Performance Comparison of Sarsa(λ) and Watkin's Q(λ) Algorithms

Karan M. Gupta
Department of Computer Science
Texas Tech University
Lubbock, TX 79409-3104
gupta@cs.ttu.edu

Abstract

This paper presents a performance comparison between two popular Reinforcement Learning algorithms: the Sarsa-Lambda algorithm and Watkin's Q-Lambda algorithm. The algorithms were implemented on two classic reinforcement learning problems – the Pole Balancing problem and the Mountain Car problem. Further work and improvements have been suggested in the conclusion.

1 Introduction

It is necessary to know which algorithm to use for a specific reinforcement learning problem. This study aims to indicate which one is a better algorithm, and under which parameters, for two types of problems. The mountain car problem is an example of an episodic control task where things have to get worse (moving away from the goal) before they can get better. The pole balancing problem is an example of an episodic control task where the agent tries to reach the goal at all times.

This study is also a part of my course curriculum at the Department of Computer Science, Texas Tech University.

2 Theoretical Background

2.1 Reinforcement Learning

Reinforcement Learning is that branch of Artificial Intelligence where the agent learns with experience, and with experience only. No plans are given, there are no explicitly defined constraints or facts. It is a "computational approach to learning from interaction". The key feature of reinforcement learning is that an active decision-making agent works towards achieving some reward, which will be available only upon reaching the goal. The agent is not told which actions it should take to reach the goal, but instead it discovers the best actions to take at different states by learning from its mistakes. The agent monitors its environment all times, because actions taken by the agent may change the environment and hence affect the actions available to the agent from the environment. The agent learns by assigning values to states and actions associated with every state. When the agent reaches a state that it has already learnt about, it can exploit its knowledge of the state space to take the best action. At times the agent takes random actions – this is called exploration. While exploring the agent learns about those regions of the state space that it would otherwise ignore if it only followed the best actions. By keeping a good balance between eploitation and exploration, the agent is able to learn the optimal policy to reach the goal. In all reinforcement learning problems, the agent uses its experience to improve its performance over time. For a detailed study of Reinforcement Learning please refer to [1].

The Sarsa algorithm (short for state, action, reward, state, action) is an on-policy temporal difference control algorithm. It is an on-policy algorithm in the sense that it learns the action-value function for the policy which is being followed by the algorithm to make the transition from a state-action pair to the next state-action pair. It learns an action-value function for a policy π. This algorithm considers transitions from state-action pair to state-action pair and learns the value of visited state-action pairs. The algorithm coninuously estimates Q^{π} for a behaviour policy π, and at the same time changes π toward greediness with respect to the learned action-value function Q^{π} .

The Q-learning algorithm is an off-policy temporal difference control algorithm. This algorithm learns the optimal policy while following any other (non-greedy, e.g. ε-greedy, etc.) policy. The learned action-value function Q directly approximates Q*. The policy being followed changes as the action-value function is learnt and more accurate values for every state-action pair are generated.

When the Sarsa and Q-learning algorithm are augmented with Eligibility Traces and TD(λ) methods, then they are known as Sarsa(λ) and Q(λ) algorithms respectively. The Q-learning algorithm was developed by Watkins in 1989, hence the name, Watkin's Q(λ). The λ in both the algorithms refers to the n-step backups for Q^{π}. These are still temporal difference learning methods because they change an earlier estimate based on how it differs from a later estimate. The TD(λ) methods differ from the basic TD method because the backup is made after n steps and not after every one step. The value of n is given by the value of λ. The practical way of implementing this kind of method is by using Eligibility Traces.

An eligibility trace is associated with every state-action pair in the form of an additional memory variable. The eligibility trace for every state-action pair decays at every step by $\gamma\lambda$, where γ is the discount rate. The eligibility trace for the state visited on that step is changed depending on the kind of eligibility traces being implemented. Ther are two ways to implement eligibility traces, Accumulating Traces and Replacing Traces. If using Accumulating Traces, the eligibility trace for the visited state-action pair is incremented by 1. If Replacing Traces are used, the eligibility Trace for the visited state is set to 1. In all their essence, eligibility traces keep a record of state-action pairs which have recently been visited, and the degree for which each state-action pair is eligible for undergoing learning changes.

$$e_t(s,a) = \begin{cases} 1 & ReplacingTraces \\ \gamma\,\delta e_{t-1}(s,a)+1 & AccumulatingTraces \\ \gamma\,\delta e_{t-1}(s,a) & Decay \end{cases}$$

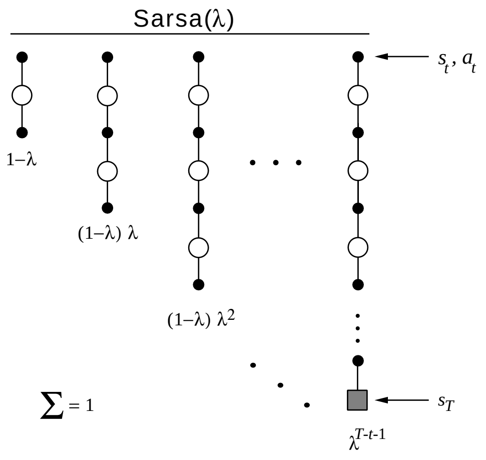

Figure 1: Sarsa(λ) Backup Diagram

2.2 Sarsa(λ)

When eligibility traces are added to the Sarsa algorithm it becomes the Sarsa(λ) algorithm. The basic algorithm is similar to the Sarsa algorithm, except that backups are carried out over n-steps and not just over one step. An eligibility trace is kept for every state-action pair. The update rule for this algorithm is:

$$Q_{t+1}(s,a) = Q_t(s,a) \; + \; \alpha\,\delta_t\; e_t(s,a) \quad \text{for all s, a}$$

where

$$\delta_t \; = \; r_{t+1} \; + \; \gamma\, Q_t(s_{t+1},a_{t+1}) \; - \; Q_t(s_t,a_t)$$

and

$$e_t(s,a) = \begin{cases} 1 & if\ s = s_t\ and\ a = a_t \\ \gamma\delta e_{t-1}(s,a) & otherwise \end{cases}$$

The backup diagram for the Sarsa(λ) algorithm is shown in Figure 1.

The Sarsa(λ) algorithm is shown in Figure 2. This algorithm uses replacing traces. The algorithm runs for a large number of episodes until the action-value function converges to a reasonably accurate value. At every step it chooses an action a_{t+1} for the state s_{t+1} that is the next state. This decision is made based on the policy that is being followed by the algorithm. Then it calculates δ_t based on the above equation. Depending on the kind of traces being used, the value of the eligibility trace for the current state-action pair $e(s_t,a_t)$ is calculated. In the

```
Initialize Q(s,a) arbitrarily and e(s,a)=0, for all s,a
Repeat(for each episode)
    Initialize s,a
    Repeat(for each step of epsiode)
        Take action a, observe r, s'
        Choose a' from s' using the behaviour policy
        $\delta \leftarrow r + \gamma$Q(s',a')-Q(s,a)
        e(s,a) $\leftarrow$ 1
        For all s,a:
            Q(s,a) $\leftarrow$ Q(s,a) +$\alpha\delta$e(s,a)
            e(s,a) $\leftarrow$ $\gamma\lambda$e(s,a)
        s $\leftarrow$ s'; a $\leftarrow$ a'
    until s is terminal
```

Figure 2: Tabular Sarsa(λ)

next step the action-value for every state-action pair is calculated using the above equation. Also, the eligibility trace for every state-action pair is decayed by $\gamma\lambda$. This process continues iteratively for every episode until a terminal state is encountered.

2.3 Watkin's Q(λ)

The Q(λ) algorithm is similar to the Q-learning algorithm except that it uses eligibility traces and learning for an episode stops at the first non-greedy action taken. Watkin's Q(λ) is an off-policy method. As long as the policy being followed selects greedy actions the algorithm keeps learning the action-value function for the greedy policy. But when an exploratory action is selected by the behaviour policy, the eligibility traces for all state-action pairs are set to zero, hence learning stops for that episode. If a_{t+n} is the first exploratory action, then the longest backup is toward

$$r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n \, max_a Q_t(s_{t+n}, a)$$

The backup diagram for this algorithm is shown in Figure 3. The eligibility traces are updated in two steps. If an exploratory action was taken, they are set to zero for all state-action pairs. Otherwise the eligibility traces for all state-action pairs are decayed by $\gamma\lambda$. In the second step, the eligibility trace value for the current state-action pair is either incremented

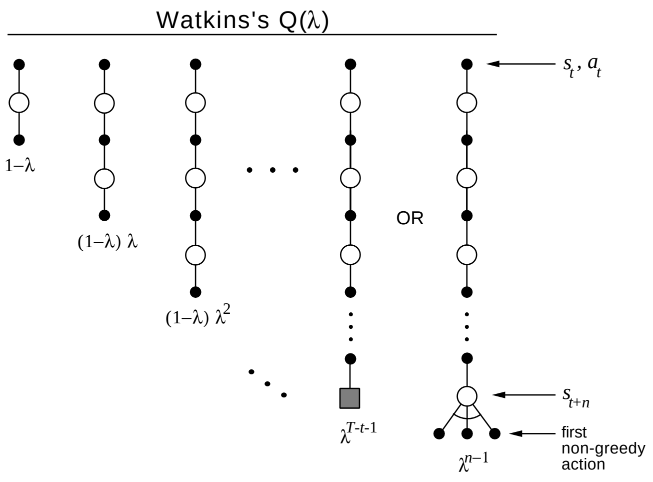

Figure 3: Watkin's Q(λ) Backup Diagram

by 1 in the case of accumulating traces or set to 1 in the case of replacing traces.

$$e_t(s,a) = I_{ss_t}.I_{aa_t} + \begin{cases} 1 & if Q_{t-1}(s_t,a_t) = max_a Q_{t-1}(s_t,a); \\ 0 & otherwise \end{cases}$$

The action-value function is defined as

$$Q_{t+1}(s,a) = Q_t(s,a) \; + \; \alpha \, \delta_t \; e_t(s,a),$$

where

$$\delta_t = r_{t+1} \; + \; \gamma \, max_{a'} \; Q_t(s_{t+1}, a') \; - \; Q_t(s_t, a_t)$$

The complete algorithm is shown in Figure 4.

The algorithm is conceptually the same as Sarsa(λ), but it updates the action-value function using the value of the greedy action at the current state.

3 Implementation

The algorithms were implemented on the Pole Balancing and Mountain Car problems. The code is written in C++. The code was compiled using version 3.2 of the g++ compiler. A makefile is included with the code. The code has been tried on RedHat Linux 8 and on Sun Machines running Solaris 9. It works comfortably on both, but takes longer to converge on the Sun machines.

```
Initialize Q(s,a) arbitrarily and e(s,a)=0, for all s,a
Repeat(for each episode)
   Initialize s,a
   Repeat(for each step of epsiode)
      Take action a, observe r, s'
      Choose a' from s' using policy Q($\epsilon$-greedy)
      $a^* \leftarrow \arg max_b$ Q(s',b)
      (if a' ties for max, then $a^* \leftarrow$ a')
      $\delta \leftarrow \gamma$Q(s',$a^*$)-Q(s,a)
      e(s,a) $\leftarrow$ 1
      For all s,a:
          Q(s,a) $\leftarrow$ Q(s,a) +$\alpha\delta$e(s,a)
          if a'=$a^*$, then e(s,a) $\leftarrow \gamma\lambda$e(s,a)
                   else e(s,a) $\leftarrow$ 0
      s $\leftarrow$ s'; a $\leftarrow$ a'
   until s is terminal
```

Figure 4: Tabular Watkin's Q(λ) Algorithm

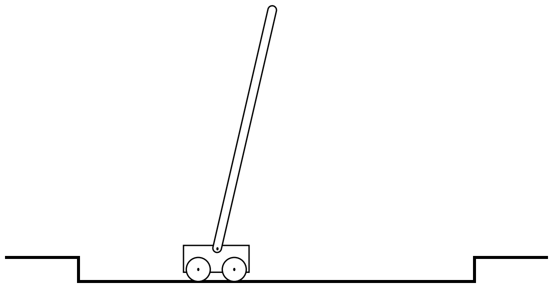

Figure 5: Pole-Balancing Problem

3.1 Pole Balancing Problem

The pole balancing problem is a classic problem of reinforcement learning. The basic idea is to balance a pole which is attached to a hinge on a movable cart. Forces are applied to the cart, which can only move along a 2 dimensional track, to prevent the pole from falling over. A failure occurs if the pole falls past a given angle from vertical. If the cart reaches the end of the track, that also accounts as a failure. The pole is reset to vertical after each failure. The implementation treats the problem as an episodic task, without discounting. The episode terminates when a failure occurs. The return is maximized by keeping the pole balanced for as long as possible. A graphical view is shown in Figure 5.

The entire state space is represented as a combination of four variables, x, $\dot{x}$, θ, $\dot{\theta}$. There are two actions: apply force to the left of the cart, and apply force to the right of the cart. Eligibility traces are associated with every state-action pair. The action-value function Q is maintained as a two dimensional array which maps to every state-action pair. The Q function and the eligibility traces are thus defined in the program as such:

$Q[state][action]$ and $ET[state][action]$.

The program starts out with an ϵ value of 0.99999 for the first one hundred episodes. After that the epsilon value is decayed every hundred episodes by multiplying it with itself. The pseudo-code for this is:

```
if ($\epsilon > 0$)
   if (episode mod 100 equals 0)
      $\epsilon = \epsilon \times \epsilon$
```

The state space is divided into regions. It would be computationally impossible to generate a single state for every possible combination x, $\dot{x}$, θ, $\dot{\theta}$. To facilitate computation, the range for every variable is divided into 15 equal regions. Every value for every parameter is assigned an index based on the corresponding region it falls in. The four indexes are then combined to map to a single state. By this method there are $15 \times 15 \times 15 \times 15 = 50,625$ unique states, which is a computationally managable number. This mapping is carried out by the $getState(x,\ \dot{x},\ \theta,\ \dot{\theta})$ function. This function takes in the values of the variables as arguments. It returns a unique number

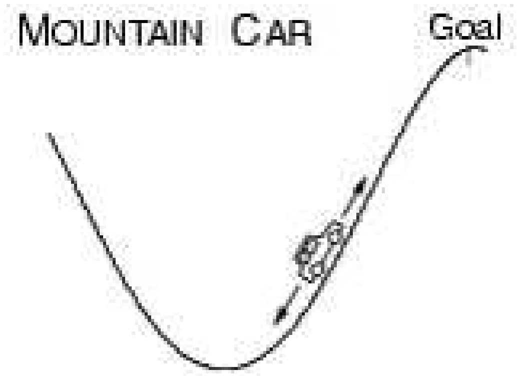

Figure 6: Mountain-Car Problem

which is the state. This function is represented mathematically as,

$$xindex = (int)\left|ceil\left[\frac{minx + |x|}{xdiv}\right]\right|$$

where,

$xrange = minx - maxx$,

$xdiv = \frac{xrange}{15}$

for the value of x. The values of $\dot{x}index$, $\theta index$ and $\dot{\theta}$ are calculated similarly. Finally the *stateindex* is calculated by a simple polynomial function:

$stateindex = (int)\ [15^3 \times xindex + 15^2 \times \theta index + 15 \times \dot{x}index + \dot{\theta}index]$

The egreedy action is decided using the function $eGreedy(Val1,\ Val2,\ \varepsilon)$. This function returns the action number to be taken. $Val1$ and $Val2$ are action-values for the state at which the decision is to be taken. To take a greedy action a value of 0.0 for ε is passed as an argument.

The code was written for both Sarsa(λ) and Q(λ) algorithms. Both the implementations use Replacing Traces to update the eligibility trace.

For both the algorithms the iteration only ends if the action-value function converges to fixed values. This was deduced by running the algorithms until the number of steps taken remains a constant for 100 consecutive episodes. This is the converging condition for the algorithms.

3.2 Mountain Car Problem

The Mountain Car problem is another classic problem in reinforcement learning. It involves driving an underpowered car up a steep mountain road. However, the car's engine is not strong enough to make the car accelerate up the steep slope. The car must back up onto an opposite slope on the left. Then the car must build up enough intertia to carry itself up to the goal. By alternatingly moving forward and backward, repeatedly, between the two slopes the car can eventually build up enough inertia to reach the top of the steep hill. The idea behind this experiment is to make the car learn how to reach its goal while minimizing the number of oscillations between the two slopes. The basic idea is shown in Figure 6.

The action-value function for the state-action pairs is again a two dimensional array represented as $Q[state][action]$. There is an eligibility trace associated with every state-action pair represented as $ET[state][action]$.

The state is a function of two parameters: the position of the car and the velocity of the car. Like the Pole-balancing problem, here also the state space is divided into equal regions for reasons of computational feasibility. However, the state space here is divided into 30 regions and not 15 regions for each parameter. This makes the state space more accurate towards the mapping to a single unique state. There are $30 \times 30 = 900$ unique states. The mapping from different values of the parameters to a single unique state is carried out by the function $getState(pos,\ vel)$. This function returns a unique value for the state depending on the values of *pos* and *vel*. This function works similarly to the $getState()$ function used in the pole balancing problem. First the *posindex* and the *velindex* are calculated using the equation:

$$posindex = (int)\left|ceil\left[\frac{POSRANGE[0] + |pos|}{posdiv}\right]\right|$$

where

$posrange = POSRANGE[1] - POSRANGE[0]$

and

$$posdiv = \frac{posrange}{30}.$$

velindex is calculated similarly. Finally *stateindex* is calculated using the equation:

$$stateindex = (int)[30 \times posindex + velindex]$$

The egreedy action is decided using the function $eGreedy(Q, state, \varepsilon)$. This function returns the action number to be taken. Q is the action-value function and *state* is the state for which the decision is to be taken. To take a greedy action a value of 0.0 for ε is passed as an argument.

The code was written for both Sarsa(λ) and Q(λ) algorithms. Both the implementations use Replacing Traces to update the eligibility trace.

For both the algorithms the iteration only ends if the action-value function converges to fixed values. This was deduced by running the algorithms until the number of steps taken remains a constant for 100 consecutive episodes. This is the converging condition for the algorithms.

4 Results

The results have been averaged over 20 episodes to get the resulting graphs. Starting from a value of 0.99999, ε was decayed every 100 episodes. Other values are: $\gamma = 1$ and $\alpha = 0.1$. Watkin's Q(λ) algorithm was observed to be much more stable for the Mountain Car Problem.

α and γ were kept constant for the experiment. The different values of λ that were used are: 0.0, 0.2, 0.5, 0.7, 0.9 and 1.0. All graphs are based on these values and are between Number of Steps Taken on the Y-axis and The Number of Episodes on the X-axis.

4.1 Pole Balancing Problem

The results for the Pole Balancing problem are skewed because the program took too long to converge. It had not yet converged for some values of λ at the time of writing of this document.

Modifications to the code could be possible solutions to this problem.

4.2 Mountain Car Problem

The results for the Mountain Car problem are very precise and informative. We must remember that the

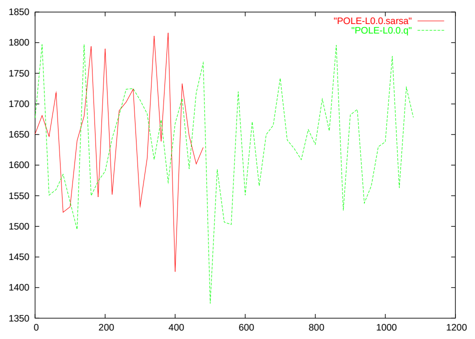

Figure 7: Pole Balancing, $\lambda = 0.0$

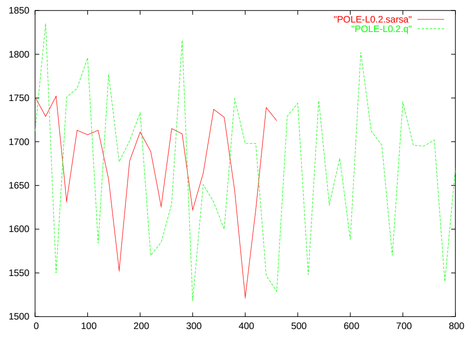

Figure 8: Pole Balancing, $\lambda = 0.2$

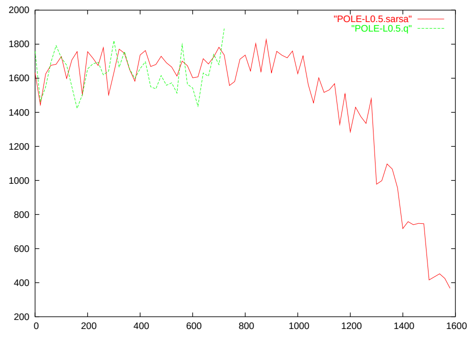

Figure 9: Pole Balancing, $\lambda = 0.5$

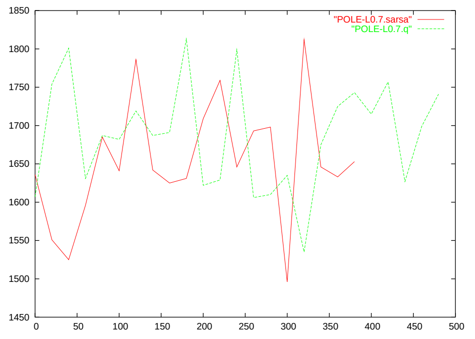

Figure 10: Pole Balancing, $\lambda = 0.7$

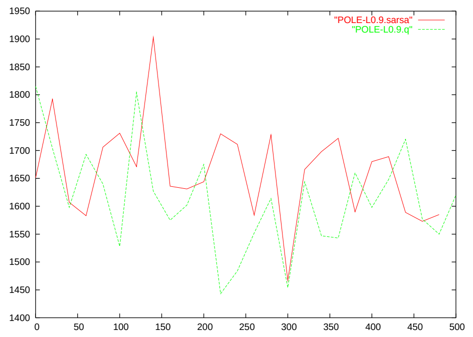

Figure 11: Pole Balancing, $\lambda = 0.9$

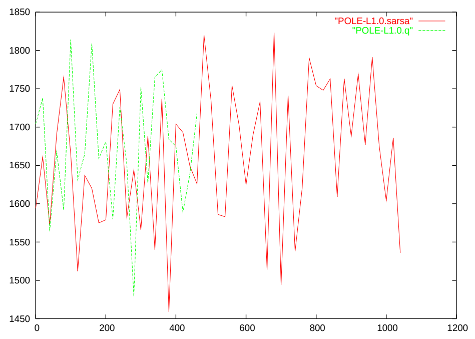

Figure 12: Pole Balancing, $\lambda = 1.0$

mountain car problem is a unique type of problem because the agent has to move away from the goal to get to it. All results have been averaged over 20 episodes. The maximum number of steps was 5000.

If we look at Figure 13 Sarsa(λ) converges earlier than Q(λ) at episode 2250, whereas Q(λ) converges at episode 2850. However, Q(λ) converges to a lower value, and hence has learnt better than Sarsa(λ). Comparing Figures 13, 14 15, we observe a trend in the way the resulta are obtained. The

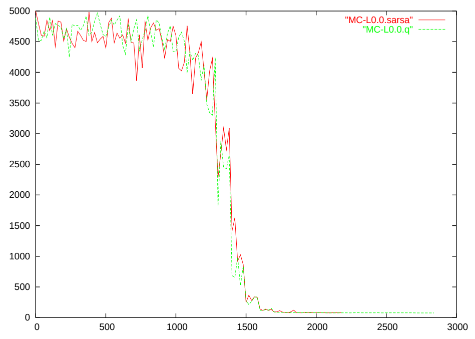

Figure 13: Mountain Car, $\lambda = 0.0$

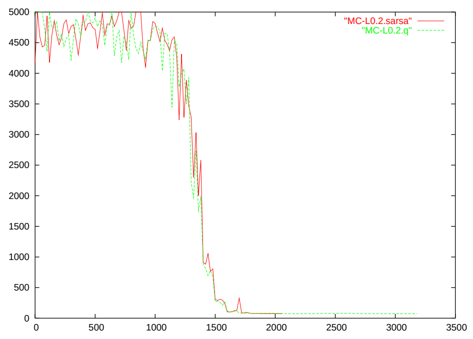

Figure 14: Mountain Car, $\lambda = 0.2$

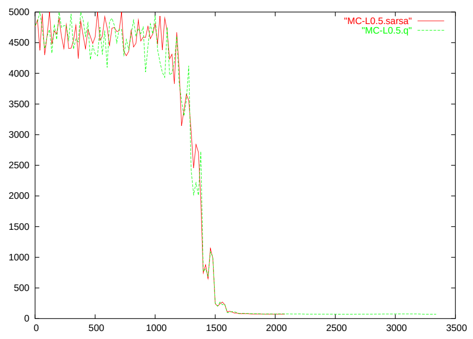

Figure 15: Mountain Car, $\lambda = 0.5$

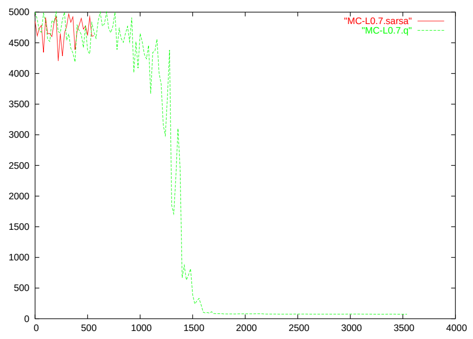

Figure 16: Mountain Car, $\lambda = 0.7$

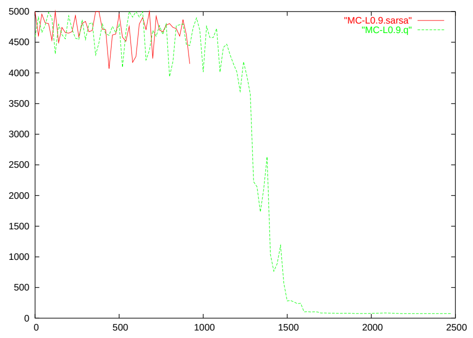

Figure 17: Mountain Car, $\lambda = 0.9$

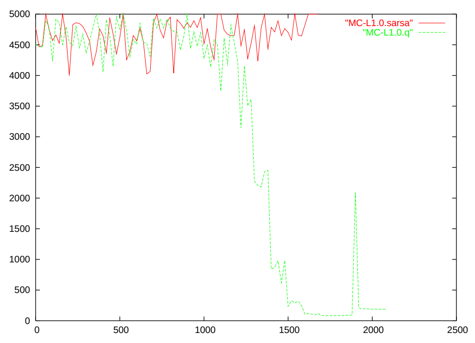

Figure 18: Mountain Car, $\lambda = 1.0$

Q(λ) algorithm always converges to a value lower than Sarsa(λ) for the same value of λ. However, Sarsa(λ) always converges earlier than Q(λ). As λ increases both the algorithms converge to a lower value than what they had with a lower λ. However, both the algorithms now take longer to converge.

Unfortunately, the programs for Sarsa(λ) for $\lambda = 0.7$ and $\lambda = 0.9$ had not converged at the time of writing of this paper. But they had been observed to have converged at an earlier time during the experiment.

Interesting results were observed for Sarsa(λ) with $\lambda = 1.0$. The relevant graph is shown in Figure 18. This is a Monte Carlo implementation. The value function converges to the worst possible value. It does not seem to be affected by any changes in ε.

5 Conclusion

Improvements to the algorithms are desired. Further experiments could include decaying the value of α during the course of a run. Furthermore, I believe that the value of α used in these experiments was too high. Also, the decay of ε could be modified to get better results.

6 Acknowledgements

I would like to thank Dr. Larry Pyeatt for providing the simulation code for both the problems. The figures for the backup diagrams and the drawings for the pole-balancing and mountain car problems have been borrowed from Dr. Sutton and Dr. Barto's website, http://www-anw.cs.umass.edu/ rich/book/the-book.html

References

[1] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, Massachusetts, England, 2002.