Rahul Zhade
CS 4641 Assignment 4

Markov Decision Processes Write-up

1. Introduction

For this assignment, I am using two different instances of GridWorld with one goal state, with varied sizes. The first instance is a smaller MDP, of size 5x5, thus with 25 goal states. The larger map was 11x11, thus around 5 times larger than the original MDP with 121 goal states. As can be seen below, I initialized each map with a reward state at the top left, represented by the blue square, with a "trap" terminal state, represented by the red square, somewhere else in the map. The gray circle represents the start state for the agent. I started with an initial stochasticity of 0.8, which meant that the probability an agent would perform the intended move is 0.8, and the probability that the agent would perform some other unintended action is $\frac{1-0.8}{3} = \frac{0.2}{3}$, since the total set of actions in a grid world are North, South, East and West.

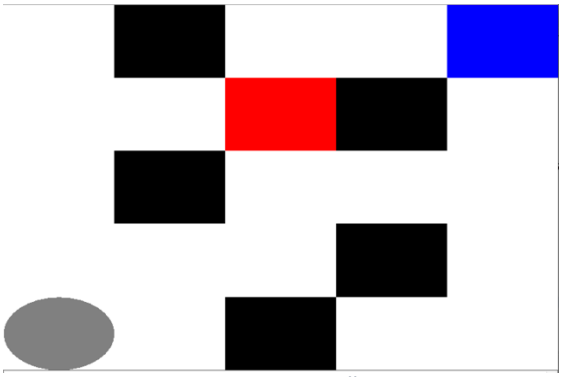


Figure 1: Smaller Map

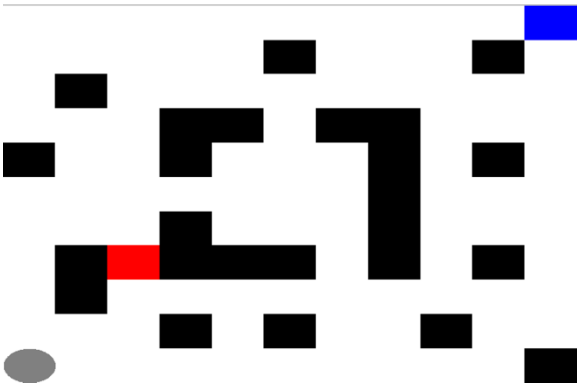


Figure 2: Larger Map

I used [Juan Jose's boiler plate code], so this made the codebase extremely extensible, allowing me to experiment with several different types of parameters. Below, I experiment with different values for the discount as well as the reward value. I also experimented with different types of reward functions, with one being a linearly increasing function, and one being a static reward function.

1.1. GridWorld

A GridWorld is a simple example of a MDP, because of the simple way in which a large state space can be modelled with such few parameters. Since the agent's actions are not completely deterministic, the MDP can be easily changed by simply changing the stochasticity. The parameters that can be changed are the reward function, the transition function (which is modelled by the stochasticity of the agent).

1.2. Novelty

The GridWorld problem can also easily be extended to the real world, especially within the field of robotics. Since sensors and actuators are not 100% accurate, as well as environments being treachorous and difficult to navigate, real world environments can be modelled with GridWorlds, with ravines and other "traps", as well as reward states, and walls all being part of the state representation of

this problem. Thus, because of the simplicity of the problem, the extensibility to create new problems, and how grounded this problem is in real life, I find the GridWorld MDP interesting.

1.3. Functions

The two algorithms that I will be analyzing are Value Iteration and Policy Iteration. These two are both iterative algorithms, which update the utility of either the state or the policy by querying possible actions. Value Iteration is a greedy algorithm which iteratively takes an action which optimizes a factor called "utility". Since the reward function and the utility function might be different, and usually are, this utility factor needs to be updated iteratively. Policy Iteration is another iterative algorithm which seeks to speed up the convergence of Value Iteration, by updating the utility of a *policy* instead of the state space. It does this by initializing a random policy, and then updating the utility of the policy iteratively, by locally finding better policies, using Value Iteration. The parameters that can be changed for this are discount.

2. Small Grid

2.1. Constant Reward Function

I started by running policy iteration and value iteration on the maps with the constant reward function, where the agent gets no reward unless it reaches either the trap state or the reward state, at which it gets a reward of either -100 or 100 respectively. My results are tabulated below:

Constant Reward Function

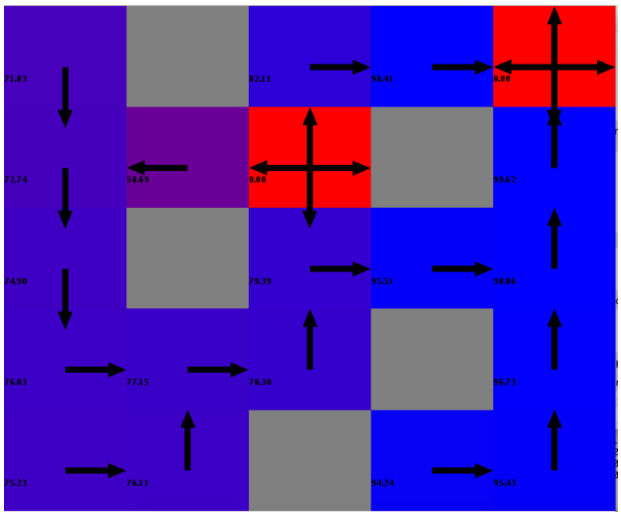


Figure 3: Value Iteration Results

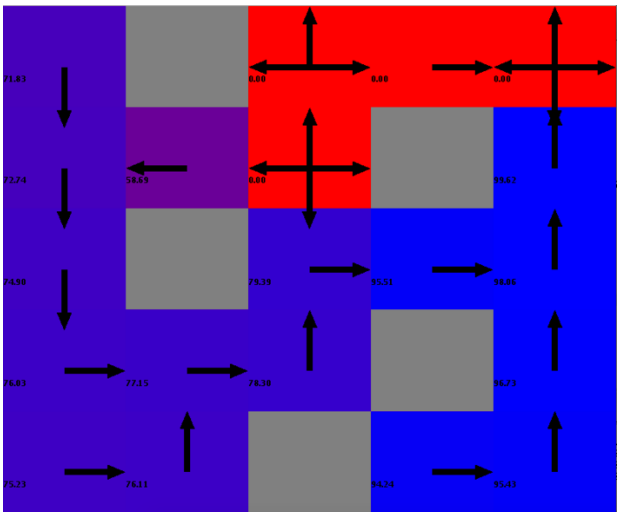
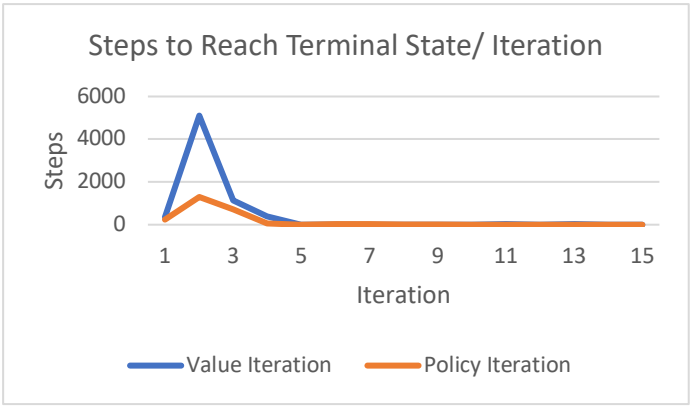
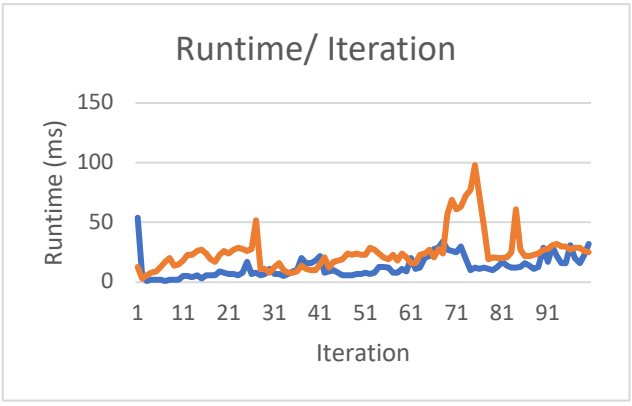


Figure 4: Policy Iteration Results





I experimented with different values of both of these, however, this combination of values seemed to be a good representation. As we can see from the graphs, value iteration took a larger number of steps to reach the terminal state, until both functions were both able to find an optimal at around 4 iterations for Policy Iteration vs. 5 iterations for Value iteration. This was expected, as Value Iteration needs to calculate utility for every single state, whereas Policy Iteration updates based on a policy, i.e. random actions. In our GridWorld above, there are 2 states that are unreachable, since the only two ways to get to these squares are through either the trap or the reward state. While Policy Iteration would skip these states, since they are unreachable by any policy, value iteration would try to use these, and update the utility for these states as well.

The differences in runtime are also expected, as Policy Iteration usually had a longer runtime per iteration than value iteration. Policy iteration needs to run value iteration within it, in order to determine the utility of the policy, which means that each iteration would take much longer to finish than an iteration of value iteration, on average. However, when I checked, usually Policy Iteration only ran one iteration of Value Iteration within it.

2.2. State by State Reward Function

The next reward function that I tried was a state by state reward, which gave the agent a reward a reward no matter what state it was at, only varying at the trap state and the reward state. For this, I gave a reward if 10 at every state, with a goal state having a reward of 400, and all other parameters remaining the same. I'll note that using a *negative* reward at all states had nearly identical results to the constant reward situation, with the only notable difference being a slight decrease in the number of steps required to reach the goal state.

State by State Reward Function

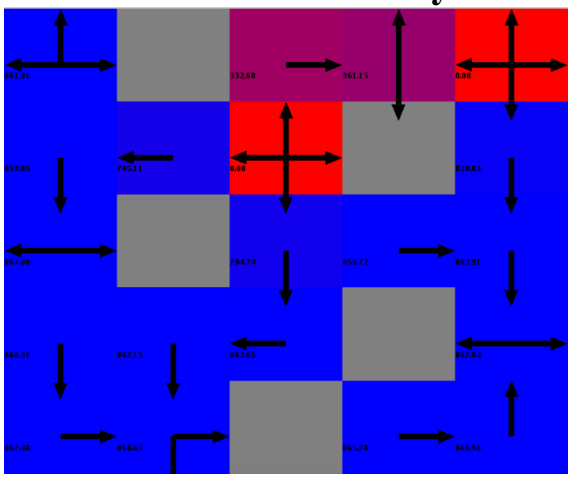


Figure 5: Value Iteration State by State

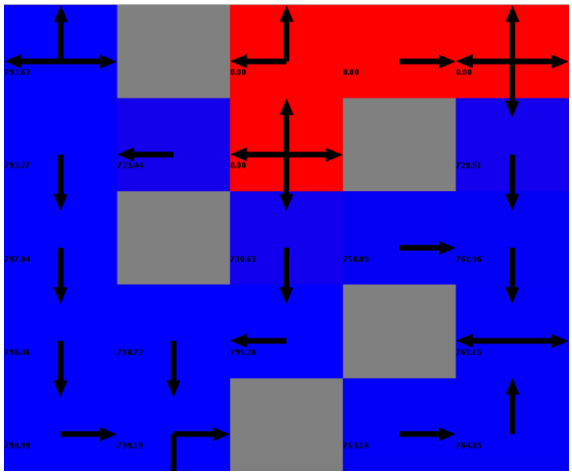
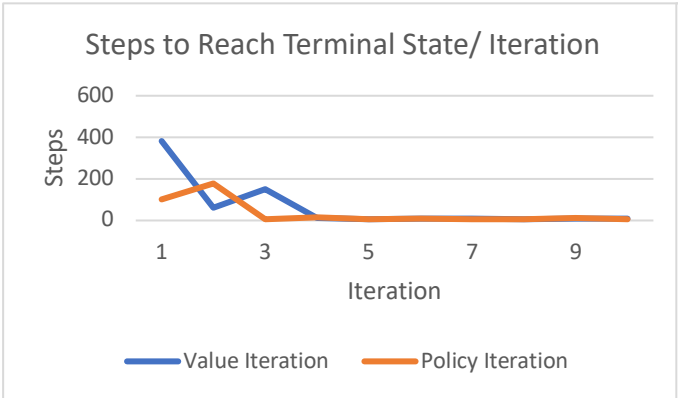
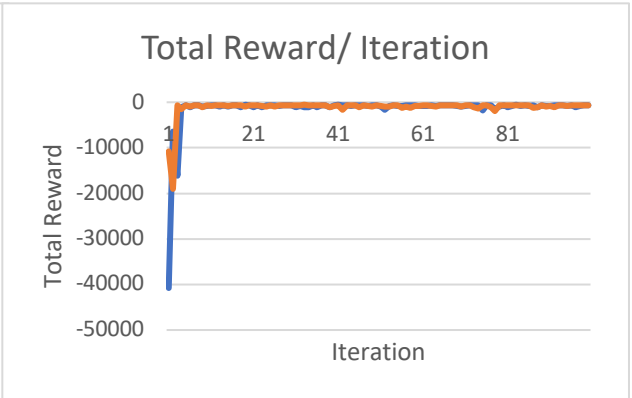


Figure 6: Policy Iteration State by State





The results for runtime were similar to those shown above, so I didn't bother showing them twice. The most notable thing is the change in optimal policy. The algorithms found that the most optimal policy was to keep moving around, since just moving around gave a reward of 10 each time, rather than getting a one- time reward of 100. This was expected, however, I just thought it was interesting to share in this report.

2.3. Linear Punishment Reward Function

The next reward function was the linearly decreasing punishment function, which punished the agent the further it got from the goal state.

Linear Punishment Reward Function

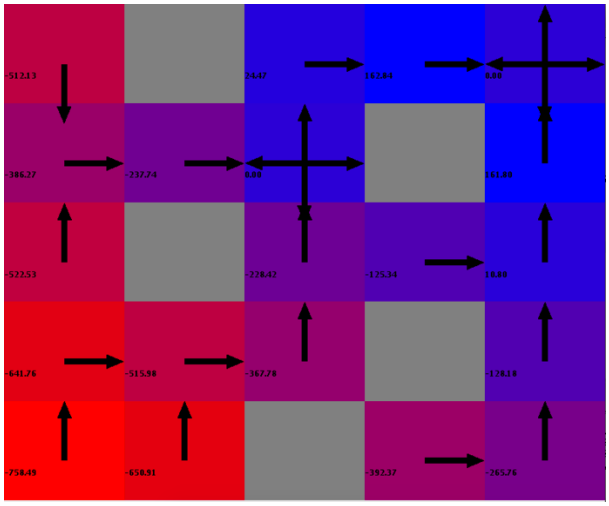


Figure 7: Value Iteration Linear Punishment

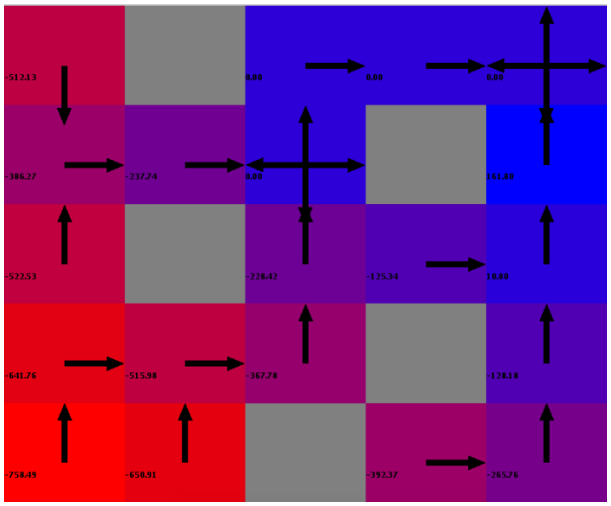
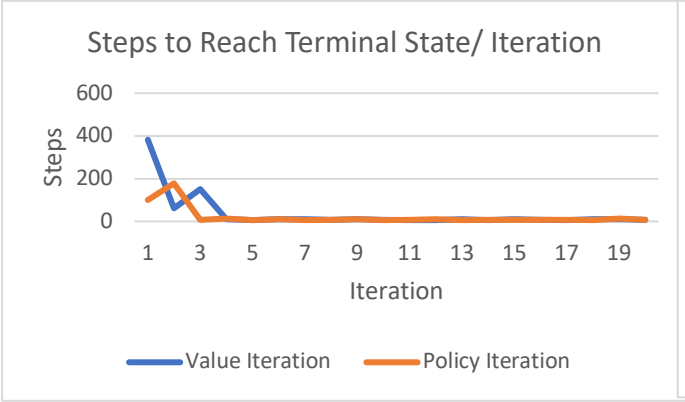
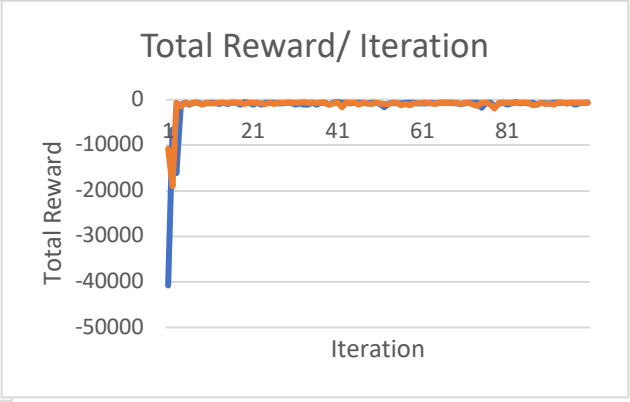


Figure 8: Policy Iteration Linear Punishment

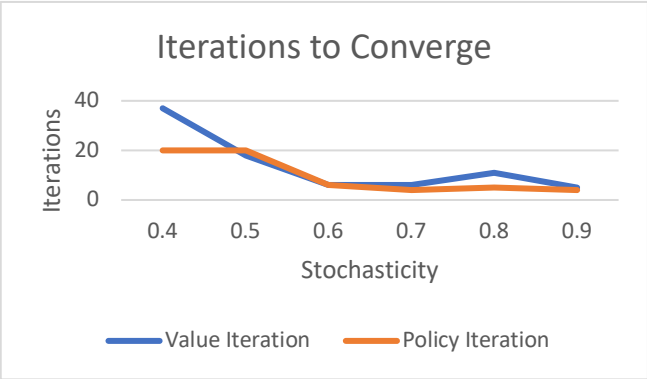




These policies were much different from the ones generated by the constant reward function, since this punishment function heavily encourages the agent to move further towards the goal state as quickly as possible, at the expense of getting close to the sink state.

2.4. Stochasticity

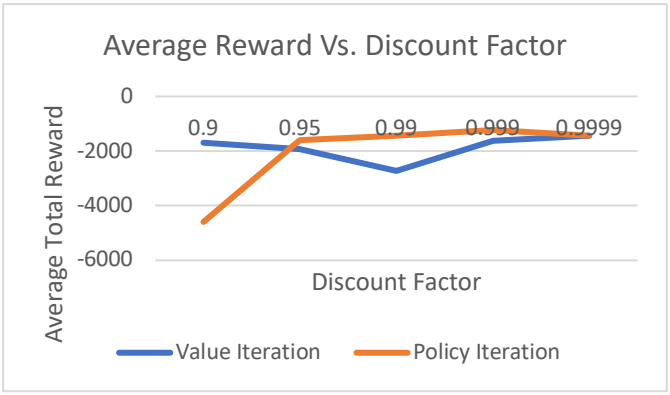
Since the policies generated for different values in stochasticity are similar, I just tabulated the number of iterations to converge for each different value for the stochasticity below. I defined stochasticity to be the likelihood that the agent performs a certain action, and thus it is represented by a probability as shown below, i.e. a stochasticity of 0.8 means that the agent is 80% likely to perform the intended action, with a 20% likelihood of performing one of the other 3 possible actions at the given state.

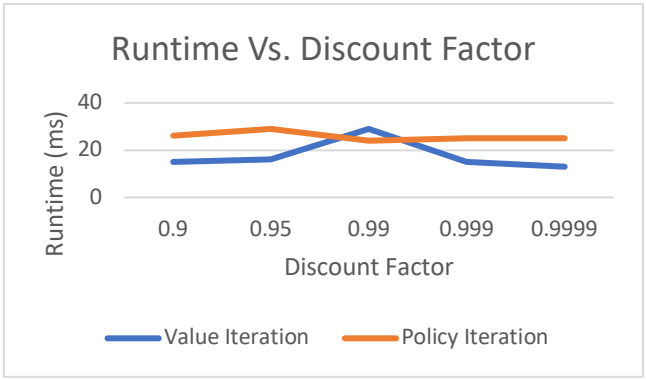


For the above, it is clear that the number of iterations to convergence is negatively correlated with the stochasticity, i.e. the higher the stochasticity, the sooner it converges. This is expected, as a higher stochasticity value, as I defined it, results in a higher determinism, which means that the agent is less likely to have *not* followed the policy until that point. The same relationship was observed for all of the reward functions that I tried, so I won't display the other graphs above.

2.5. Discount Factor

I also varied the discount factor to check the performance of the algorithms, with the other parameters held constant, using the linear punishment reward function. Below I tabulated my results. Please note that the x- axis **is not** linear, and is simply a selection of different discount factors that I tried out.





The Average total reward that I mention above is the average total reward across the last 10 iterations. Since the reinforcement learning algorithms bounce around different values for the average reward, I used the average reward, since that was found to be more stably linear, rather than the total reward found at convergence. As we can see above, policy iteration performs fairly badly for lower discount factors, in terms of both total reward found, and in runtime. Policy Iteration is fairly stably worse than value iteration for run time, and I attribute the weird spike in runtime for value iteration at 0.99 to be a statistical outlier. However, I attribute the bad performance of the Policy Iteration algorithm for lower discount factors because a lower discount factor forces the algorithm to converge quicker. Since policy iteration took around 6- 7 iterations to converge anyways, using a lower discount factor makes it converge to a suboptimal policy.

2.6. Other

I also altered the state representation of the MDP by changing the position of the reward state, the position of the sink state, and even the number of sink states. Altering these aspects of the MDP changed the optimal policy that was found by the algorithms, however, did not affect performance much.

Increasing the number of sink states increased the number of iterations required to converge, as well as the runtime for policy iteration, however, nothing significant enough to justify a graph here. The

In conclusion, for these smaller states, the difference in runtime between policy iteration and value iteration was small enough, while the difference in policies generated was insignificant. While both algorithms ended up finding similar policies in the end, policy iteration converged much quicker, in terms of iterations, than value iteration. It might be interesting to note whether the similarity in policies is because of the small state size, or whether it is because the algorithms both converge at the same policy. From what we learned in class, it is the latter.

3. Large Grid

3.1. Constant Reward Function

I used the same method, for the same parameters for this graph. Again, results are tabulated below:

Constant Reward Function

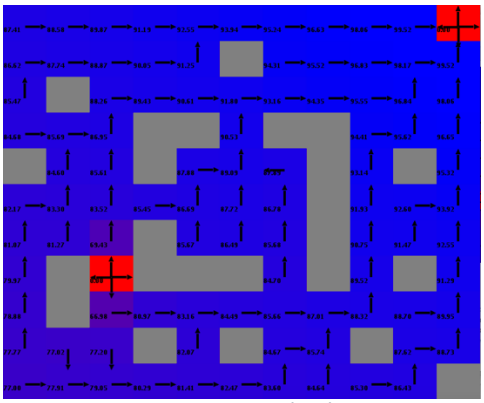


Figure 9: Constant Reward Value Iteration

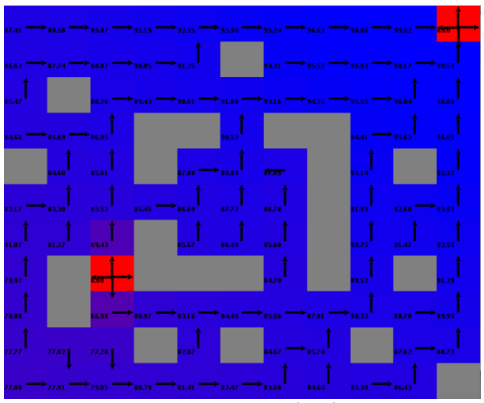
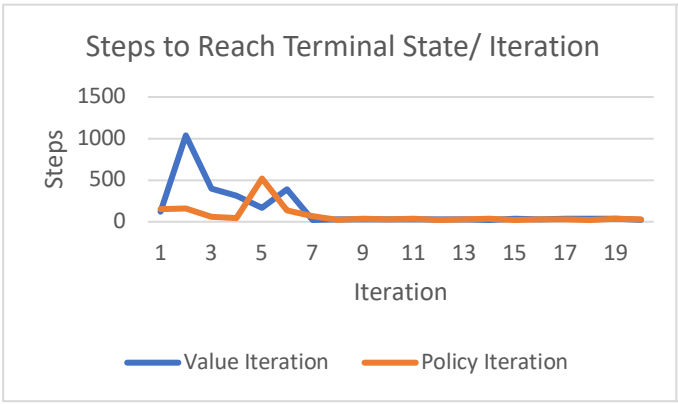
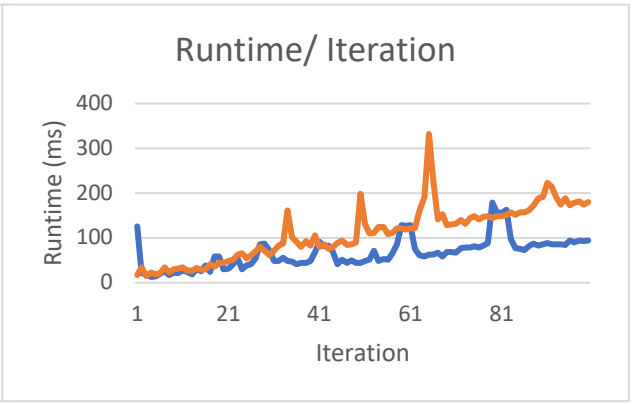


Figure 10: Constant Reward Policy Iteration





So above, a major difference can be observed in the runtime vs. iteration graph. Policy Iteration took significantly longer per iteration to run than value iteration for this state space, while for the smaller MDP, the two times were comparable. This is because policy iteration is less efficient for larger state spaces. It takes much longer, since policy iteration runs value iteration within itself. More iterations of value iteration need to be run within policy iteration for larger state spaces. This was verified by the output of the program, which told me that, while the smaller MDP averaged around 2- 3 iterations of value iteration per iteration of policy iteration, this larger MDP averaged around 6- 7 iterations for this state space. It was interesting the way that this happened though, as most of the time policy iteration would require only one iteration of value iteration, but every so often it would take 18 iterations of policy iteration.

3.2. Linear Punishment Reward Function

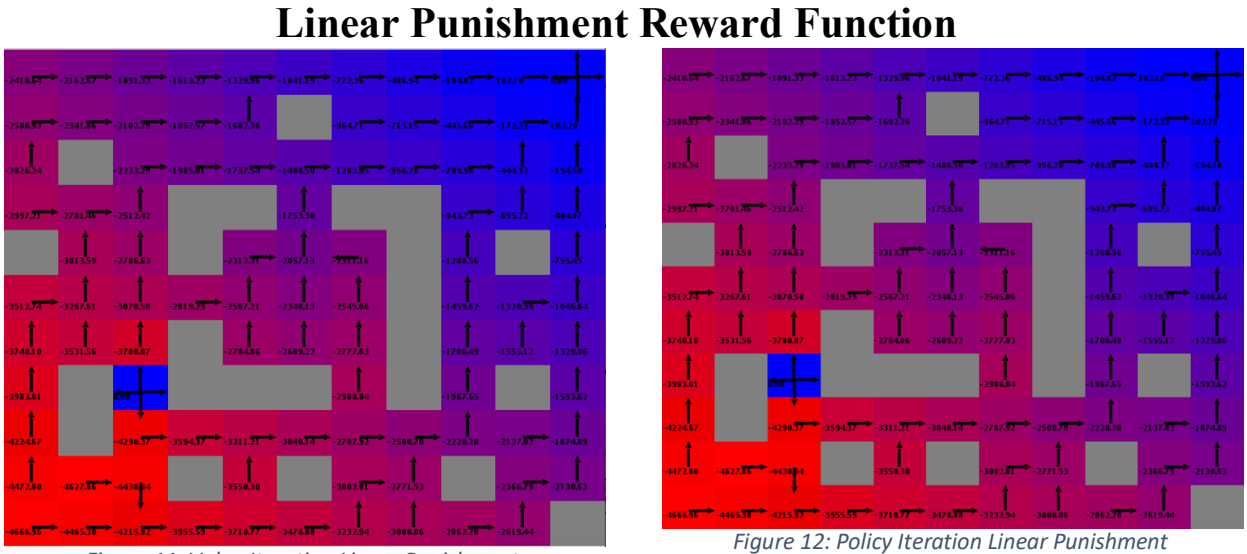
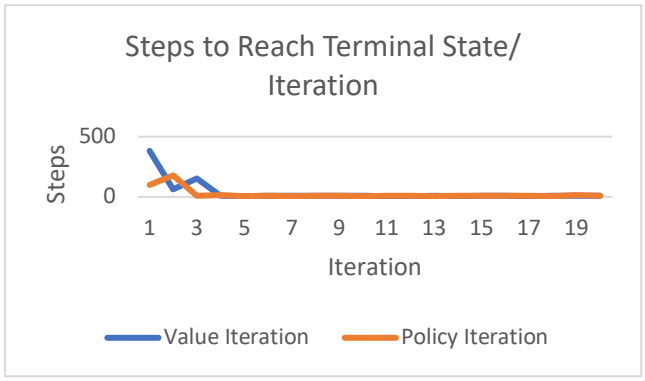
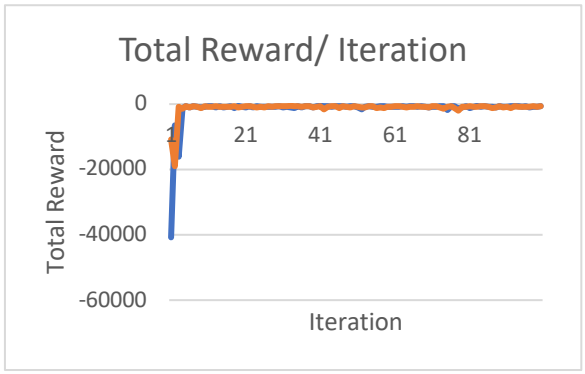


Figure 11: Value Iteration Linear Punishment

Figure 12: Policy Iteration Linear Punishment

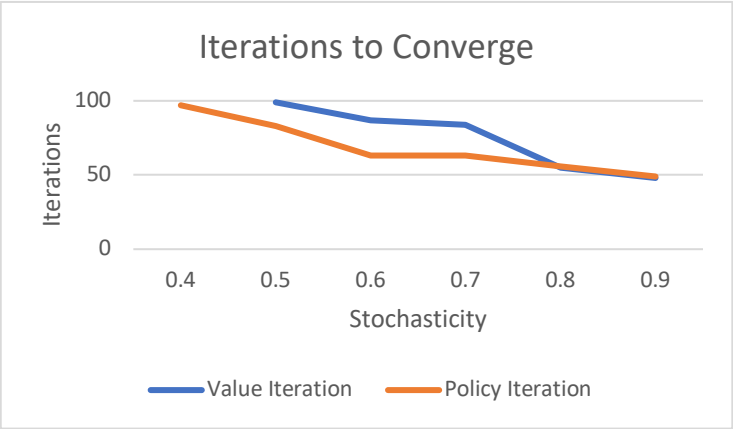




The linear punishment reward function, again, resulted in a similar policy as the constant reward function did, however, again, the sense of urgency can be observed in the policy. I increased the punishment parameter here to 10000 in order to make it not worth going to in the 100 iterations that I limited the iterations. The same patterns were being noticed here for steps to reach terminal state as well as total reward/ iteration. The same pattern was also observed with runtime/ iteration, with Policy Iteration taking significantly longer in time to perform its iterations as the number of iterations increased, however, it converges much quicker than value iteration in terms of the number of *iterations*.

3.3. Stochasticity

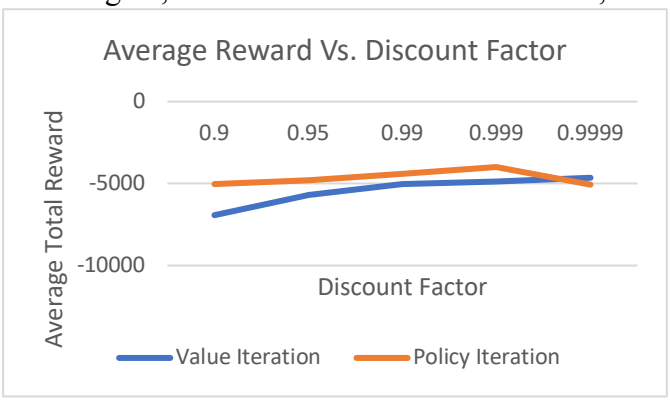
Below, I varied the stochasticity as I did for the smaller MDP, and then observed how many iterations each algorithm took to converge.

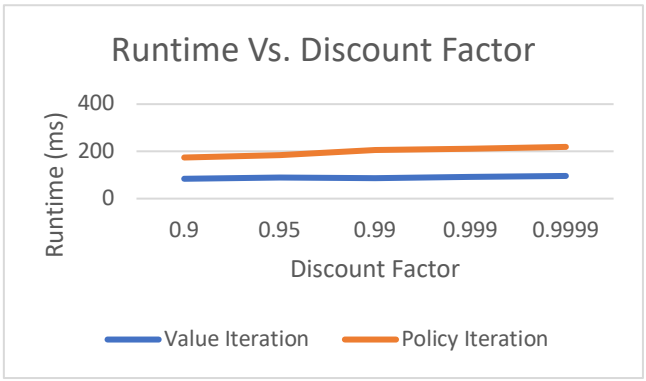


As shown above, policy iteration required less iterations in order to converge than value iteration, as was also found in the smaller MDP. I will note that for stochasticity values of 0.4, the value iteration did not converge in the 100 iterations that I ran for it, and so no values were entered in the graph for this data point. However, the time required to converge was much larger for policy iteration than for value iteration, as was explained above. It is quite interesting how the number of iterations was very similar for both PI and VI for higher values of stochasticity, however, they diverged when the environment became more random (lower values of stochasticity).

3.4. Discount Factor

Again, I used the same method as above, in order to find the values for each discount factor.





As I theorized above, policy iteration had a greater runtime than value iteration for all values of the discount factor. My theory regarding the average total reward was also confirmed, as the average total reward was found to decrease as the discount factor decreased. However, it seemed like too high a discount factor results in a decrease in the average total reward. This dip was observed both in the small MDP and the large MDP, so it seems like too high a discount factor results in too dangerous policies found, resulting in a decrease in the reward, as the policy takes the agent close to the sinks, causing the decrease in the total reward.

3.5. Other

Again, I tried to see how varying the number of sinks, locations of sinks, and changing "hallways" affected the performance of the algorithms. "Hallways" are parts of the map that have one block to pass through, with sinks on either side. Since any random movement except for the intended one would cause the agent to fall into the sinks, this is a good problem to observe. Again, I did not want to graph anything, as the most notable thing that was changed by the model

representation was the policy that was generated. It was interesting how the policy for different hallways was more and more likely to choose hallways as the value for the discount factor increased, since a higher discount factor is correlated with a higher value for future rewards, even at the cost of present danger.

4. Conclusion

In conclusion, policy iteration seemed to a better, more versatile algorithm to solve GridWorld problems. While both algorithms seemed to find similar algorithms, when run for a long enough time, policy iteration converged much quicker than value iteration in terms of number of iterations. This observation was constant, across all values of stochasticity that I checked, with policy iteration performing much better than value iteration in more random environments. However, policy iteration suffered in terms of time required per iteration, especially in larger state spaces. While the time per iteration was similar for the smaller MDP that we tested, the difference between the two algorithms became apparent for the large MDP, as the difference between the runtime per iteration was quite significant as the iteration number became larger. While higher values for the discount factor were better for most cases, it led the agent through more dangerous situations, while lower values for the discount factor led to lower values for the total reward, indicating suboptimal policies found.