Jorge Betancourt

12/04/18

CS 4641

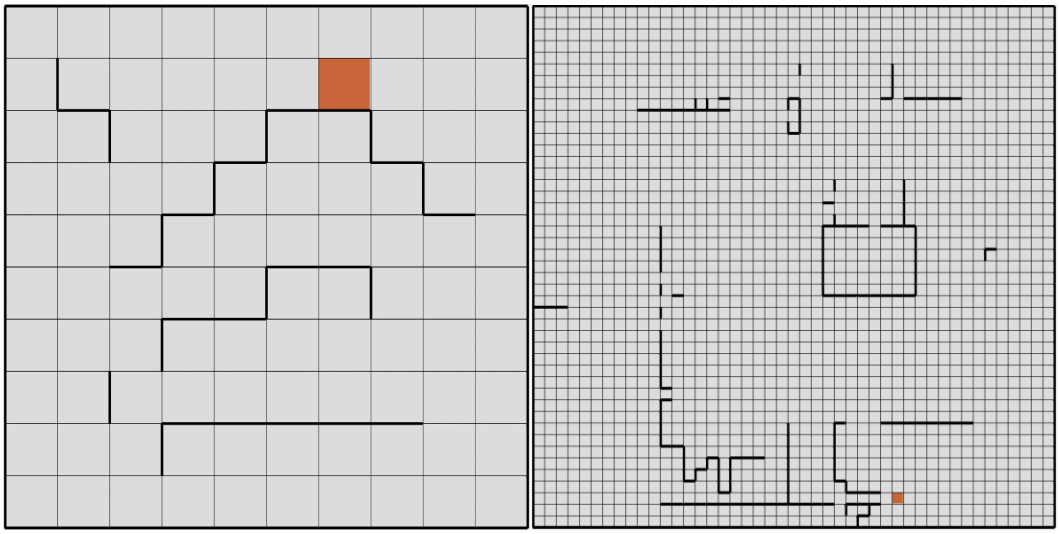
Assignment 4: Reinforcement Learning

Problem Description and Intro

Markov Decision Processes

Markov Decision Processes (MDP) are models of problems that are discrete, time stochastic, where we assume the Markov Property: that the effects of an action taken in a state depend only on that state and not on the prior history. Every MDP contains states, actions, rewards for each action in each state, and transitions. We use these models to simulate many real-world scenarios, like vacuum robots traveling a learned map of a house. For these experiments, I will simulate this application of MDPs by generating maps that simulate homes where every state that is not the goal state has a reward, or punishment, of -1 and colliding with a wall provides a punishment of -50. This should encourage the agent to prioritize the goal state while simulating a realistic fear of damage to our hypothetical vacuuming robot if it were to collide with its environment.

Maps



I chose these two maps because of the very large difference in domain size. The first map only has 100 states while the second has 2025 states. There is a possibility that the action in a state does not lead to the corresponding transition, the agent can 'veer' and end up going in any direction from the current state except for the opposite of the desired action. Because there will be a percentage of error dependent on the PJOG parameter, the fastest path to the goal may not be the most rewarding.

The first map is interesting in itself because of the difficulties certain paths would have over others. The bottleneck on the right of the map should show different values than on the left where there is more room for the agent. By changing the PJOG parameter, we will see how the agent takes into account the 'risk' of an area that would otherwise be risk free if the agent's actions were deterministic.

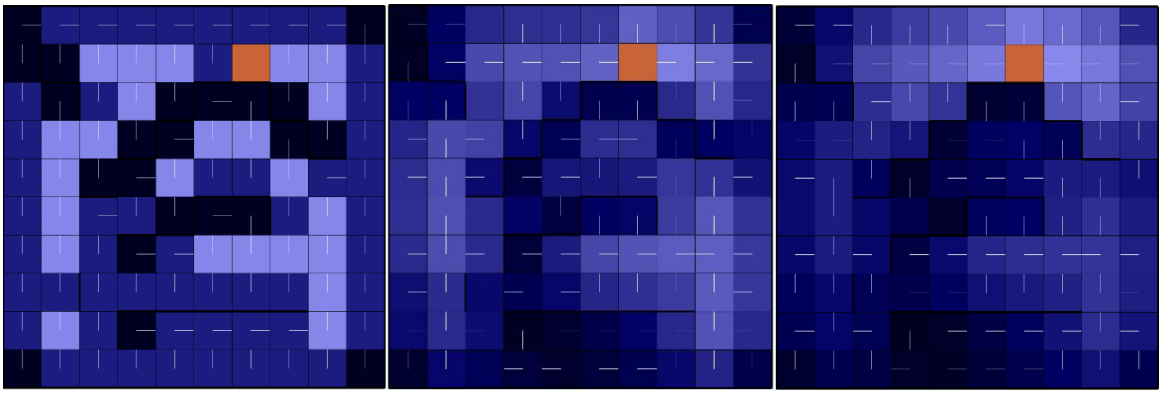
The second map provides insight into how the algorithms work with a large number of states. With the large number of states, this map will be able to highlight the advantages and disadvantages of the different RL algorithms as well as the way PJOG can affect exploration of an agent in an unknown state space.

Solving the MDPs

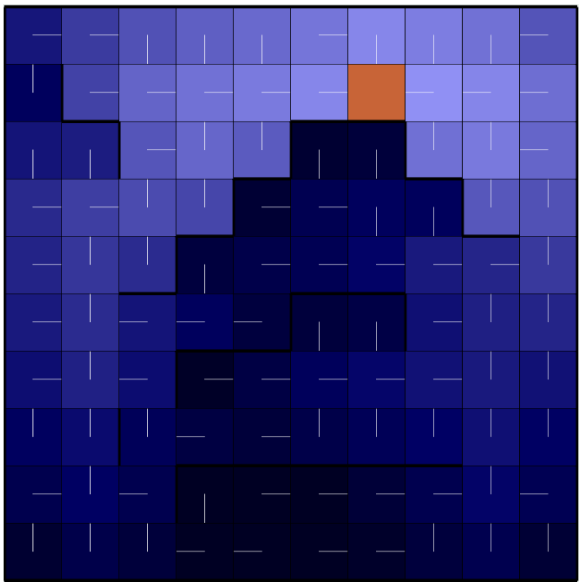
Value Iteration

Value iteration works by iteratively calculating the values for all states and determining the max rewarding action to take (in this case the one that will lead to being punished the least). We estimate the reward of each state in the state space.

The first problem led to expected results:



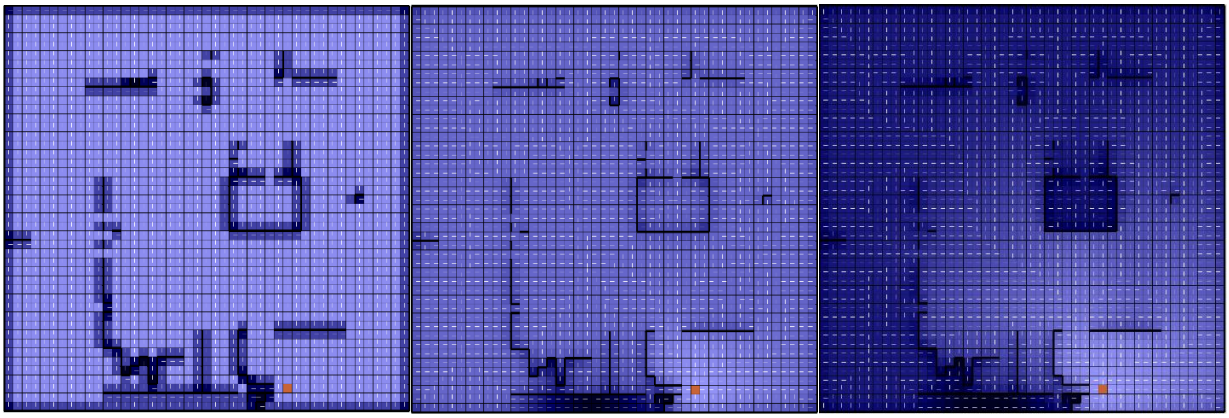
1 Step 6 Steps 60 steps



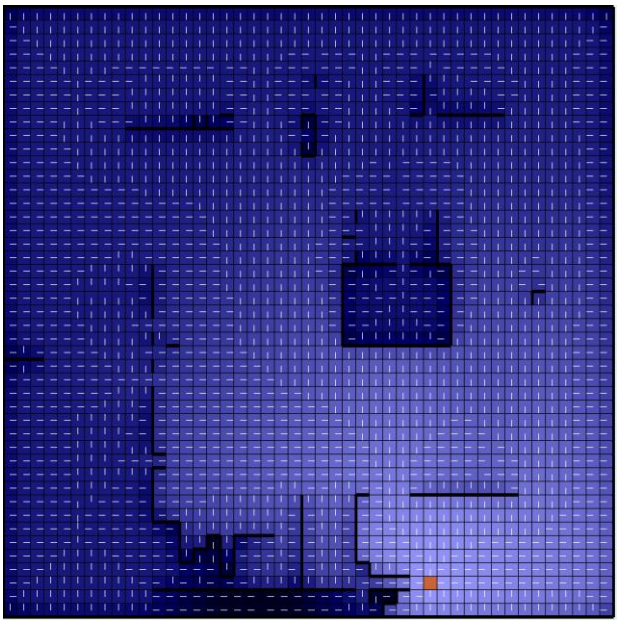
Convergence: 76 steps, 12ms

We can see that after convergence, there is a lighter shade of blue indicating higher value on the left side of the map, where there is more room allowing error. This sharp change in value for what seems like a slight shift in layout is most likely due to the harsh punishment we have assigned wall collision.

The second problem also provided interesting results, showing the first con to relying on value iteration.



1 Step 26 Steps 78 Steps

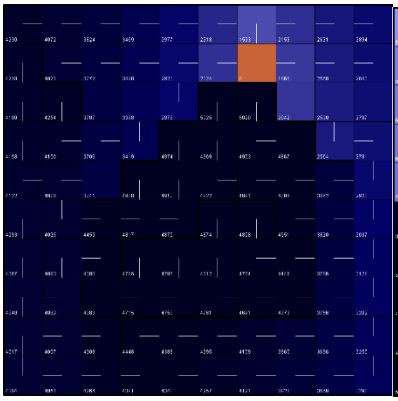


Convergence: 181 steps, 2866ms

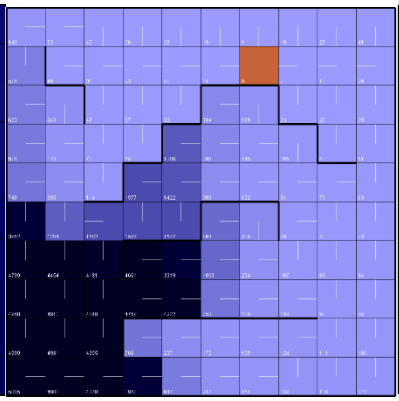
We can see that while the algorithm did converge on good values for each state, it took much longer given the amount of states in the space. This is because of the nature of value iteration, where it evaluates every action in a given space and finds the most rewarding action. 4 actions for 2025 states is many calculations. One of the ways to combat this is through policy iteration.

Policy Iteration

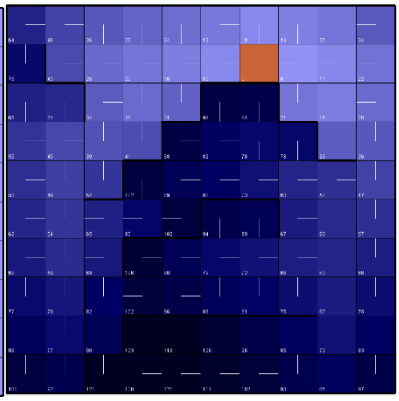
Instead of going through every action in every state, we decide to compute values for a fixed policy. This should reduce the amount of calculations done per iteration. When running on the first problem we notice implementation differences reflected in the step images.



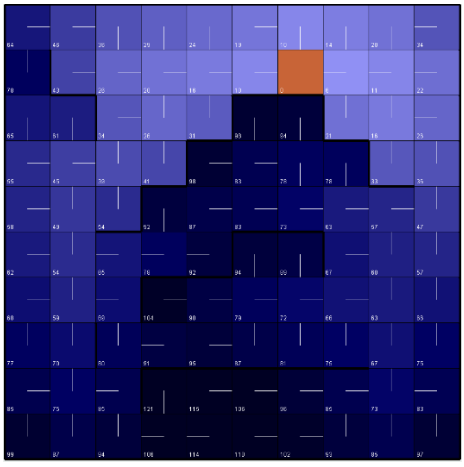
1 Step



2 Steps



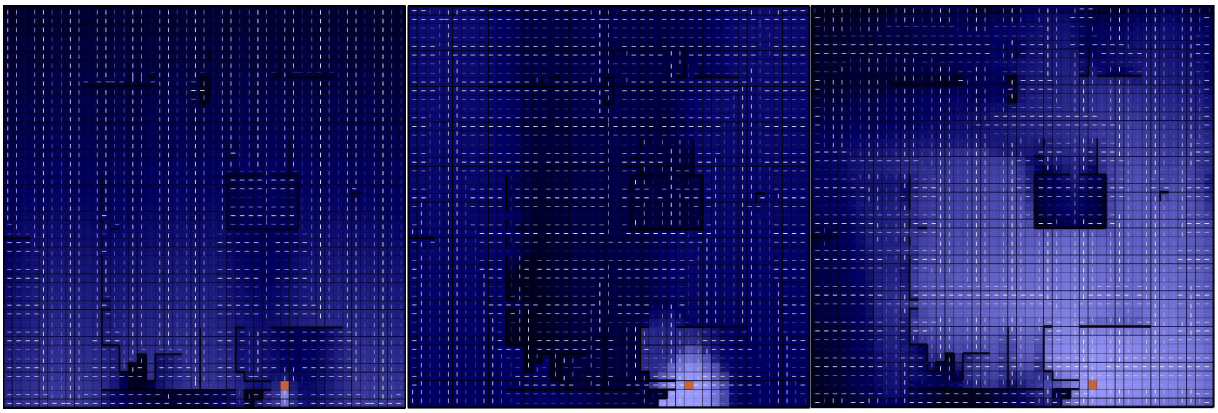
6 steps



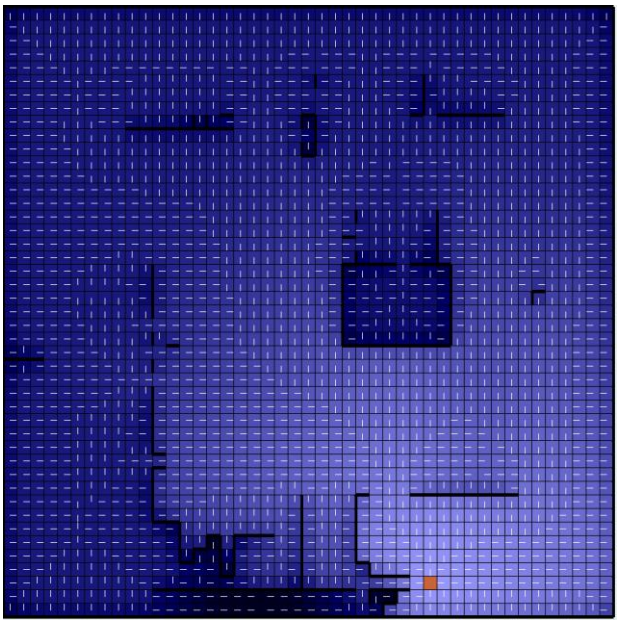
Convergence: 8 steps, 34ms

We see that the number of iterations drops drastically for policy iteration (from 74 to 8) while the time it took to run until convergence went up. This is because policy iteration has 2 steps per iteration: convergence of the state values and the policy update.

We see in the following MDP how the extra time per iteration is worth the amount of iterations conducted trying to estimate the values of MDPs with large state spaces.



1 Step 4 Steps 7 Steps



Convergence: 15 steps, 2122ms

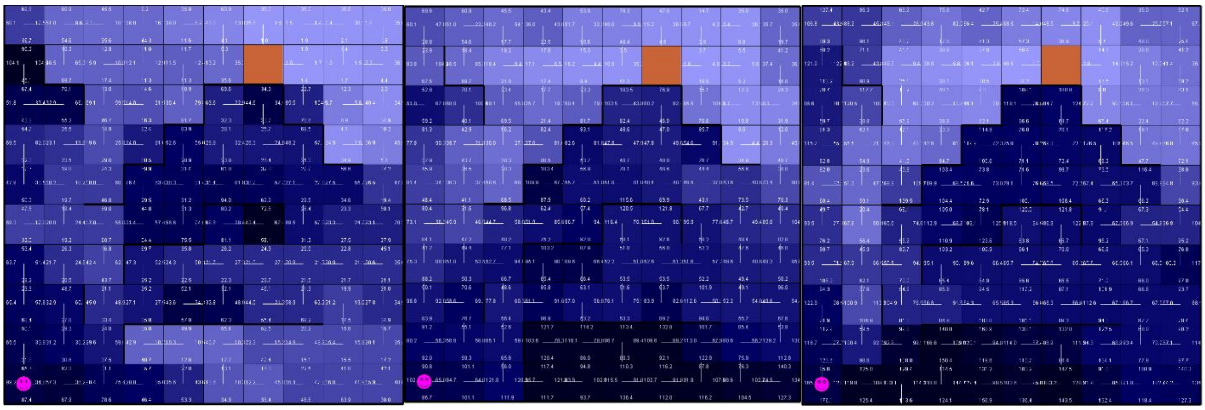
The number of iterations was cut down by an impressive amount, and we can see that reflected in the run time. We should also notice that the results of both value iteration and policy iteration are the same. This is because they are both trying to estimate the same thing, the value at each

state. The only difference is value iteration determines it directly while policy iteration does it indirectly based off the policy which is turn is determined by the value.

Q-Learning

The largest con from both of these two iterative algorithms is that they both require a complete knowledge of the state space. Something not common when working with changing environments like a house in the vacuum problem we tried to model in our MDPs. Because of this, we need an algorithm that explores the state space and builds a knowledge of the values at each state as it explores. This is where Q-Learning comes in. In an attempt to avoid converging on a globally inefficient local optimum, we will keep PJOG as it is to incentivize exploration as well as epsilon, a hyperparameter that determines the likelihood the agent decides to take a new path and explore. For these experiments I will be using two different exploration values, 10% and 25%.

Epsilon = 0.1:

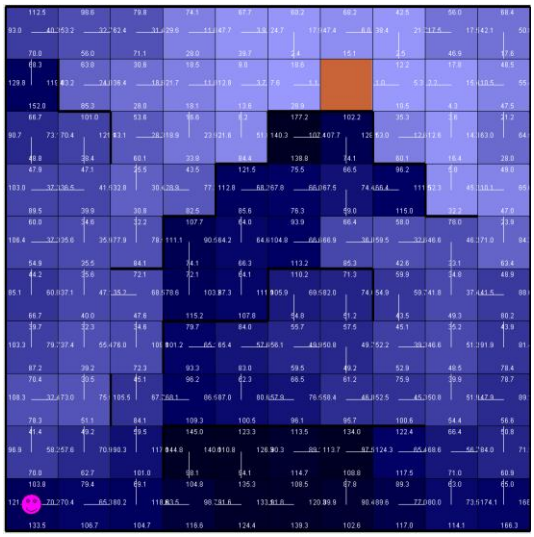


25 Episodes 50 Episodes 100 Episodes

I was impressed with how quickly the learning algorithm was able to gain a sense of the state space with no prior knowledge. It looks very similar to how the other algorithms were able

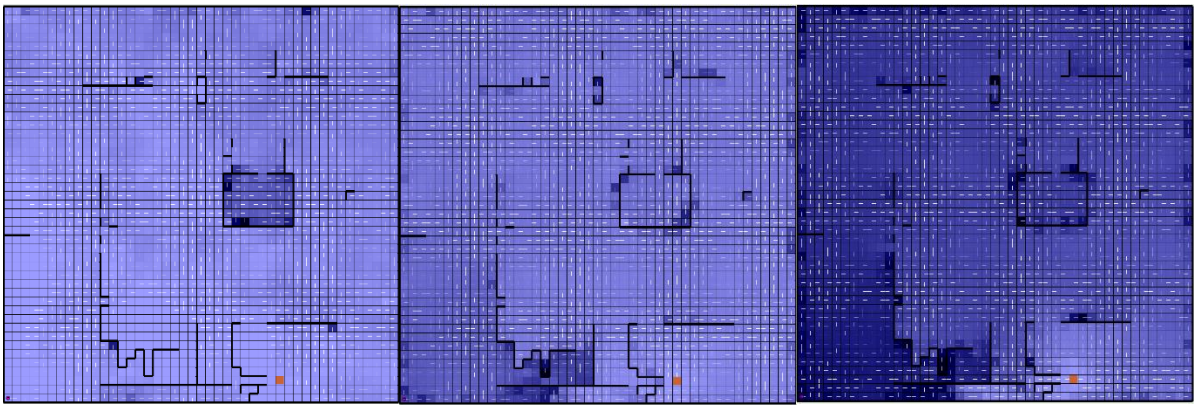
to determine that the left path is safer and almost just as efficient as the right path as seen in the white path markings as well as the lighter shade of blue in the grid.

Epsilon = 0.25, Episodes = 100



Here we can see that the agent was more willing to explore the right side of the maze and found paths leading through the bottle neck. This shows that with a higher epsilon we are able to search more of the state space.

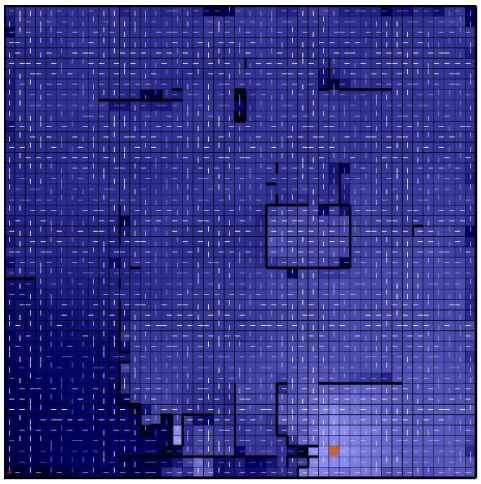
Epsilon = 0.1:



25 Episodes 50 Episodes 100 Episodes

Again, I was impressed with how quickly the learning algorithm was able to gain a sense of the state space with no prior knowledge. Due to the high PJOG value, it correctly identified the tunnel towards the bottom as a non-optimal path to take. It also understands to avoid the walls by moving directly opposite to them to avoid the -50 reward.

Epsilon = 0.25, Episodes = 100



Here we see darker shades of blue near the starting location. I believe this is due to the high activation chance of epsilon which caused the agent to explore the walls near the start, thereby devaluing the value of the state.

It is important to see that Q-Learning is overall faster than the other two algorithms because it only has to iterate along certain parts of the state space it is exploring during that episode as opposed to the entirety of the state space. However, this has a trade off in where the optimal path may not be found due to it not being explored in any episodes of Q-Learning's cycle.

Conclusion

It is important to see the benefits of all the reinforcement algorithms and the times each one should be used. Since the 2 algorithms that use the knowledge of the state space converge to the same result, it is more so a question on which algorithm is more efficient given the size of my state space. As for Q-Learning, it is extremely important to understand how action error can affect the agent's ability to learn and its willingness to explore. Fine tuning epsilon as well as the MDP's parameters (like making all spaces' r = -1 and the amount the agent is punished for wall collisions) can change the way the states are explored and the paths that are optimized.