MP1 Part 2 Writeup

Ron Wright

September 29, 2014

1 Responses to Required Part 2 Questions

1.1 Responses to Question 1

a. With $\gamma = 0.9$, $\alpha = 0.1$, and $\varepsilon = 0.0$, I noticed that the agent does not earn any rewards, and in the steady state, the agent fails to move to any position that involves risk. This is most likely because the agent is waiting an infinitely long amount of time for the obstacles to disappear (i.e. the agent is too greedy when it comes to making sure it does not lose value on its reward).

b. With $\gamma = 0.9$, $\alpha = 0.1$, and $\varepsilon = 0.1$, I noticed that the agent gains mostly positive rewards that sum to around 130 on average. This is because the agent is more willing to take risks to earn rewards (i.e. the agent is less greedy in this case), but the actions chosen are not too random.

c. With $\gamma = 0.9$, $\alpha = 0.1$, and $\varepsilon = 0.5$, I noticed that the agent gains mostly negative rewards that sum to around -30 on average. This is because the choices of actions the agent is executing are too random. This indicates that there is a tradeoff between greediness and randomness, and the goal is to find an optimal value of ε that balances these two extremes.

1.2 Responses to Question 2

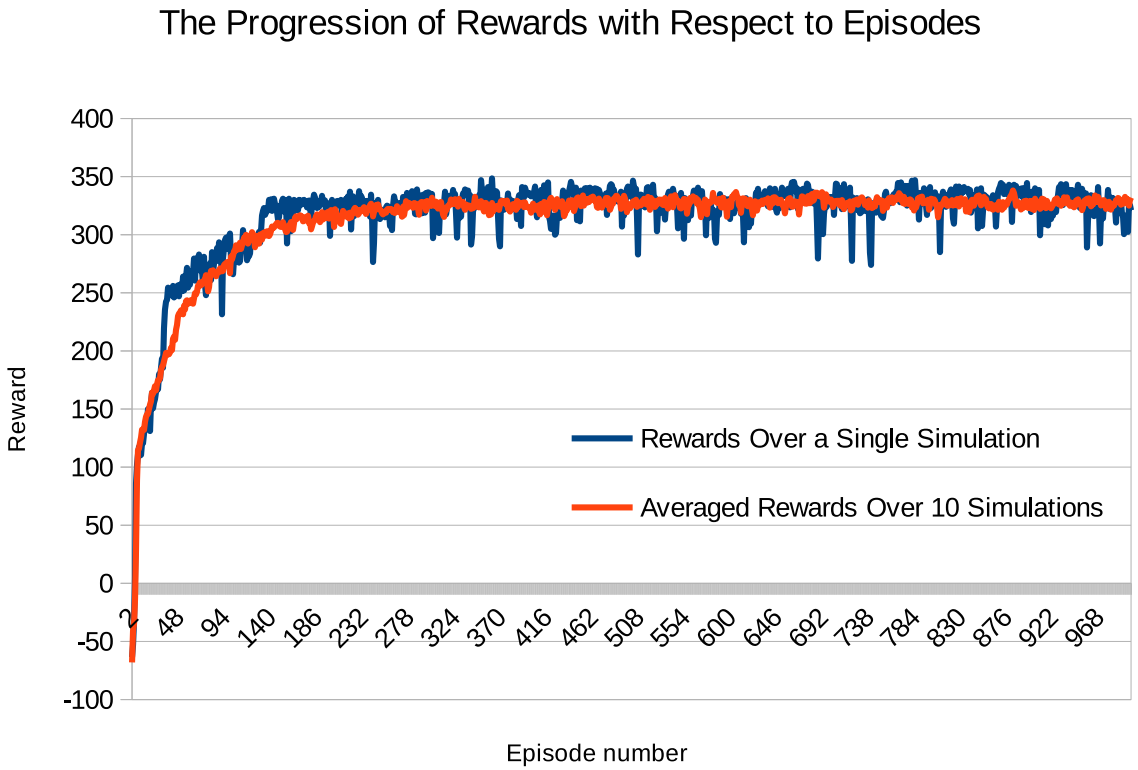
a. With $\gamma = 0.9$, $\alpha = 0.1$, $\varepsilon = 0.05$, and `knows_thief` set to `n`, I noticed that the agent gains mostly negative rewards that sum to around -13 on average. This is because the agent is more willing to take risks to earn rewards (i.e. the agent is less greedy in this case). His choices are not too random, but he incurs a severe penalty when encountering the thief, which substantially factors into the total reward. He is also limited by not knowing the state of the thief, which is the thief's row position.

b. With $\gamma = 0.9$, $\alpha = 0.1$, $\varepsilon = 0.05$, and `knows_thief` set to `y`, I noticed that the agent gains mostly positive rewards that sum to around 280 on average. This is because the agent is no longer turning a blind eye to the thief's position. That is, the state includes the row position at which the thief lies, and the agent will eventually know how to use that knowledge to avoid states in which the agent and the thief run into each other.

c. For this part, I decided to use an algorithm known as the Nelder-Mead simplex method, which is an algorithm for zeroing in on the global minimum of a cost function with respect to one or more variables (in this case, the learning rate α and the greediness control paramter ε). I chose this algorithm, since I assumed that the simulations exhibit reliable steady-state behavior, and there is no direct way to find the gradient of the reward function. I set the reflection, expansion, contraction, and shrink coefficients to the standard values of 1, 2, $-1/2$, and $1/2$, respectively. The cost function is the negation of the average total reward of the last half of the simulation computed over 10 simulation runs with 1000 episodes and 10,000 steps. I selected to have the agent know about the thief's state, since the maximum reward obtainable would be zero if the agent did not know about the thief's state. To prevent α and ε from stepping outside the $[0, 1)$ domain, I assign any out-of-bound sets of variables an infinite cost (or for floating point numbers, the largest possible value), which effectively transforms this unconstrained optimization problem into a constrained one. Nevertheless, the average total reward may include undesired transients at the start of the simulation (i.e. initialization bias), so I chose to include only the last half of the simulation (i.e. the last 500 episodes) for averaging the total reward. I ran the optimization algorithm five times, and found the optimal values of α and ε to be approximately 0.3344 and 0.002805, respectively. The reward averaged over all simulations over the last 500 episodes at those optimal values of α and ε was approximately 330.8.

1.3 Responses to Question 3

The plot of rewards over a single simulation along with the plot of averaged rewards over ten simulations are shown in a single figure below:

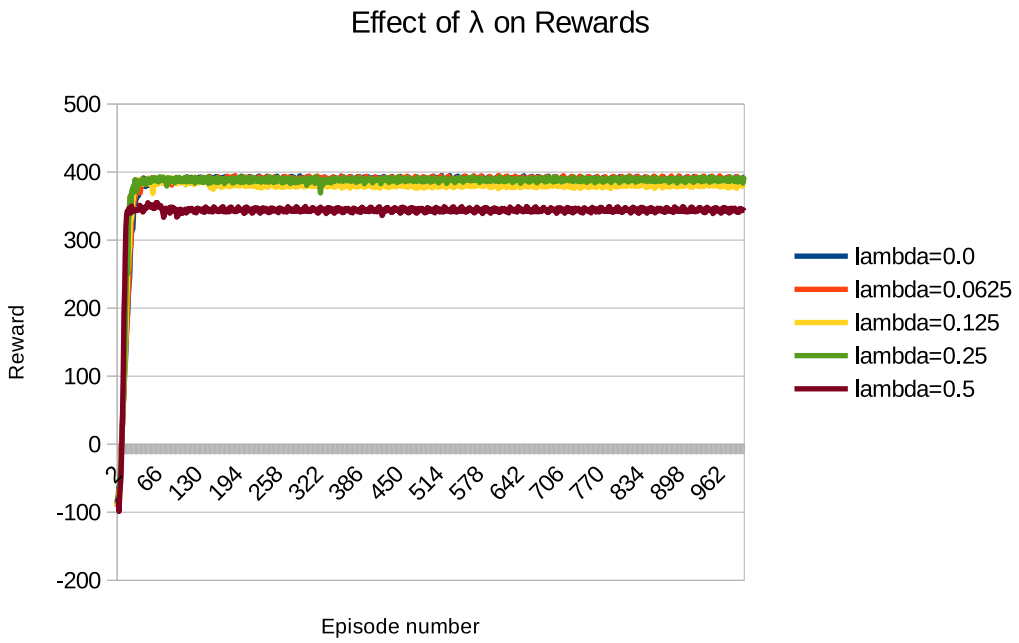


This plot supports the fact that there is initialization bias from the start of the simulation to about the 300th episode. This is because the Q values are initializing at the start of the simulation, and eventually, the Q values settle down to their steady-state values. We see this happening after the 300th episode, where the reward starts leveling off to a relatively constant steady-state value centering around approximately 330.8. It is also observed that as the simulation approaches steady-state, the reward values in a single episode inhibit increasingly more variance. This is because the Q values are partially driven by random values (e.g., used to choose random actions), and as more random values contribute to the simulation, the variance always increases (i.e. $Var(c_1X_1 + c_2X_2 + \ldots + c_nX_n) = c_1^2Var(X_1) + c_2^2Var(X_2) + \ldots + c_n^2Var(X_n)$, where $X_1, X_2, \ldots, X_n$ are random variables, and $c_1, c_2, \ldots, c_n$ are constants). Unsurprisingly, the initialization bias remains after averaging the rewards over ten simulations, but the random nature of the individual rewards at each episode in the runs cancel out. This smoothing is due to the law of large numbers, where a large enough sample mean will result in convergence to the true mean.

2 Eligibility Trace Experiments

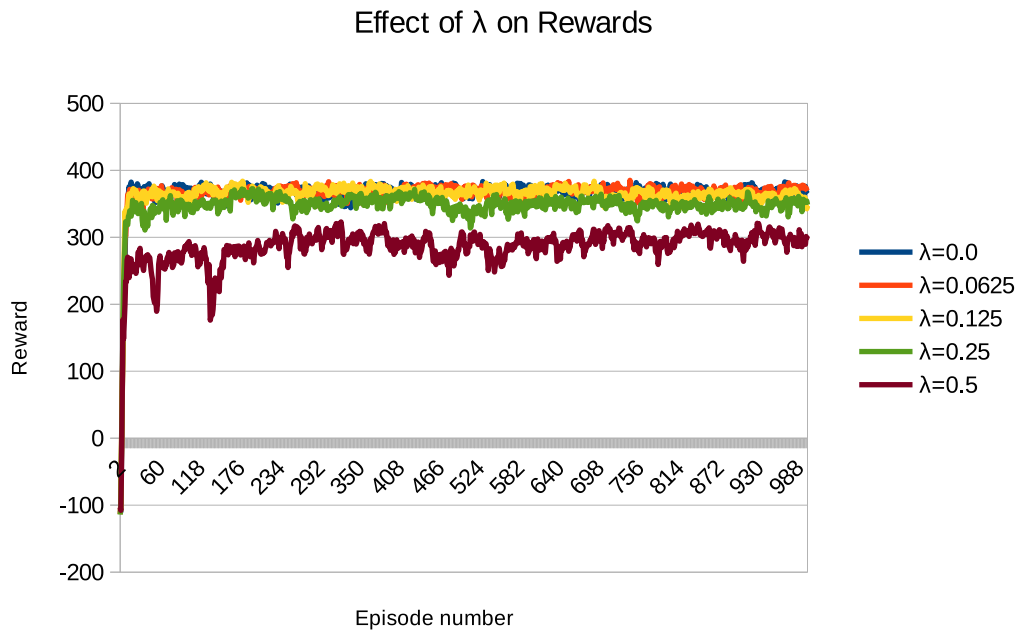
I ran out of time to compute the estimated optimum values of the learning rate, ε, and λ. However, I had to choose a different starting guess for the Q values, otherwise I always end up with bad results. This is also probably why I ended up with larger numbers for the rewards and a smaller initialization bias. Nevertheless, I was able to study the effects of λ for two example pairs of the learning rate and ε. These come from simulations of the world with the thief, and the agent knows about the thief.

The plot below shows the average rewards over four simulations with 1000 episodes and 10,000 steps using various parameters of λ. Here, $\alpha = 0.1$ and $\varepsilon = 0$:



At $\lambda = 0.125$, the average reward is slightly more affected. However, the average reward is even more affected at values as large as $\lambda = \frac{1}{2}$.

Another plot below also shows the average rewards over four simulations with 1000 episodes and 10,000 steps using various parameters of λ. This time, $\alpha = 0.3344$ and $\varepsilon = 0.002805$, the same optimal values for the plot in #3 corresponding to simulations without eligibility traces:



It is much more obvious in this plot that the average reward is affected for values as small as $\lambda = 0.125$, which means that λ has a slightly greater effect for this particular example.

We can conclude by saying that λ negatively impacts the average reward over the simulation run, and that the effect of λ is partially dependent on the randomness of the reward values.

Contents of `QLearningAgentWithEligibilityTrace.java`:

```
/* Author: Mingcheng Chen */

import java.util.ArrayList;
import java.util.Random;

public class QLearningAgentWithEligibilityTrace implements Agent {
  public QLearningAgentWithEligibilityTrace() {
    this.rand = new Random();
  }

  public void initialize(int numOfStates, int numOfActions) {
  this.discount = 0.9;
  this.rate = 0.1;
  this.epsilon = 0.0;
  this.lambda = 0.0;
    this.qValue = new double[numOfStates][numOfActions];
    this.eligibilityTrace = new double[numOfStates][numOfActions];
    this.numOfStates = numOfStates;
    this.numOfActions = numOfActions;
    for (int i = 0; i < numOfStates; ++i)
    {
```

```
22        for (int j = 0; j < numOfActions; ++j)
23        {
24          // Apparently, leaving the Q matrix initially zero leads to zero
25          // rewards for any non-zero lambda
26          this.qValue[i][j] = 1.0;
27        }
28      }
29    }
30
31    public double getDiscount()
32    {
33      return discount;
34    }
35
36    public void setDiscount(double discount)
37    {
38      this.discount = discount;
39    }
40
41    public double getRate()
42    {
43      return rate;
44    }
45
46    public void setRate(double rate)
47    {
48      this.rate = rate;
49    }
50
51    public double getEpsilon()
52    {
53      return epsilon;
54    }
55
56    public void setEpsilon(double epsilon)
57    {
58      this.epsilon = epsilon;
59    }
60
61    public double getLambda()
62    {
63      return lambda;
64    }
65
66    public void setLambda(double lambda)
67    {
68      this.lambda = lambda;
69    }
70
71    private double bestQValue(int state) {
72      // Compute max w.r.t. actions for the given state
73      int bestAction = -1;
74      for (int j = 0; j < numOfActions; ++j)
75      {
```

```
76        if (bestAction == -1 || qValue[state][j] > qValue[state][bestAction])
77        {
78          bestAction = j;
79        }
80      }
81      return qValue[state][bestAction];
82    }
83
84    private int chooseBestAction(int state) {
85      // Compute argmax w.r.t. actions for the given state
86      ArrayList<Integer> bestActionArr = new ArrayList<>();
87      double bestQValue = 0.0;
88      for (int j = 0; j < numOfActions; ++j)
89      {
90        if (j == 0 || qValue[state][j] >= bestQValue)
91        {
92          if (j > 0 && qValue[state][j] > bestQValue)
93          {
94            bestActionArr.clear();
95          }
96          bestQValue = qValue[state][j];
97          bestActionArr.add(new Integer(j));
98        }
99      }
100     return bestActionArr.get(rand.nextInt(bestActionArr.size()));
101   }
102
103   public int chooseAction(int state) {
104     double randValue = this.rand.nextDouble();
105     int chosenAction;
106     if (randValue < epsilon)
107     {
108       // Choose action randomly
109       chosenAction = this.rand.nextInt(numOfActions);
110     }
111     else
112     {
113       // Be greedy with current Q
114       chosenAction = chooseBestAction(state);
115     }
116     return chosenAction;
117   }
118
119   public void updatePolicy(double reward, int action,
120                            int oldState, int newState) {
121     // Implementation based off of
122     // http://webdocs.cs.ualberta.ca/~sutton/book/ebook/node75.html
123     double delta = reward + discount * bestQValue(newState)
124         - qValue[oldState][action];
125     eligibilityTrace[oldState][action] += 1.0;
126     for (int i = 0; i < numOfStates; ++i)
127     {
128       for (int j = 0; j < numOfActions; ++j)
129       {
```

```
130         qValue[i][j] += rate * delta * eligibilityTrace[i][j];
131         eligibilityTrace[i][j] *= discount * lambda;
132       }
133     }
134   }
135
136   public Policy getPolicy() {
137     int[] actions = new int[numOfStates];
138     for (int i = 0; i < numOfStates; ++i)
139     {
140       // Always be greedy in the final step (from Piazza post 103)
141       actions[i] = chooseBestAction(i);
142     }
143     return new Policy(actions);
144   }
145
146   Random rand;
147
148   private double discount;
149   private double rate;
150   private double epsilon;
151   private double lambda;
152
153   private int numOfStates;
154   private int numOfActions;
155   private double[][] qValue;
156   private double[][] eligibilityTrace;
157 }
```