Frozen Lake

The frozen lake problem is a grid world problem. The concept is that there is a lake that is frozen. The upper left corner is the start location while the bottom right is the goal. There is frozen tiles and hole tiles. If a hole tile is reached, then it is considered a terminal state and the round is over. Since the lake is frozen, it is also slippery which creates a 1 in 3 chance of sliding to the left or the right tile of whichever direction the action is going. This creates a situation where the optimal policy is often not to go to the goal directly, but rather getting to the goal in a way that involves avoids the risk of falling into a hole as much as possible while still getting to the end goal. All rewards are 0 except for the end state where it is a reward of 1. The actions are limited to 10,000 actions or a terminal state when the policies are being evaluated. The evaluation will involve running the policy through 100 simulations and seeing how many times the policy achieves getting to the goal.

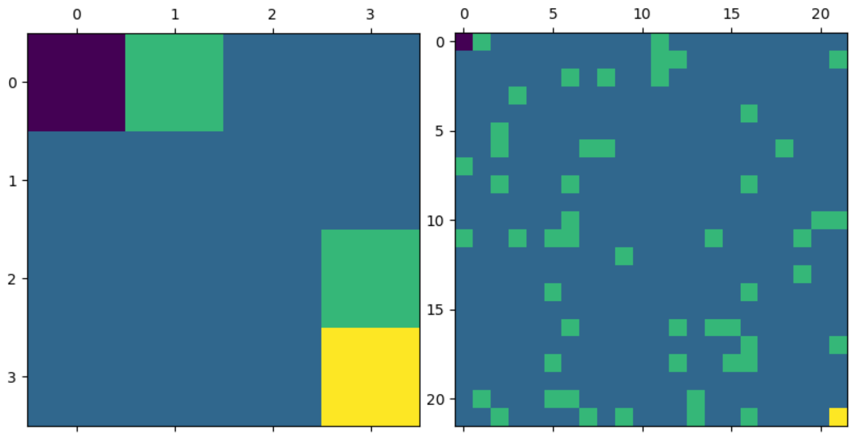
The first map 4 x 4

Start	Hole	Frozen	Frozen
Frozen	Frozen	Frozen	Frozen
Frozen	Frozen	Frozen	Hole
Frozen	Frozen	Frozen	Goal

Optimal Policy

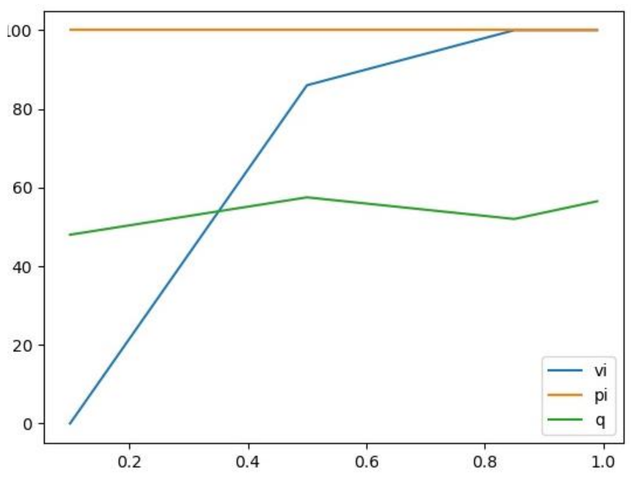
Left	Hole	Right	Up
Down	Down	Left	Up
Down	Down	Left	Hole
Down	Down	Down	Goal

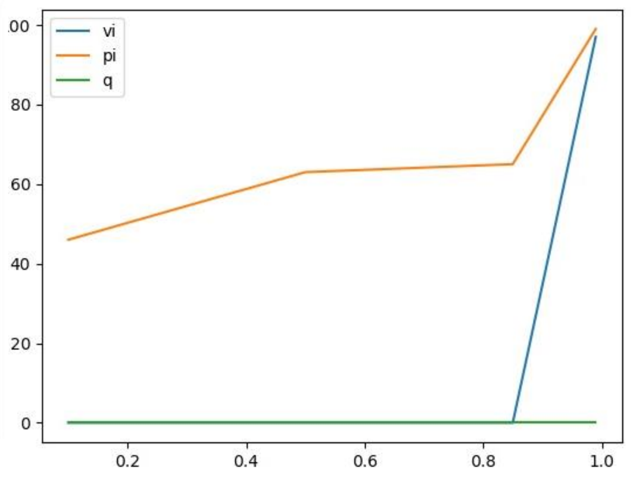
An optimal policy for this map is shown above. The key to this problem is not just to get to the goal, but to also avoid sliding into the holes. For the current map, an optimal policy can be achieved with 0 chances of failure meaning on 100 runs all 100 can make it to the goal. For example, the start move should be to go to the left. This is because there is a hole to the right of the start, and thus, any other move risks a 1 in 3 chance of slipping into the hole. By going left, it will eventually lead to going down through slipping and getting to the next state. Any space next to a hole, must have a move that takes it into the exact opposite direction of the hole to guarantee safety. The space beneath the first hole is down and the one to the right of it is right, this makes it so that slipping into the hole is avoided. The same can be said for the other hole on the map. Another interesting part of this strategy is that the optimal move next to the goal is not to go right directly to the goal, but rather go down and eventually slide to the goal. That way it does not even end up with the risk of going to the one above it, and thus, having to be next to the hole and potentially have to take an additional move to get back to where it was since it can only go left at that spot to guarantee avoidance of failure. Larger problem sizes can create more difficulties and can even create maps that have no guarantee at success. This can occur in the smaller space as well, but it is less likely. Situations where holes are diagonal from one another could force moves that are not guaranteed to succeed. An increase in those kinds of situation and a 1 in 3 chance in slipping will reduce the reward amount for any optimal policy over 100 runs.



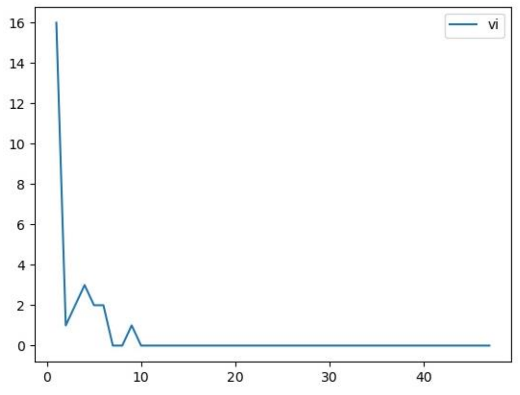
The images above represent two frozen lakes of different sizes. The purple square is the start, and the blue squares are the frozen tiles. The green squares are the holes, and finally, the yellow tile is the end goal. The first lake is the same as the above example and is a 4 x 4 map. The second map is that of a 22 x 22 map. There are areas of the map that cannot be crossed without some risk of slipping. Those areas can be avoided altogether in the larger map if one is careful. It involves some potential movement loops to avoid falling in. The key is still to always move in the opposite direction of the hole if next to one. The map that was generated for the 22 x 22 in this case does have many optimal solutions that can avoid any chance of slipping into a hole and reach the end goal. If location (3, 5) and location (3,10) were holes, that would be a situation where there would be no way to guarantee that all holes could be avoided while moving towards the goal. It would create a situation where moving opposite of 1 hole would either be moving directly towards a different hole or with a hole to the left or right for any given route. The larger problem size does create situations where the reward takes more time to propagate for something like Q learning. Another interesting observation is that there are multiple optimal strategies. When there are large portions of frozen tiles, there are many moves that will achieve getting closer to the goal, but it will have no impact on how likely a policy is to get to the end goal and get the reward. There are many locations where going right or going down can get closer to the goal and have no real difference in the utility of one state over the other.

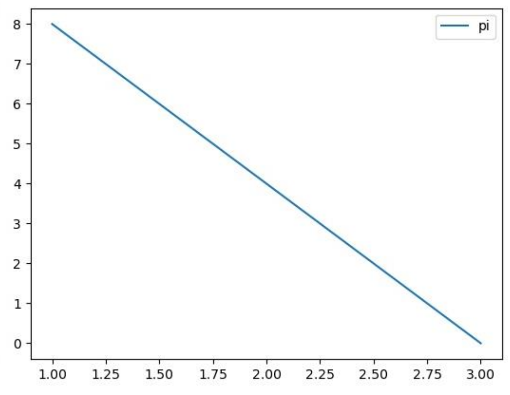
Value and Policy Iteration

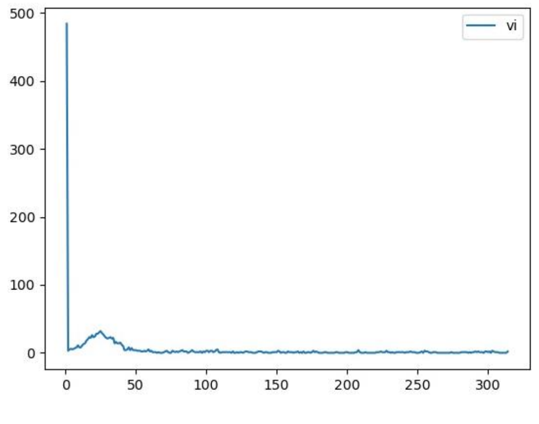


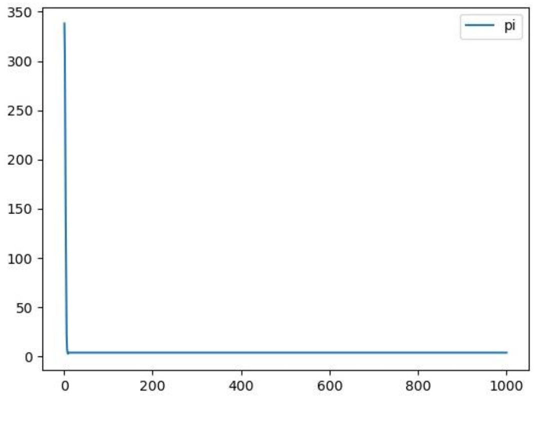


The charts above show the results of trying different values for Gamma for policy iteration and value iteration. The chart on the left is for the 4 x 4 and the chart on the right is for the 22 x 22. It can be noted for both cases a higher value in gamma was preferred for value iteration. This is because the reward does not come until the end so discounting it too much will lead to an insignificant amount of learning for later rewards. Policy iteration did not need that for the first problem since it was so simple it was able to figure it out no matter how discounted the latter rewards were. The larger map though it ran into a similar problem although had a smaller floor than value iteration did. Q learning is also presented here and the value of .5 was used for the 4x4. This is also a sneak peek on how Q learning did for the larger map (hint not good).

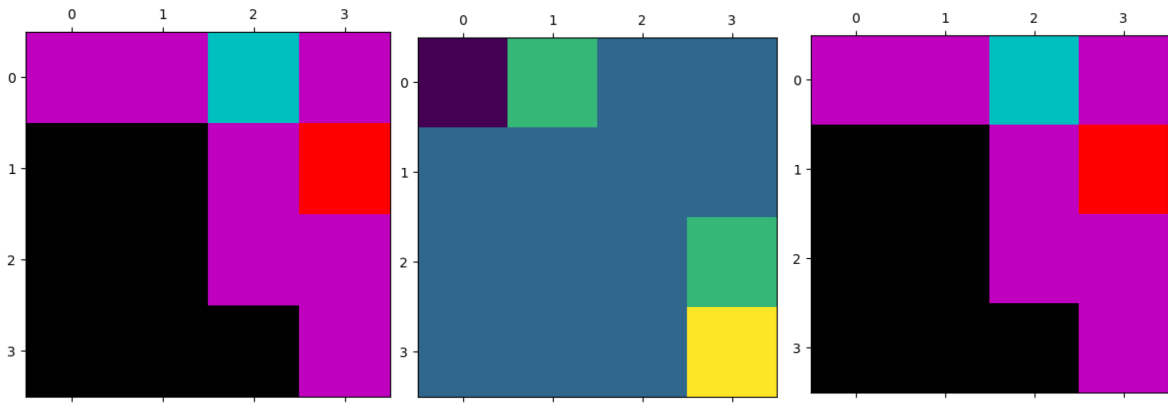




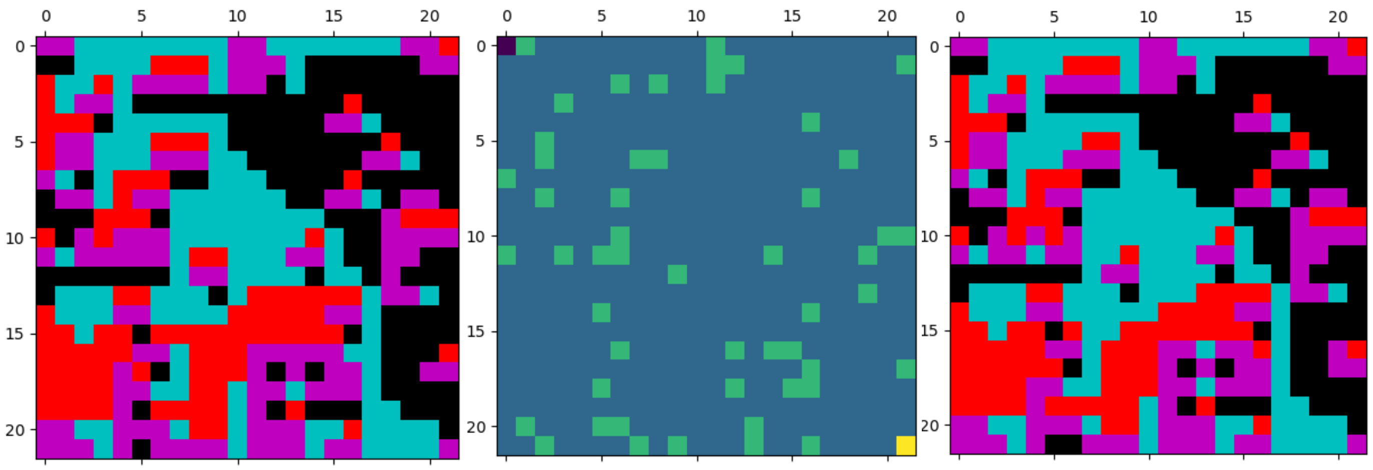




For value iteration and policy iteration, lets first exam the convergence plots. The plots represent iterations on the x axis, and the change in actions at that iteration on the y axis. Above are 4 plots, the first two plots are for the 4 x 4 map and the second two are for the 22 x 22 version. For the first problem, both value and policy iteration make many changes right from the beginning. Value iteration changed the actions of every state, and policy iteration changed half. Value iteration is updating all the actions at 1 time based on its newly calculated rewards for each utility. As it propagates the reward from the goal, it updates less actions, but with more impact. Only those actions which help to propagate the increased value in utility faster and stronger are updated as iterations continue. Therefore, an initial spike occurs, and then a drop with fewer and fewer updates being made. Policy iteration instead updates the policy directly and needs fewer overall changes. It calculates the utility of the policy and updates the policy based on any changes that indicate a more optimal policy exists. In the 4x4 case, it only needs 4 iterations (3 spots to change policy), to reach convergence. The initial 8 changes, then 4 additional updates on the next iterations, and then no changes occur because an optimal policy has been reached. This makes sense given that many of the utility values will not change when looking at the next version. For the larger problem, a similar pattern can be seen for value iteration. It has a bunch of initial updates based on rewards and future rewards from the neighbors. Then as it continues to iterate small increased chances of rewards get back propagated through to make states in the change prefer actions that bring them to the states with increased rewards. After about 50 or so iterations, small policy changes are being made. It takes a few iterations as the values are being updated by small amounts not necessarily enough to change the policy at first, but enough to be noticeable as an improvement. Eventually, this gets to a point where there's no significant spikes, and the changes are less than a threshold set in the algorithm, and thus, the algorithm stops. The reason it does not reach a point of 0 changes is because there is more than 1 optimal policy, and thus, it could continue to update values and swap out actions at various states for a long time. This is emphasized even larger in the policy iteration graph. The policy iteration graph makes a ton of changes very quickly, it finds essentially an optimal policy, but then it just continually does 4 changes to the policy repeatedly until the max number of iterations is met. This is because it can continually change the policy around and find something that is slightly more "optimal". Thus, convergence may not be reached or may take a long time to get there. It took policy iteration longer than value iteration for this reason. Overall, it highlights something interesting about policy and value iteration. When there are multiple optimal policies, convergence can be reached to one of them, but if there are not thresholds to stop them from iterating further, they can iterate until they are told to stop or at least for a very long time.



The images above represent the optimal policy for the 4x4 problem that was reached by value and policy iteration. Value iteration is on the left and policy iteration is on the right although in this case, it does not matter, because they reached the exact same policy. Purple represents the action go left, black represents the action go down, cyan represents the action go right, and red represents the action up. It can be noted that at every location that there is a hole the optimal strategy was go in the opposite direction of the hole.



These are some of the optimal policies for the 22 x 22 grid size. The same applies as before: value iteration on the right, policy iteration on the left, purple represents the action go left, black represents the action go down, cyan represents the action go right, and red represents the action go up. Again, wherever a hole is, the action is to go in the opposite direction of that hole. One interesting thing is with the larger size of the problem, there are locations where being surrounded by holes on either side, the action is to move in the direction of one of the holes. This makes sense given that it is more likely to slide than not so sliding would make it, so a hole is avoided in that case. It also is the only action that can be taken that involves reducing the chance of sliding into one of the holes by going in the opposite direction. There are some policy differences including in a location where there are holes on either side. Value iteration decided to go left where policy iteration decided to go right. Both the left and right have a hole on either side of that location but moving left or right gives a 2 in 3 chance of not ending up in the hole because of sliding so both actions are equally optimal at that point. Another interesting observation is that the actions taken directly avoid going near any sections with more holes and prefer to generally go through sections with more ice. That makes sense as well given that there's less chance of falling into a location that requires taking a chance at slipping. Overall, a larger number of states make it more likely that multiple optimal strategies exist for this problem. It also makes it take longer to converge as well as take more iterations. Also, it can mess up something like policy iteration or value iteration with a small epsilon by having it run through more and more iterations for small changes in the utility functions since multiple optimal policies exist.

Algorithm	Map	Iterations	Time	Reward on 100 runs
vi	4x4	47.000000	0.000000000	100.000000
pi	4x4	4.000000	0.000000000	100.000000
vi	22x22	314.000000	0.084774971	100.000000
pi	22x22	1000.000000	7.712792635	100.000000

Overall, both policy iteration and value iteration were able to achieve optimal strategies as seen in the table above for both maps. It took policy iteration less iterations for the first map to converge than it did for value iteration. The same may not appear to be the true for the larger map, but if policy iteration had not got stuck in a cycle, it can be seen that it took less iterations on the convergence plot to get to a point where it was just cycling through optimal policies. Value iteration ran faster for the larger size, and both ran so fast for the smaller size that the clock did not even register a difference in the time from start to finish! That shows the power of these kinds of algorithms for problems that are well defined.

Q Learning

Q Learning has a much more difficult task at hand than either policy iteration or value iteration as it needs to discover the information that either of them get from the start. I do not expect Q Learning to do as well, because it will take some time to propagate the reward back. By then the learning rate and exploration rate will decay which can make it more difficult to propagate rewards from one state onto the next states. The exploration rate is a little less relevant for this problem. It is still important to explore and try options, but given that there are only 4 actions most of which will result in 0 rewards until propagation occurs through the Q table, exploring is of a little less value than in cases where there are more intricate reward systems and action sets. Overall, I hypothesize that the Q learning algorithm will get some sort of solution for the smaller problem that will avoid some of the holes but not all. For the large 22 x 22 problem, I expect that Q learning will have an extremely difficult time with that problem. I theorize this because the problem is not one with intermittent rewards. The reward is only in the end state, and the end state is quite a distance and large number of potential state action transitions from the starting state. Additionally, negative penalties are not given for falling into holes or positive rewards are not given for getting closer to the goal. They are only given once the goal is reached. This requires Q learning to find a path in which it is able to get to the reward successfully without hitting a terminal state of a hole and to be able to successfully update enough of the table to create a path to the reward. I do not have high hopes for Q Learning's ability to do that on the larger more difficult map.

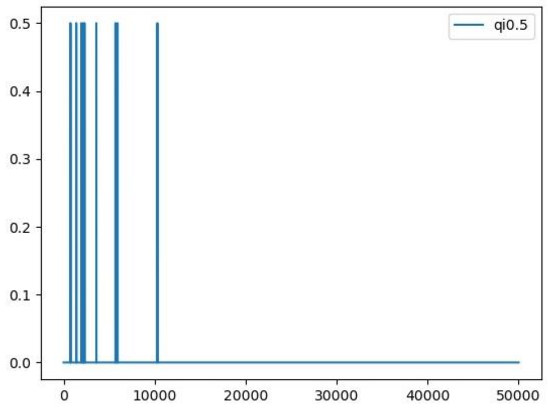
Hyperparameter Tuning for Exploration Strategy

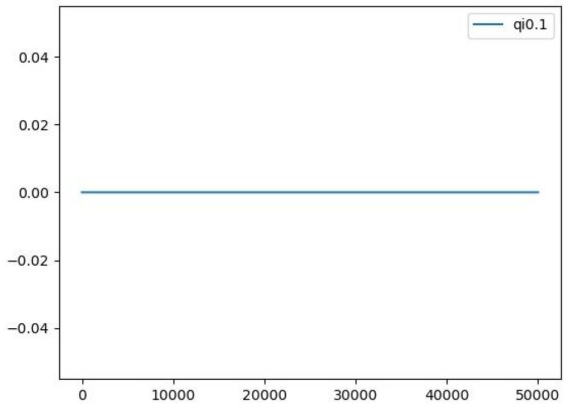
Constant Epsilon	Size	Gamma	Rewards on 100 runs
0.15	4	0.1	18.333333333333332
0.15	4	0.5	36.0
0.15	4	0.85	53.666666666666664
0.15	4	0.99	39.0
0.25	4	0.1	16.0
0.25	4	0.5	16.666666666666668
0.25	4	0.85	47.666666666666664
0.25	4	0.99	44.666666666666664
0.33	4	0.1	68.6666666666667
0.33	4	0.5	32.333333333333336
0.33	4	0.85	44.666666666666664
0.33	4	0.99	29.0
0.5	4	0.1	25.0
0.5	4	0.5	16.333333333333332
0.5	4	0.85	33.666666666666664
0.5	4	0.99	40.666666666666664

Decay Function for Exploration	Size	Gamma	Rewards per 100 runs
log	4	0.1	43.0
log	4	0.5	65.33333333333333
log	4	0.85	57.666666666666664
log	4	0.99	51.333333333333336
exp	4	0.1	39.666666666666664
exp	4	0.5	36.0
exp	4	0.85	40.0
exp	4	0.99	37.666666666666664
geom	4	0.1	21.0
geom	4	0.5	34.666666666666664
geom	4	0.85	32.666666666666664
geom	4	0.99	26.666666666666668

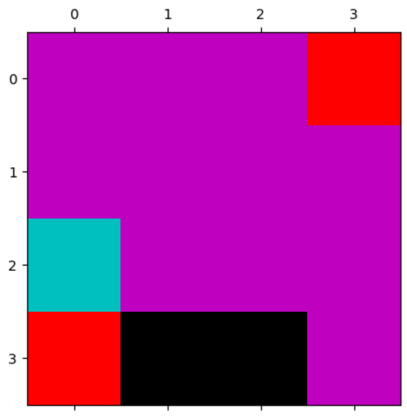
The above tables show the hyperparameter tuning results for the exploration rate. I tried both a set of constant values and a set of decay functions with different gamma values. A constant rate between .25-.33 allowed for the best results for constant values. Of the decay functions 1-

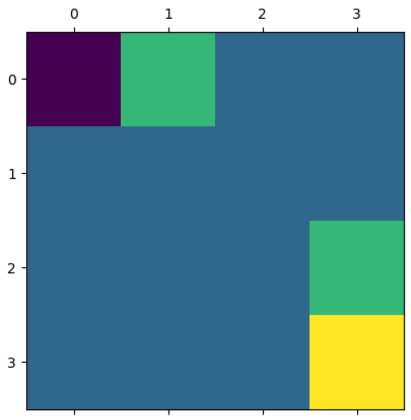
(1/log(n+2)) where n is the iteration number appeared to do the best. This makes sense given that this function gets to a minimum of around .25. Thus, it has a bit higher of an exploration rate in the beginning and slows down to what ended up being the second best exploration constant rate of around .25. Exploring too little didn't get enough of sampling of the space, but exploring too much leads to a lack of exploiting the knowledge that was gained to find truly valuable paths. This can be seen in the results above as the exponential and geometric decay functions decay too quickly in comparison to the log function leading to a lack of trying things. The exploration rate of .50 is too high though as every other action is pretty much random and it ends up with similar results to the exponential and geometric, but for different reasons, since it does not exploit the knowledge gained enough and is too random as information builds. I utilized the log based exploration strategy because it had on average the best results across different gamma values. The .33 constant exploration looks like it would have potentially worked well on a low discount rate, since it would be able to sample the space pretty decently, and thus would likely determine better actions for each individual state with slight interest towards the future results. I would show the results for the larger problem size for q learning, but they are uninteresting as all the results are 0. Q learning did not complete it at all. I tried varying gamma, learning rates, and the decay function, and I could just not get Q learning to play ball. This will be discussed further later.

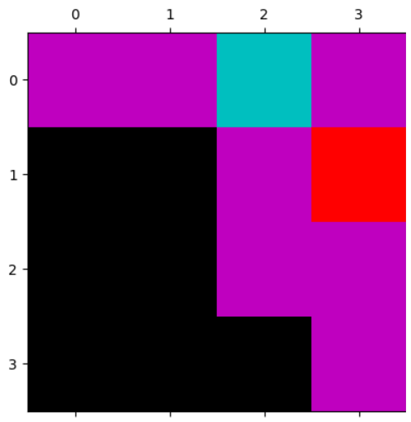




Q Learning for the small problem size had several changes to the policy in the beginning 10,000 or so iterations and then it just continued to run making no updates to the policy. This would be the point of convergence for Q learning. This is because beyond this point, the learning rate of 1 / _math.sqrt(n + 2) is quite small so any new updates are unlikely. It appears that Q Learning hit a local minimum towards the beginning with a bunch of changes rapidly, and then took quite some time to update additional policy actions. This makes sense given the decaying learning rate and smaller amount of exploring less changes were occurring, and they were quite discounted when they did occur. Thus, only changes that continually showed benefit would end up getting propagated enough to make a change in the policy itself. The graph on the right is for the 22 x 22, and it shows the sad story that Q learning was unable to learn this complex of a problem with the underlying implementation. It shows the limitations a problem can create for a learner that does better with incremental rewards when the problem only has rewards at the very end of the problem. This makes it extremely difficult for Q learning to ever even get to the point of realizing the reward let alone figuring out an optimal path to it. It makes it so there is pretty much no difference for Q learning for going anywhere and doing almost any action. It is getting no feedback and makes it so falling in a hole is the same as moving to any other location. Compared to policy and value iteration convergence to the optimal policy is so much more difficult for Q learning since it is blindly learning and stumbling about. This problem does not have intermittent rewards to say keep going this way. The rewards only show up at the end and propagating that back requires a lot of exploration that ends up in just the right places.





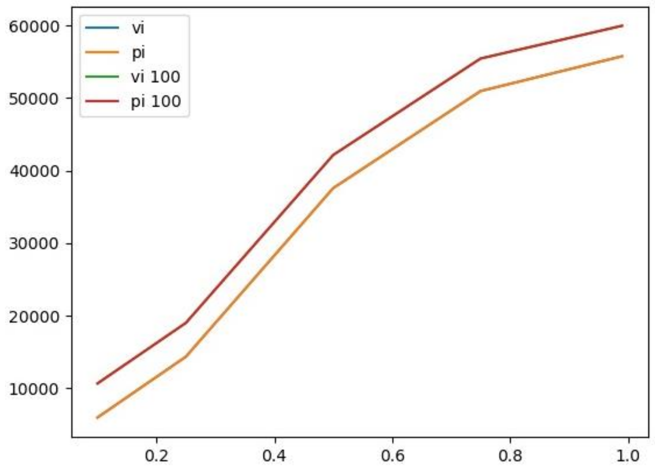


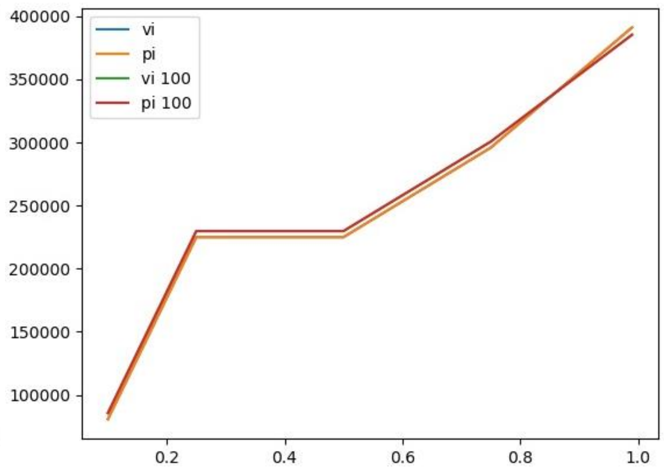
Q Learning's policy on the left can be compared to the optimal policy reached by policy and value iteration's policy on the right. Q learning was unable to learn that avoiding the holes at all cost was essential for success. In fact, in location (0,2) it has an action that would lead it directly to a hole instead of away from one if it successfully executed. This makes sense as Q learning does not see a difference between falling in the hole and going to a square. Especially if reward has never been realized for sets of actions out of that state. The explorative nature of Q learning is in display here as well as the square (3,0) decided on the action up. This does not make a lot of sense given the goal is to the right and there is nothing to avoid either. Overall, Q learning was able to get parts of the optimal strategy for the smaller problem, and just being able to avoid the hole in the beginning increases its chances substantially. Then having the go left in column 2 is good except in the spot next to the hole. That's because it reduces the likelihood of getting to a space next to or falling into the hole next to the goal. This could just be luck though as the base actions were initialized to 0. As shown in the table below, the run time for Q learning is much higher than either polic y iteration or value iteration, but that is expected as it has more computing to do and more iterations to run since it needs to determine things that policy and value iteration get for free. The rewards on 100 runs were 61 times it made it to the goal, which is pretty decent considering that it is a 1 in 3 chance of not getting the desired direction at any given point, and if too many of those occur, it can end up right next to a hole. The large problem size for this problem just obliterated Q learning. It was unable to pick up on the scent of a small reward so far away when every action just seemed to give it 0 rewards. In comparison to policy and value iteration, it just shows the difficulty of determining the optimal policy with lower information, and that this p

Algorithm	Problem size	Average Time	Reward on 100 Runs	Gamma
Q	4 x 4	1.512611389	61.000000	0.50

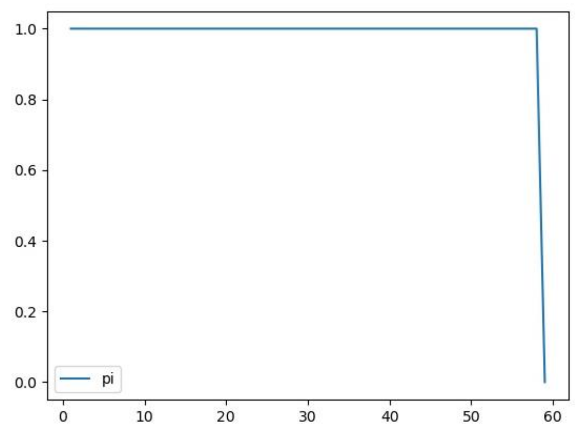
Forest

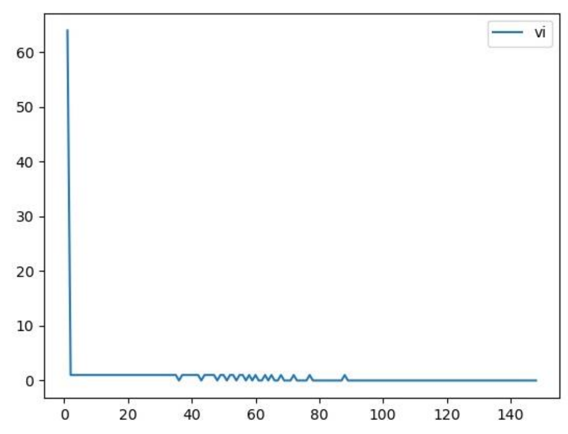
For the next problem, it is a forest management problem. The general idea is that there is a forest with a given age. That forest can either be chopped down and receive a reward for harvesting the lumber, or it can be preserved by waiting to chop it down. If the forest reaches the oldest age of a forest, then a reward is received for being in that state and every subsequent time that forest is in that state that same reward is received. There is danger to be aware of though. The danger is that the forest could burn down in any given year and revert the forest back to the original state. The optimal policy will be one that determines the right point at which to wait for a forest to have a chance of getting the reward for making it to the final state. Waiting too long make it unlikely to make it to the final state over burning down, and thus, missing out on small gains from chopping. Since chopping a forest down reverts it to the youngest state, once chopping occurs or when in t he initial state, the optimal policy is to wait for the forest to grow for 1 cycle, then chop it down and wait again for the next and so on. This does depend on the probability of the forest burning down as well as the reward for making it to the end state. I set the reward for waiting at the oldest age to be the total number of states of the forest^1.2. I also set the probability to burn to be 1/((the total number of states of the forest^.60). This creates a more interesting problem, because if the probability of burning and the reward at the end are always the same, then changing the total number of forest ages does not impact the point at which waiting should begin instead of cutting. This makes it so its more variable and as the problem size increases the expected value calculation changes with it. For this problem I measured reward in two different ways. Each ran 100 simulations that start with the forest in a random state from 0 to the number of states; one ran until state 0 was reached meaning the forest was in its initial state. The other ran 100 iterations and stopped whether it ended up at the terminal state or not. I thought this would highlight interesting differences showing the potential of when waiting can lead to rewards. I thought it was also fare to show what a set number of time iterations would do with the given policies since it would emphasize the penalty that can come from waiting and not receiving the reward as well as show what could happen if the reward is never reached in the larger state. The forest can be up to 64 in the smaller version of this problem, and it can be up to 666 in the larger version. This defines the number of states there can be.

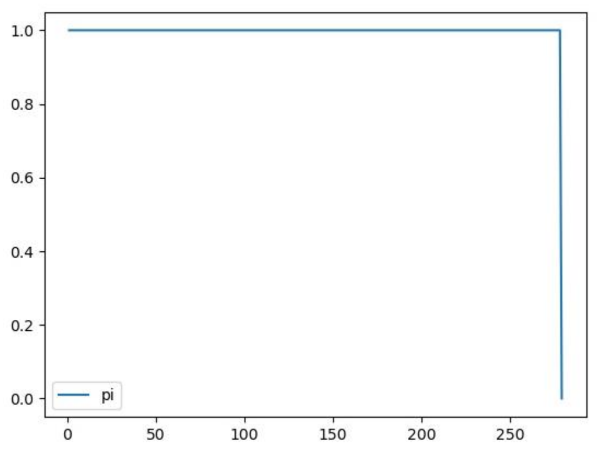


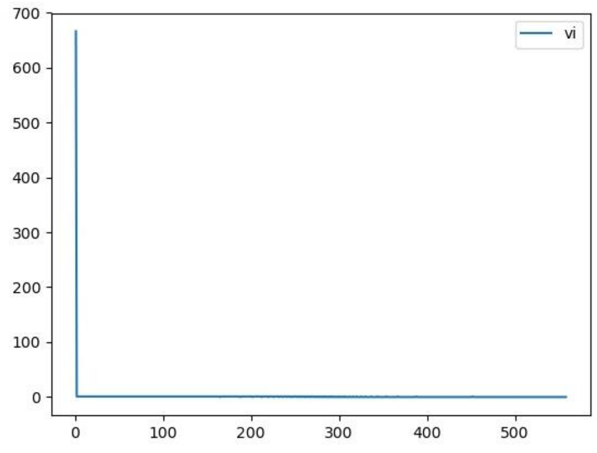


This is the gamma vs reward chart for value iteration and policy iteration. The average reward is lower with lower values of Gamma since the future reward is more heavily discounted. This just results in making it to the final state less often which means not getting the larger pay off at the end for having waited. As gamma increases, that max reward is reached more often and gives large boosts to the sco re. Gamma being as large as possible in this case is the best solution. This allows the probability of the forest being burned down vs the reward for reaching the final state to dictate what the cutoff point should be without gamma interfering at all. The two curves follow each other for the scores on 100 actions vs running until state 0. In the first instance, the 100 iterations is slightly ahead, which makes a ton of sense because generally state 0 would be reached through cutting or burning occurring. Thus, the same policies being used would have similar results, but on average, the 100 actions would get to do a little extra cutting and boost up the point a bit. In the larger case, its creepy how close the two are. This m ust indicate on average the 0 state is reached around 100 states or so. It is also interesting that at the very end the run until state 0 rewards edge out the 100 meaning it was on average running for more than 100 iterations. This shows the policies being used must be have more than 100 wait states. Also, notice that policy iteration and value iteration have the same graphs, and thus, only one line is showing up since it is on top of the other one! This gives insight into the fact that there is only 1 optimal policy for this problem, and policy iteration and value iteration arrived at the same one.

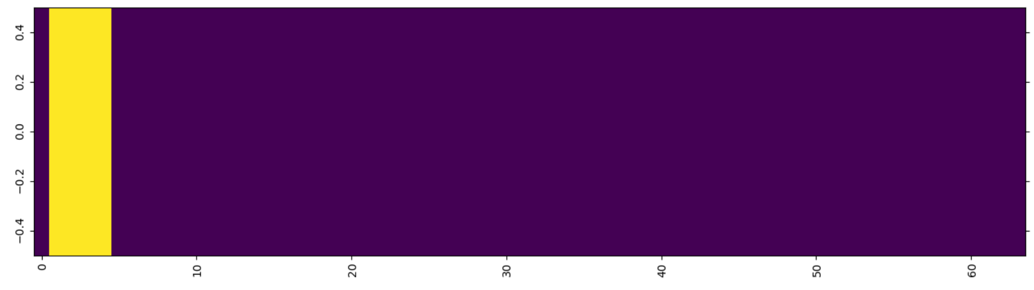


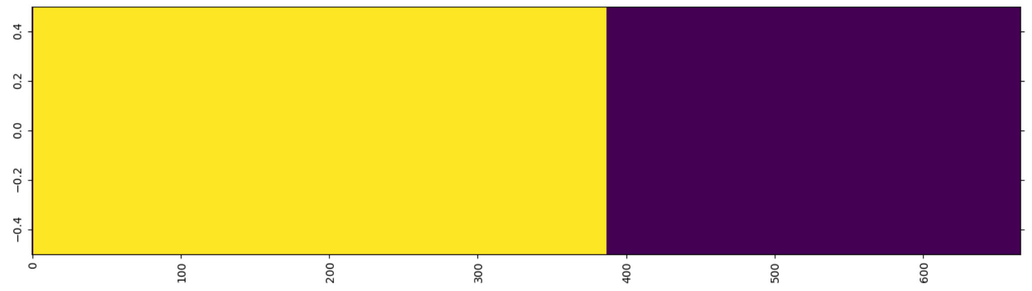






The convergence plot for value iteration looks very similar to that of the previous problem. In both cases, it makes a ton of updates and converges rather quickly and just waits until the updates are below its threshold to stop. It changes the number of states all at once for the policy update. This makes sense since it can quickly calculate the values and grab the one that returns the max for the state. There are only two actions, and one set of transitions for each plus the fire which makes this a simple and quick calculation for value iterations. Policy iteration on the other hand is doing something interesting here. It seems to be changing 1 action at a time. It is looking at one step into the future with a different action and calculating the new max policy with the 1 change. It continues to do this until it reaches the number of iterations that matches the number of states that it decides the wait action is appropriate for, because it is only calculating the update at t+1 for any given iteration so as time moves forward the value of waiting shows up in more and more of the states. Different number of states did not impact the overall the convergence. It did impact the time by making it take longer to compute overall. Additionally, value iteration took more iterations to converge, but it ran faster than policy iteration. Each of its iterations are simpler and do not involve recalculating the t+1 policy. In this case, policy iteration was also forced to do that multiple times as mentioned before due to the nature of the problem.





The optimal policy that was reached by both value iteration and policy iteration for each problem size is shown above. For the smaller problem size, waiting most of the time was the preferred action, where in the larger problem that became infeasible after a certain point. This is because even with a smaller probability of burning, eventually that will just become statistically likely to occur before reaching the end state. There is a clear cut off point for each problem size. This is the optimal policy for each of these problems, and both converged to the same thing. Value iteration was faster to converge to the policy than policy iteration was. This is because value iteration could quickly calculate and propagate the values back and update the ideal policy in pretty much 1 foul swoop. Policy iteration on the other hand had to make individual changes each iteration to come to the same conclusion. This highlights a case where value iteration is much faster because of how it computes the optimal policy. The fact that there is a limited number of variations of both actions and states helps to make this a faster and simpler calculation for value iteration. Below is the table with the overall results, run times, iterations, and rewards achieved by both. They have the same policy and thus, the same rewards. The time was obviously longer for the larger problem.

Algorithm	Size	Iterations	Time	Run until 0	Run 100 iterations
vi	64	148.000000	0.008024693	55734.654598	59937.654598
pi	64	59.000000	0.016000748	55734.654598	59937.654598
vi	666	558.000000	0.146872044	391160.118211	385364.590255
pi	666	279.000000	3.784347296	391160.118211	385364.590255

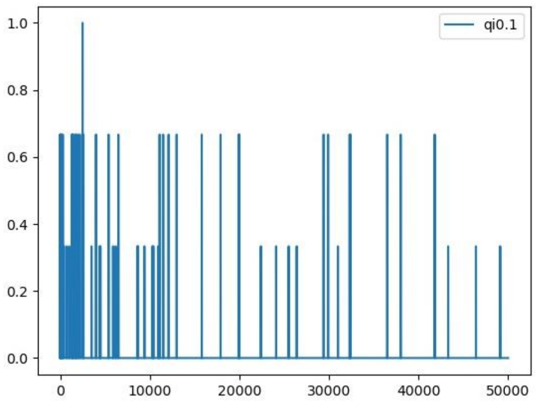
Q Learning

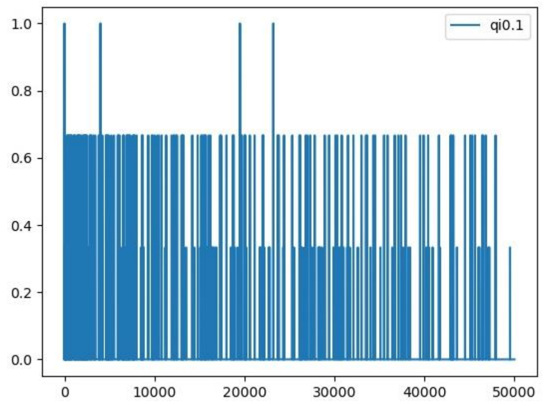
Q learning for this problem will likely fare a little bit better, but it will likely have difficulty really converging to anything. It has a lot of rewards that are being thrown at it that are of a similar size. To figure out the point in which waiting results in higher rewards requires q learning to explore enough to get enough waits in a row to get the reward and update the Q table. This would then need to be propagated through additional actions that would result in trying this same action before the state in which it was previously determined to be a good location to wait at. This would need to happen enough times for the pattern to start to emerge. Then it would need to be acted upon enough while still randomly trying it at states before that to determine an optimal location in which to start waiting from. This requires a high amount of exploration, but only in the end states, which is going to prove difficult for Q learning. The same can be said for figuring out to always cut in the beginning so many states. Cutting will result in a reward early on which will make Q lean towards that, but the exploration will lead to it wanting to try the other options and if the state after waiting has a cut with a reward, it could make it think that waiting then cutting provides a reward but cutting at certain points does not. Since it does not know the exact structure, it will have a lot to sort through.

Decay	Size	Gamma	Run until 0	Run 100 iterations
log	64	0.1	5968.002244251486	10631.66891091816
log	64	0.5	5967.33557758482	10631.335577584827
log	64	0.85	5963.33557758482	10609.335577584827
log	64	0.99	5964.33557758482	10612.335577584827
exp	64	0.1	1126.2337260773434	5831.233726077342
exp	64	0.99	1126.2337260773434	5831.233726077342
geom	64	0.1	1126.2337260773434	5831.233726077342
geom	64	0.5	2154.467452154686	6856.467452154687
geom	64	0.99	1126.2337260773434	5831.233726077342
log	666	0.1	128826.4514110352	133504.11807770186
log	666	0.5	80751.60563096503	85424.60563096503
log	666	0.99	80751.60563096503	85424.60563096503
exp	666	0.1	96.0	4931.0
exp	666	0.99	96.0	4931.0
geom	666	0.1	99.0	4937.0
geom	666	0.99	99.0	4937.0
0.15	64	0.1	1126.2337260773434	5831.233726077342
0.15	64	0.99	1126.2337260773434	5831.233726077342
0.25	64	0.1	1126.2337260773434	5831.233726077342
0.25	64	0.85	2154.467452154686	6854.467452154687
0.25	64	0.99	1126.2337260773434	5829.233726077342
0.33	64	0.1	1126.2337260773434	5830.233726077342
0.33	64	0.99	1126.2337260773434	5829.233726077342
0.5	64	0.1	1126.2337260773434	5830.233726077342
0.5	64	0.85	2153.467452154686	6852.467452154687
0.5	64	0.99	1125.2337260773434	5826.233726077342
0.15	666	0.1	97.33333333333333	4927.333333333333
0.15	666	0.5	97.0	4926.0
0.15	666	0.99	97.0	4926.0
0.25	666	0.1	97.0	4923.0
0.25	666	0.99	97.0	4923.0
0.33	666	0.1	98.0	4921.0

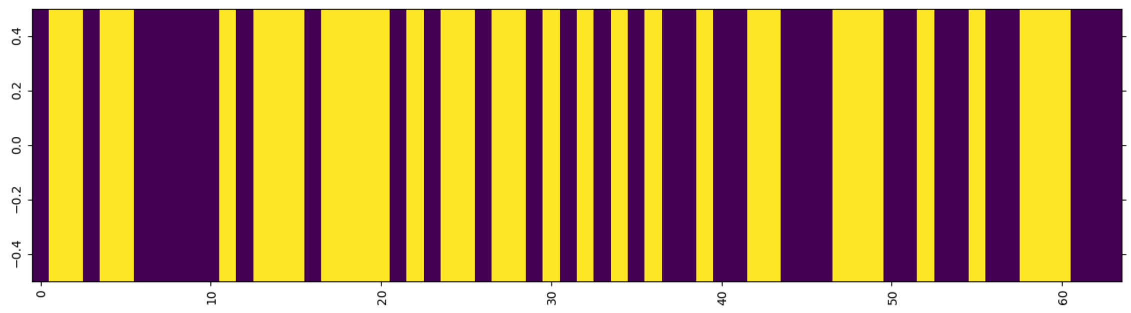
0.33	666	0.99	98.0	4921.0
0.5	666	0.1	80760.60563096503	85534.60563096503
0.5	666	0.99	80760.60563096503	85534.60563096503

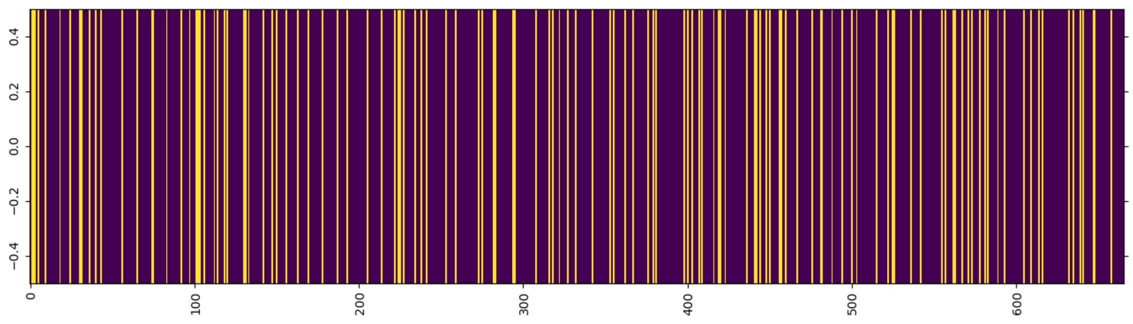
The tables above show the exploration strategies that were tuned for this problem. Some combinations were cut from the final table, because they were the same as the others of their kind. An interesting thing is that the .15, .25, .33, and geometric decay function for 666 all have the smallest expected value possible. This would just be always cut. They did not do anything but keep the cut actions as they explored since it gave the most immediate pay off. The others all seem to have a random combination of wait and cut spread throughout them. The exploration strategy that seems to be the best is the logarithmic decay with a gamma equal to 0.1. Which says explore around a bit but do not pay much attention to future rewards. This is odd to say the least. I think this shows overall the Q learning strategies are not converging. Thus, limiting future rewards and just trying and updating the state as is will likely lead to learning to at least wait in the last few states and pretty much cut everywhere else. The reward at the end is large enough that even with a gamma as small as that it will still affect the localized neighbors towards the end. A gamma this low essentially only allows close neighbors reward to have any impact. This does make sense given that if Q learning is having trouble converging then considering future rewards too heavily is just confusing it more. Thus, becoming more localized allows it to learn those end wait states. Log also makes sense since it allows it to change out and try in those local areas. It also goes to a minimum of 25 percent exploration giving it the chance to change things out in the later states and see what happens. If doing that enough occurs, then it will change the ideal action for a given state.





Convergence for Q learning started to occur for the smaller problem. For the larger problem, it was still active, although the gap between the most recent change and the previous was wider that any before indicating a slow down in changes. It does not appear to be converging though and the gap could easily be from a decaying learning rate and exploration rate. This is shown for an average of 3 runs. As rewards are discovered, clearly actions are being taken and more so in the beginning. There are so many changes and small rewards throughout that converging is difficult for an algorithm such as Q learning. No reward is necessarily larger than the others when it comes to cutting, burning results in lost reward opportunities, and although waiting has the highest potential reward, discovering that from any given point can be rather difficult when random action s are being explored, and the wait action needs to be taken so many times in a row to realize that. This is showing how Q learning can take quite a bit of time to converge if it does at all. When there's not a clear large increase in rewards, it can be hard to tell, and when increases come later in the game from exploration, it can often be too late for q learning since the learning rate is often decaying as well. Too high of a learning rate and the strategy will have trouble converging as well since there will be too much change occurring to settle on anything.





The policy selected by Q learning is shown above. In the smaller case, it is pretty interesting because it does seem to be favoring the cut action. It can be noted that it was able to learn to wait in the first state before cutting since no reward is given. Towards the end, the last 3 or so states, it figured out waiting was a decent strategy for an increased reward. Since the reward was so large compared to everything else, it was able to overcome the small value for gamma. It did learn some small clusters of actions it appears when it had cut after cut for chunks in the beginning. In the middle, it looks like it had no idea what was going on though. Overall, Q learning had trouble determining the reward structure to go after and was quite a bit off from the optimal strategy determine by value and policy iteration. For the larger problem, it is a mess. There is also some waiting occurring at the end which indicates the same as for the small problem. The same can be said for the beginning. The larger problem shows the difficulty that can occur when all the rewards are uniform. It is like the first problem except that there were at least some rewards to encourage Q to try new actions and update the Q table with. It did not create an environment that was conducive to Q learning still. Q did not do nearly as well as either optimal policy. Somewhere between a fourth and a third of what the optimal policies achieved for rewards. This shows the limitations that Q learning can have. Obviously, when the transitions and rewards are not clear, it had its place, but for well defined problems, it just does not stand a chance.

Algorithm	Size	Iterations	Time	Reward to 0	Reward 100 actions	Gamma
q	64	50000.000000	1.494956970	8761.303310	13417.969977	0.10
q	666	50000.000000	2.523960352	128826.451411	133504.118078	0.10

Limitations

Some limitations for this project were with Q learning mainly. Q learning could have been allowed to run for more iterations to allow it a better chance of convergence. Episode sides could have been played with to allow it to potentially learn patterns within the larger problem spaces better. The learning rate could have been adjusted for different decay functions and constant values as well. I did play with many of these things, but with no clear improvements and limited time, I was unable to get to these things in. Additionally, each of the problems could have had a different structure that would have been more helpful to Q learning. Shaping the problem to the learner is never an ideal situation, but it would have allowed Q learning to have a better notion of how it was performing for the frozen lake problem. This could have been achieved by either adding checkpoints throughout the frozen lake, or by having small positive rewards the closer to the goal a location was. Furthermore, hole could have had small negative values and spots next to holes could have been worth less than spots not next to holes. This of course would change the nature of the problem, but it would allow for Q learning to have a better shot at learning. The forest problem could have had exponential rewards for waiting that either summed to or peaked at the final reward for waiting in the final state. This would have encouraged Q learning to try waiting out and learn about the potential benefits of waiting at certain points. Again, this would have changed the nature of the problem, but it might have given Q learning a better change at figuring out a more optimal strategy than what it came up with.

Conclusion

Overall, for any given Markov decision problem having all the information makes it so an optimal policy can be determined, or at least some version of one can be within constraints. Value iteration and policy iteration showed this quite well for these problems. They were easily able to figure out the smaller problem sizes, and for the larger ones, it just took a little more time or a few more iterations. In fact, they are so well designed for these problems that sometimes they can be too good like in the case of policy iteration with the large frozen lake map. It was able to find so many optimal solutions it had trouble deciding. That is where it is on the implementation to determine when that optimal convergence is reached and to cut it off there. It was fun to see it run through the iterations and continually swap out 4 actions in the policy repeatedly. These are very powerful tools for well-defined problems. In the real world though, problems are often not well defined that we want to solve. They often reflect the situation that Q learning found itself in. A minimal amount of information with a set of actions and a set of feedback or rewards for those action and the new position one finds oneself in. This is much more like a real human being. Take the frozen lake for example, if the goal was a spot of non-frozen ground to allow a person to continue on their way, then it would make sense on a smaller lake that it would be easier to find and easier to avoid falling in. On a larger one on the other hand, that would be much more difficult especially if the goal is far out of site. The forest problem could have a similar analogy. Managing a forest and keeping it maintained could pay off in dividends of biodiversity, people getting to enjoy and explore nature, and plants being able to grow and thrive, but if that forest if going to burn down, collecting the lumber allows for a forest elsewhere to potentially stay in tact and for some profit to be made. That is a complex decision to make, and one that Q learning also had difficulty with. Sorting through rewards that were unclear to it and unknown to it. There was always the temptation to cut, and yet, the thought of what that forest could become might lead one to hold off and see what occurs. We do not always know the optimal action. If everything were able to be perfectly described in a Markov decision problem, then this space would not have algorithms like Q learning since value and policy iteration would be good enough. Exploring even if it results in a state of confusion can be worth the potential insights gained. I can see where different exploration rate, learning rate, and even algorithms in reinforcement could perform better and that this is just the beginning of the tip of the iceberg when it comes to reinforcement learning. Even though Q learning did not stack up to its opponents, it was working with a blindfold and noise cancelling headphones where they had omnipotent knowledge of the problem from the start, and still did a decent job in a case or two.