CS 242 Final Project: Reinforcement Learning

Albert Robinson
May 7, 2002

Introduction

Reinforcement learning is an area of machine learning in which an agent learns by interacting with its environment. In particular, reward signals are provided to the agent so it can understand and update its performance accordingly.

For this project I explored different reinforcement learning techniques and tested the effectiveness of those techniques under different sets of circumstances.

An Brief (Non-Technical) Overview of Reinforcement Learning

Reinforcement learning is one of the more recent fields in artificial intelligence. Arthur Samuel (1959) was among the first to work on machine learning, with his checkers program. His work didn't make use of the reward signals that are a key component of modern reinforcement learning, but, as Sutton & Barto point out, (1998) some of the techniques he developed bear a strong resemblance to contemporary algorithms like temporal difference.

Work in the 1980s and 90s led to a resurgence in, and a more detailed formalization of, reinforcement learning research. Consequently, most of the techniques currently employed are recent. Most notably, Sutton (1988) formalized the temporal difference (TD) learning technique.

There are some general ideas that go along with most reinforcement learning problems. A reinforcement learning agent has four main elements: a *policy*, a *reward* function, a *value function*, and sometimes a *model* of the environment interacted with. Note that in this case, "environment" refers to anything outside the direct decision-making entity (i.e., anything that is sensed by the agent is part of the environment, even if it is inside the physical agent in a real-world implementation like a robot). The agent considers every unique configuration of the environment a distinct *state*.

A policy is a function that tells the agent how to behave at any particular point in time. It is essentially a function that takes in information sensed from the environment, and outputs an action to perform.

A reward function is a function that assigns a value to each state the agent can be in. Reinforcement learning agents are fundamentally reward-driven, so the reward function is very important. The ultimate goal of any reinforcement learning agent is to maximize its accumulated reward over time, generally by attempting to reach, by a particular sequence of actions, the states in the environment that offer the highest reward.

A value function is an estimation of the total amount of reward that the agent expects to accumulate in the future. Whereas a reward function is generally static and linked to a specific environment, a value function is normally updated over time as the agent explores the environment. This updating process is a key part of most reinforcement learning algorithms. Value functions can be mapped from either states or *state-action pairs*. A state-action pair is a pairing of each distinct state and each action that can be taken from that state.

A model of the environment is an internal, and normally simplified, representation of the environment that is used by agents to try and predict what might happen as a result of future actions. A more sophisticated agent might use a model to do planning of its future course of action (as opposed to doing simple reaction-based, trial-and-error exploration).

Reinforcement Learning Techniques

There are many different techniques for solving problems using reinforcement learning. Sutton & Barto (1998) identify three basic ones: Dynamic Programming, Monte Carlo, and Temporal-Difference. Following are brief descriptions of each.

Dynamic Programming (DP)

Dynamic programming is a technique for computing optimal policies. An optimal policy can, by definition, be used to maximize reward, so DP can be very useful under the right circumstances. The drawbacks of DP are that it requires a perfect model of the environment and it can require a considerable amount of computation.

One of the most basic DP methods is to compute the exact value function for a given policy, and then use that value function to produce a new policy. If we assume that we have a good value function that assigns a value to each state, a sensible policy is often to simply move to the adjacent state with the highest value. ("Adjacent state" in this sense is defined as any state that can be reached with a single action.) In this way, the policy can be updated using the improved value function. This technique is called *policy improvement*. The problem with policy improvement is that computing the exact value function can take a considerably long time, since it requires iterating many times over every possible state. Even simple problems can have environments with a number of states so large that such iteration is realistically impossible.

Other methods, such as iterating over the policies themselves, or localizing iteration over relevant parts of the environment, exist, but in many cases similar limitations remain. One of the most severe such limitations is the requirement of a perfect model of the environment.

Monte Carlo (MC)

Monte Carlo techniques, on the other hand, require no knowledge of the environment at all. They are instead based on accumulated experience with the problem. As the name might suggest, MC is often used to solve problems, such as gambling games, that have large numbers of random elements.

Like DP, MC centers on learning the value function so that the policy can be improved. The simplest way of doing this is to average each reward that a given state (or state-action pair, if that is the type of value function being used) results in. For example, in a game of poker, there are a finite number of states (based on the perceptions of the player) that exist. An MC technique would be to keep track of the rewards received after each state, and then make the value of each state equal to the average of all the rewards (money won or lost) encountered following that state (in that particular game). Assuming an "agent has Royal Flush" state had been encountered at all, the value for that state would probably be very high. On the other

hand, an “agent has a worthless hand” state would probably have a very low value. Obviously, accumulating useful value data for states requires many repeat plays of the game. In general, since MC learns with experience, many repetitions of problems are required.

It is possible to “solve” problems using MC by exploring every possibility and then generating an optimal policy. However, this can take a long time, (the number of variables in a human poker game make the number of states huge) and for many problems (like blackjack), the randomness is so great that, as Sutton & Barto (1998) note, a “solution” doesn’t result in winning even half the time.

Temporal Difference (TD)

One of the problems common to both dynamic programming and Monte Carlo is that the two techniques often don’t produce information that’s useful until a huge number of possible states have been encountered multiple times. It is possible to get over this problem with some DP methods by localizing updates, but in that case, the problem remains that DP requires a perfect model of the environment.

One solution to these problems lies in the method of temporal difference (TD), which combines many of the elements of DP and MC. TD was formalized largely by Sutton (1988), though earlier influential work was done by Samuel (1959), Holland (1986), and others.

Like MC, TD uses experience to update an estimate of the value function over time. Like MC, after a visit to a state or state-action pair, TD will update the value function based on what happened. However, MC only updates after the run-through of the problem, or *episode*, has been completed. It is at that point that MC goes back and updates the value averages for all the states visited, based on the reward received at the end of the episode.

TD, on the other hand, updates after every single step taken. The general methodology for basic TD, sometimes called TD(0), is to choose an action based on the policy, and then update the value of the current state based on the sum of the reward given by the following state and the difference in values between the current and following state. This sum is often multiplied times a constant

called a *step-size parameter*. This technique of updating to new estimates based partly on current estimates is called *bootstrapping*.

TD works well because it allows the agent to explore the environment and modify its value function while it's working on the current problem. This means that it can be a much better choice than MC is for problems that have a large number of steps in a given episode, since MC only updates after the episode is completed. Also, if the policy depends partly on the value function, the behavior of the agent should become more effective at maximizing reward as updating continues. This is called using TD for *control* (as opposed to simply predicting what future value), and there are a number of well-known algorithms, such as *Sarsa* and *Q-Learning*, that do it.

TD is often used with state-action pair values rather than simply state values. Since the value of a given state-action pair is an estimation of the value of the next state, TD is considered to *predict* the next value.

TD(0) predicts ahead one step. There is a more generalized form of TD prediction called *n-Step TD Prediction*, characterized by TD(λ). This uses a mechanism called an *eligibility trace* that keeps track of which states (or state-action pairs) leading up to the current state are responsible for the current state, and then updates the values of those states to reflect the extent to which they made a difference. As Sutton & Barton (1998) point out, the generalized MC method can be considered a form of TD(λ) that tracks the entire sequence of actions and then updates all the visited states (without using a discount factor) once a reward has been reached. MC, in other words, can be considered a form of TD that is on the opposite end of the spectrum from TD(0). TD(0) only predicts one state, but MC "predicts" every state (though in this sense "prediction" refers to learning about past events).

Using TD(λ) in this way results in some of the same problems that MC has, in that some information isn't learned about a state until well after that state has been encountered. However, if multiple episodes are expected in the same environment, the information learned during one episode will become useful in the next episode. Also, if the same state is visited twice, the information will be immediately useful.

More Advanced Techniques

Much more sophisticated and complicated techniques have been developed that make use of combinations or altered versions of the above methods. In general, computation time is a major issue in reinforcement learning because many problems require real-time updating, and techniques that rely on information that won't be updated until many steps in the future can have difficulty doing some things. This is why models of the environment are sometimes implemented in agents, because they allow agents to use planning techniques on the model that coincide with experience in the actual environment. Assuming the model is accurate, the agents will be able to make optimal decisions much more quickly.

A particularly famous use of reinforcement learning techniques, aside from the aforementioned groundbreaking work by Samuel (1959), has been the program TD-Gammon (Tesauro, 1992, 1994, 1995). This is a Backgammon-playing program that combines TD techniques with a neural network that aids in prediction of future values. In one of its more recent incarnations, TD-Gammon after only two weeks of training (by playing against itself) was rated on a level almost equal to that of the best human Backgammon players in the world.

Clearly the development of methods that combine and enhance the basics of reinforcement learning can result in great achievements. In order to better understand the fundamental techniques of reinforcement learning, I implemented a number of different problems and algorithms so I could analyze how they worked.

Problems I Worked With

Following are descriptions and results of different problems (both environments and reinforcement learning techniques) that I explored. They are listed roughly in the same order that their respective techniques are listed above. For more complete information about using the program files, see the readme file (robinson5.txt). Many of the problems I worked on are based on those mentioned by

Sutton & Barto (1998) in their book *Reinforcement Learning*. This is noted where applicable.

Dynamic Programming

Simple Iterative Policy Evaluation (DPGrid.java)

This problem, taken from Sutton & Barto (1998) p. 92, used a simple 4x4 grid as an environment, with terminal states set to the upper left and lower right corners, and every other state having reward –1. (See Figure 1.)

0 (Term.)	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	0 (Term.)

Figure 1. DPGrid Reward Layout

The goal of the problem is to find values that lead to an optimal policy, by iterating with a simple iterative policy evaluation algorithm until the values are stabilized. The final grid produced by this solution is in Figure 2.

0	-11	-15.5	-16.5
-11	-14.5	-16	-15.5
-15.5	-16	-14.5	-11
-16.5	-15.5	-11	0

Figure 2. DPGrid Value Results

The values are diagonally symmetric, as would be expected given that the terminal states are also diagonally symmetric. The policy resulting from values like these simply calls for moving towards the adjacent state with the highest value, so it is clear that from any point on this board the values lead to a policy which finds a terminal state as quickly as possible.

mazeWorld evaluation (mazeSolver.java)

This program uses an algorithm similar to the one above to develop state values for a world containing a 10x10 maze. The maze's layout is shown in Figure 3:

		X							End
	X								
X									
		X	X	X					
					X				
			X			X			
			X				X		
	X	X						X	
X									X
Start									

Figure 3. The mazeWorld Layout (X = Wall)

The results for this problem were messier than the ones for DPGrid, since the layout is more complicated. Notably, following a policy of going to the highest adjacent value will not let this maze be completed, because there are areas where the values are highest in a corner, which would obviously cause such a policy to get stuck in the corner. A better solution would have to check for loops to make sure that didn't happen. This problem with the algorithm is interesting, and it shows how the algorithm can be less effective under certain circumstances.

Monte Carlo

N-Armed Bandit (NABOptimalAverage.java & NABScoreAverage.java)

The N-Armed Bandit problem is introduced in the beginning of Sutton & Barto (1998), on p. 26. It is a simplified Monte Carlo problem that demonstrates how the technique can improve the playing of semi-random games over time.

The environment of the problem is a device that allows *n* actions (or levers – the problem is based on the principles of the 1-Armed Bandit slot machine). Each action has an average reward, but on any given use of the action it returns a reward randomized over a normal distribution with the average as a mean. The goal of the agent is to maximize total reward over repeat playing by learning which actions have the highest average reward.

The implementation of the solution is simple. The agent simply keeps track of the average value for each action so far, and its policy is to pick the action with the highest average value with frequency 1 - ε. This is called a *ε-greedy* method. With ε probability it chooses a completely random action.

I attempted to duplicate the results Sutton & Barto found, so I duplicated their experiment exactly. They measured average reward over number of plays for different ε values, and percentage of optimal action chosen over number of plays for different ε values. The results are graphed in Figures 4 and 5, respectively.

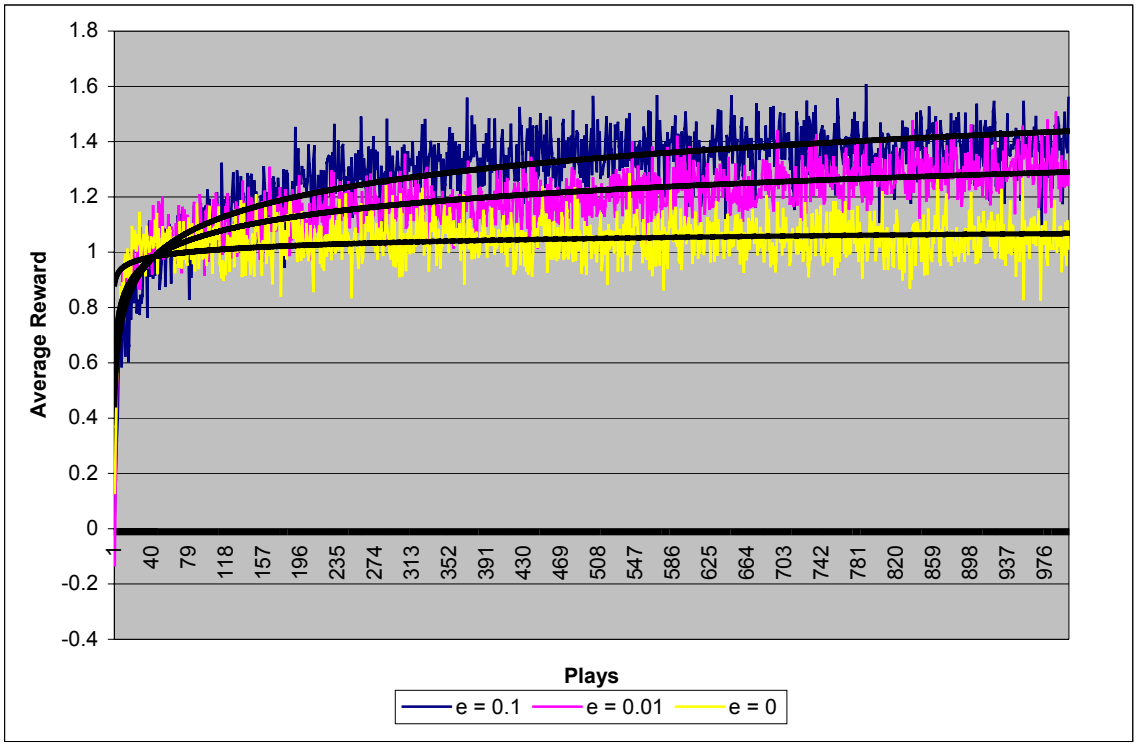

Figure 4. Average Reward over plays for N-Armed Bandit with a simple MC algorithm.

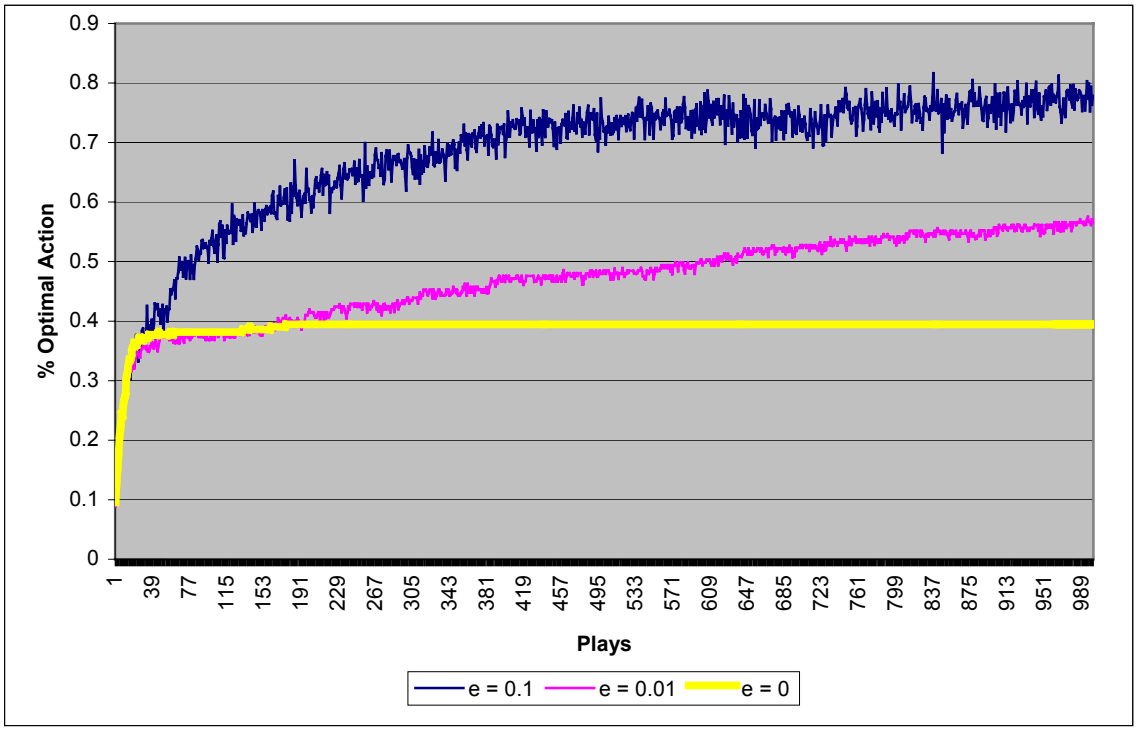

Figure 5. Percentage of time optimal action was chosen.

The data are averaged over 2000 runs. Each run had 1000 episodes. The graphs are almost identical to the data Sutton & Barto reported, so I am fairly confident that my implementation of the algorithm was accurate.

The data are notable for a few reasons. Figures 4 and 5 both demonstrate how important proper choice of ε can be in a ε-Greedy algorithm. With ε set to 0, no random exploration occurred, so the optimality leveled out quite dramatically. The average reward was 1 in this case, which is to be expected since the random rewards were chosen with 1 as a median.

Also it is interesting that the choice of $\varepsilon = 0.1$ made average reward climb higher sooner than $\varepsilon = 0.01$ did, but the slope of $\varepsilon = 0.01$ is greater. This means that the lower greed value will eventually overtake and pass the higher one. This also makes sense, because in the long run the highest averaged value is more likely to be an accurate representation of the best action to choose, so a higher likelihood of greedy action is good.

Temporal Difference

mazeWorld (mazeTester.java)

I designed a modification of the GridWorld environment that features walls inside it so that exploration becomes akin to exploring a maze. The layout I used for this problem is in Figure 3.

Along with the DP algorithm I ran on this environment, I implemented two TD algorithms, *Sarsa* and *Q-Learning*. I had these algorithms explore the maze and tracked their performance over time, in terms of how many steps it took each one to find the goal from the start. Their relative performance over 300 episodes is in Figure 6.

Sarsa differs from Q-Learning in that Sarsa is *On-Policy*, whereas Q-Learning is *Off-Policy*. That means that Sarsa's value update depends on what future state-action pair is chosen by the policy. With Q-Learning, the highest value available is chosen for updating, regardless of what choice the policy makes.

Of course, the policy still guides Q-Learning value updating in that it decides where to go, but in terms of looking ahead for prediction, policy has no effect in Q-Learning. Because it does not rely on the policy, Q-Learning is an inherently simply algorithm.

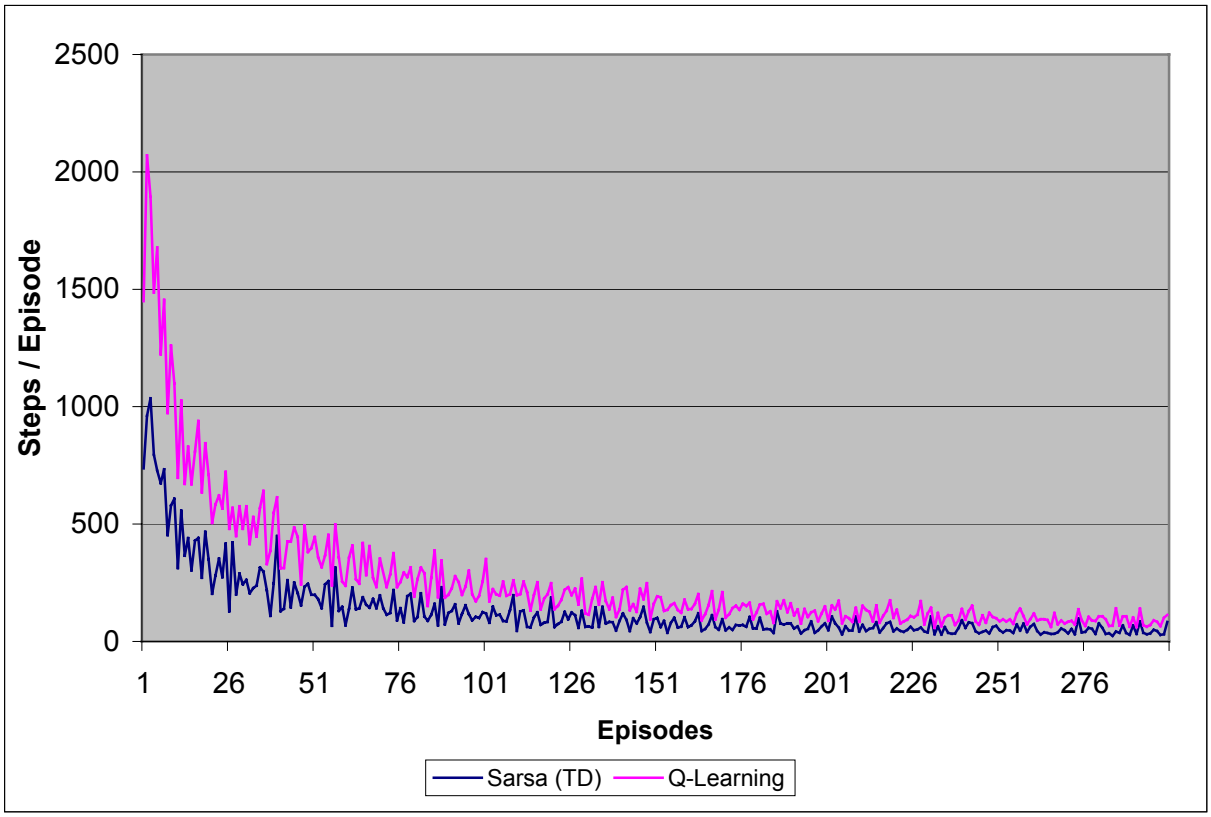

Figure 6. Steps per episode taken as the number of episodes increases for mazeWorld exploration by two TD algorithms Sarsa and Q-Learning.

Both algorithms clearly level out at a minimum, which is effectively the minimum number of steps needed to get to the goal. However, Sarsa stays consistently above Q-Learning the whole time, though not by much. This makes sense, because Sarsa is acting on more information that Q-Learning is. If the two algorithms were acting less greedy ($\varepsilon = 0.1$ here), Q-Learning might perform better because it would not suffer from "learning" about potentially bad random actions that the policy makes. As it is, Q-Learning is slightly less effective.

Windy Grid World (WGWtester.java)

Windy Grid World is an environment from Sutton & Barto (1998), that is explained on p. 147. It is a GridWorld that is modified to contain a vertical "wind" vector. This vector modifies all movement so that, for a given column, any move made will also push the agent up by an amount specified by the vector. With the Goal state right in the center of a collection of level 1 and 2 wind values, the task of reaching the goal becomes significantly more difficult than it would otherwise be. The agent starts on the left side of the grid, so it must cross across the top of the grid, over the Goal, and then come down the far right side of the grid (which has wind level 0) far enough that its trip to the left causes the wind to drive it to the goal.

This is the type of problem that TD can be good at solving, because the algorithms don't "know" about the wind. Instead, the wind is simply perceived as part of the environment and factored in to the values, so after some initial learning they solve the problem fast. Figure 7 shows a comparison between the performance of Sarsa and Q-Learning on this world.

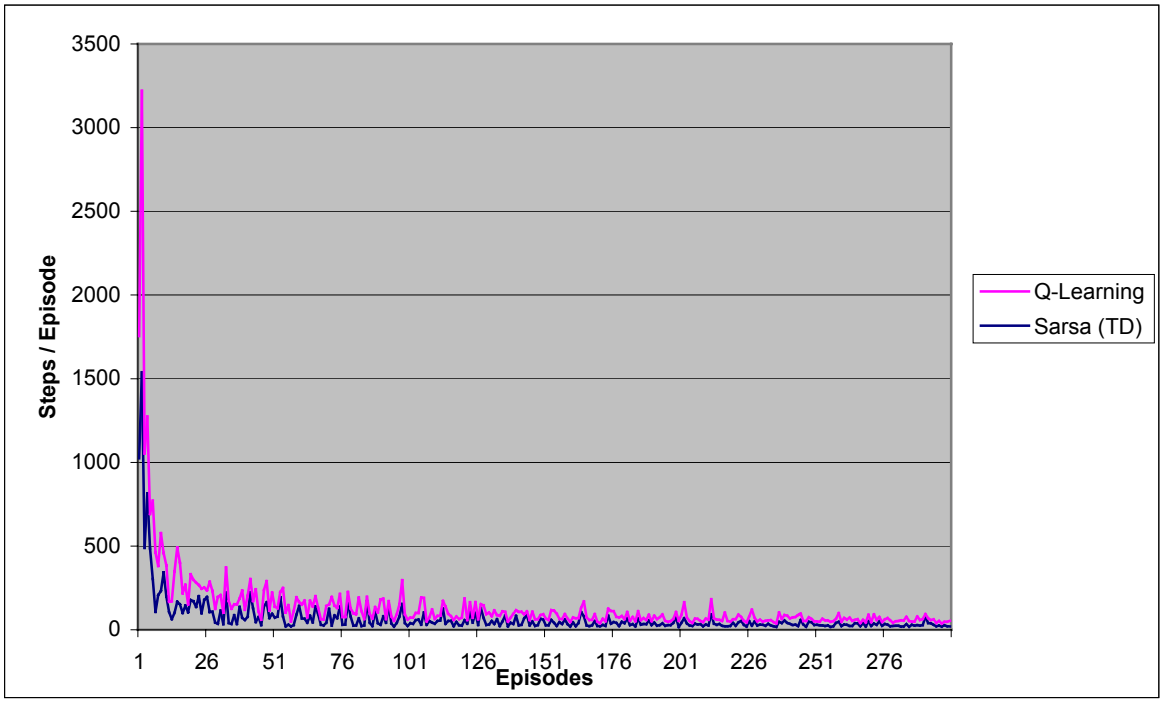

Figure 7. Sarsa vs. Q-Learning on WindyGridWorld.

The performance here was very similar to that on the mazeWorld, and the discussion there applies here. It is interesting that even though the problems appear different from an external viewpoint, (mazeWorld is about dealing with walls, WindyGridWorld is about dealing with shifts in movement) the fact that those differences are part of the environment and not the agent itself means that to the agent, the problems are actually the same.

As can be seen from comparing Figure 7 to Figure 6, initially the WindyGridWorld problem required a lot more exploration than mazeWorld, but once that exploration was done both algorithms almost immediately plunged in step time so that their episodic performance was close to optimal every time.

Future Ideas

Given time, an interesting project would be one that combined TD learning techniques in a large-scale, semi-random environment, with an evolutionary algorithm.

One such project would be a life simulation with predator and prey agents that moved around in a semi-dynamic world over a series of time steps. While moving through the world the agents would collect information using TD techniques about the values of certain areas, as pertaining to certain impulses (hunger, sleeping, etc.). Then, when some random variable triggered that impulse, the agents could make use of the built-up value functions for that particular impulse to find a semi-optimal path to what they needed. Without a representation like this, an agent would need either a full representation of the world (cheating), or it would wander around semi-blindly. Both predator and prey would reproduce at certain times, and a genetic algorithm would use some fitness function to determine which agents reproduced. Part of the reproduction would include a combination of the value functions that had been built up by the agents, as well as, perhaps, some other learned policy.

There a few difficulties with this project. The first is coming up with a world complex enough to make evolution of value functions worthwhile. Such a world would greatly

increase computational requirements. The larger problem is balancing such a world with the proper use of variables.

Nevertheless, if a stable version of this world could be developed, it would be very interesting to experiment with, and it would do a great job of showing off the power of combining reinforcement learning techniques to solve larger problems.

References

Holland, J. H. (1986). Escaping brittleness: The possibilities of general-purpose learning algorithms applied to rule-based systems. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (eds.), *Machine Learning: An Artificial Intelligence Approach*, vol. 2, pp. 593-623. Morgan Kaufmann, San Mateo, CA.

Russell, S., and Norvig, P. (1995). *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal on Research and Development*, 3:211-229.

Sutton, R.S. (1988). Learning to predict by the method of temporal differences. *Machine Learning*, 3:9-44.

Sutton, R.S., and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, Massachusetts.

Tesauro, G. J. (1992). Practical issues in temporal difference learning. *Machine Learning*, 8:257-277.

Tesauro, G. J. (1994). TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6:215-219.

Tesauro, G. J. (1995). Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38:58-68.