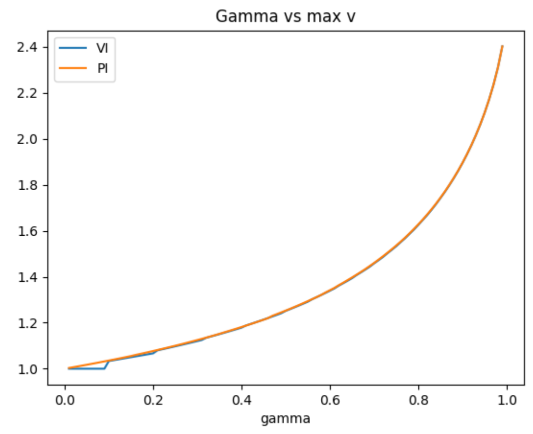
Markov Decision Processes

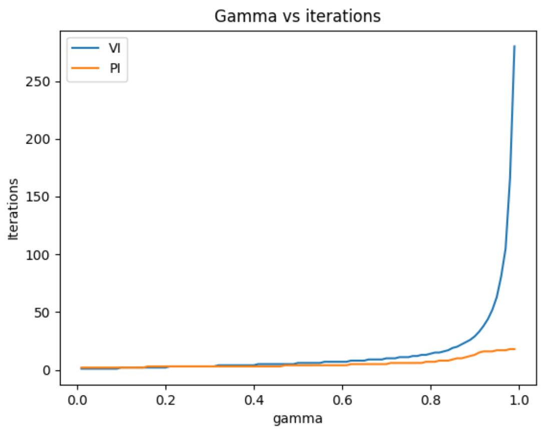
Cody Phrampus
cphrampus3@gatech.edu

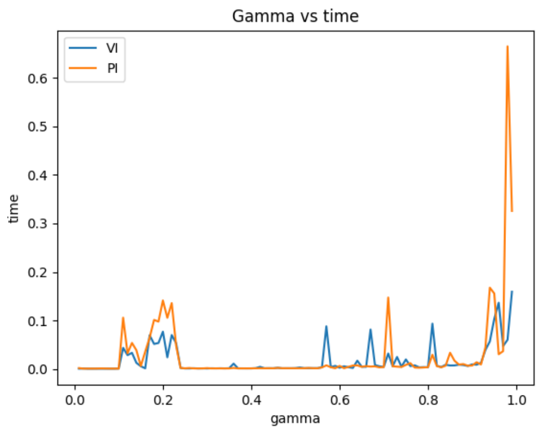
1 Problem 1 - Gridworld - Frozen lake

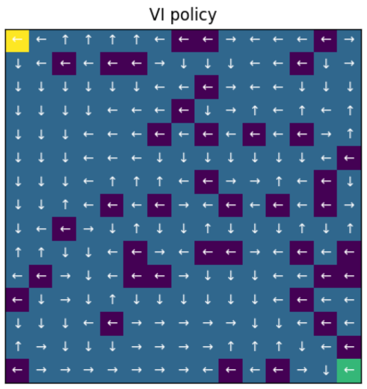
The frozen lake problem, pulled in from the environment in OpenAI gym, is a gridworld problem that consists of an agent trying to navigate over a frozen lake which has holes it can fall into (OpenAI). The problem introduces randomness to the movement via the ice being slippery, so there is only a ⅓ chance the agent moves in the intended direction with the other ⅔ causing the agent to move in a direction perpendicular to the one it intended, ⅓ for each side. This problem does not penalize movement, i.e., there is no negative reward per step, nor is there any negative reward for falling into a hole, the problem simply starts over, and there is only a single reward, 1 for reaching the goal state. A 15 by 15 grid was used for this problem.

1.1 VI/PI









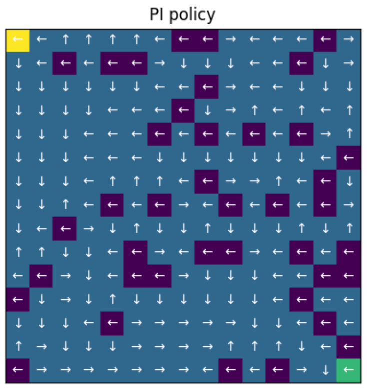
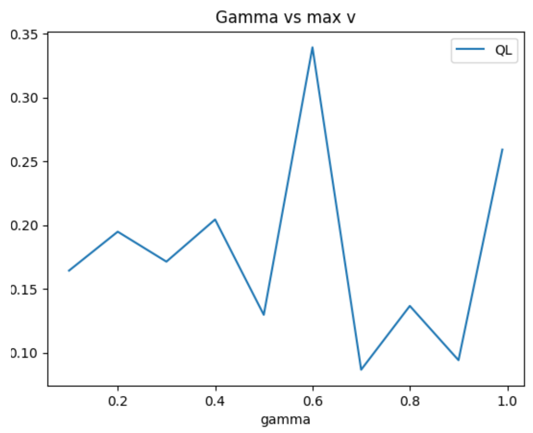
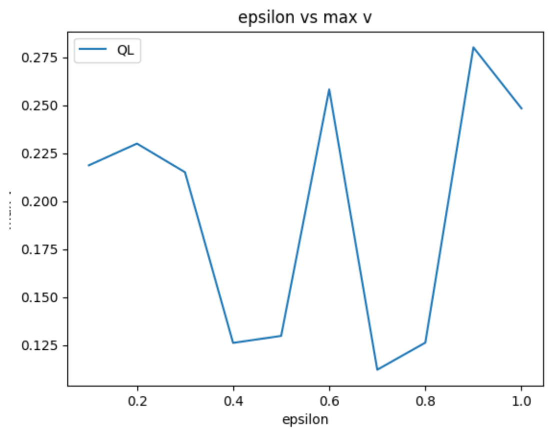


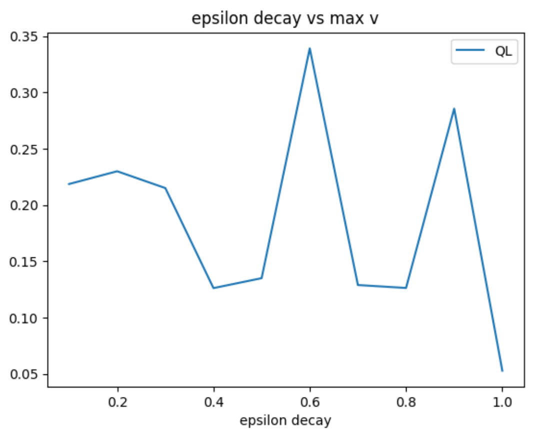
Figure 1, 2, 3, 4, 5—Gamma vs reward tuning (top left), Gamma vs iterations (top right), gamma vs time (middle) Policy for VI (bottom left), and Policy for PI (bottom right)

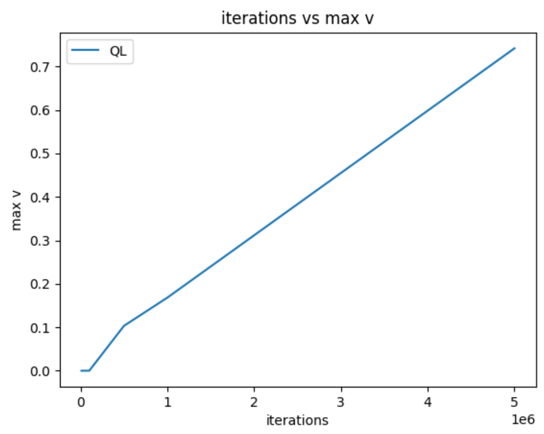
Value Iteration and Policy Iteration both performed as expected for this problem, which is to say well and fairly quickly. Both came up with the same policy, as can be seen in the visualizations. Value iteration, as expected, took many more iterations, over 250 compared to about 20, to reach the same optimal policy, but also did so more quickly. The fact that is is taking more iterations than Policy Iteration makes sense, the steps/iterations for Value Iteration are much smaller and can continue even after the optimal policy has been reached, due to small changes in the values that the algorithm will continue with even though they do not affect the overall policy. Policy Iteration takes much longer timewise, but in many fewer iterations, due to the fact that the steps it takes are larger, but also more costly and intensive to run. This is similar to MIMIC where PI would run more quickly if the value function had a high cost, but since it is cheap in this instance, VI was able to come out ahead. This was not the case in all instances, as can be seen above, there were some instances, namely around 0.6, 0.7, and 0.8, where PI actually ran faster. However, given how quickly both ran until 0.99, it is not a big consideration for this problem at this size. The convergence for both of these algorithms was checked automatically within the library code for a maximum change, epsilon, of 0.0001. For this requirement, the PolicyIterationModified version of the agent in the library was used, as the original does not allow for passing in an epsilon in this way. A couple of interesting points show up in the optimal policy, namely with respect to jumping into holes. We can see this with the hole towards the top left near the start and the bottom right near the goal. It seems confusing at first, as jumping into a hole ends the game and this is the best policy the algorithms could find. So what is going on? This has to do with the particular way that this problem is stochastic. Unlike the gridworld problem in the lecture, where the agent had an 80% chance of moving in the direction it chose, the agent here only has a ⅓ chance, with ⅓ also going to each perpendicular direction. This has an interesting effect similar to the agent in the lecture, the agent can choose a direction that, with absolute certainty, it will **not** move in. On the surface, this may not seem to help, the agent still randomly moves in one of 3 directions, so why does the fourth one matter? If we look at the top right example, a hole on the left and right and open squares to the top and bottom, we can see why this makes sense. The agent wants to move to the top or bottom, since that means getting closer to a reward and not being in a hole. In a deterministic world, the agent would simply move, say, down, but due to the stochastic nature of the problem, the agent would move down ⅓ of the time and left or right ⅔ of the time, which means falling into a hole. However, if the agent tries to go right, i.e., jump into the hole, there is a ⅓ chance to end up in the hole and ⅔ to end up going up or down to safety. Thus, it is, counterintuitively, better to try to jump into the hole. This process can be applied symmetrically to going left or up. This is only in the case of there being a risk of falling into a hole by being between multiple of them like this, if there is a hole in isolation, the best move is to also move away from it as to ensure never slipping in, as can be seen in a few places in the policy visualization.

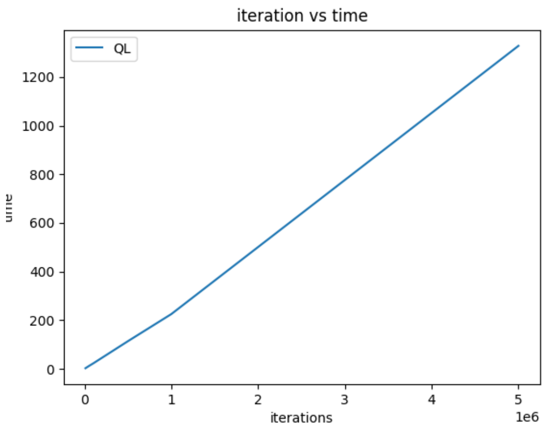
1.2 Q-Learning











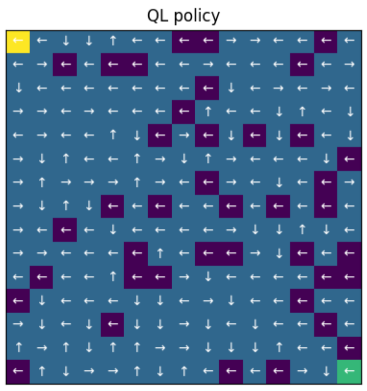


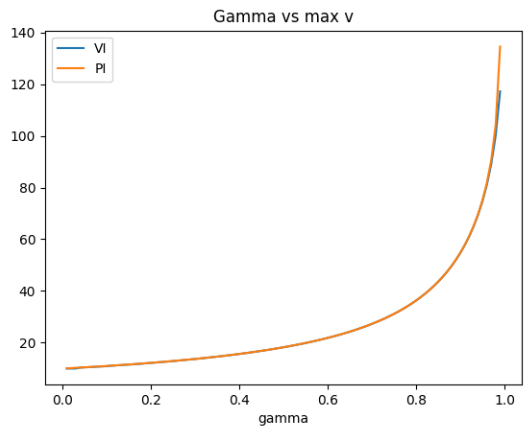
Figure 6, 7, 8, 9—Gamma vs reward tuning (top left), Epsilon vs reward (top right), epsilon decay vs reward (middle left), reward vs allowed iterations (middle right), runtime vs allowed iterations (bottom left), and Policy for QL (bottom right)

Q-learning had a lot of trouble with this problem, as can be seen by how poorly it scored even after 1 million iterations, where it was capped for time and tuning. With this cap, QL is only making it to a reward of about 0.2-0.3, compared to the 2.4 of VI and PI. One of the big issues with this problem for q-learning is the lack of rewards, there is only a single reward and it is for completion. This means that the learner gets no incentive to finish quickly, as wandering around for 10 steps and 10 thousand steps both yield 0 reward. It also gets no disincentive for falling in holes, as it is simply a 0 reward end state, so it must wander its way to finding the goal state, wandering in the dark, to discover the fact that there even **exists** a state that ends the game on a higher reward. Adding these kinds of rewards, a small negative reward for each step to encourage quick completion and a negative reward for falling in a hole to encourage avoiding them, could help the agent perform significantly better by getting more feedback and getting it regularly. As noted in the lectures, it is important that the step penalties not be too large, as it could encourage the agent to simply end the problem, rather than get to the goal, as well as that they not be positive, as then the agent has no reason to ever end the problem. This was not implemented for this assignment, but would be the next step in improving q-learning performance. If this were done, it would be important to make sure that the rewards used simply enhanced the environment and were not added to induce specific behavior for q-learning. The tuning graphs are in line with what one would expect thinking about this problem and q learning. It ends up with a gamma of 0.6, something that we may expect to be higher given that VI/PI did better given a higher gamma. This lower value could be due to the fact that there are no step rewards or disincentives for falling in holes, which means that "planning too far ahead" without knowing where it actually needs to go could be detrimental, since it does not have the rewards passed in like VI and PI did. A relatively high epsilon makes sense for this problem, 0.9 in this case, as the agent needs time to move around and "discover" the landscape, but the decay of 0.6 still means that it will stop exploring sooner and start exploiting the knowledge it has. As can be seen above, the runtime unsurprisingly increases linearly with the number of allowed iterations, which was limited to 1 million for the sake of tuning and testing, which took roughly 5 minutes. It should also be noted that because the library implementation has no early stopping mechanism, such as based on Q values stabilizing, the runtime would increase linearly regardless of the algorithm converging. It can be seen in the rewards vs iterations plot that the algorithm was still improving even after 5 times as many iterations, getting up to a lofty 0.7. The number of iterations necessary for the algorithm to be on par with VI and PI was not discovered, but it would likely be unfeasibly high, as would the runtime, to make it a useful competitor.

2 Problem 2 - Non-Gridworld - Forest

The forest problem is a non grid world forest management problem pulled in from the example module of hiive-mdptoolbox (Example module). The essence of the problem is to determine each year whether it is better to cut the trees down for a reward of 1, resetting the forest to state/year 0, or wait, getting no reward but letting the forest continue to grow. Each year there is a probability p, 0.1 in this case, that the forest catches on fire and burns down, resetting it to year 0. If the forest is in its oldest state, 5000 in this case, the reward for waiting in this state is r1, 4 in this case, and the reward for cutting is r2, 2 in this case. The goal of the agent is to maximize reward from any given state, thus balancing high rewards at greater risk and lower rewards immediately, but resetting the forest's state.

2.1 VI/PI



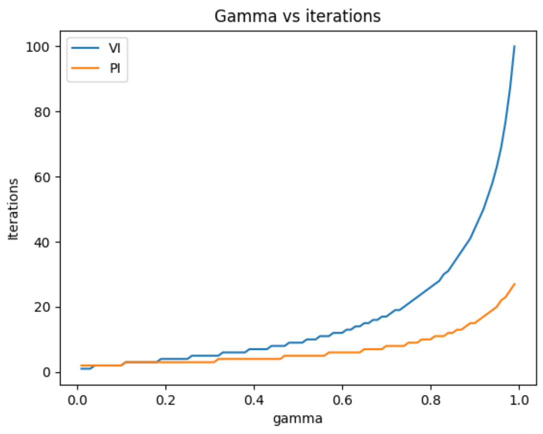


Figure 9, 10—reward vs gamma for VI and PI (left) and iterations vs gamma (right)

Very similar to the first problem, a high gamma is rewarded here as it is allowing better look ahead capabilities for the algorithm to determine whether it is better to hold out for the bigger rewards available at the final state or to "cash out" to avoid the risk of losing it all, to relate it to a gambling problem. The optimal policy that both VI and PI arrived at was "Wait Cut*4981

Wait*18". This makes a good amount of logical sense, for any state that is not close to end, in this case 18 places away, the risk of the forest burning down is too high to justify the additional reward, even if the discounted value is higher than what it would get now. Waiting in the last state also yields a higher reward than cutting the forest, so it makes sense to take that action if possible. However, 18 years (state transitions) is a long time to wait, as the probability of the forest not burning down, i.e., getting the reward for waiting, is only about 15%, however the delayed reward for waiting is about 3.34 which appears to be high enough to warrant the 85% chance of getting nothing. These are not necessarily odds that I believe I would take, but the algorithms made out quite well, getting 120-140 reward, so it seems to have worked for them. As with the first problem, value iteration managed to run more quickly, due to a low value function cost, but in this case took 4 times as many iterations and did not perform quite as well as the more methodical policy iteration.

2.2 Q-Learning

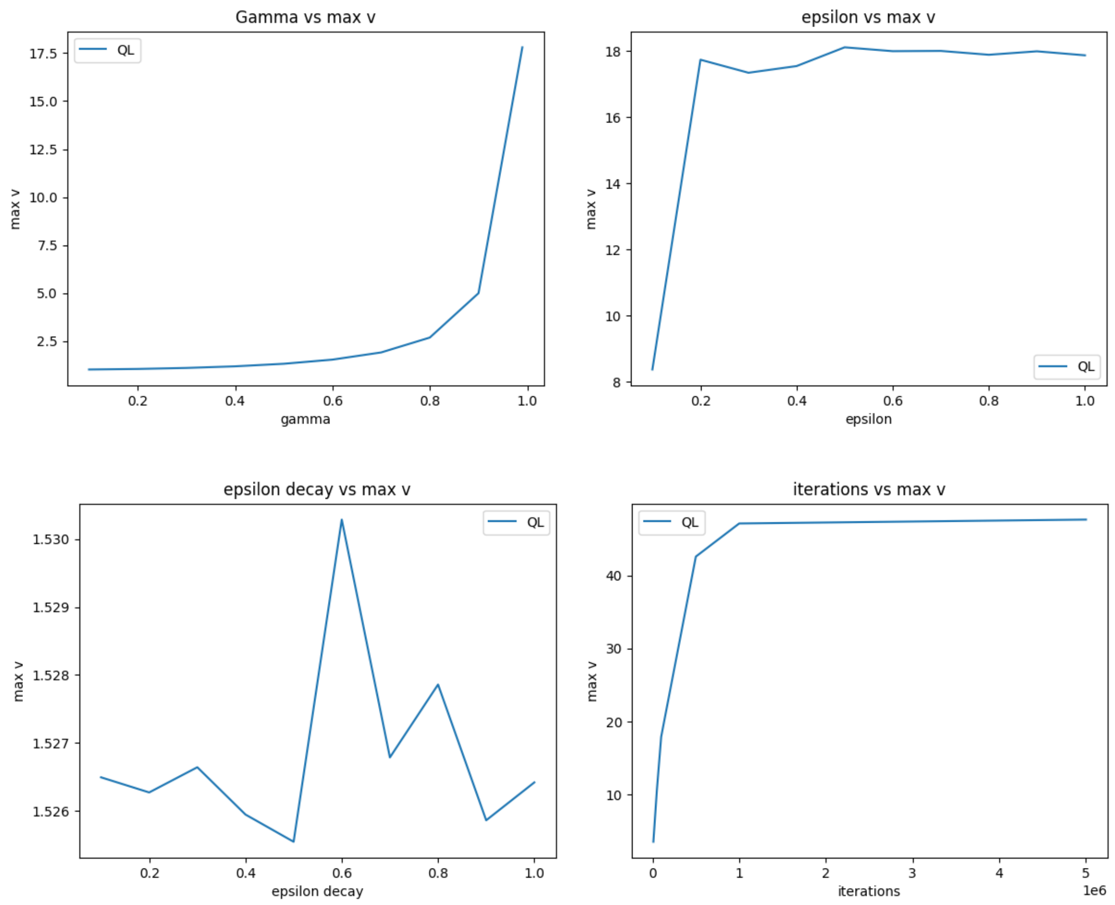


Figure 11, 12, 13, 14—tuning for QL by gamma (top left), epsilon (top right), epsilon decay rate (bottom left), and number of iterations (bottom right)

Similar to the first problem, q-learning had issues here. This is likely due to the fact that with so many states, the agent is simply not able to visit enough of them often enough to generate an accurate view of the problem. This is especially the case because it is likely not seeing the long term rewards very early, since there is only 1 state in which they matter, and the agent would need to take both actions in that state, or have managed to wait until that state to see them again. During the tuning process the reward for epsilon peaked at 0.5, likely due to getting the “idea” of the problem, thus not needing as much exploration time, since there are rewards scattered about, but the agent did not really see the big picture of managing the forest to avoid it burning down and waiting for the end rewards when within some reasonable time of it. The epsilon decay value peaked at 0.6, likely for the same reason, it is not hard to imagine the agent ending up in random states and learning to cut in that state because it got a reward before. However, QL does not possess any kind of ability to generalize this information, i.e., learning “cutting here gives a reward and it did in 500 other random states too, maybe always cutting is best,” because it needs to update its values for each state each time it visits it, which it would prefer be infinitely often. It may not be visible in the plot above, but the score continues to improve up to 5 million iterations, but only by a fraction of a point, making the significant time difference very much not worth it. The agent is already running for over 8 minutes for 1 million iterations, which is very significantly longer than VI/PI, as expected, and not doing as well, so spending over 5 times as long to do about the same is not very reasonable. Overall, QL only gets to about a third of the score of VI/PI. The policy, if copied in as with the VI/PI policy, is well over a page, as ql switched its actions quite frequently, likely leading to the forest burning down quite often due to bad luck. A snippet of the beginning of the policy is as follows:

“WC*3 W*2 C*2 W*4 CWC W*2 C W*4 C W*3 C W*6 C W*3 C*2 W*6 CWC W*2 C W*8 C W*9 C W*3 C W*4 C W*2 C W*2 C W*2 C W*3 CWC W*2 C W*5 C*2 W*2 CWCWC W*4 C W*4 CWC W*6 CWCWC W*4 C W*3 C W*7 C W*3 C*2 W*2 CWCW C*2 W*3 C W*2 C W*3 C*2 WCWC W*9 C W*3 C W*2 C*2 W*3 C*2 W*4 C W*5 C W*2 C*2 W*2 C W*6 C W*6 C W*2 CW*4 C W*2 CWCWC W*3 CWC W*4 C W*2 C W*3 C W*5 C W*4 C*2 WC W*5 C W*2 C W*3 C*2 W*4 CWCWC W*3 C W*5 C W*2 CWCWCWC…”

As noted before, there is no real reason to wait in any state until the very end, as it is better to get **some** reward rather than banking on the functionally 0 probability of making it to the end. However, this frequent oscillation between actions is likely not due to QL actually finding that waiting is a reasonable action in these early states, as it never is, but more likely due to the fact that many states were simply not visited often enough for the algorithm to get a good idea about them. Given enough time and iterations, QL would learn the problem well enough to be passibly good, but it is certainly not the algorithm for the job, especially if the reward matrix is available, it makes very little sense to have an agent try to learn it by wandering around.

3 References

1. Markov decision process (mdp) toolbox: Example module. (n.d.). Retrieved from https://pymdptoolbox.readthedocs.io/en/latest/api/example.html
2. OpenAI. (n.d.). A toolkit for developing and comparing reinforcement learning algorithms. Retrieved April, from https://gym.openai.com/envs/FrozenLake8x8-v0/