CS7641 Machine Learning Fall 2021
Project 4 : Markov Decision Processes

Ruhan Li
(rli445)

Abstract

Two Markov Decision Processes (MDP) problems were chosen with a "small" number of states and a "large" number of states, respectively. Three algorithms, including value iteration (VI), policy iteration (PI), and Q-learning were applied to solve these two MDP problems.
The implement of algorithms and curves are based on python and with public libraries, including hiive.mdptoolbox, gym, numpy, and matplotlib.
Link of code: https://github.com/Hanlarious/ML/tree/main/A4

I. Brief Introduction

Problems

First problem: Frozen Lake (small grid world problem). In this problem, there's a start point, a goal point, and multiple holes and normal walkable lake surfaces. The purpose is to walk from the start point to the goal point without falling into the holes. Once the agent reaches the goal point or falls into the holes, one episode will be ended. If the goal point has been reached, the reward will be 1, otherwise 0.
I formulated this problem using the gym library. I wanted to observe the behaviors with different state sizes, therefore, I implemented a smaller size 8*8 (16 states), and a bigger size 50*50 (2500 states).
Second problem: Forest Management (large non-grid problem). In this problem, the agent will play a role of forest manager who needs to decide whether to maintain the forest for wildlife or to cut woods for profits. By default, maintaining forest gives the reward value of 4 whereas cutting woods gives the reward value of 2.
I formulated this problem using the hiive.mdptoolbox.example library. To observe the behavior of different state sizes, I generated a smaller size of 1000 states, and a larger size of 10000 states.

Algorithms

Value Iteration: starts from the end and works backward. [1] I used "Utility" as a metric to evaluate the policy, which adds up the current reward if one step is taken and the discounted (gamma) future rewards. We aim to get the maximum reward, so we always want to take the step with the maximum utility. The convergence criterion for VI is when the utility no longer changes or the change is very small. In other words, the variance (delta) is smaller than the threshold (0.0001 by default of the library).
Policy Iteration: is slightly different. It starts with a random policy, and update the policy to improve the utility. In other words, it calls the written policy evaluation and then do the greedy policy improvement[1]. Similar to VI, it also aims to get the maximum rewards. The convergence criterion of PI is when policy no longer changes.
Q learning: is different from the other two model-based algorithm, it is a model free algorithm that "seeks to find the best action to take given the current state."[2] Although all three algorithms aims to maximize the total reward, Q learning learns polices from outside actions, so it does not require policy. When the agent learns an optimum reward, it will update it to the Q table that it maintains. I tuned the epsilon and learning rate for this algorithm.

Why interesting

The major advantage of reinforcement learning is that it can learn actively, and can obtain feedback from the environment. This is also one of the crucial advantages of reinforcement learning compared with traditional machine learning. Another advantage is that reinforcement learning algorithms learn policies that can be executed in a dynamic environment. Therefore, it is more artificial intelligent than other supervised and unsupervised learning.
The advantages of reinforcement learning are exactly what are needed to solve the Frozen Lake and Forest Management problems, since most of the time we want the solutions to be generated in a more intelligent way. As I understand, one of the major features of these two problems is that they require solutions in dynamic environments. Because each step would have further impact on the next step. I think it will be interesting to see what parameters will influence policymaking, and how they will help to find the optimal policy.
On top of that, because we can generate problems with random maps, it will be interesting to see the results of different maps. For instance, for the Frozen Lake problem, with different random seeds, there can be different numbers of holes located in different grids. I believe that the difficulties are

[1]The greedy approach is to apply the state value function after policy evaluation to look one_step_lookahead on each state, that is, to find the action value function (q(s,a)) of the current state.

different.
Last but not the least, there are some uncertainties for each problem, which make them more interesting. Specifically, for the Frozen Lake problem, there are chances to slip, which means even if you have an optimal policy, there are still chances that the agent fall into the hole because of slippery. For the Forest Management problem, at each state, there is a probability that the fire burns the forest and a new generation of trees will grow.

Challenges

The gym library seems to have problem dealing with large number of states. Specifically, when I was evaluating the discount factor vs utility, number of iterations and execution time for the lake size 50*50 (state size 2500), it got stuck at discount factor = 0.7 for more than 48 hours without returning me anything. I checked my implementation and switched to another computer with higher performance, but got stuck at the same point for another couple of hours. I also modified the size of lake to 32*32, but got stuck at discount factor = 0.6. I did not continue to shrink the lake size since I want some obvious changes in sizes to observe behavior.
Originally, I wanted to put the metrics, including utility, execution time, and number of iterations to converge, for the bigger size problem and smaller size problem within one MDP problem together to compare, but I later found that because of the huge differences in values, sometimes the curves were smoothed into straight lines. For example, for the Forest Management problem, when I plotted the change of utility and execution time while epsilon changed for Q learning, I got two straight lines that were far apart for two different sizes, but when I plotted them separately, I found that they actually have ups and downs. So when I saw two straight lines, I would do an extra step to separately plot them to see if there's any information that I missed. I will have further detailed explanation later when we see the graphs.
The convergence of Q learning is hard to characterize. Therefore, I had to take the best policy that given by VI and PI as a baseline, and try to get Q learning experiments to yield the same. However, no matter how I tune the parameters, my Q learning has way worse performance than VI and PI.

II. Frozen Lake

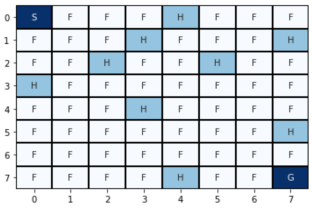


Figure 1: Frozen Lake (8*8)

Value Iteration

As I mentioned in the introduction part, I formulated two random Frozen Lake problems and fixed the random seed to ensure the constant results. The 50*50 lake is too big to generate a heatmap, but it will be similar to the 8*8 one (Figure 1), just a lot bigger.

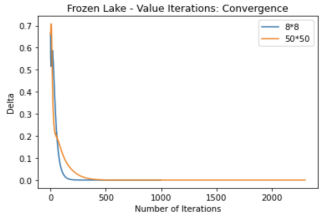
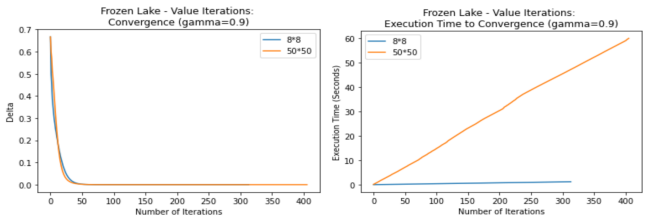


Figure 2: VI: convergence

The return value suggests that the 8*8 (smaller size problem) converged at the 997^{th} iteration, whereas the 50*50 (larger size problem) converged at the 2304^{th} iteration.
I wanted to compare their execution time to convergence, but I met another problem with the public library. It looks like the performance with high gamma value is not stable. I have built a list to store the execution times for each iteration, but with gamma = 0.99, only about a half of the execution times had been put into the list[2]. To verify whether it's a library defect or my poor implementation, I ran exactly the same code on another computer, and about double of the execution time records were stored in the list[3]. I double-checked my code to make sure I implemented it correctly, but I could not figure out what else could trigger this problem. Therefore, in order to see the behavior, I changed gamma to 0.9, with which the correct number of execution time were added to the list.



(a) gamma=0.9 (b) gamma=0.9

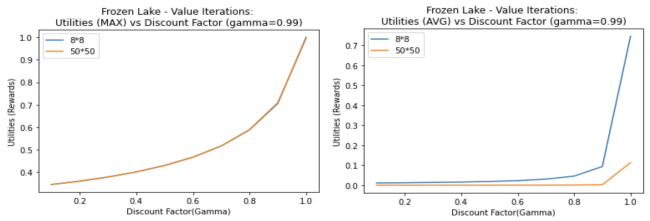
Figure 3: VI: convergence & time

The graphs show that with a smaller gamma, the convergence curves of the smaller size problem and the larger size problem becomes more similar (Figure 3a), and their numbers of iterations for convergence have been reduced to 314 and 406 respectively.

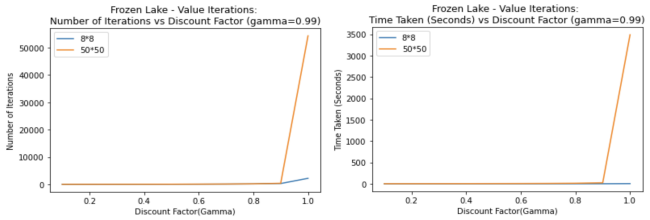
[2]For the smaller size problem, which converged at 997 iterations, I supposed to have 997 records of execution time, but I only had 314 records, which equals to the number of iterations of the convergence when gamma=0.9. I thought the library could not work with gamma greater than 0.9, but it worked just fine with PI and Q learning.

[3]I supposed to have 997 records, but got over 2000 records.

Inspired by this, I thought it would be interesting to see the change of utility, execution time, and number of iterations to converge with different gamma values.



(a) Utilities(MAX) (b) Utilities(AVG)

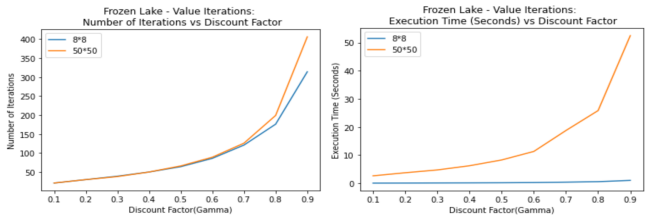


(c) Number of Iterations (d) Execution Time

Figure 4: VI: Gamma(0.99) vs metrics

For the two state sizes, it appears that the maximum utilities are almost the same (Figure 4a), whereas the mean utilities are pretty different (Figure 4b). The smaller size problem has much higher average utilities, and the trend is more obvious when the gamma is bigger, especially after 0.8. This is very reasonable since the larger the lake, the more holes there are and the longer the route to the goal point. So it is more likely that the agent will fall into the hole in a larger map.

For the number of iterations to converge and the execution time, because of the sharp increase after gamma = 0.9, the curves of gamma smaller than 0.9 are compressed into a straight line (Figure 4c&d). Therefore, I decided to plot the curves for gamma smaller than 0.9.



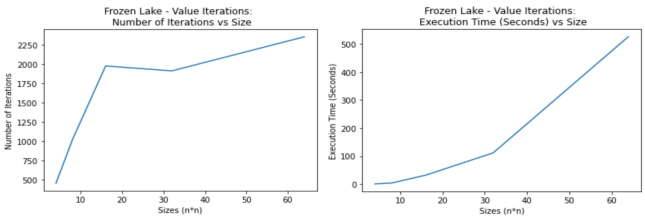
(a) Number of Iterations (b) Execution Time

Figure 5: VI: Gamma(0.9) vs metrics

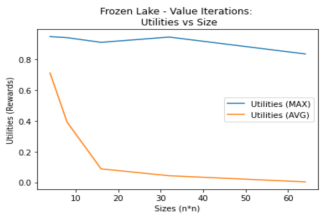
The graphs have verified my guess: there are different behaviors between two different sizes when gamma is smaller than 0.9, especially the execution time. With a larger lake, the execution time gradually increases until gamma = 0.8, and surges after that (Figure 5b). The curves of number of iterations to converge do not have that obvious differences (Figure 5a). Curves for both sizes are gradually increasing, while the larger size problem has slightly larger number of iterations to converge.

Furthermore, I noticed that gamma = 0.9 is a watershed where metrics values surge after that (Figure 4). As I understand, the discount factor gamma is applied to trade off the importance of current reward and future rewards, the greater the gamma value, the more important future rewards. Since the ultimate goal is to maximize the final reward, and with higher gamma values, especially greater than 0.9, the utilities are much higher, I decide to set gamma to 0.99. The drawback is that the number of iterations to converge, and the execution time are a lot longer than those of smaller gamma values. I think it becomes a problem of tradeoff again. In the situation where I only care about the rewards, I will definitely choose the gamma value of 0.99, but in the situation where execution time and number of iterations matter as well, I think other gamma values that are equal to or greater than 0.9 are also acceptable.

I do not believe that two different state sizes can tell the whole story. To understand the behavior more comprehensively, I plotted the metrics curves from size 4*4 to 64*64. The graphs suggest that the number of iterations to converge



(a) Number of Iterations (b) Execution Time



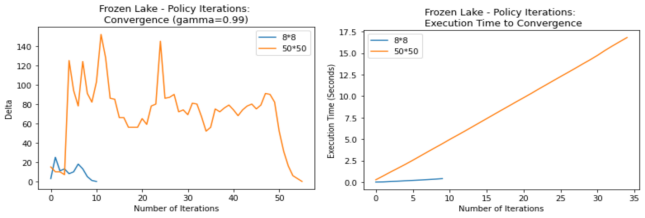
(c) Utilities

Figure 6: VI: Sizes(n*n) vs metrics

rapidly increases until around size 16*16 (256 states), and then drops a little bit, and then slowly increased after size 32*32 (Figure 6a). The increase rate of execution time becomes larger after around size 32*32 as well (Figure 6b). For the utilities, the maximum utilities do not vary a lot while state sizes increases, whereas the average utilities dramatically drop to a quite low value with size 16*16, then continue to drop as size increases(Figure 6c).

Policy Iteration

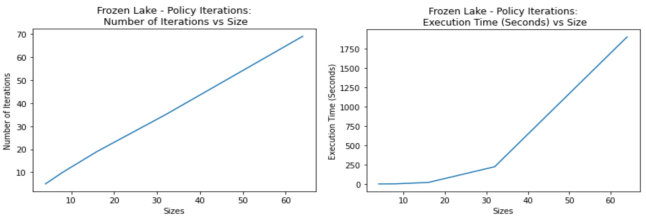
It is obvious from the convergence graph that the delta has changed dramatically before convergence, especially the problem with larger number of states (Figure 7a). I think this



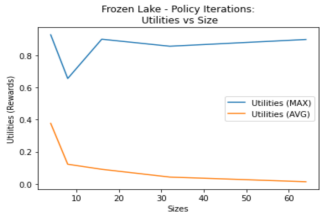
(a) criterion: delta (b) execution time

Figure 7: PI: convergence & time

is understandable since the algorithm repeatedly does "policy evaluation" and then "policy improvement", the policies can sometimes be terrible in performance. The execution time curve suggests that larger size of states takes a lot longer to converge.



(a) Number of Iterations (b) Execution Time

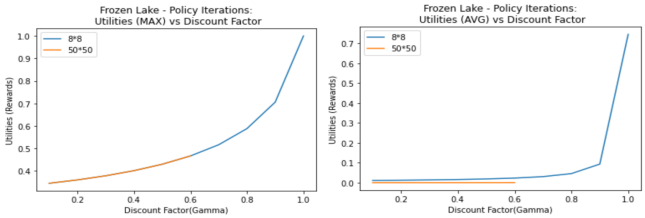


(c) Utilities

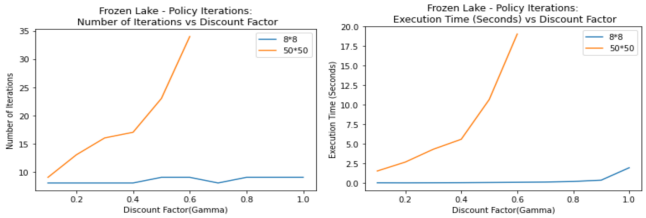
Figure 8: PI: Sizes(n*n) vs metrics

I also plotted the metrics with state sizes from 4*4 to 64*64 for PI (Figure 8). As expected, the number of iterations to converge and the execution time increases steadily as size increases. And the mean utility drops significantly with size 8*8 then steadily decreases as size increases.

When plotting the metrics vs different gamma values, I had difficulties getting values after gamma 0.6 for the large size (this is what I mentioned in the challenges part). Although I used multiple computers to run, and shrank the state size, I still could not get complete values. I guess this is because the gym library does not support state sizes greater than 8*8, therefore, I hard-coded a bigger map, which is too big for the library to handle. I have some supporting evidences for my guess. I plotted the values for the larger state size when gamma is smaller than 0.6, and compare them with the smaller state size, hoping to get some clue. As is shown in Figure 9c&d, the number of iterations to converge and the execution time increase exponentially after the gamma value of 0.4, therefore, I infer that the increase may



(a) Utilities(MAX) (b) Utilities(AVG)



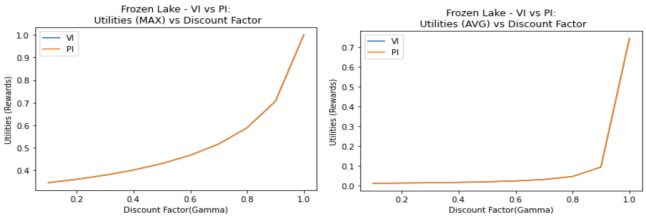
(c) Number of Iterations (d) Execution Time

Figure 9: PI: Gamma vs metrics

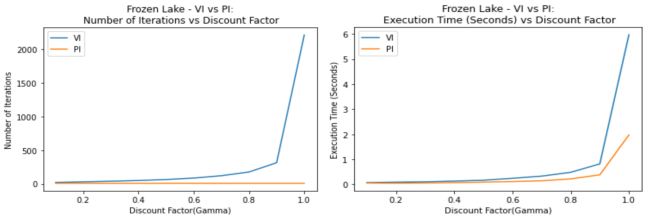
become even faster as the gamma value increases.

VI vs PI

The behaviors of VI and PI have some similarities and some differences, I think it is more interesting to put them together to compare. I compared metrics with different gamma values



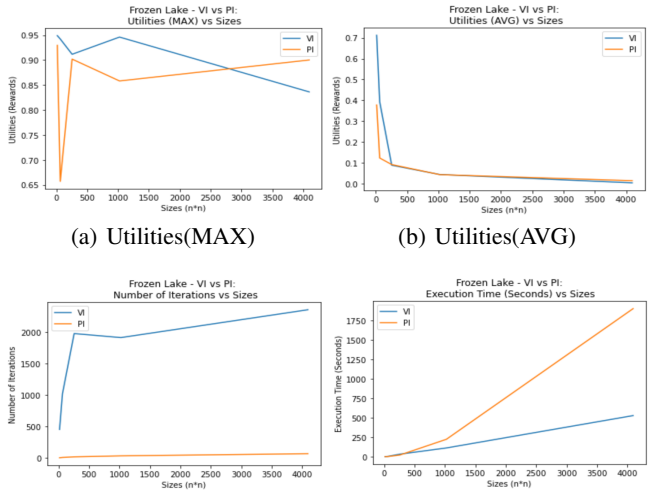
(a) Utilities(MAX) (b) Utilities(AVG)



(c) Number of Iterations (d) Execution Time

Figure 10: VI&PI: Gamma vs metrics

first. For the utilities, VI and PI have the same curves (Figure 10a&b), which suggests that they probably return the same optimal policy. However, for number of iterations to converge and execution times, VI takes significantly longer time and more iterations to converge, especially when gamma is greater than 0.9 (Figure 10c&d). This tells me that gamma has greater impact on VI than on PI.



(a) Utilities(MAX) (b) Utilities(AVG)

(c) Number of Iterations (d) Execution Time

Figure 11: VI&PI: Sizes(n*n) vs metrics

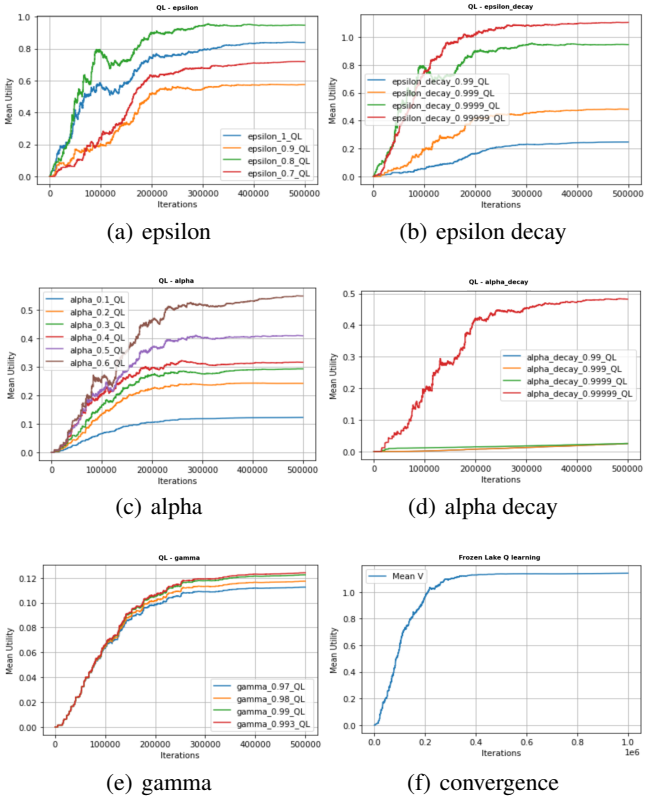
I also plotted the metrics with different state sizes. Generally speaking, VI has greater average utilities when the state size is smaller than 256 (16*16), but they have similar average utilities after that (Figure 11b). The large difference in max utilities (Figure 11a) suggests that even though these two algorithms may come to the same optimal policy at the end, their implementations and the logics behind them are different. Another point is that VI takes way more iterations to converge compared with PI, whereas PI takes longer time to converge, especially then the state size becomes larger. Other convergence and policy comparisons will be discussed later with Q learning algorithm.

Q Learning

Q learning is a bit more complicated. The exploration strategy that I'm using is epsilon greedy, which I think can balance the exploration and exploitation the best for the algorithm. The strategy is to give a large epsilon value at the beginning, and let the agent explore the environment and randomly select an action. As the agent knows more about the environment, lower the epsilon so that the agent starts to take actions based on its knowledge of the environment. On top of that, after taking an action in the current state, the Bellman equation is used to calculate Q-values to update the Q table. There are some other exploration strategies, but I feel like epsilon greedy is easy to understand and implement, and over time, the actions with the best utilities will be selected more frequently.

Furthermore, I tuned the parameters including epsilon, epsilon decay, alpha, alpha decay, and gamma for Q learning. There are different metrics to observe the performance, among which I chose the average utility as my metric to judge the performance and convergence.

It is pretty obvious that epsilon = 0.8 (Figure 12a), epsilon decay = 0.99999 (Figure 12b), alpha = 0.6 (Figure 12c), alpha decay = 0.99999 (Figure 12d) are the definite win-



(a) epsilon (b) epsilon decay

(c) alpha (d) alpha decay

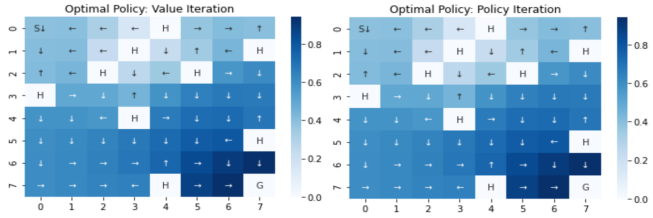
(e) gamma (f) convergence

Figure 12: QL: tuning and convergence

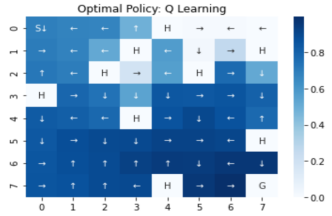
ners among all candidates. For gamma, value 0.993 seem to have a slightly higher average utility (Figure 12e). Therefore, these are the parameters I picked for the Q learning algorithm.
I combined these parameters and plotted the average utility with different iterations (Figure 12f). It looks to me that it converges after like 400000 iterations.

Comparisons

With the tuned parameters, I generated the optimal policies of each algorithm. The VI and PI give me the same optimal policy (Figure 13 a&b), whereas the Q learning returns a very different policy (Figure 13c). I understand the VI and PI policy easily: the closer to the goal point, the higher the reward. However, the policy that Q learning returns is quite different. it seems like its logic is to give higher rewards when moving to the area with less hole, and give lower rewards around the holes. However, when taking a look at the arrows, I found that some arrows in Q learning policy are pointing to the holes, which means the agent did not receive the environment information very well.
I think this is because of the exploration strategy that I selected. When exploring and collecting environment information, the algorithm found that in the area with the majority of holes (upper right area), the agent was easier to fall into the holes, therefore, it tends to lead the agent to some safer area with fewer holes (lower left area) and of course the goal point. Therefore, the Q learning algorithm gives higher rewards to the grids where it wants the agent to move to. However, in the exploration process, the agent failed to detect all holes, or when received contradictory information, for instance, going towards the goal point and avoiding holes, the agent failed to make a correct decision.



(a) VI (b) PI



(c) QL

Figure 13: Policies

The problem setting is that the agent only gets 1 reward when reaches the goal point, which is a case of success. If the episode ends with the agent falling into the holes, the reward for the episode will be 0. Therefore, after having the final optimal policies determined by VI PI and Q learning respectively, I made the policies run 500000 times, and recorded the times that they successfully reached the goal point[4]. The results are shown in the Table 1.

Algo	VI	PI	QL
Success rate(%)	58.21	60.01	22.125
Execution time(s)	2.8	1.5	15.4
Convergence iteration	997	11	around 400000

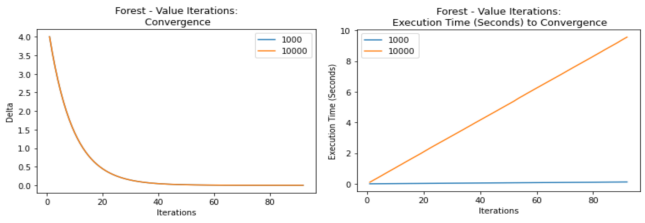
Table 1: Comparison: three policies

From the table, we can see that Q learning has the lowest success rate but longest execution time and largest number of iterations to converge. On the other hand, even though the policies returned by PI and VI are the same, the success rates are different. This is because there are chances for the agent to slip. Therefore, even if the success rates are different, they are very close. Furthermore, PI has the shortest execution time and the smallest number of iterations to converge. As for the reason, I think it may be related to the convergence criteria. The convergence criterion of VI is the variance of rewards (value function), whereas the criterion of PI is the strategy. As I understand, these two functions are model-based. Ideally, they need to traverse all states to update values or policy. My inference is that traverse states and update policy is faster than update values. Q learning is even slower since it takes more time to complete each episode, especially at the beginning.

III. Forest Management

As I mentioned in the brief introduction part, I ran the experiments with a smaller state size of 1000, and a larger state size of 10000. The implementation logic is the same as the Frozen Lake problem.

Value Iteration



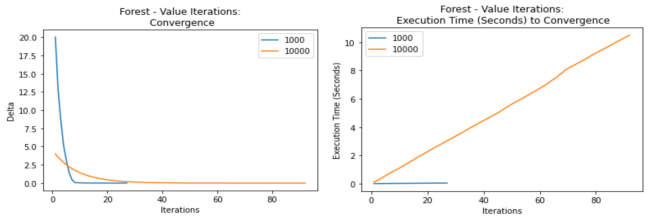
(a) criterion: delta (b) execution time

Figure 14: VI: convergence & time

The graphs suggest that the change of variance of rewards (delta) as the number of iterations increases for both smaller and bigger state sizes is very similar (Figure 14). The only difference is the execution time. It is reasonable that larger state size takes longer to converge.

[4]The technique of calculating success rate is to divide the total number of rewards by the number of episodes

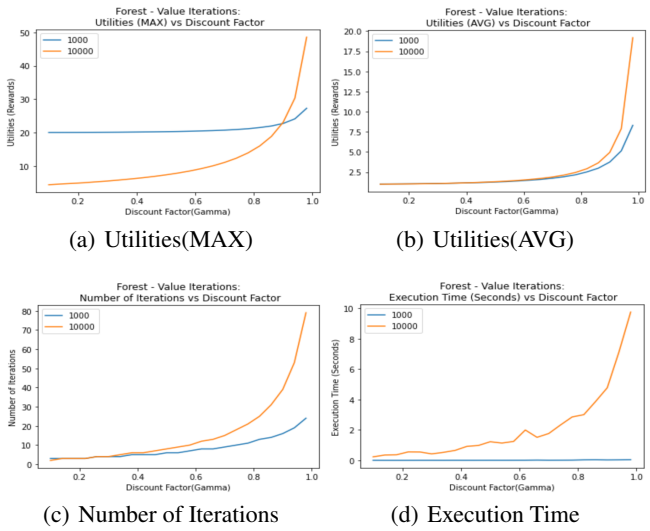
Different sizes of states having the same behavior looks strange to me, therefore, I printed out the run stats of these two sizes, and found that their values of rewards, max v, mean v, and error were really similar. To verify if this is a coincidence, I changed different sizes like 10 and 100 vs 10000, but still they generated very similar graphs. After hours of tuning, I figured that as long as the parameters like rewards, possibility of wildfire, and gamma were the same, the behaviors would be quite similar regardless of the size of the states. This looks boring to me, therefore, I decide to change their parameters. For 1000 states, I think I need to care more about profits, so I decide to give a large r2 value for cutting woods to sell (I know this isn't a good thought, but I want to see some different behaviors). And because I encourage to cut woods, I want to set the possibility of wildfire higher as a punishment.



(a) criterion: delta (b) execution time

Figure 15: VI: convergence & time

The graphs show that with a higher r2 value and higher p value, the problem with a smaller number of states converges faster (Figure 15). By "faster" I mean less time and fewer iterations.



(a) Utilities(MAX) (b) Utilities(AVG)

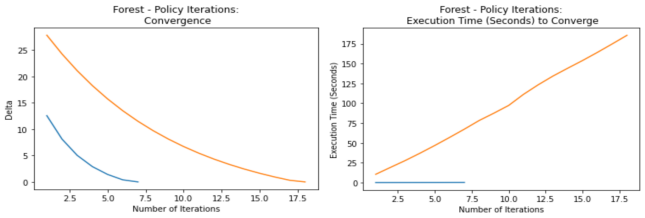
(c) Number of Iterations (d) Execution Time

Figure 16: VI: Gamma vs metrics

Since I already found that the behavior changes have nothing to do with the size of states, I will discuss the influence of reward setting. The size 1000 has higher maximum utilities until gamma reaches 0.9 (Figure 16a), which is a result of me giving the size 1000 a much higher r2 value. However, the average utilities of this size is smaller than the bigger size with default r1 and r2 values, especially when gamma is greater than 0.8 (Figure 16b). I think this is because the bigger the gamma, the more the agent considers future rewards.

The same as previous analysis, the problem with larger r2 value and p value takes less time and fewer iterations to converge (Figure 16c&d).

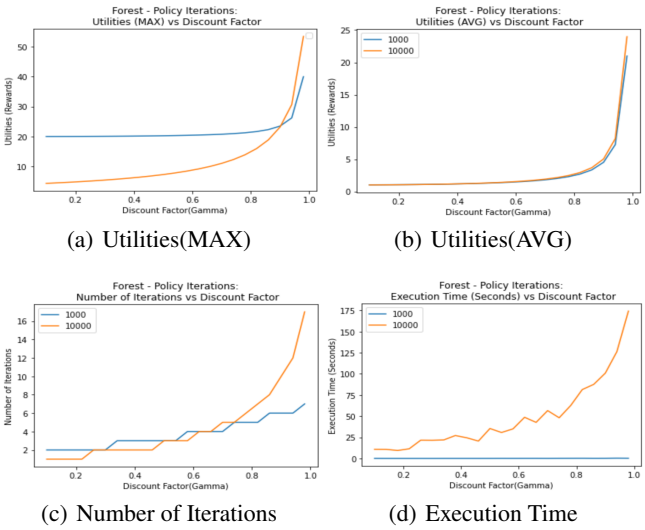
Policy Iteration



(a) criterion: delta (b) execution time

Figure 17: PI: convergence & time

Similar to VI, PI algorithm also suggest that problem with a higher r2 value takes less time and fewer iterations to converge (Figure 17).



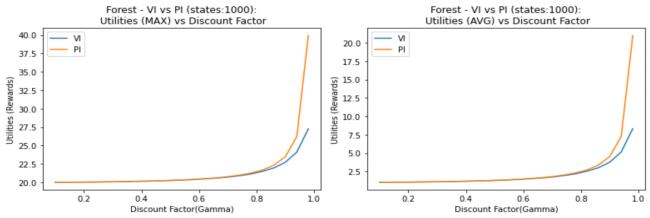
(a) Utilities(MAX) (b) Utilities(AVG)

(c) Number of Iterations (d) Execution Time

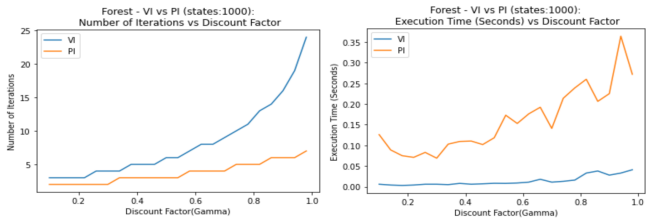
Figure 18: PI: Gamma vs metrics

Similar to VI, the problem with a higher r2 value has higher maximum utilities until gamma reaches 0.9 (Figure 18a), and it has similar average utilities compared with the problem with default parameter values (Figure 18b). What is slightly different is that the size 1000 problem has slightly higher number of iterations to converge until gamma reaches 0.6 (Figure 18c). But it always takes less time to converge (Figure 18d).

VI vs PI



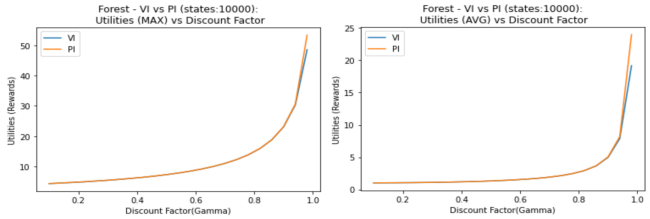
(a) Utilities(MAX) (b) Utilities(AVG)



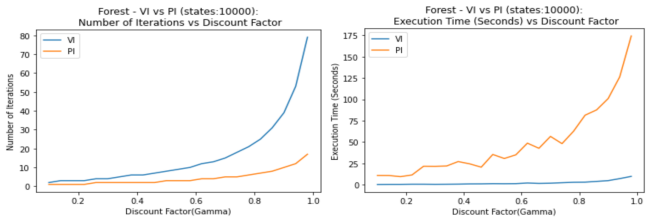
(c) Number of Iterations (d) Execution Time

Figure 19: VI&PI: Gamma vs metrics (1000 states)

The graphs show that PI has higher maximum utilities (Figure 19a) and average utilities (Figure 19b). It takes fewer number of iterations (Figure 19c) but longer time (Figure 19d) to converge. This is different from my findings of the Frozen Lake problem. I decide to run the same experiment with the 10000 size problem with all default parameter values to see whether the difference is related to parameter adjustment.



(a) Utilities(MAX) (b) Utilities(AVG)

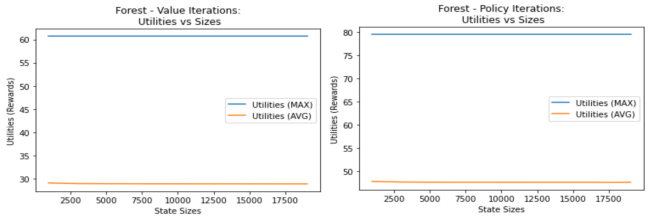


(c) Number of Iterations (d) Execution Time

Figure 20: VI&PI: Gamma vs metrics (10000 states)

The behavior of the 10000 states problem (Figure 20) is similar to the behavior of the 1000 (Figure 19) states problem, except that PI does not appear to be significantly larger than VI regarding the maximum and average utility. But I care more about why PI takes longer clock time to converge when dealing with large size problems. I did a lot of research but did not find a reliable answer. I'm guessing it may be because PI needs to traverse all states and update the policy per iteration with a set of complex calculations, with very large number of states, this process becomes very time-consuming compared with the time VI spends to traverse and update values.

To verify that the performances are not relevant with problem sizes, I plotted the utilities vs problem sizes for both VI and PI

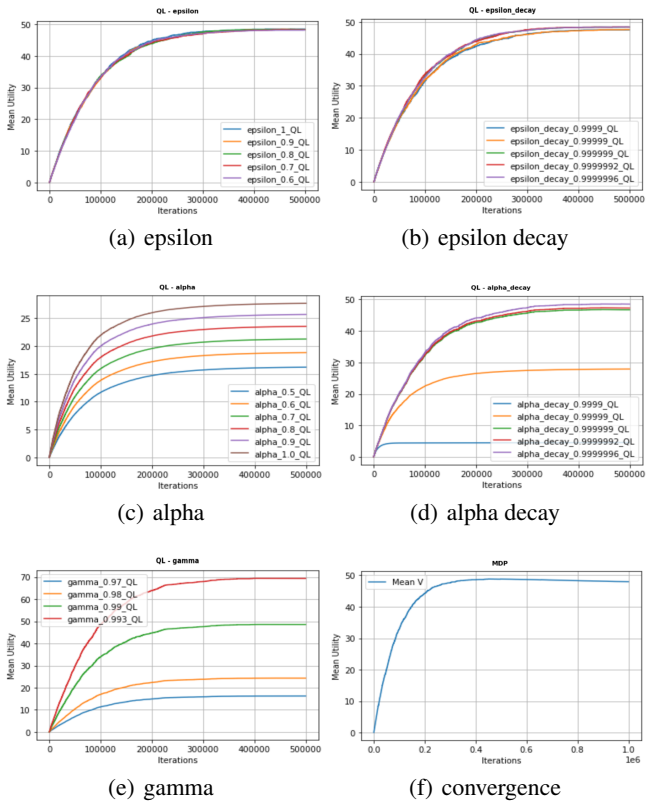


(a) VI (b) PI

Figure 21: VI&PI: size vs utilities

It is clear that no matter how the sizes vary, with the same parameters, the performances are constant.

Q learning



(a) epsilon (b) epsilon decay

(c) alpha (d) alpha decay

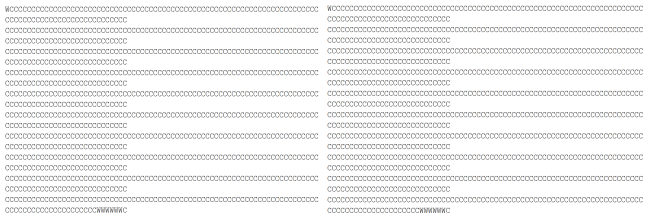
(e) gamma (f) convergence

Figure 22: QL: tuning and convergence

With the same idea, I also tuned the parameters including epsilon, epsilon decay, alpha, alpha decay, and gamma for this problem.
It is a bit difficult to tell which epsilon and epsilon decay values are the winners, but since they have very similar average utilities values, I think there should not be huge differences between different epsilon values. I feel like the green one (0.8) has slightly higher utility value (Figure 22a), so I picked it. As for the rest (Figure 22), I selected the epsilon decay of 0.9999996, alpha of 1, alpha decay of 0.9999996, and gamma of 0.993.
With the combination of selected parameters, I plotted the average utility with different iterations (Figure 22f). It looks to me that it converges after like 400000 to 500000 iterations.

Comparisons

Using the tuned parameters, I generated the optimal policies of each algorithm. The same as the Frozen Lake problem, the VI and PI give me the same optimal policy (Figure 23 a&b), whereas the Q learning returns a very different policy (Figure 23c). In the graphs, "W" stands for waiting, and "C" stands for cutting. We can tell that VI and PI have almost all "C"s whereas Q learning has more "W"s.
As for the reason, I think it may be related to my adjusted r2 value for the problem setting. As I understand, VI and PI guarantee to find the optimal policy. Since the rewards of cutting is much higher, I think they tend to make the agent cut more woods. For Q learning, it collects environment information while taking steps, that is to say, it is possible that the information it gains are not complete.



(a) VI (b) PI

(c) QL

Figure 23: Policies

To further compare the three algorithms, I made Table2.

We can get some useful information from the table. Although VI and PI give the same optimal policy, they do have different Utility value. This means that they find the optimal policy through different methods. The same as the Frozen Lake problem, the optimal policy found by Q learning has the worst performance: with low utility values, especially the average utilities, as well as the longest execution time and biggest number of iterations to converge. Different from the Frozen Lake problem, even though PI still has the smallest number of iterations to converge, VI takes the shortest time to converge.

Algo	VI	PI	QL
MAX Utilities	29	60	18
AVG Utilities	10	41	0.08
Number of "C"s	993	993	360
Execution Time(s)	0.04	0.3	7.36
Convergence Iteration	24	7	400000-500000

Table 2: Comparison: three policies

IV. Conclusion

For different problem types and sizes, the performances of each algorithm are different. But there are some common rules. Specifically, VI and PI normally converge faster with few iterations than Q learning. Even though VI and PI normally returns the same optimal policy, their success rates or utilities are different. Honestly, I thought Q learning should perform better than VI and PI, because theoretically, it takes a long time to learn environment information and make decision accordingly, it should have a good performance. I guess the problem is not the Q learning algorithm. My Q learning experiments are not that successful, maybe because I did not choose the best exploration strategy. I'll save it for future improvement: explore better exploration strategies.

References

[1] David L Poole and Alan K Mackworth. *Artificial Intelligence: foundations of computational agents*. Cambridge University Press, 2010.

[2] Andre Violante Andre Violante. "Simple Reinforcement Learning: Q-learning". In: (2015).