Markov Decision Processes

Jarrett Alexander

November 27, 2022

1 Introduction

The following experiments and evaluations were conducted to determine some of the nuanced differences within model-based methods (Value and Policy Iteration) as well as the larger differences with model-free methods (Q Learning) when solving MDP's. Further, experiments into the balance of exploration/exploitation and the impact learning rate has on both are evaluated when looking at model-free methods.

2 MDP's

2.0.1 Forest

The forest problem is defined with N different states[Rol22]. These states represent the age of the forest. N is arbitrary in this case. At each state, there is a chance P that a fire will burn down the entire forest. This will reset the forest to the very first state again. With chance 1-P, the forest will survive into the next year. At each state, the forest manager has 2 options. The manager can wait a year to give wildlife a place to live. Second, the manager can decide to chop down the whole forest and sell the wood for instant reward. There are 3 possible rewards given. First, a reward of 1 is given if the manager chops the forest down and sells all the wood. However, this resets the forest to the initial state again. Second, a reward of R1 is given if the forest reaches the final state. This value will be manipulated to observe different behaviors. Finally, a reward of R2 is given if the forest is cut down when it reaches the final state. N will be 100 for the following experiments to represent many generations of managers working on the same forest (this is not completely accurate as the forest age restarts after each cut/burn). R2 and R1 will be 4 and 2 respectively. P is 10%.

The dueling rewards should give this problem some interesting behaviors. Depending on the morality of the manager, R1 may or may not be bigger than R2. A higher R1 would lend itself to a manager who wants to preserve nature and is more likely to let the forest grow while risking a wildfire. A money driven manager may decide to cut the forest down every year regardless of the R1 and R2 values because they know they can get a reward of 1 no matter what every year that they do that. This would lead to an infinite horizon on this problem as well. It is possible for a manager to just cut the forest down every year and never let it grow to the final reward state if that final reward is not large enough. The introduction of the discount rate for the different algorithms will help here. Adding discount will allow us to compare infinite sums of rewards that otherwise would lead to only ever reaping short term reward. The most interesting states for this algorithm will be the final state (where r1 or r2 is applied) and a transition state somewhere along the way where the policy makes a change to achieve that better final reward state regardless of the short term reward it can get at that point.

Random exploration of this problem would lead to poor performance because it would almost guarantee a cut down forest before reaching the final state. Depending on reward, this would never reach the optimal reward. It would also not learn anything about the chance of a fire and would act as if they could not happen.

The fire provides a balance to the two conflicting rewards. While the manager may not want to chop down the forest because the final reward of letting it live is so great, the instantaneous reward of chopping down the forest is still greater than letting it burn to the ground. If the manager waits, there is a P chance that the forest burns down and they would receive nothing. However, chopping it down guarantees a reward of 1 in the short term but makes it much harder to reach that long term R1 goal. This balance is hard to strike and is greatly dependent on the values assigned to N, R1, R2, and P.

2.0.2 Frozen Lake

The frozen lake problem is an extension of the grid world style problem given in class and is a built in environment to Open AI Gym[BCP+16]. Similar to the in class example, the agent's goal is to reach the goal square without falling into any of the holes. Also, similar to the lecture, the agent is not guaranteed to move the direction that they intend due to the ice. This will mean that the optimal policy may not just attempt to find the goal in the shortest number of squares but that it will also be tasked with avoiding the holes with some distance. The number and location of the holes can be manipulated to make it more or less difficult for the agent. Also, the size of the space can be increased so that the policy that needs learned will have to cover many more states.

The agent always starts in state 1 which is in the top left of the state space. The positive goal state is in the bottom right of the state space. The default state size is a NXN grid with a random placement of holes and a reward square. There are only 4 actions that can be taken. The agent can attempt to move up, down, left, or right. The goal and hole states are end states for the game. The goal state is the only state with any reward. All other states have a reward of 1. Therefore, the only reward that can be achieved is 1 and that can only be achieved if a goal state is reached. All others result in 0 reward. 2 different state sizes will be evaluated below. For every experiment, a state size of 20x20 is used.

Random exploration would yield poor results on this problem. There are more holes than goal states and therefore random exploration is more likely to end with 0 reward than positive award. Further, the states that are next to the holes are going to be the most interesting states for this problem. Due to the slippery ice, it is possible to accidentally fall in the ice even if the agent is attempting to move away. Therefore, the policies should attempt to prevent the agent from ever entering these states in the first place. Depending on the map, this may not be possible.

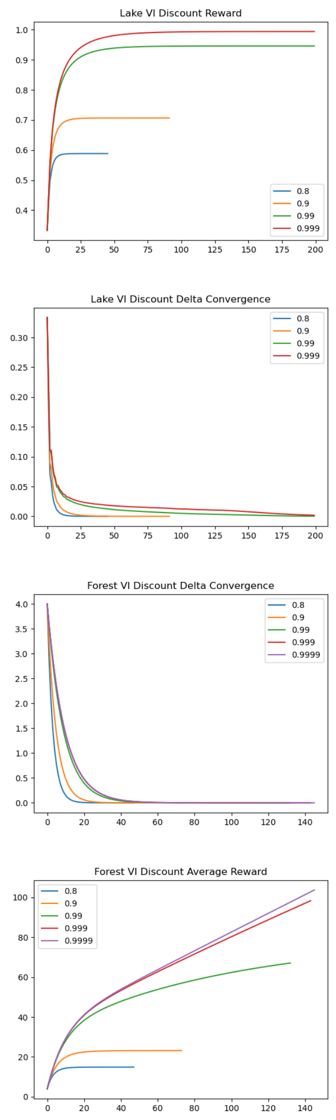


Figure 1: Value Iteration Discount Rate

3 Model Based Methods

To evaluate the differences in the different model based approaches, we look at gamma tuning and intermittent policies during training to determine the difference with each algorithm as well as when each algorithm converges. In the final section of this paper will be a comparison of run times and other performance metrics with model-free methods as well.

Average reward, average change in utility per iteration, and average value (utility) are all used to determine convergence. Average reward is important because it shows which values give the most reward. This is not totally reliable as this reward can be scaled with some of the parameters. Average change in utility per iteration is used to determine convergence because when this approaches 0, learning is nearing completion. Finally, average value shows which values help propagate utility across the different states the fastest. The charts for this are omitted to save space and referenced only when needed.

3.1 Value Iteration

The Hiive MDP Toolbox implementation of Value Iteration was used[Rol22]. It allows for tuning of max iterations, epsilon (an error threshold metric), and gamma which is the discount rate. For the sake of these experiments, only gamma is tuned. Max Iterations is set to a value that prevents any early stopping. Epsilon is set similarly to where it will not affect the algorithms ability to find the optimal policy.

3.1.1 Gamma Tuning

Gamma determines the amount of downstream value to add to the current state/action utility. A higher value will heavily weight an action that allows for any reward in the future. A smaller value will tend to look for actions that yield short term results. This will be explored in the following demonstrations. Gamma was looked at with values of .8, .9, .99, and .999.

Looking at figure 1, we can see that a higher gamma value increased the average reward taken on each iteration. The higher gamma value also allowed it to learn more quickly early on. This is required because there is only a single state that contains any reward for this problem. If the algorithm is able to propagate that reward as utility in each state, it will learn which direction to attempt to go. It also still allows the algorithm to learn that the hole states are states to avoid because the long term reward of those states is 0. Regardless of gamma, we see that moving to a hole state has 0 utility. This quickly allows value iteration to avoid moving to those states. This will be seen further when looking at the intermittent policies generated.

Finally, the forest problem is affected in a much different way from gamma tuning. Unlike the lake problem, reward can be had at each state. However, if the final stage is reached then a higher reward is given. As can be seen in the reward chart, a higher gamma appears to be approaching a linear increase as gamma approaches 1 (due to the infinte sum approaching an infinite value). This trend only stops because the updates to the utilities of each state reach a small enough value that it stops. By increasing gamma, the algorithm is able to see that allowing the forest to grow to the final state is more and more important. In this case, it is hard to determine if this actually is meaningful. When looking at the policies directly it is also shown that they are almost always a string of "cut" commands followed by a number of "leave" commands near the final state. The number of leave commands gets larger as gamma increases. This again is because gamma allows each state to weight future values much more than just the current reward. While this makes perfect sense, it creates an impossible to leave situation for a forest manager that is looking years and years into the future. The manager will always cut down the forest and never allow the forest to reach a mature age. The fix for this would be to shrink the time frame that the manager is focused on. A change of rewards would also dramatically change this outcome.

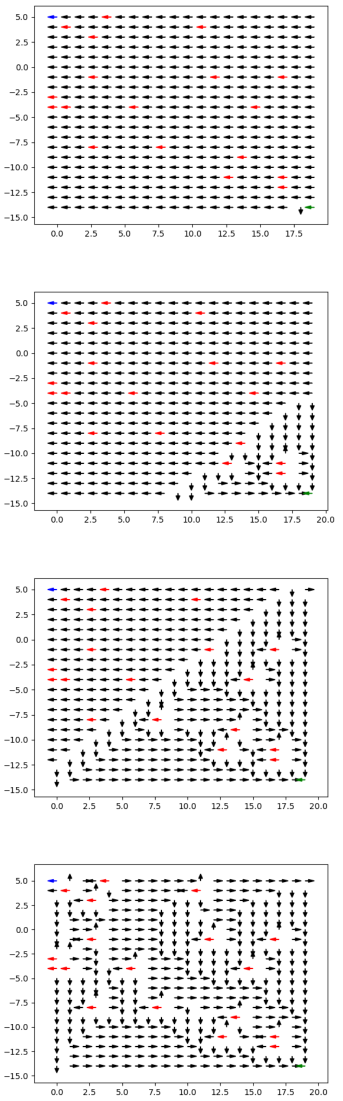


Figure 2: 1, 10, 20, and 200 Iterations

3.1.2 Intermittent Policy Evaluations

Gamma is important for value iteration because it almost acts as a learning rate for MDP's with very low reward space. Next the experiments show intermittent policies that demonstrate this sort of learning rate. For the sake of space, only the small lake and forest problems are shown.

Value iteration appears to learn in a "wave" for the forest problem. After a single iteration the only action that changes in the entire policy is the one right next to the reward state. This is moved to down which after a single iteration is equivalent to left as well as up. All 3 would have been valid. The utility at all states has not been updated yet and therefore it has a 1/3 chance of getting the reward for this state regardless of that direction as long as it is not left. As the reward propagates as utility across the board we see that the previously down arrows become right arrows which points the agent towards the rewards state more reliably. Also, the edge of the "wave" is always pointing down which likely means that there is a tie breaker occurring here. As it reaches more and more hole states the algorithm is also able to point states immediately away from those states. By iteration 20, the policy is almost totally optimal. This demonstrates this algorithm perfectly as it is looking at each state and all actions and then looking for actions that provide the most utility. That utility only exists near the reward state until enough iterations have passed. A higher gamma tends to prevent falling in the hole states faster. They remove the ability for any long term reward. A high gamma puts high emphasis on long term reward and therefore is able to determine much quicker that falling in a hole is a bad thing.

The forest problem's policy is a string of all cut states followed by 20 leave states. Forest does not change at all with subsequent iterations. At first this is strange but it makes sense because it is able to see the short term and long term reward at each and every state/action pair. This would immediately allow the algorithm to determine that cashing in on short term reward is the best action until you are within N years of the final state in which the long term reward becomes much more valuable. This is discovered for each state within a single iteration because there are only 2 actions and the discounted long term reward remains fixed. For the problem size 100, this "leave" action begins at year 80. The remaining 20 years are left alone.

3.2 Policy Iteration

The same experiments and setups from above were ran on Policy Iteration as well. Again, the hiive mdptoolbox package was used[Rol22]. Max iterations are still able to be used for this algorithm so it was set to a value that would not hinder performance.

3.2.1 Gamma Tuning

Similar results across both problems were seen for discount rate tuning with Policy Iteration. It showed the best performance for higher values of gamma in both problems. Again, gamma serves a similar role in policy iteration as it does in value iteration. Therefore, it is expected that the results are the same. A higher gamma allows the algorithm to quickly determine where the reward is and how to get to it in the case of the lake problem. For the forest, it is able to determine when and why the reward is good to take in the short term over the long term reward provided by the "leave" action.

Different to value iteration is the shape of the curves created by policy iteration. The charts are much less smooth because policy iteration is able to learn in much fewer iterations than value iteration. It approaches the same average rewards and average utilities as value iteration but does so in much few iterations. This tends to be faster and we will explore that in a later section.

Also, a high gamma value is able to terminate very quickly. The higher gamma improves extremely quickly and finds the optimal policy for the lake problem and then stops because there is nothing left to improve on. Lower gamma values like .99 do not stop quickly because they are still moving that rewards across the board. This high gamma is important early on in learning as it weights all possible states that could be landed in based on the current state and action given by the current policy. The reward for almost every state is 0 and therefore does not contribute during the first (few) iterations. This will become more clear in the next section as we look at each iteration and how it explores the state space.

A higher gamma once again led to a higher average reward for the forest as well. Because the discount rate allows the algorithm to compare infinite sums, it is able to determine that there is a greater long term expected output near the final state if the forest is left alone. Interestingly, the curves for gamma show that the average reward and value are actually static. This is not the case. The scale of the chart is so large that it is hard to tell that they are actually increasing. The delta convergence chart shows that there are updates. A higher gamma takes longer to converge but allows for a higher average reward.

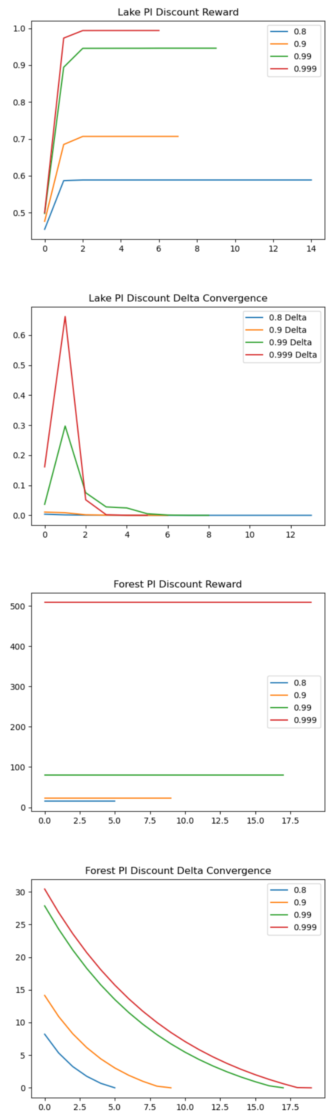


Figure 3: Policy Iteration Discount Rate

3.2.2 Intermittent Policy Evaluations

After the first iteration, there was a single change to the policy similar to value iteration for the lake problem. The only change moves the state to the left of the reward state to be down instead. The reason for this is the same as above. However, after only 2 iterations, every non-terminal state has been updated. Also, the policy is avoiding the hole states. Further, they are gaining utility from the reward state which is reachable by only moving to the right near the reward state (due to the randomness of moving). Other states have changed to moving up as well instead of left. This is once again due to the fact that each states new action is determined by a combination of possible future utility as well as the current reward. The utility of each state is calculated at the given iteration and this is what is able to cause a mass change in updates across the entire policy.

Unlike value iteration, policy iteration is unable to find an optimal policy within a single iteration for the forest problem. Each state has a utility of just the reward that can be obtained in the short term except the final state. This means that policy iteration starts by saying "cut" for every state except the final state. As more iterations occur, it is able to see that the "cut" state closest to the final state should likely be a "leave" state instead because that has a higher utility. As can be seen, with more iterations, more "leave" actions are added to the end and it is less likely to cut down the forest. This is a major difference in PI and VI.

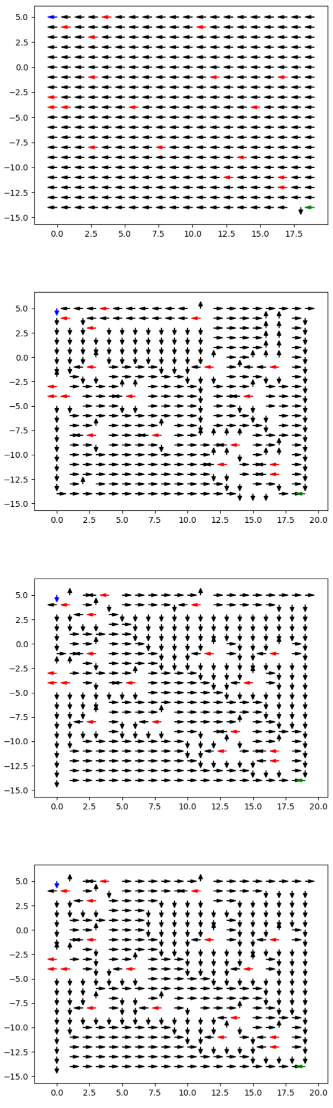


Figure 4: 1, 3, 5, and 10 Iterations

4 Q Learning

Once again, the Hiive MDP Toolbox implementation was used for the experiments with Q Learning[Rol22]. There are many available hyperparameters that can be tuned in order to experiment with the behavior of this algorithm. Unlike Policy and Value iteration, this algorithm is model free and does not use any prior knowledge of the environment in order to generate a policy. It must learn the environment and the optimal policy as it goes.

4.1 Hyperparameter Tuning

Each parameter was tuned separately and then used for each following experiment. This was repeated backwards as well to ensure that all parameters were optimal for each other training. The following experiments were done on the lake problem and the forest problem. The evaluated parameters were the discount, learning rate, learning rate minimum, epsilon (exploration), and epsilon minimum. The experiments with learning rate are omitted for space.

Similar to before, a combination of the average reward, average change in utility, and average utility will be evaluated. While average reward is the most important, it is also important to see how quickly this is achieved. Average utility of each state will also demonstrate how quickly Q Learning is able to discover the reward states and take advantage of them. The reward chart is a rolling mean of 50000 iterations to help smooth out the noise. A new metric is added when evaluating some parameters for Q Learning. As was mentioned before, there are many optimal policies that induce an inefficient path to the final reward state simply because there is only as single reward state. To detect "efficient" policies, each policy is run to see how many steps it takes to reach the final reward state. This is set to a limit of 1000 steps. If a hole state is found, a value of 2000 is given to show that this is worse than simply wandering around.

4.1.1 Discount

Similar to the other algorithms discount places value onto the future. The difference with Q Learning is that it is unable to know exactly what states lead to reward without exploration. Therefore, differing behavior occurs when modifying the discount rate.

For the lake problem, the highest average utility is once again the highest discount rate. While this makes sense (it is able to update values the quickest due to that rate), it is also misleading. Every non-hole state in the lake problem is able to achieve reward at some point. As long as you move away from the holes when next to them, you are guaranteed to eventually get the reward. While this is "equivalent" to the policy that directs the person to the reward the fastest, it is not "optimal" because humans would prefer to get the reward quicker. This is a limitation on this problem that could be fixed by creating a slightly negative reward at each iteration. Looking at the average reward chart, it can be seen that discount rates of .8, .9, and .95 are all pretty equivalent at finding optimal policies which have a high likelihood of achieving the reward. The discounts of .99 and .999 suffer from the described problem. They are placing so much weight on future returns that they don't really care if they are able to get there quickly or not. For this problem, a discount rate of .95 is used because it is able to "spread" the reward across the states as utility quickly but it does not put too much weight on future returns such that an inefficient policy is created.

For the forest problem, a similar pattern shows up as to value iteration. As discount increases the average value of each state increases. However, the average reward is much lower for states as discount increases. As discount increases the final state gains more and more weight because it is able to cash in on that large reward. This is not optimal however as the manager misses out on reward along the way. Therefore, the highest average reward occurs when the discount rate for Q Learning is lower. That forces the manager to chop down the forest more often and gain the single point of reward at each iteration. The discount rate of .9 appeared to approach the same asymptote as the discount of .8 but still was able to spread the final reward state out more evenly among the rest of the states. .9 will be used going forward for this MDP.

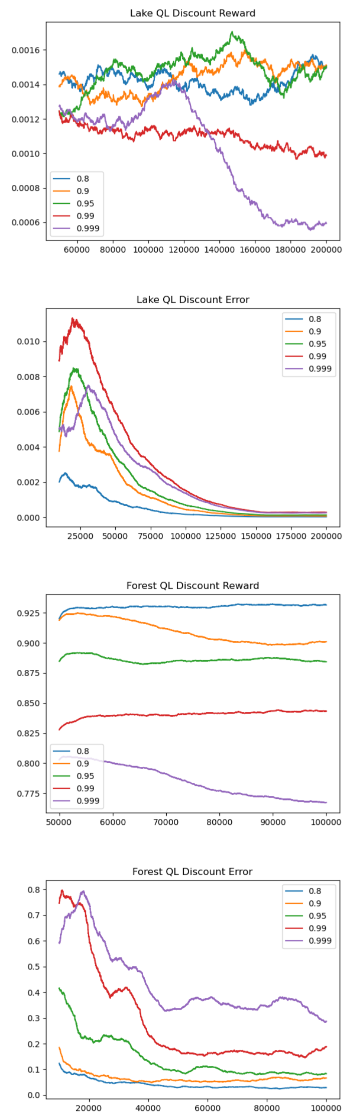
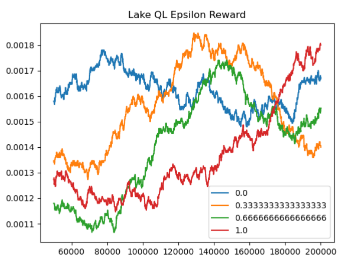


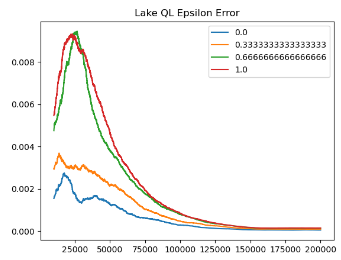
Figure 5: Q Learning Discount Rate

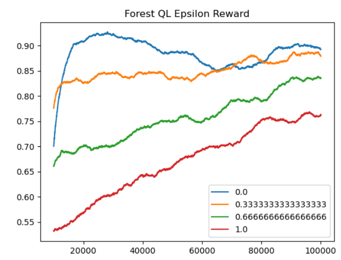
4.1.2 Epsilon

Epsilon is the likelihood that the action taken at a given state is random instead of the current best action. Basically, this is the exploration parameter. Similar to alpha, there is a decay parameter which lowers the exploration as time goes on. This decay is also set to .99999. While some experiments were completed with this decay rate, the impact of epsilon and epsilon minimum were more interesting in this case. 4 values of epsilon were tried. .1, .33, .66, and 1. There was also an epsilon min of .1.

As is expected, a larger value of epsilon caused the most exploration early on in the search for an optimal policy. Looking at the error function, we can see that the largest epsilon starting values are sorted from the top down the largest changes early on. However, for the lake problem, this does not seem to have any long term impact on the final policy. The higher exploration values actually performed the worst on the early iterations of the training. The lowest values of epsilon seem to be able to find the the correct policies to maximize reward in this case without major exploration. Most states in this problem will either be aimed right or aimed down because that is the way the agent must travel in the long term. Further, the starting policy is to always move left. By labeling all of them left knowing that most of them will change to the right or down, there is no real need to keep exploring once this change happens. The algorithm does not need to experiment with left again or up because it has already tried them and shown that they are not correct unless near a hole state. It also does not take long to determine the location of the hole states and how to avoid them. And finally, there is already a 2/3 chance that the agent will not move the direction that you desire it to move. This induces further exploration in the agent without needing to force it any other direction. Therefore, a lower epsilon makes sense here. The final policies for each starting epsilon are pretty much identical by the end of the training. Smaller epsilons just get there more quickly for this problem. .33 will be used for this MDP going forward.







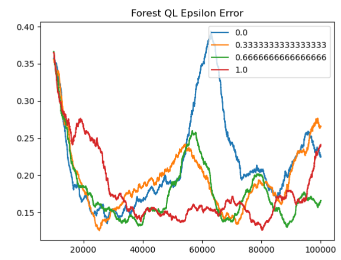


Figure 6: Q Learning Epsilon

Nearly identical results are seen from the forest problem as well. A higher epsilon seems to hinder the algorithms ability to actually learn anything meaningful because it is "wandering" around too much. Taking random actions sometimes is important because it lowers the odds of landing in a local optima. However, this also means that you will likely be exploring a lot of space that may or may not actually reveal anything beneficial. In the case of the forest problem, there are only 2 actions. If you have found one to be better and then explore another action for a given state and it appears to be worse, it doesn't really make much sense to try it out more often than the current best action. Starting with a high epsilon would do this. Again, the final policies output by these different starting locations are all pretty much identical. They are a string of all "cut" actions followed by 5-10 "leave" actions. The speed in which the algorithm approaches this appears to be the difference when action space is so low. .33 was also a sweet spot for this MDP.

A possible way to mitigate this issue with MDP's that have small action spaces is to increase the rate in which the epsilon decays down to the minimum. Small experiments were done with this and it showed to improve performance when using high values of epsilon. Early on it is able to explore all possibilities more effectively and then only explore here and there once near an optima. This is very similar (or identical) to the results seen on simulated annealing from the randomized optimization assignment.

Learning Rate and Exploration While the learning rate experiments are omitted, it did have a large impact on the this epsilon and the minimum epsilon. A high learning rate paired with high exploration generated quite a bit of noise in the final output policy for the lake problem. It would consistently find inefficient paths to the reward state because at each iteration it was learning too much from the new exploration. A balance was required here because a low learning rate with high or low amounts of exploration did pretty much nothing. It would not learn anything from future iterations. Also, a low learning rate with high epsilon would reliably find the efficient path to the reward state. The problem was that it took many more iterations to do so. Therefore, epsilon and its minimum value cannot be evaluated in a vacuum. They are heavily tied because high learning rate will learn quickly and efficiently if and only if the algorithm is setup to prefer exploitation over exploration. For example, say the algorithm is starting with a policy that is already almost totally correct. It should require only a small change. Therefore, exploration should be low and the learning rate can be very high. However, if starting with a very incorrect policy (like the one that lake starts with), a high exploration and low learning rate must be used. This should be eval-

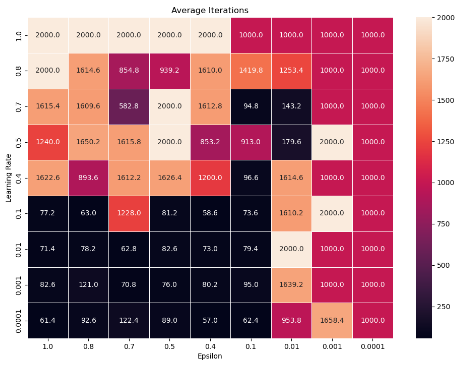


Figure 7: Learning Rate / Exploration Average Steps

uated for every single problem and a balance must be struck here anytime Q Learning is applied. Failure to do so will require much more training time or will result in noisy and inefficient policies.

Figure 7 is a heat-map of different exploration and learning rate parameters. The value on the map is the average number of steps it takes the agent to complete a run using the policy created by those parameters. As can be seen in figure 7, when given 100000 training iterations, a high epsilon and low learning rate pairing reliably found efficient policies towards the reward state. Low exploration would tend towards "wandering" around and never finding a reward state but also never falling into a hole. Finally, high learning rates and exploration would reliably steer the agent towards hole states because they are trying to explore so much and learn so much from single iterations.

4.1.3 Epsilon Min

The minimum epsilon value determines how much exploration will be occurring near the end of the training as long as the decay allows it to get there. This has a very large effect on the outcome of the algorithm because it determines exploration during the entire run, not just the start. For this, 4 values were used. 1, .5, .1, and .01. If epsilon min is larger than epsilon at the start then epsilon min will be the starting epsilon.

This parameter had a large impact on the lake problem. High minimum epsilon values did allow the algorithm to converge cleanly to an efficient policy. With a high learning rate, there are many inefficiencies in the policy created with the high exploration rate later in the algorithm. This balance was discussed in the heatmap. A lower learning rate is required when the minimum epsilon is high. It did tend to learn the most early. This is a perfect demonstration of the trade off between exploration and exploitation. While the algorithm is able to explore the entire board and learn from the entire thing, it is putting too much weight into the new observations and never really improving. However, with a lower long term epsilon, the algorithm can take advantage of that early exploration but then only change a few things here as is completely necessary in the long term. The policies here demonstrate this well. With high values of epsilon there is quite a bit of randomness pointing in many different directions. As it shrinks, there are fewer and fewer actions that appear to be random. Low values of epsilon min have a similar issue. There is nothing forcing exploration and then many of the actions are not updated. This shows that a value too large or too small are both going to hinder the convergence of the algorithm. .1 will be used going forward because it strikes a balance between high average reward and still allows for exploration during later phases of training.

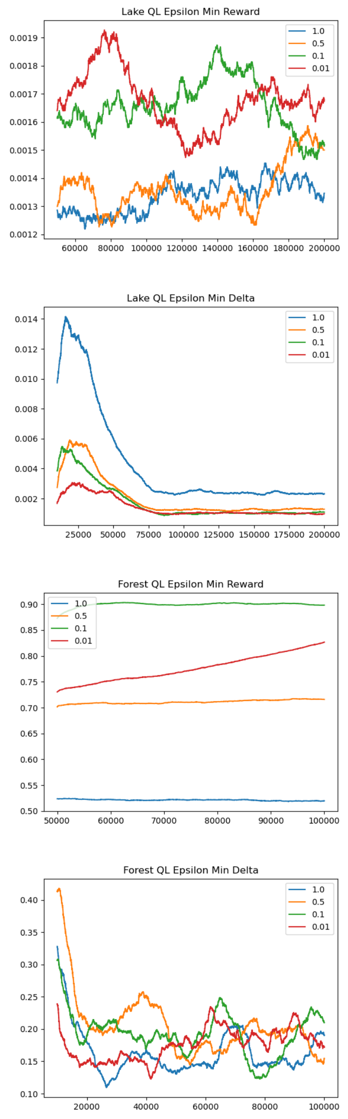


Figure 8: Q Learning Epsilon Min

The policy found with a low exploration factor is full of "leave" actions which were shown to be suboptimal through both Value and Policy Iteration. The algorithm is unable to explore enough and determines that it should just leave all of those states as "leave" actions. Higher values of minimum epsilon did cause the algorithm to perform poorly as well. Due to the high exploration, the algorithm found that cutting down the trees every year was optimal for pretty much every year. This reduces any long term possible reward because the forest will never reach the final state. The minimum values of .1 and .5 allowed the algorithm to do both. They also had the best average reward out of the 4 options. .1 will be used going forward.

Lower values of learning rate would help improve this problem. However, back to exploration vs exploitation, it takes much longer to converge when exploration is high but learning rate is low. A balance must be struck for each individual problem and lots of tuning is required.

4.2 Intermittent Policy Evaluation

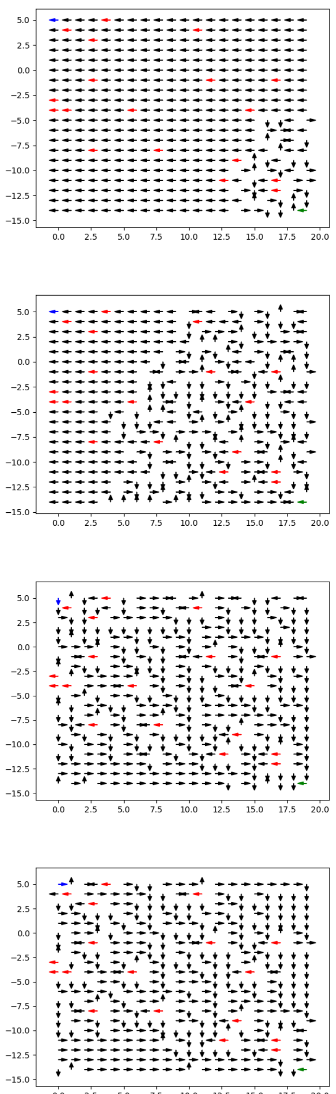


Figure 9: 10000, 20000, 100000, and 200000 Iterations

Similar to the value iteration, Q Learning seems to learn the optimal policy by updating values near the reward start outward. This makes sense because it is the only source of reward for the lake problem. Differing to VI and PI however is the direction optimality of the policy as it moves away from this reward state. Due to the randomness and the exploration required by Q Learning, we can see that the actions are not updated perfectly each time. They are updated, but maybe to a direction that is suboptimal. This is a benefit however. Without these "incorrect" updates, Q Learning may not end up learning the entire environment that it is trying to map. As more and more iterations go on, it is then able to learn which actions are actually best at a given state. With a large state space like a 40x40 lake problem, a lack of exploration guarantees that the output policy is suboptimal. Forcing incorrect exploration again allows the algorithm to learn from the previous iterations mistakes.

Forest showed the exact same outcome. As it iterates, it is able to learn more and more about the long term reward that comes at the end of the run. It also learns about the short term reward from chopping down the forest. There are scattered "leave" actions throughout states and these slowly shift over to "cut" actions. Similarly, "cut" commands are added early on near the final state. These are removed by the 1000000th iteration of the algorithm and the policy is very similar to those found during value and policy iteration.

5 Comparison

As is expected, Value Iteration and Policy Iteration outperform Q Learning on model based problems. The ability to know the entire model of the problem and accurately estimate utility at each iteration is a large benefit. They are also able to find the optimal policy every single time and they both converge to the same policy. This is something that we see Q Learning was unable to do. It did not land on the same policy for the forest nor the lake. This is shown in the number of leave states being different and the average number of steps to reach the reward state being larger. Also, Q Learning required much more effort in tuning. Comparatively it took significantly longer to tune each parameter and strike a balance among them. PI and VI only required some minor tuning to the discount rate for problems with sparse reward.

The model free methods were significantly faster as well. This is again because they are able to model the environment and represent the correct utility with much fewer iterations. The idea of "exploration" is much less apparent with these algorithms. VI and PI were able to represent the correct utilities of each state/action in under 100 and 20 iterations respectively. It took Q Learning anywhere between 50000 and 200000 for the given problems.

100 Year Forest			
Algorithm	Time (s)	Training Iterations	Leave States
VI	.006	100	20
PI	.007	10	20
QL	3.6	50000	19
20x20 Lake			
Algorithm	Time (s)	Training Iterations	Steps to Complete
VI	.003	100	131.6
PI	.001	20	137.4
QL	20.98	200000	156.4

Table 1: Algorithm Comparison

For the forest problem, PI and VI performed almost identically. The difference comes in during the

Lake Problem. PI is able to converge faster as it is only dealing with updating policies and calculating utility from there. VI works opposite which requires more iterations to converge. This is still a small difference in time. This does scale however. On a separate test of a 40x40 lake problem, PI performed significantly faster than VI again at roughly .03 seconds for PI and .08 for VI.

6 Conclusion

Value Iteration and Policy Iteration are effective tools for solving MDP's. They take advantage of the ability to see the model of the problem and that allows them to converge significantly faster than model free methods. Policy Iteration seems to be a faster algorithm typically because it is only concerned with calculating a value given a policy instead of the opposite. This allows it to solve MDP's like the lake problem very quickly.

While it is harder to tune and requires more training time, Q Learning is able to applied to problems that VI and PI cannot. Q Learning can discover the underlying model of the problem on its own. VI and PI require it as input. Therefore, complex to model problems or problems that have continuous state spaces are much easier to solve with Q Learning or variants of it. VI and PI can be applied here but a lot of work must be done to the problem beforehand to turn it into and MDP (and this may be more work than it is worth, or will remove nuance from the problem). Q Learning is affected heavily by input parameters as well as the starting policy. If it is impossible to give a ”good” starting policy, Q Learning needs to be forced to explore more than it would if the starting policy was near optimal. Domain knowledge and input bias to the model are more important when dealing with model free methods than model based methods.

Overall, Policy Iteration performed the best for the problems discussed here. Q Learning however has a much higher ceiling for problems that it can solve but it requires much more tuning and time to work effectively.

References

[BCP$^+$16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[Rol22] A. Rollings. mdptoolbox, hiive extended remix. `https://github.com/hiive/hiivemdptoolbox`, 2022. Accessed: 2022-11-27.