Joshua Philipose

Dr. Charles Isbell

CS 4641

4/14/19

Markov Decision Processes Analysis

Introduction

Throughout this paper, we will be exploring Markov Decision Processes (MDPs) and reinforcement learning (RL) algorithms, primarily focusing on Value Iteration, Policy Iteration, and Q-Learning. An MDP is a state-based system or process where the transition between states has an element of randomness (stochasticity) to it, rather than being fully deterministic. There is a transition model that acts as the "physics" of the system by governing the transitions according to a probability distribution. Essentially, if an agent decides to take action, there is some probability that a different action may be done instead – think of a robot navigating a room; it may choose to make a 180 degree turn and move backwards but may randomly slip on some obstruction and move right instead. Because a realistic environment often includes some of this noise and stochasticity, we include this in our transition model. When the MDP enters a new state, it is presented with some sort of award. The beforementioned transition model and the system by which rewards are distributed may be known or unknown when attempting to navigate an MDP. Regardless, our end goal is to find a policy π that tells us what to do in any given state. Obviously, our task is easier if given the transition model and reward system. When we have both, we can make use of value iteration or policy iteration to find our optimal policy. This journey is detailed in later sections. If we don't have access to the transition model, however, we can still utilize reinforcement learning to approximate the ideal policy. For this report, I choose to use a Q-learner as my reinforcement learner, mostly because it was the learner I'm most familiar with.

Markov Decision Processes

Gridworld Maze (Small and Large)

The MDPs I chose to explore were both gridworld mazes of differing size and complexity. The first gridworld was a small, 10x10 maze with a relatively high number of walls for its size. I chose this setup to mimic the environment a learner may face if its task is to navigate inside a residential or commercial building. An example may be a robotic gurney that needs to navigate to its patient's room inside a hospital. I added a wall cost of 50 (relatively high) for this gridworld because a robotic gurney should take care not to bump the walls and risk the health of its patient. For the second gridworld, I selected a much larger 45x45 maze with a relatively few number of walls for its size. This setup is intended to mimic the environment an agent may face if it had to explore a countryside or a rural town. There are relatively few obstructions in such an environment, reflected by the relatively low number of walls. An example may be a robotic mail carrier that needs to deliver mail to a sparsely populated rural town. I added a relatively low wall cost (20) to this gridworld because in practice, it's fine if a robotic mail carrier bumps into something here and there, as long as it is efficient in delivering mail.

Methods

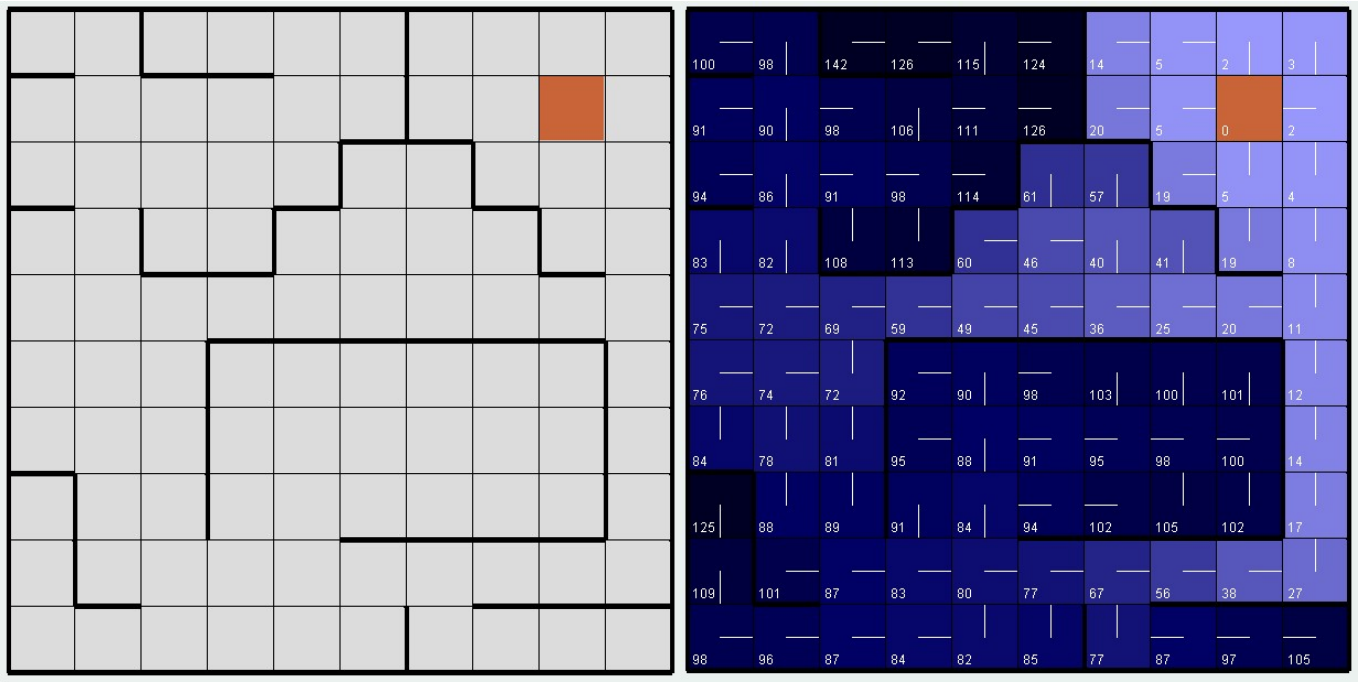
The two implemented MDP algorithms (value iteration and policy iteration) and Q-learning were run multiple times, with different hyperparameters, on the two MDPs defined above. These runs were done by utilizing Carnegie Mellon's (CMU's) Reinforcement Learning Simulation, a GUI based Java application created by two CMU grad students that created

the MDP gridworlds and ran the beforementioned algorithms. Throughout this report, I've included a few screenshots from this application to help explain my experiments and findings.

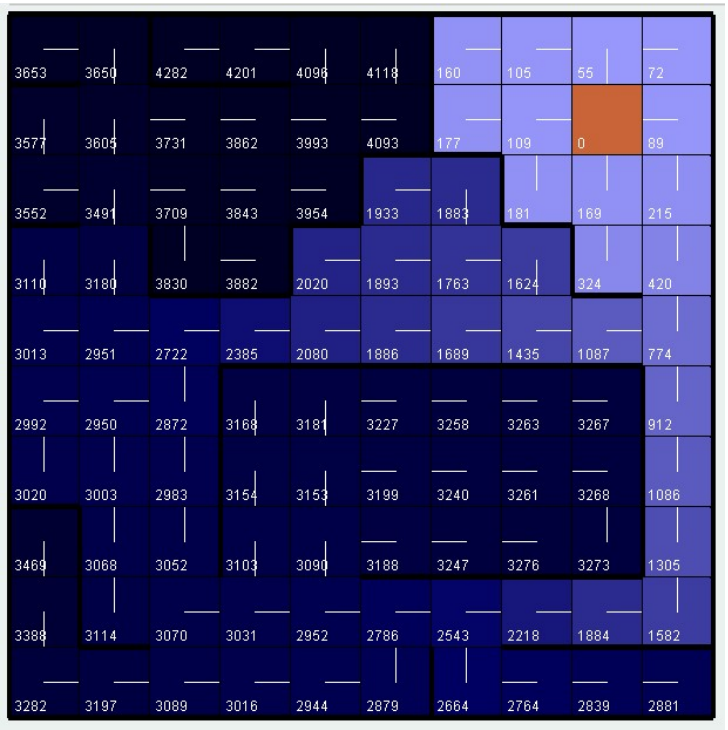
Experiments

Value Iteration:

Value iteration is an algorithm that essentially solves the Bellman equation for each state, providing us with a utility value and suggesting to us what action we should take upon reaching that state. In CMU's implementation, the utility in each state is the calculated as the cost it will take to reach a goal. The Bellman equation sums up the reward of the current state as well as the maximum of the expected future rewards multiplied by a discounting factor (gamma). Gamma helps us to discount future rewards – the intuition can be explained by the fact that a dollar today is worth more than a dollar we'd get a month from now, because present rewards should be weighted more than future ones. This discounting value is indicative of our domain knowledge and will vary from problem to problem. Since the Bellman equation includes a maximization function, we cannot solve it with a system of linear equations, and therefore must run it iteratively until we converge. Convergence is defined by the lack of change in utility value of any state beyond that of a certain value of precision. Once the algorithm has converged, we will have a policy π that we can use to guide an agent through the MDP. We begin our exploration of the smaller gridworld with a vanilla run-through, the result of which are seen below.



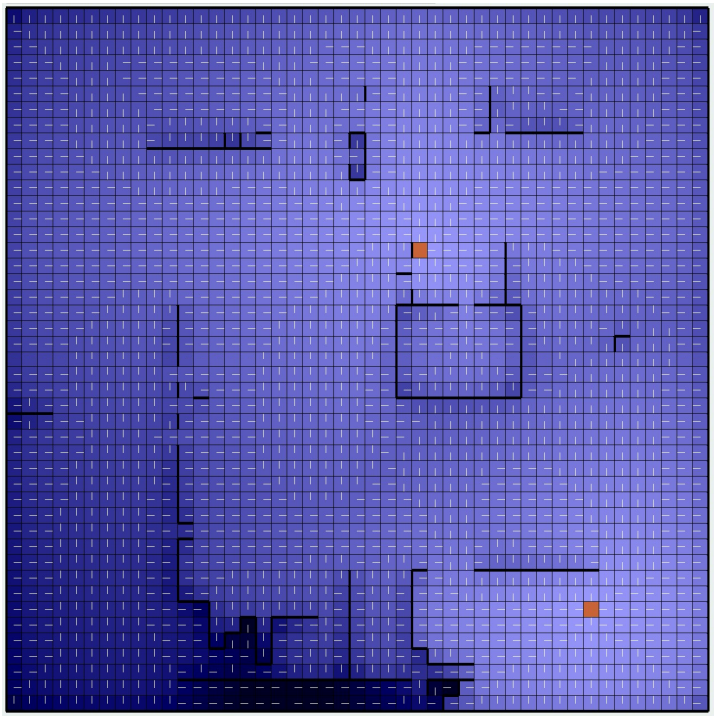
Above you can see the original gridworld on the left, which has many walls and tight passageways between rooms, resembling a manmade building. On the right, we see the results of our vanilla run of value iteration, which converged after 118 steps in 435 milliseconds. The relatively high wall cost caused the resulting policy to tend to avoid walls where it could, most notably in the 6x3 rectangular room in the center. The highest cost (lowest utility) state had a cost of 142. This can be attributed to the state's relative distance away from the goal state (represented in orange) as well as the high number of walls along the policy's implied path to the goal. Now that we've established our baseline expectations, let's proceed to manipulating two important hyperparameters: PJOG and precision. PJOG is the probability of error in taking an action – if PJOG is .3, we have a 30% chance of making an unintended action. If our potential actions are up, down, left, and right, we have a 10% chance each of going up, down, or left, and a 70% chance of going right if our intended action is to go right. PJOG therefore represents our domain knowledge of how stochastic our environment is. Precision is the threshold value at which convergence is reached. A smaller number for precision usually means larger number of iterations and a resulting policy that is closer to our optimal policy.



Value Iteration (10x10) – PJOG Variation		
PJOG	Steps	Time(ms)
.1	47	64
.3	118	135
.5	278	353
.7	2788	3556

First, I ran value iteration with PJOG values of .1, .3, .5, and .7 on our 10x10 gridworld. You can see the times and steps taken for each of those above. The leftmost map above displays the resultant policy for our 10x10 gridworld with a PJOG value of .7 – it displays no attempt to get away from the walls, only a focus on getting to the goal. This is because the stochasticity is so high that value iteration recognizes we cannot achieve fine tuned control of our agent – much like a drunk person stumbling home, the focus is on getting to the goal, not doing so elegantly. I've omitted the graphs and visualizations for precision because it offered little in terms of significant results. The lower precision became, the more time it took, but there were diminishing returns for changes to the final policy. The default precision of .001 served our purposes perfectly well, and further meddling was not needed.

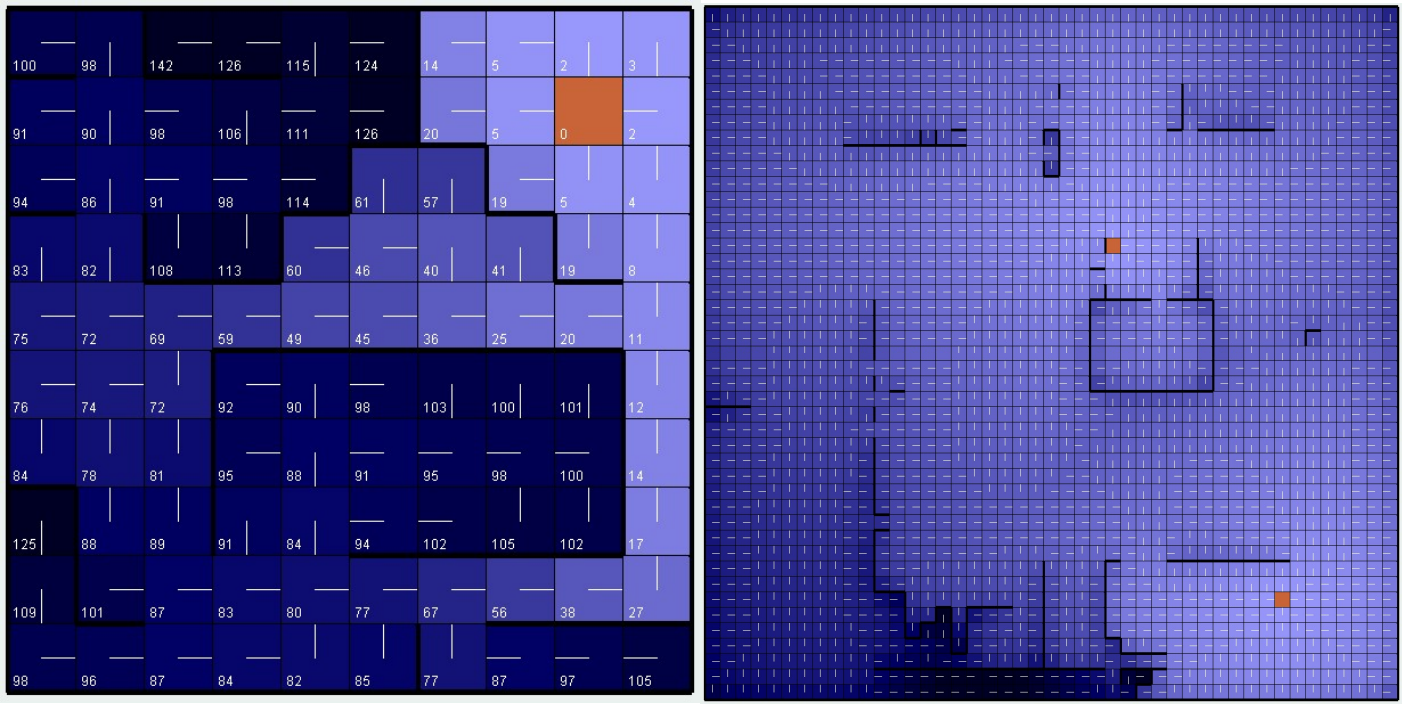
I then replicated our experiments with the 45x45 gridworld, which is more open with relatively fewer walls. Our findings were neatly replicated here, except of course the far higher runtime required to calculate the many more iterations. We needed more iterations to fully propagate the rewards from our goal states to the rest of the states, which in turn needed to be fine tuned with further iterations. When I attempted to run value iteration on the 45x45 gridworld with a PJOG value of .7 and the standard precision of .001, the algorithm ran for 397.788 seconds (6.62 minutes) before convergence – a stark contrast to the 3 second convergence of the same experiment on the 10x10 gridworld. To add a baseline value for the 45x45 gridworld, running the same setup with a PJOG of .3 took just 13.743 seconds. This large increase in runtime is definitely due to the sheer difficulty of finding an ideal policy within such a stochastic, random world. If the agent's movements are so random that it does the wrong action 70% of the time, it is much more difficult to tell it what to do with such a degree of certainty that the



change in utility value from one iteration to the next is less than the precision threshold. Increasing the precision threshold (thereby making it easier to attain) had the effect of decreasing runtime, further bolstering my hypothesis. A potential confounding factor, however, is the fact that I added two goals to this gridworld with the intent of seeing how and where each goal was prioritized. As you can see in the picture, there is a line almost equidistant from the two goals where the target goal changes.

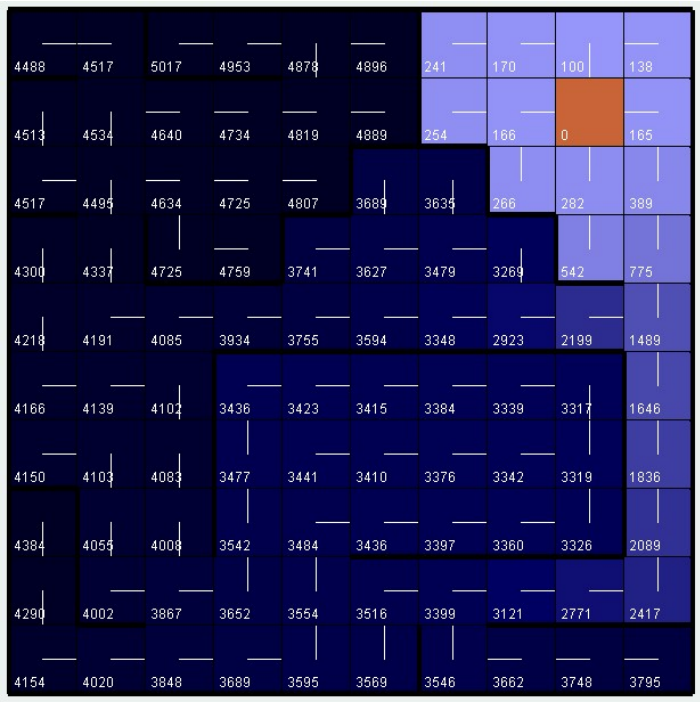
Policy Iteration:

As we mentioned above, the main cost of value iteration comes from its nonlinearity – since we have a maximization function, can't simply solve the equation with a system of linear equations. However, policy iteration solves this problem by generating a policy before attempting to solve the equation, this eliminating the need for a maximization. Therefore, we can turn our utility update equation into a system of linear equations that we can quickly solve with well-studied methods. Once we solve the equation, we update our policy and repeat until convergence. Other than this point, policy iteration is similar to value iteration, sharing some elements like future reward discounting. However, though the overall runtime should be faster, each individual step for policy iteration will be slower, taking bigger jumps towards the optimal policy. This is because we're calculating a full system of linear equations in each step. Another thing to keep in mind is that instead of combing through the search space in policy iteration, we're actually navigating through the policy space. Now that we've explained policy iteration, let's jump into our standard vanilla run.



Above are the results of running policy iteration on our 10x10 gridworld (left) and our 45x45 gridworld (right). Policy iteration converged with far fewer iterations (6 and 14, respectively), as expected. Our 10x10 run took 401 milliseconds, which was on par with the same setup for value iteration. However, our 45x45 run took 37.217 seconds, which was almost triple what it took for value iteration to do the same. This is quite a surprising find, as policy iteration is supposed to be quicker than value iteration. However, this discrepancy may be due to the fact that the resulting system of linear equations for the 45x45 gridworld is quite large, and therefore took my system a long time to run. I was not able to find any literature online on this matter, so I will have to leave a more precise explanation to another report. That said, both value and policy iteration delivered very similar resultant policies, with minor deviations. A closer analysis of the policies delivered by both value and policy iteration revealed that the ways in which they were different were not big changes – the utility of going one way versus the other changed only slightly, leading us to believe that if we had let the algorithms run for longer, they would have converged into a singular policy that was consistent with both algorithms.

For CMU's implementation of policy iteration, there are four hyperparameters that I could have varied: maximum value, iterations, PJOG, and precision. I did not see a point in limiting the maximum value for utility as our default implementation never seemed to get even close to the default maximum value. Changing iteration cap also seemed pointless, because policy iteration did not come close to the default cap either – I suspect that both iteration cap and maximum value for utility were only implemented to cut short algorithm runs that did not converge. Precision didn't seem to be as important for policy iteration as it was for value iteration, because we use precision to determine the minuteness of the change in utility value to determine convergence, but convergence is determined by lack of change in policy in policy iteration. Therefore, precision shouldn't have a significant effect on the outcome at all. With this in mind, I ran a quick experiment on PJOG, alternating between values of .1, .3, .5, and .7 as mentioned in the last segment. The results of this experiment are below.



Policy Iteration (10x10) – PJOG Variation		
PJOG	Steps	Time(ms)
.1	7	535
.3	6	410
.5	6	524
.7	3	128

The gridworld above is the 10x10 environment run with a PJOG of .7. Upon first glance, the results of this experiment seem to be very similar to the same experiment done with value iteration. The same abandonment of wall-aversion is clearly shown – for the same reasons explained in value iteration, policy iteration just focuses on getting to the goal, as shown in the gridworld above. However, the real nugget of information is seen when looking at the time taken for runs. Policy iteration takes less time when PJOG is very high – in fact, it takes three times less time to run when PJOG is .7 than when PJOG is .3. This is again due to the fact that we value simple directions (and by extension, policies) when our world is more stochastic, because we may not be able to carry out more complicated policies. However, because we converged more quickly, we see a large benefit in using policy iteration over value iteration for highly stochastic MDPs. The specific policy matters less when stochasticity is high, and so policy will converge much faster than the utility value that value iteration uses. Indeed, we see that time taken increases with PJOG in value iteration but is inversely proportional to PJOG in policy iteration.

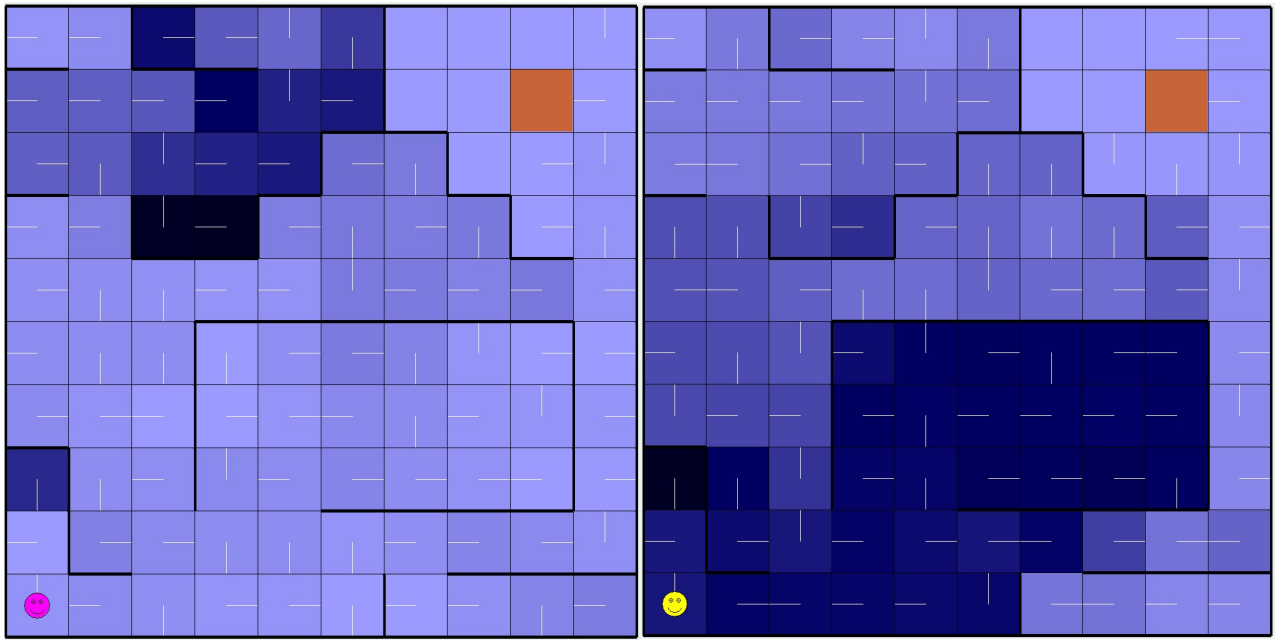
Q-Learning

While value iteration and policy iteration are all well and good, what happens if we don't have access to the transition function? If we cannot know the underlying probability distribution of our stochastic environment, we cannot calculate a policy without some exploration. To explore our MDP and overcome the lack of knowledge about a transition function, we turn to reinforcement learning techniques, specifically Q-learning. Q-learning starts an agent into an arbitrarily picked location in the MDP. As the agent explores, the Q-learning algorithm updates its Q function, which approximates the

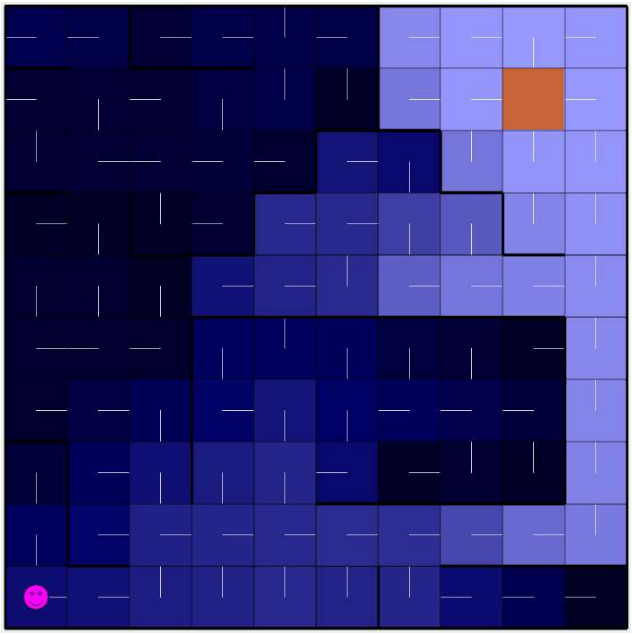
expected utility of each state and action. The Q-learner also implicitly approximates the MDP's underlying transition function as it explores the MDP. The following graphic of the Q-learning update rule was taken from Wikipedia's page on Q-learning:

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

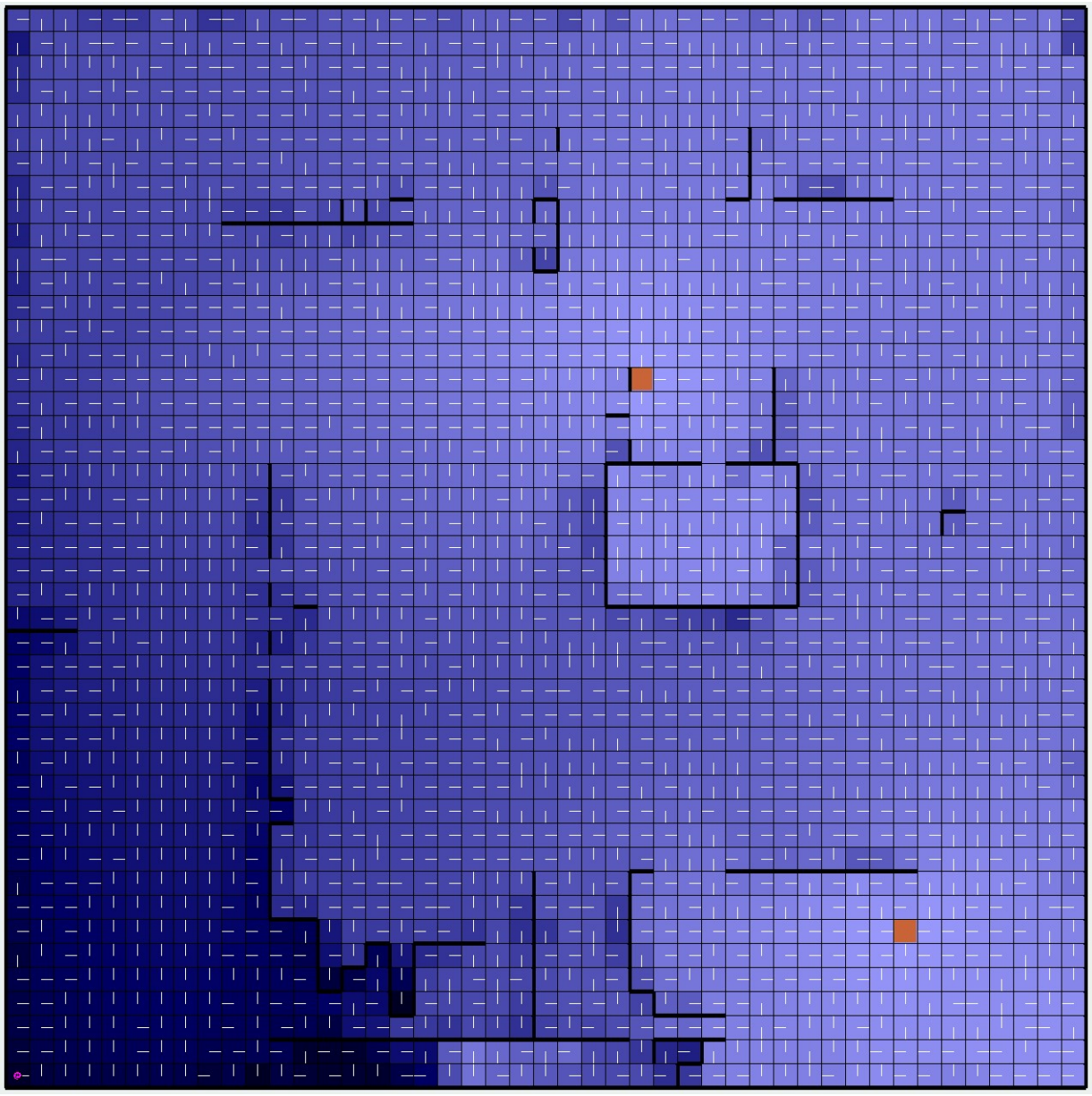
As you can see, the format is similar to that of the value and policy iteration equations – there's a discount factor for future rewards, as well as the current reward and an estimate of future reward. In addition, Q-learning utilizes a learning rate to weight the change that new information will have on the Q-function. While exploring, the agent differs from the Q function's suggested action with probability epsilon. This allows further exploration of the implications of various actions in similar states instead of strict adherence to a policy after just one exploration of a state. Now that we've laid the groundwork, let's jump into our vanilla Q-learning run.



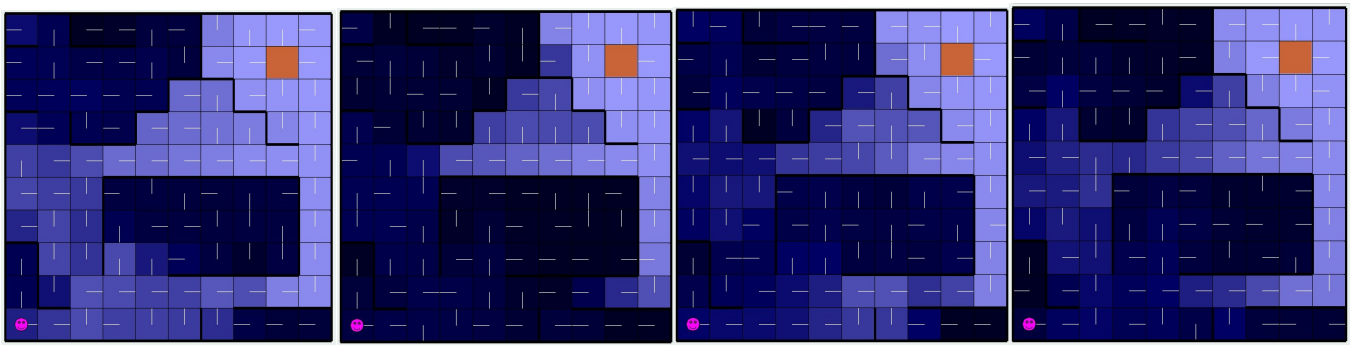
Starting with the top left image in going in clockwise order, the images are of the Q-learner's Q-function results for the 10x10 gridworld after 10 episodes, 100 episodes, and 10000 episodes. As you can see, the learner's Q-function gives better and better directions with successive iterations. I was not able to get a time for each run because CMU's application did not have that functionality, but the Q-function was updated with blazing speed – far faster than the time it took to calculate value or policy iteration. However, comparing Q-learning to value or policy iteration is a bit unfair, as they have more information and therefore spend the time to generate precise results, while Q-learning quickly gives us its best approximation. As we can see on the right, even after 10,000 cycles a few of the instructions given to us by the Q-function have us running into walls or forming loops. With our 45x45 gridworld, we actually had difficulty getting the Q-learner to see enough of the world with lower iterations, forcing me to increase the iteration cap to enable sufficient time for exploration. Though it took far longer



for the 45x45, the Q-learner did just as well with its resulting q-function as it did for the 10x10 in the end, as pictured below. In both, there are still many sub-optimal decisions made, but these could be resolved with more time and iterations.

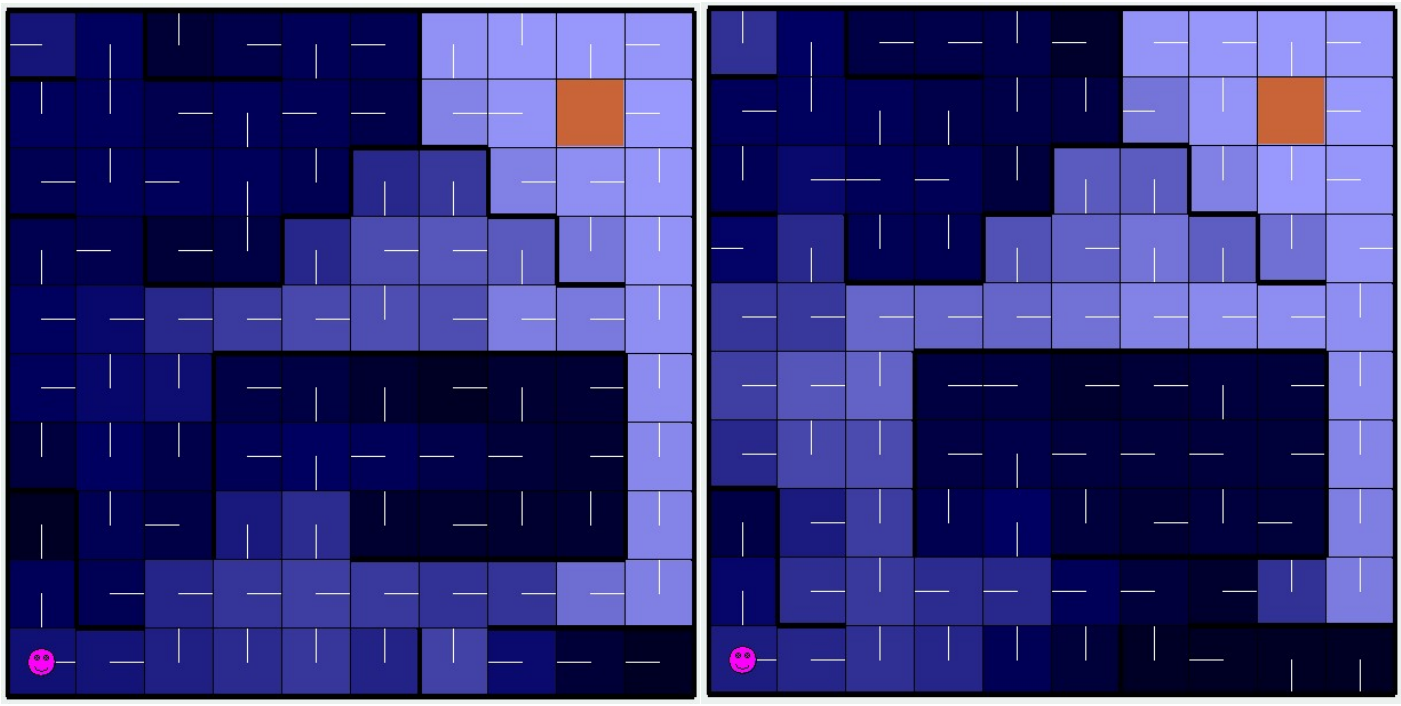


Let us then dive straight into our experiments on learning rate and epsilon. First, I varied epsilon, predicting that as epsilon increased, so too will the quality of the resulting Q-function and the amount of exploration done. Varying epsilon shows us the balance between exploration and exploitation. When epsilon is high, we explore more, and when epsilon is low, we prefer to exploit the MDP through the paths we've already learned.



Above, from left to right, are the resulting Q-function outputs with an epsilon function of .1, .3, .5, and .7. As you can see, the movements become more scattered as epsilon increases. All the four Q-learning run-throughs we're capped at 1000 episodes, but we know that if we let them run for as long as it needed, we would see that in general, a higher epsilon resulted in a better final function, at the expense of time. An epsilon greedy (low epsilon) approach means we take the first path we come across with little exploration but choosing to focus on exploration gives us a more robust policy at the end that we might have missed had we gone an epsilon greedy route.

Finally, let's vary learning rate. Learning rate increases the weight we give to new information, which is great if the new information is better than what we've already learned. When we're in an MDP where exploration is preferable to exploitation, we should keep learning rate high, else it should be kept low. This again exemplifies the need for superior domain knowledge when conducting these experiments for practical purposes. Below, I've ran Q-learning with a learning rate of .3 (on the left) and of .7 (on the right).



From this experiment, we can see that the smaller gridworld performs better with a higher gridworld. I verified these results with the same findings on our 45x45 gridworld. The comments on exploration vs exploitation from the epsilon segment above still stand here, but it seems that epsilon has a higher impact on the resulting Q-function than learning rate does.

Conclusion

Overall, both value and policy iteration were able to give us appropriate policies with which to navigate the MDP. We were able to gain an intuition for how varying precision and PJOG affected their outcomes as well. Furthermore, we were able to successfully use Q-learning to approximate the optimal policy with it's Q-function when we were deprived of information about the transition function. I'm glad we were able to see the relative strengths and weaknesses of all these algorithms, and I hope to continue learning about the inner workings of and the future applications of these algorithms in due time. For now, I am content with what we've learned in these past eight pages, and I hope the reader is as well.