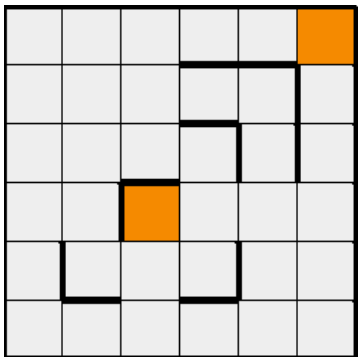
Markov Decision Processes Assignment

Yuanheng Wang 902905501

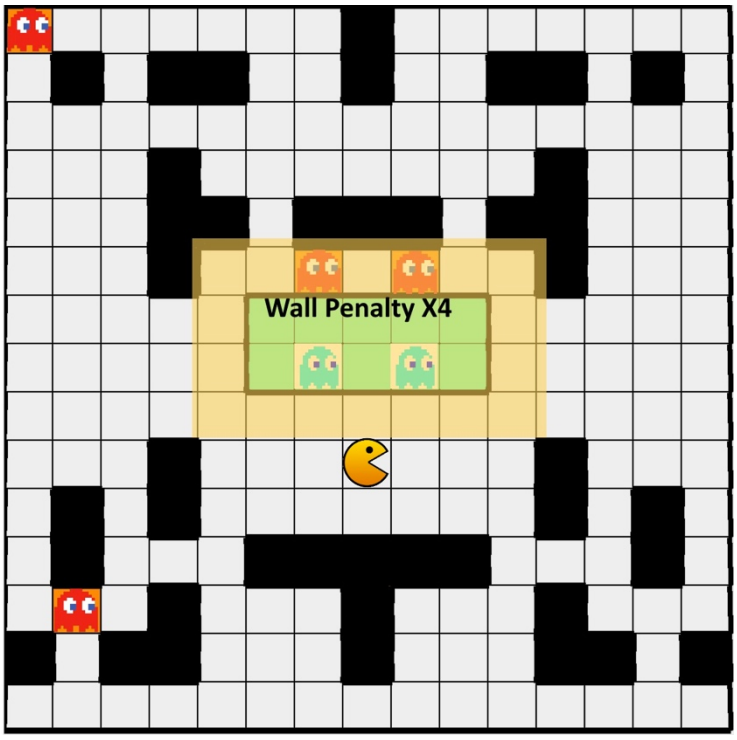
For this assignment, I implemented CMU Reinforcement Learning Simulator, as it is a really easy to use and is very good at visualizing the MDP problems.

My small MDP is simply a maze of size 6x6 with two goals. Staying in the same square incurs a -1 penalty, and hitting the wall (black lines) will incur a -50 penalty. The goal states are the orange states. The top-right corner goal is fairly free of obstacles, but is far away from most of the grids. On the other hand, the other goal is closer to the center, but it is surrounded by many walls, which can incur potential loss if actions start getting random.

My large MDP is a map of size 15x15, and is a map inspired by Pac-man game. In this problem, however, scenario is simplified. Instead of asking Pac-man to eat all the pac-dots that exist, the goal state is to eat one of the enemies after Pac-man consumes the flashing dots and all ghosts become vulnerable. As we can see here, there are four red-Blinky ghosts in this map: one on the top-left corner, which is the furthest point on the map, there is one near the bottom-left, which is closer to the center of map, but surrounded by a few walls, which is a little harder to get. What is worth mentioning is that there are two ghosts right next to the spawning pool: If pac-man decides to go and eat those ghosts, it is possible it got attacked by the newly-born ghosts. I simulated this situation by increasing the wall penalty four times around the spawning pool area. Finishing off one of the ghosts indicate the victory of Pac-man.



Small maze, above.
Large Pac-man map, right.

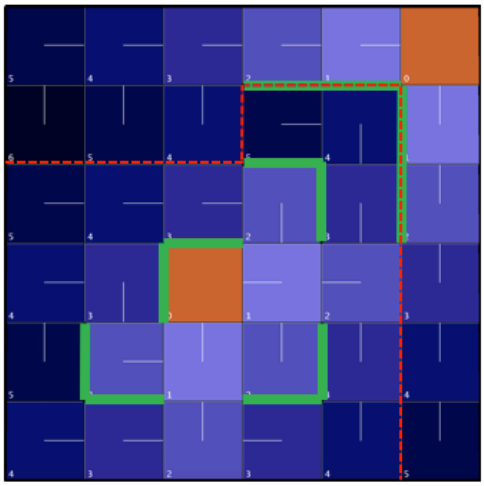
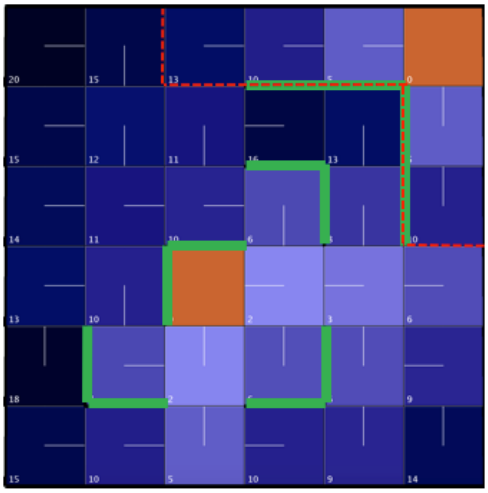
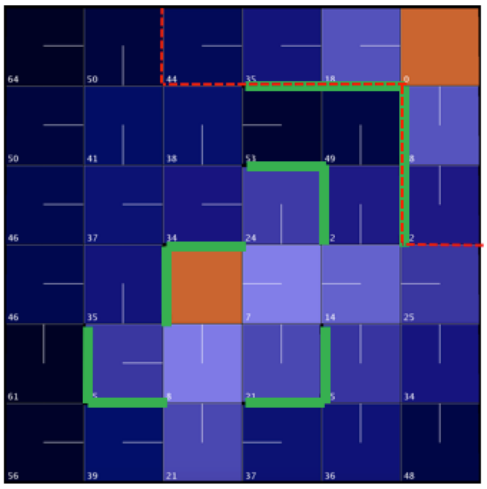
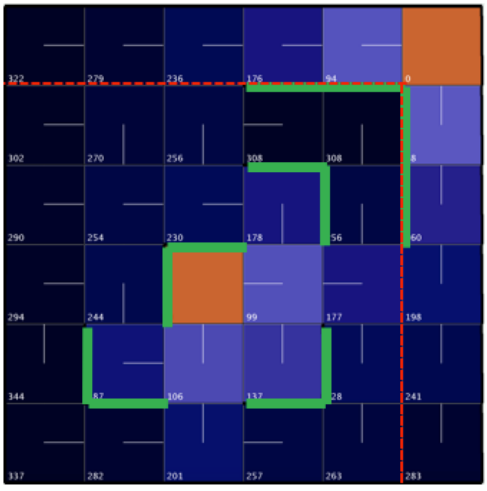


PJOG is used to model the noise in the environment. This value represents the probability of the agent not taking the right action given the command. For example, if agent wants to take action UP, then agent take that action with probability (1-PJOG) and it takes one of remaining action with probability PJOG/3. PJOG value is varied during the run for different algorithms.

Small Maze:

Value Iteration:

I run each problem with varying values for PJOG until it converged. After writing down how long it took down to converge in terms of both wall clock time as well as number of iterations, I examined the policy it generates. In order to make the wall clearer, I changed the color of all the walls in the middle of the map to green (boundaries of the map serve the same functionality as walls with same value of penalty). To better visualize the difference, I partitioned the map into different sections with dashed red lines. Each section corresponds to one goal state and any square in that section will be lead to that goal state eventually.

<table>
<tr><th colspan="2">Small Maze – Value Iteration</th></tr>
<tr><td> 
PJOG = 0</td><td> 
PJOG = 0.1</td></tr>
<tr><td> 
PJOG = 0.3</td><td> 
PJOG = 0.6</td></tr>
</table>

From the graph above, we noticed a few things. When PJOG is 0, which means the environment is deterministic, the policy guides the agent at any state to the goal state with shortest Manhattan distance. After introducing randomness into the system, however, fewer states would lead the agent to top-right goal state. Notice that the number displayed at the left-bottom corner of each state is actually the expected penalty that state will get given from that state and onward if the agent act optimally. That means, the larger the number is, the more penalty it will get. If the goal state has a certain reward value of R, then the utility of each state is U = R – penalty. The darker grid color indicates a higher penalty it will receive.

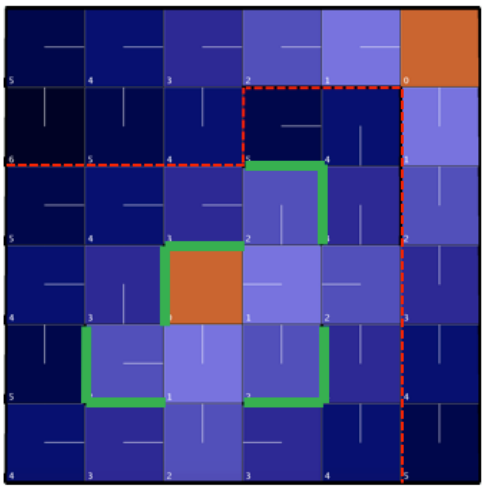
From the graph above, we find that changing the PJGO doesn't change the partition as long as it is not zero and below 0.5. When there is uncertainty, agent seems to prefer to move to the middle goal state. This makes sense because although the goal state in the middle of the map seems to be surrounded by many different shapes of walls, it is easier for the agent to reach because there are no adjacent walls next to each other. On the other hand, in order to get to the top-right goal state, no matter which direction the agent tries to approach, it has to pass through two consecutive walls without hitting any of them to minimize the penalty. This tasks becomes hard once the system becomes nondeterministic. The agent would rather choose to go around to the further goal state since the penalty of each extra step is far lower than the penalty of touching a wall.

One thing I noticed and thought interesting was once the PJGO gets above 0.5, the policy would look like the fourth picture of the graph, where it tricks the user to think that the agent at bottom right and top left would be redirected to the top right goal state (I tested multiple values that are greater than 0.5 and it shows me the same result). Nonetheless, after second thought, I think the reason system does that is because the system becomes a highly stochastic process with a high PJGO value that telling agent to go up would actually make it more likely to see the agent perform other actions. After counting all the probability and discount factor into consideration, pointing "wrong" direction will likely end the agent in the right goal state.

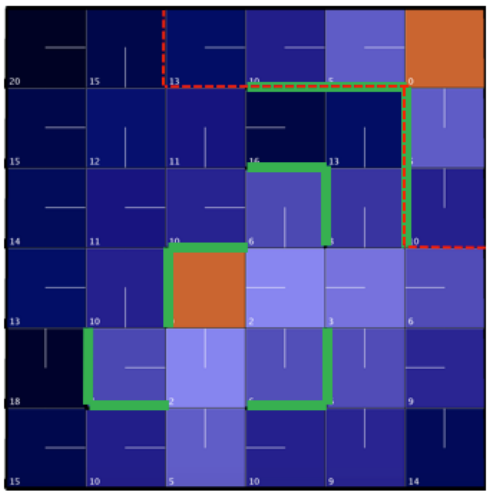
Policy Iteration:
Below is the graph I got by performing policy iterations algorithm.

By comparing the graph of value iteration and policy iteration, I find that the policy as well as the actually value for each state is exactly the same. This is not surprising as they both will find the optimal policy if there exists one. The only difference is that policy iteration actually doesn't care about the true value of each state but instead directly finds the true policy. For this case, they both render the same value for each state, however. One thing worth further investigation is that whether policy iteration always guarantees generating the true value in the process of looking for optimal policy.

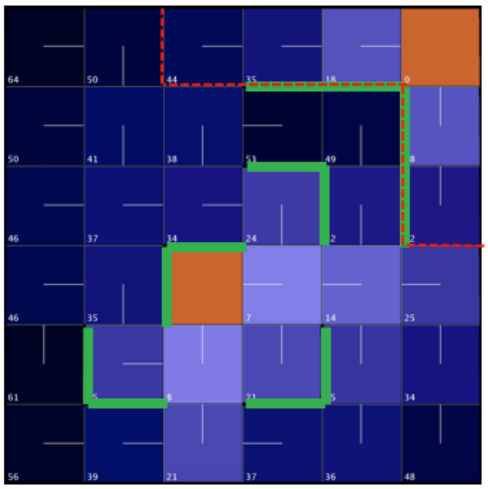
Small Maze – Policy Iteration



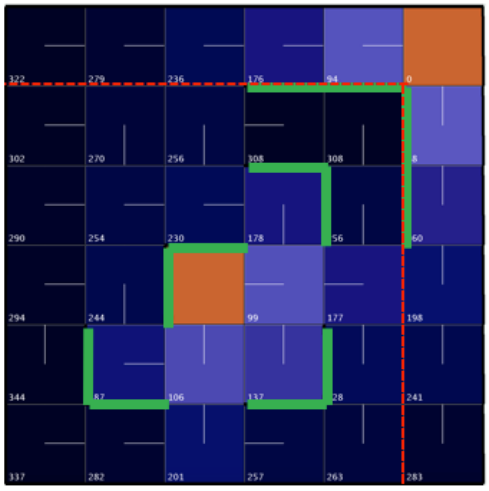
PJOG = 0



PJOG = 0.1



PJOG = 0.3



PJOG = 0.6

Difference between value Iteration and policy Iteration:

		PJIO = 0	PJIO = 0.1	PJIO = 0.3	PJIO = 0.6
Value Iteration	Steps taken to converge	7	21	43	168
	Time taken to converge	1ms	5ms	5ms	19ms
Policy Iteration	Steps taken to converge	6	5	5	4
	Time taken to converge	6ms	9ms	8ms	18ms

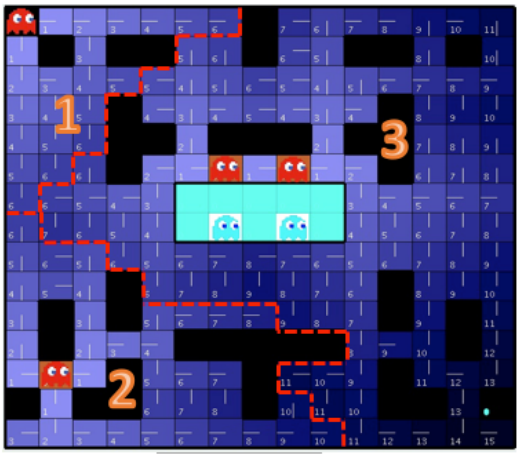
From the table we can see both converge fairly fast in terms of wall clock time. But as uncertainty increases, the steps value iteration need to take grows exponentially. This happens because policy iteration only needs to adjust actions at each state instead of calculating the true value, therefore fewer steps is needed. But as PJIO grows, the time each iteration takes also goes up.

Large Pac-man Map:

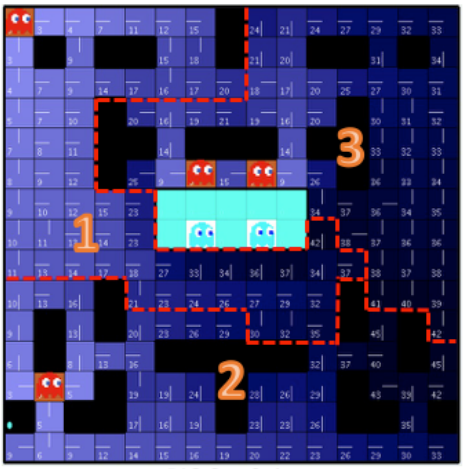
Value Iteration:

As I did it before, I ran each problem with varying values for PJOG until it converged. After writing down how long it took down to converge in terms of both wall clock time as well as number of iterations, I examined the policy it generates. For graph when PJOG = 0.3 and PJOG = 0.6, since the grid becomes really dark, I switched the color of walls to grey so it's easier to distinguish walls with states. For each partition, I overlaid a number above simply for reference purpose.

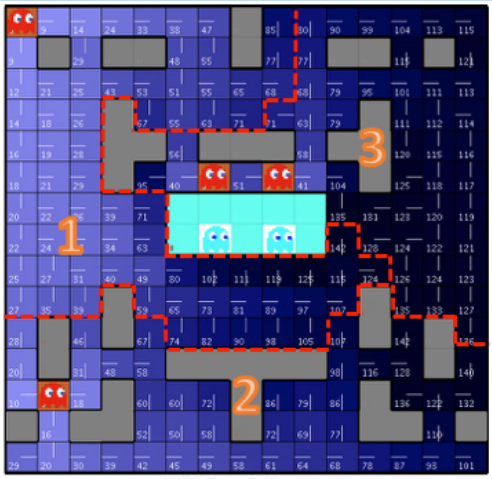
Large Pac-man Map – Value Iteration



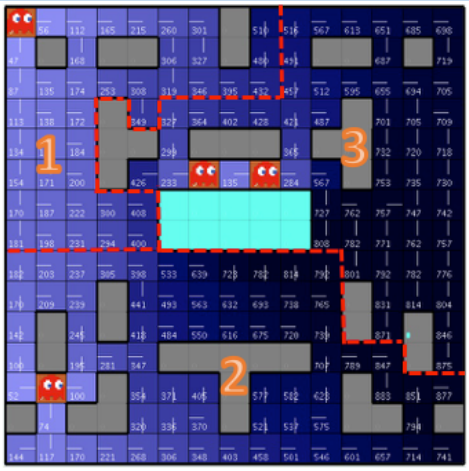
PJOG = 0



PJOG = 0.1



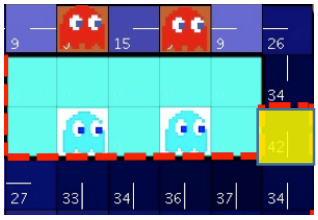
PJOG = 0.3



PJOG = 0.6

And I found this large map indeed showed some really interesting patterns. Just a reminder again, those walls next to the sky-blue spawning pools incur very high penalties. Meanwhile, there are two goals states next to the pool, which serves as incentive for Pac-man to get close that area. Thus, when PJOG is zero, the partition that two ghosts are at covers almost 2/3 of the

map since their locations are relatively near the center. However, as soon as PJOG is increased to 0.1, things get interesting and the division of the map dramatically changes.
The first notable change is that partition 1 covers much more space than it does when PJOG = 0. Although it's one of the most isolated point on the map, its partition extends to as far as to the right of the spawning pool. Take a look to the closer screenshot to the right, we found the agent at yellow state, though only 4 steps away from the goal state, would rather not risk the high penalty and instead travel 20 steps to the top left corner goal state since the path is fairly clear of obstacles. Another risk aversion action we can see is that all the states underneath the spawning pool would guide the agent to take a DOWN action in order to get away from the high penalty wall, just as what was described in the Udacity MDP lecture. For those split in the 3rd partition, the typical strategy is to go up from the right side of the wall, then loop back down to the ghosts.



The second and third scenario doesn't differ too much from each other as partition 1 and 2 controls majority of the map space. However, when PJOG passes the 0.5 threshold, which indicates the system get into the turmoil stage (during the game play, more ferocious ghosts are spawned, player is drunk and can't well control the move of Pac-man), the policy will tell the agent to try to stay off the walls and get close to any of the goal states. Since ghost at stage 1 is relatively far away, under the uncertainty of 0.6, it becomes unlikely the agent can get there, hence result in partition 2 grows as the system decides try bumping into some low penalty walls then traveling for too far.

Policy Iteration:

After running policy iteration, I realize the result is exactly the same in terms of utility and policy. Therefore, assumption that policy iteration provides true utility is verified.

Difference between value Iteration and policy Iteration:

		PJIO = 0	PJIO = 0.1	PJIO = 0.3	PJIO = 0.6
Value Iteration	Steps taken to converge	16	50	152	475
	Time taken to converge	18ms	61ms	175ms	551ms
Policy Iteration	Steps taken to converge	11	11	10	5
	Time taken to converge	1448ms	380ms	284ms	548ms

Since the problem space is much more complicated, taking more time to converge than the small maze is not surprising. When comparing the results under same PJIO value, we find that Policy iteration takes far fewer iterations then Value iterations. This makes sense because Policy Iteration will terminate when the policy ceases to change. That means, once the order of the actions is clear, Policy iteration stops. This is unlike Value Iteration, which continues iterating until the individual state updates fall below a certain threshold. Thus, Policy Iteration should take fewer iterations. However, because it needs to evaluate all of the Markov chain each iteration for each beginning state, each iteration in Policy Iteration will take longer. Value

Iteration only needs to evaluate its neighbors, thus explaining its lower wall clock time per iteration.

Another interesting phenomena is that for Policy Iteration, the more undeterministic the environment is, the fewer steps it takes to converge. And when PJOG is equal to zero, it takes a long time to converge, which is counterintuitive. One of the reasons might be the structure of the map where two goal states are being surrounded by extra penalty walls. After changing the walls to regular penalty values, the iteration takes 995ms, which is improving but is still higher than the other cases. I believe this is a case worth of further investigation.

I also played around with different values of precision during the running of Value Iteration on the large Pac-man map. This is the threshold when value difference between two iterations is smaller than that number, the algorithm stops. I tried some different numbers, and for this particular problem, it seems like the maximum value I can set is 0.1, and anything bigger than it will not give me the true utility of the state. There was some minor improvement in terms of time elapse but I assume if the problem gets even more complicated, the benefit of time saving would be obvious with picking an appropriate precision value.

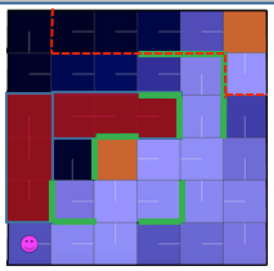
In summary, value iteration seems to be more efficient regarding wall clock times. So if the cost of iteration is cheap, I would choose Value Iteration. However, if the experiment is expensive to repeat, I would choose Policy Iteration as it requires fewer iterations to achieve the same goal.

Comparing the small maze and large Pac-man map, both seems to take roughly 30-35 times longer for the large map and the size of the larger map is 6.25 times larger than the small map. This seems like an exponential growth regarding size vs time. Stepwise, large map needs 3X more steps than small maze for each PJOG value for Value Iteration algorithm, but only 2X more steps needed for Policy Iteration Algorithm. This is understandable as more states in the system, more calculation is needed to compute the value of each single state for each single iterations and more values means more adjustment to be made. But for the policy, since it's always determining the optimal action among UP, DOWN, LEFT and RIGHT, the problem space is limited and number of iterations shouldn't be

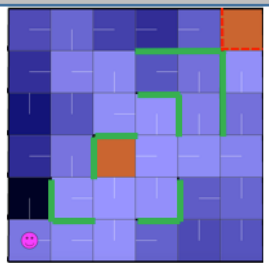
Q-Learning

I ran Q-learning algorithm on the two MDP problems above, and recorded the performance of the algorithm on each map with various PJOG, iteration number, Learning rate and epsilon. Below are the results I got. Notice that I've shaded the area where the agent will be trapped and never be able to reach the goal state.

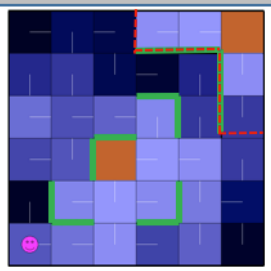
Small maze – Q-learning



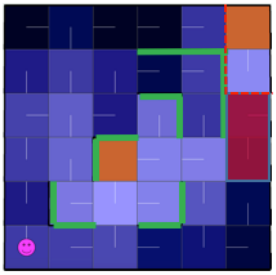
PJOG = 0.1 Epsilon = 0.1 Learning rate: 0.7
Iterations: 100



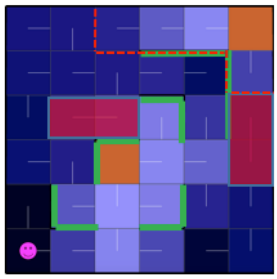
PJOG = 0.1 Epsilon = 0.1 Learning rate: 0.7
Iterations: 500



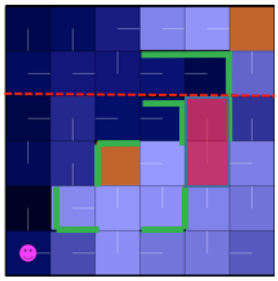
PJOG = 0.1 Epsilon = 0.1 Learning rate: 0.7
Iterations: 1000



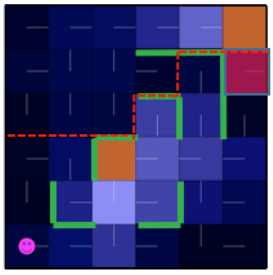
PJOG = 0.3 Epsilon = 0.1 Learning rate: 0.7
Iterations: 500



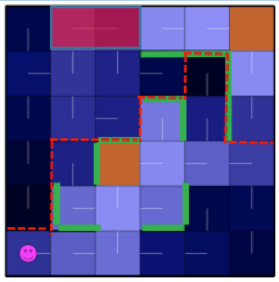
PJOG = 0.3 Epsilon = 0.3 Learning rate: 0.7
Iterations: 500



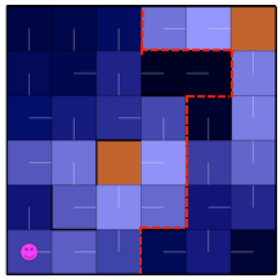
PJOG = 0.3 Epsilon = 0.6 Learning rate: 0.7
Iterations: 500



PJOG = 0.6 Epsilon = 0.1 Learning rate: 0.4
Iterations: 1000



PJOG = 0.6 Epsilon = 0.1 Learning rate: 0.7
Iterations: 1000



PJOG = 0.6 Epsilon = 0.1 Learning rate: 0.9
Iterations: 1000

Before diving into the analysis of the results and effect different parameters have on Q-learning, I found that Q-learning is a very time consuming algorithm. For small maps, with an iteration number of 1000, it always takes around 5 minutes. For large Pac-man map with an iteration run of 5000, it even took up to 30 minutes to conclude to the final result. Value Iteration and Policy Iteration do much better in terms of wall clock time. This was somehow expected as Q-learning is a model-free learning where it assumes agent has no domain knowledge at all (including transition matrix, reward, etc.), therefore for each iteration the Q-learner needs to update the estimated value of the previous state after the immediate award. Learning about the environment takes time, and the time spent on exploration can't be used on exploiting, therefore the time commitment is much more.

From the graph, we can find that one of the most significant factors that affect the quality of Q-learning is the number of iterations. While iteration number is 100, there are many dead-end states where the agent will never be guided to the goal state. By increasing the number of iterations, the generated policy will endeavor to eliminate those situations. As iteration number keeps growing, we can see that the partition gets closer to the true partition outcome we got

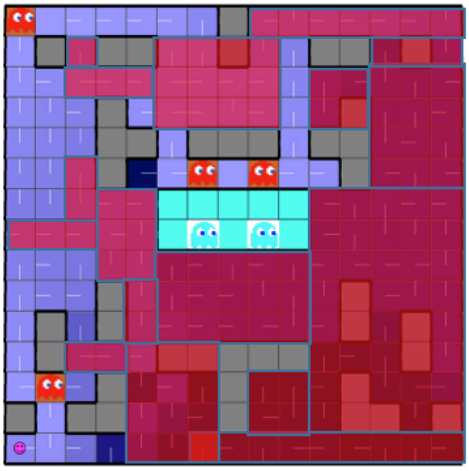
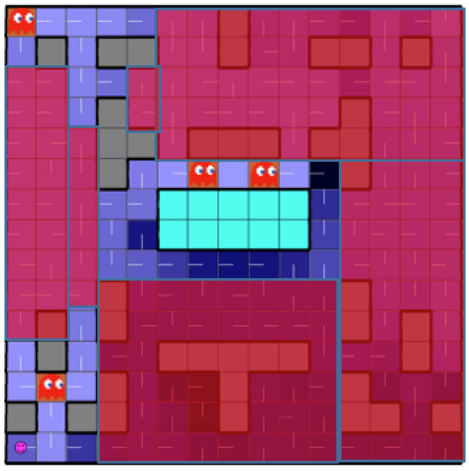
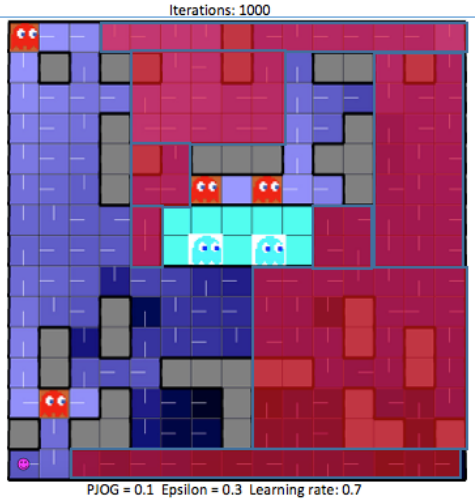
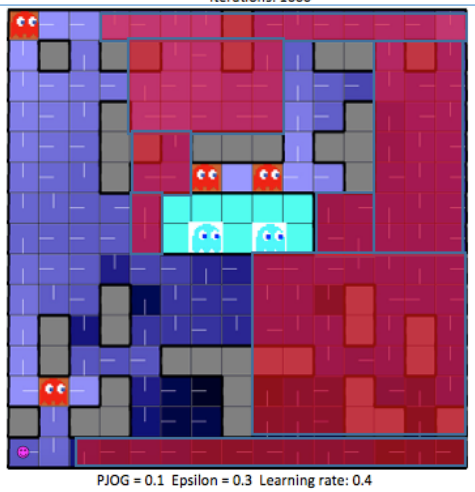
from Value Iteration and Policy Iteration and although not displayed on the graph above, the estimated value of each state gets closer to the true state value. This is the proof that as the learner understands the environment better, it will perform better.

Second thing to notice is that increasing the epsilon value appropriately can improve the accuracy. Epsilon represents the chance that Q-Learning chooses a random action, rather than the one it was planning to take. The idea behind this is similar to that of simulated annealing. By doing this kind of random exploration, it will reduce the chance of Q-learner gets stuck in a local optimal. We found that when we tune the epsilon from 0.1 to 0.3 in the second row, the partition gets closer to the true ones. However, after setting it up again to 0.6, we found it deviates from the true policy. This is because the iteration number is limited, and by exploring the problem space too much, it has less time to perform correct actions. In order to prove that, I reran the problem with 1000 iterations with epsilon value of 0.6. It turns out to be doing much better job.

I also played around with different learning rates and the results I got were sort of random. No matter what value I set, when the PJOG value was 0.6, it didn't perform as well as I thought. The most similar policy I got (in terms of the number of matching actions), however, was when the learning rate was 0.4. The reason that Q-learning didn't do well in this case might be that randomness was too high (0.6), and it became fairly difficult to capture the true story behind the scene. That is, what action will eventually lead to the goal. The reason that 0.4 learning rate does better is because it allows the agent to remember the knowledge of the maze better by updating the estimated state value little by little.

For the large Pac-man map, Q-learner performs miserably. After trying even 10,000 iterations, the policy it came up with was not even close to the optimal policy. A lot of "dead zones" are still being generated. Although we can find that increasing iteration numbers, increasing epsilon to a certain level and decreasing the learning rate a little bit does decrease the number of dead zones, which was expected. The poor performance is highly likely due to the complexity of the problem space, especially the structures of the walls and those high penalty walls. Since the map is large, it's also very hard for the Q-Learner to remember where each wall is located at. Thinking from a game play perspective, it's like asking Pac-man to take actions without knowing anything else except the adjacent squares that's surrounding him, not knowing where the Pac-dots are and where a ferocious ghost will jump out to attack.

Nonetheless, as we can see from the last two outcomes of the Pac-man map, we can see that the left side of the map, which is less occupied by walls, have a clear policy that can take agent to the goal state. For the high penalty area, we can see that Q-learner took the high penalty into consideration and has only a few actions that would take agents to that zone, thus resulting in a small partition. This justified the reasoning I had above why Q-learning doesn't do well in this MDP problem. In order to improve the Q-Learning performance for this problem, more iteration number is needed and the learning rate should be set lower in order to allow the agent to explore the entire space.

Large Pac-man Map – Value Iteration	
 PJOG = 0.1 Epsilon = 0.1 Learning rate: 0.7 Iterations: 1000	 PJOG = 0.1 Epsilon = 0.1 Learning rate: 0.7 Iterations: 1000
 PJOG = 0.1 Epsilon = 0.3 Learning rate: 0.7 Iterations: 5000	 PJOG = 0.1 Epsilon = 0.3 Learning rate: 0.4 Iterations: 5000

Summarization:

Value and Policy Iteration are relatively fast at finding the optimal policy. However, they require domain knowledge of a problem's state space and model. Compared to Policy Iteration, Value Iteration is faster in terms of wall clock time, but slower in terms of iterations. On the other hand, Q-Learning does not need access to the information to work. However, this means it will take huge amount of time to interleave learning and execution in order to capture the rules of the world. As problem complexity grows, the effort that Q-learner needs to input also skyrocket. The quality of its resulting model it is highly dependent on what parameters it is given. These parameters, such as epsilon, LR, and iteration count, would have to be determined based on one's domain knowledge of the problem for Q-Learning to be effective.