vs. Prioritized Sweeping		
First 50000 games	6257 ± 1948	$13\% \pm 4\%$	$\surd$
Second 50000 games	8235 ± 1019	$16\% \pm 2\%$	$\surd$
Total	14493 ± 2282	$14\% \pm 2\%$	$\surd$

Table 8: Opponent Modelling vs. Prioritized Sweeping

In Prioritized Sweeping all that data is saved. Updates are then performed in the most useful places first. Using the ability of Prioritized Sweeping to save all data and use it when relevant to offset the larger table size of the minimax update gives minimax Prioritized Sweeping.

4 Generalizing Algorithms

All the algorithms above have one 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 and Moore, 1995] never converging to a solution although some good results have been obtained with this method [Tesauro, 1995]. Methods that are stable do 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, 1995]'s method using Decision Trees is described and we extend it to handle continuous state values.

4.1 Fitted Prioritized Sweeping

As pointed out in [Boyan and Moore, 1996], generalizing function approximators are not a problem if the approximation is not iterated. That is, 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.

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 4 contains the Fitted Prioritized Sweeping Algorithm.

Each state/action pair was represented by a vector of attributes. For Hexcer there were 8 attributes. These were the x and y coordinates of each player, the difference in x and y coordinates between the two players, the location of the ball (On X, on O or in the middle of the board), and the action selected (1 through 6). 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

```
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 4: The Fitted Prioritized Sweeping Algorithm

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.

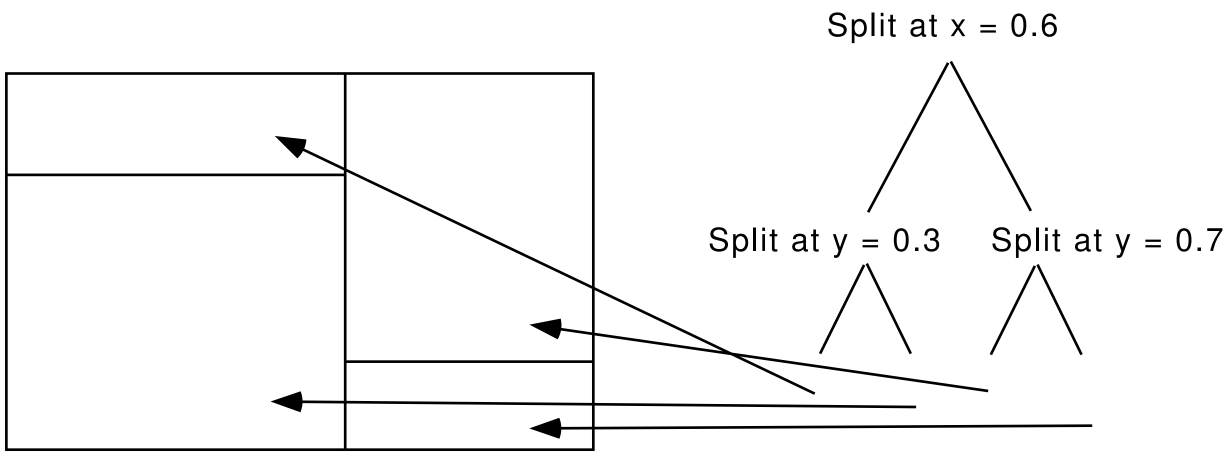

Figure 5: A simple recursive partition of a rectangle

Consider Figure 5. We start off with a two dimensional space of data. This is first split at $x = 0.6$. Each side of this split is then itself split recursively. Once down two levels there was not enough data separation to justify another split so the recursion stops.

4.2 Fitted Prioritized Sweeping in Hexcer

This generalizing algorithm was tested against standard Prioritized Sweeping. As can be seen in Table 9 it learns significantly faster than standard Prioritized Sweeping. While effective, this algorithm still requires a prior discretization of the world. It cannot handle continuous state spaces.

4.3 Continuous U Tree

The U Tree algorithm [McCallum, 1995] is a method for using a decision tree [Quinlan, 1986] instead of a table of values in the Prioritized Sweeping algorithm. In the original U Tree

	Prioritized Sweeping vs. Fitted Prioritized Sweeping		
First 500 games	75 ± 74	$15\% \pm 15\%$	$\surd$
Second 500 games	133 ± 81	$27\% \pm 16\%$	$\surd$
Total	208 ± 135	$21\% \pm 14\%$	$\surd$

Table 9: Prioritized Sweeping vs. Fitted Prioritized Sweeping

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 we see no reason why this could not be done.

Continuous U Tree is different from all the previous algorithms 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 one of the previous algorithms can be used. We used Prioritized Sweeping over this new discretization.

In order to form this discretization, data must be saved from the agent’s experience. Unfortunately, there is no prior discretization that can be used to aggregate the data. The datapoints must be saved individually with full sensor accuracy. In previous algorithms, each different, but discrete, sensory input would correspond to a different state and hence would require its own Q table entry. Here that is not the case. Areas of similar sensory input are grouped together by the algorithm to form states. 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 vector of state attributes for Continuous U Tree and Fitted Prioritized Sweeping. No action attributes are added to this vector however (i.e. only the first 7 of the 8 attributes listed above for Fitted Prioritized Sweeping are used).

The discretization formed is a tree-structure found by recursive partitioning. 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. 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.

Calculating the value of a datapoint, $q(I, a)$ is again based on the Bellman equations. Each datapoint ends in a state, i.e., the resulting sensory input I' of a transition, (I, a, I', r), will fall within a particular state in the discretization, say s'. This 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

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

transition, r, we can assign a value to the initial sensory input/action pair.

$$q(I, a) = r + \gamma V(s') \tag{6}$$

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 6 contains the Continuous U Tree algorithm.

```
During Game:
    Find current state in discretization, $s$, using current sensory input, $I$
    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 thresh-
        old
```

Figure 6: The Continuous U Tree Algorithm

Empirically, when we used this algorithm including self transitions the state values dropped to 0 very quickly. The large states gave large numbers of self-transitions which reduced the value of the state. To avoid this we ignored self-transitions when performing the Prioritized Sweeping.

4.4 Continuous U Tree in Hexcer

Table 10 shows that Continuous U Tree performs better than normal Prioritized Sweeping, although the differences for the second 500 games are only significant at the 5% level. Table 11 shows that continuous U Tree also performs better than Fitted Prioritized Sweeping. Although the results for the total are statistically significant at the 5% level, none of these differences are significant at the 1% level.

5 Impact of Watching the Opponent

In the previous section we noted that tabular Reinforcement Learning techniques require a large amount of data to learn. To overcome this we introduced generalization. Another way

	Prioritized Sweeping vs. Continuous U Tree		
First 500 games	175 ± 93	$35\% \pm 19\%$	$\surd$
Second 500 games	197 ± 99	$39\% \pm 20\%$	
Total	372 ± 183	$37\% \pm 18\%$	$\surd$

Table 10: Prioritized Sweeping vs. Continuous U Tree

	Continuous U Tree vs. Fitted Prioritized Sweeping		
First 500 games	254 ± 101	$51\% \pm 20\%$	
Second 500 games	277 ± 73	$55\% \pm 15\%$	
Total	539 ± 91	$54\% \pm 9\%$	

Table 11: Continuous U Tree vs. Fitted Prioritized Sweeping

to help overcome this is to acquire more data. One way to do this is to watch your opponent move.

This data has different properties to watching our own movement. The quality of the data depends on our opponents level of skill. If our opponent is a better player then the data on how they play is useful. If our opponent is poor, then the data is less useful.

A problem with using opponent data occurs with the algorithms that use some form of opponent modelling (The opponent modelling and the Prioritized Sweeping algorithms). From our opponents point of view we are the opponent. If we just flip some axes in the state representation, then anything we do will be incorporated into our model of our opponent. We need to be able to use the extra data without disturbing our opponent model.

In Hexcer it is possible to get 4 datapoints from each move. Firstly the original datapoint is duplicated by vertical mirroring. This isn't opponent watching, but gives us more data. These two points are then mirrored horizontally - watching our opponent.

To incorporate opponent watching into Prioritized Sweeping we needed to be careful of modelling ourselves. To make sure that this didn't happen, any data gained by modelling our opponent was only used to add links to the state transition graph. Once a link was known about it's transition count was not incremented when that link was seen through a flipped world view.

For opponent watching, the opponent move probabilities were not updated for data gained watching the opponent.

Tables 12 and 13 show the usefulness of opponent watching. We can see in table 12 that if your opponent is the same as you it does not help much in absolute terms, although the difference is statistically significant. If your opponent has better exploration than you (Table 13) then the difference seems minimal (Compare with Table 9).

	Prioritized Sweeping vs. Prioritized Sweeping w/ Opponent Watching		
First 50000 games	23200 ± 956	$46\% \pm 2\%$	$\surd$
Second 50000 games	25074 ± 373	$50\% \pm 1\%$	
Total	48274 ± 1091	$48\% \pm 1\%$	$\surd$

Table 12: Prioritized Sweeping vs. Prioritized Sweeping w/ Opponent Watching

	Fitted Prioritized Sweeping vs. Prioritized Sweeping w/ Opponent Watching		
First 500 games	112 ± 48	$22\% \pm 10\%$	$\surd$
Second 500 games	66 ± 43	$13\% \pm 9\%$	$\surd$
Total	178 ± 75	$18\% \pm 8\%$	$\surd$

Table 13: Fitted Prioritized Sweeping vs. Prioritized Sweeping w/ Opponent Watching

6 Conclusion

Standard Reinforcement Learning algorithms work in an adversarial domain. Algorithms, like Prioritized Sweeping, that have been shown to be more effective in single agent domains are again more effective in the domain studied here. However, the exponential explosion in state space size as the number of agents increases limits the problem size accessible by these algorithms.

The Minimax Q Learning algorithm proposed by Littman has a major problem in that it requires your opponent's help to explore the space. Without minor generalization in the table it is almost useless. Even with this generalization it is slower to learn than single agent methods. It does offer the advantage of being able to learn to play optimally.

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 the table based algorithms.

Finally we investigated learning from looking at the opponent. This was seen to be effective if your opponent knows more than you. You have to be careful that any opponent modelling does not become confused.

References

[Baird, 1995] Leemon Baird. Residual algorithms: Reinforcement learning with function approximation. In A Prieditis and S Russell, editors, *Proceedings of the Twelfth International Conference on Machine Learning*, pages 30–37, San Francisco, C.A., 1995. Morgan Kaufmann.

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

[Boyan and Moore, 1996] Justin A. Boyan and Andrew W. Moore. Learning evaluation functions for large acyclic domains. In L. Saitta, editor, *Machine Learning*. Morgan Kaufmann, 1996.

[Kaelbling *et al.*, 1996] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[Littman, 1994] Michael L. Littman. Markov games as a framework for multi-agent reinforcement learning. *Machine Learning*, 11:157–163, 1994.

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

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

[Owen, 1995] Guillermo Owen. *Game Theory*. Academic Press, San Diego, California, 3 edition, 1995. ISBN: 0-12-531151-6.

[Press *et al.*, 1992] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipies in C: the art of scientific computing*. Cambridge University Press, 2nd edition, 1992.

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

[Samuel, 1959] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Research Journal*, 3(3), 1959. Reprinted in ’Readings in Machine Learning’ by Shavlik and Dietterich.

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

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

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