MACHINE LEARNING - OMSCS 7641
Spring 2022
Reinforcement Learning

Introduction

In this assignment, I implement policy iteration, value iteration and Q-learning to Frozen Lake and Forest problems. I first talk about reinforcement learning algorithms. Then, I analyze the Frozen lake and Forest problems first separately, then together.

Reinforcement Learning Algorithms

Reinforcement learning can be divided into two as model-based and model-free learning. Model-based learning includes Markov decision processes such as policy iteration and value iteration, and other dynamic programming algorithms such as optimal control. Model-free learning includes gradient-free algorithms such as Q-learning and SARSA, and gradient-based algorithms such as policy gradient optimization.

Value iteration optimization the value function given as:

$$V(s) = \max_{a} \sum_{s'} P(s'|s,a)(R(s',s,a) + \gamma V(s')) \tag{1}$$

where the value function is the maximum value of any acceptable action. P is the transition matrix that gives the probability of being at position s' given initial position s and action a. As the action is probabilistic, it is possible to move in an undesired location or not move at all. R is the reward of successfully reaching s' given s and a, gamma (γ) is the importance of the value of the sum of the future actions. V(s') is the best value function at step s'. It is also called as the Bellman's equation where

$$V(s) = \max_{\pi} E(r_0 + \gamma V(s'))\ . \tag{2}$$

Bellman equation states that the value of the s is maximum of the summation of the reward and discount rate multiplied by the value of the best future path. The above approach is recursive and iterative. The iterations are repeated until the value function converges to a value with a small error defined as epsilon.

Policy iteration computes the value function for all the possible actions:

$$V_\pi(s) = \sum_{s'} P(s'|s,\pi(s))(R(s',s,\pi(s) + \gamma V_\pi(s')) \tag{3}$$

Once $V_\pi(s)$ table is calculated, it is used to compute the best policy

$$\pi(s) = argmax_a E(R(s',s,a) + \gamma V(s')) \tag{4}$$

The above two equations are repeated until convergence of the policy. Policy iteration converges in fewer iterations than the value iteration.

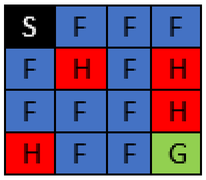
Q-learning is a model-free algorithm. It means that it doesn't use the probability and reward matrices. This algorithm would be useful in a case where we don't have the model i.e. we don't know the probabilities and rewards. Q-learning tries to reach a goal state initially randomly. Once a goal is reached or a fail occurs e.g. dropping in a hole, the episode is over. We learn from the previous experiences and store the previous experiences in a Q-table. This algorithm can be used to win a PlayStation game as well.

It would have to play the game many times until it learns the game. This algorithm can be inefficient because many experiences are required. However, it is useful when there is no model.

There are several important parameters in Q-learning that are used for tuning; gamma, alpha, alpha decay, epsilon and epsilon decay. alpha is the learning rate. Gamma is the discount factor as discussed in value and policy iterations. Learning rate is used to learn from the new actions. Large learning rate makes big changes in the quality function used in Q-learning. This is initially necessary as the model doesn't know anything. However, with time the model builds knowledge, and hence we should rely on existing knowledge rather than making big jumps. Therefore, we should decay the learning rate as the number of iterations increases. On the other hand, epsilon has a different purpose. If epsilon is 0, the algorithm always follows the higher Q. This can potentially lead to a local optimum. Therefore, a higher epsilon value introduces stochasticity, and it can go in a random direction. Over time, we should also decay epsilon.

Exploitation and exploration are two contrasting concepts. Exploration is about going in unexpected directions to avoid local optimum, and exploitation is about trusting your knowledge (the quality function) and not going in random direction when we are close to the solution. The parameters above are tuned to optimize between exploration and exploitation.

Frozen Lake



Frozen lake is a grid-based environment. The environment can be created at gym and then exported to mdptoolbox for training. I created a 4*4 grid as shown on the right. In this image, S, F, H and G stand for Start, Frozen, Hole and Goal in that order. Let us say that we are at the frozen lake and somehow want to go to the goal. However, some of the blocks are hole and others are frozen. If we drop into a hole, it is a fail. On top of that, the movements are probabilistic. There is a 33% chance we will go in the direction desired, 33% change we will go perpendicular in clockwise direction and 33% change we will go perpendicular in counterclockwise direction. If there is a wall, in any movement, we stay at the current location without penalty. The only reward is 1 and it occurs when the goal is reached. There are 4 possible actions; Left, Down, Right, Up which are coded as 0, 1, 2 and 3.

Why is it interesting?: It is interesting because it satisfies the requirement of a small grid-size problem. This a commonly solved problem which is available at gym.

Policy Iteration: mdptoolbox.mdp.PolicyIteration takes T, R and discount as input. I initialize the policy to all zeros. T and R are imported from gym. I vary discount rate to see which one gives the highest reward, average of the V matrix, where V_i is the value of a state. Further, I also report the number of episodes required for each discount rate scenario. Finally, I do a simulation by coupling the final policy after run with the step function of openai gym. I repeat the simulation 50 times and store whether it was able to reach the goal state (state=15). I store, for each discount rate scenario, the number of successful simulations as wins (wins out of 50 simulations).

I can see from Figure 1 that that as the discount rate increases rewards and wins start to increase. I pick discount rate as 0.9 instead of 0.99 because even though the reward for the latter is higher, the number of iterations for the first is lower while both of them successfully reach the goal 40 times out of 50. Discount rate is shown as the gamma in the above equations. As the reward is only gained when we reach the goal, and as the goal is many steps away, a lower discount rate will reduce the value of any state. It can be seen from Eq (3) that the V value function is inversely proportional to the gamma value. As the reward is only at the last step, the highest importance is given to the last s'.

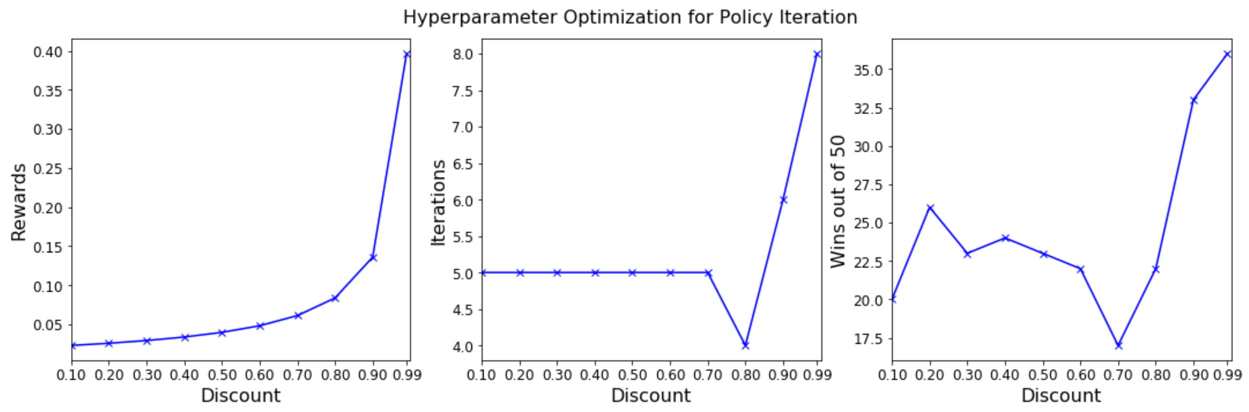


Figure 1. Left: Discount vs Rewards, Middle: Discount vs Iterations, Right: Discount vs Wins

I also study the behavior of the algorithm as the problem size is varied between 4*4 and 8*8. For the first map, I kept the discount rate as 0.9 whereas for the larger problem I took the discount rate as 0.8. Otherwise, the number of iterations were skyrocketing. Figure 2 shows that the reward decreases as the problem size gets bigger. The reason is the only reward is at the goal, and the discount rate penalizes the reward if it is farther away. Further, the lower discount rate for the larger problem also penalizes the V function. The number of iterations required for 8*8 increases from 6 to 10 as it is a more difficult problem. Also, computational time increases from 0.004 seconds to 0.01 which is 2.5 times of the first. This is a linear increase which is in line with the time complexity of policy iteration, O(|S*A|).

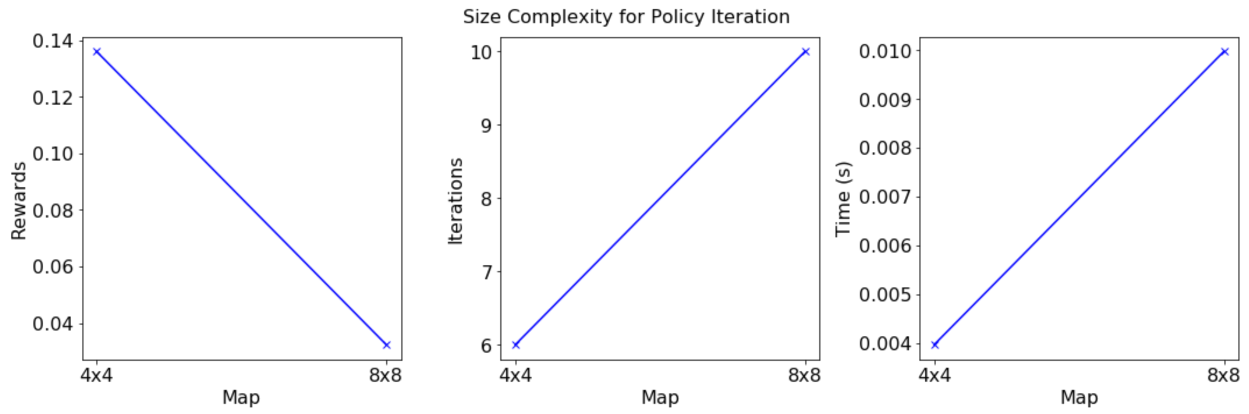


Figure 2. Left: Problem Size vs Rewards, Middle: Problem Size vs Iterations, Right: Problem Size vs time

Value Iteration: Similar to policy iteration, I create a value iteration model of Frozen lake. I do hyperparameter optimization as shown in Figure 3. The analysis for the hyperparameter optimization is the same as that for policy iteration. Again, it can be seen that 0.9 discount factor gives the most wins out of 50. Therefore, this value is the best for this case. One difference from policy iteration is that value iteration needs way more iterations for converge. For example, at discount 0.9, policy iteration converges in 6 iterations while value iteration needs 26.

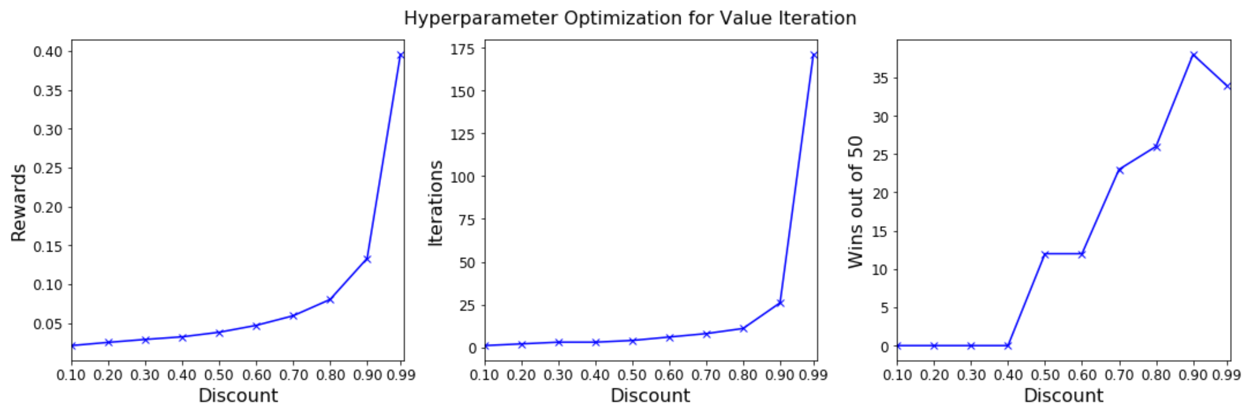


Figure 3. Left: Discount vs Rewards, Middle: Discount vs Iterations, Right: Discount vs Wins

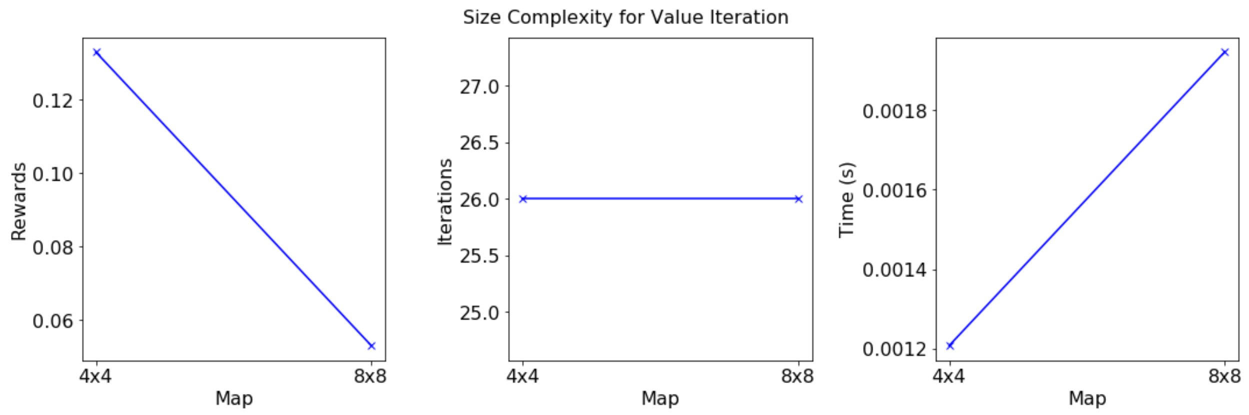


Figure 4. Left: Problem Size vs Rewards, Middle: Problem Size vs Iterations, Right: Problem Size vs time

Similar to policy iteration, I vary the map size from 4*4 to 8*8. This time, I used a discount rate of 0.9 for both cases as 8*8 did not require too many iterations to converge. Figure 4 shows that the rewards decreases with increasing map size. The reason for that is explained in policy iteration. Unlike, policy iterations, number of iterations do not increase as the map size gets bigger. The reason for that may be that the number of iterations are independent of the size of S, unlike policy iterations. However, computational time increases linearly which is in line with its time complexity O(|A*S|).

Q-learning: The background information for Q-learning as well as the definition and importance of epsilon and alpha are given above in the algorithms section. Here, I first optimize the five hyperparameters; gamma, alpha, alpha decay, epsilon and epsilon decay. All the other parameters are defaulted. For example, maximum iterations are 10000. I do a grid search by using the following values: gamma = [0.5, 0.6, 0.7, 0.8, 0.9], alpha = [0.1, 0.3, 0.5, 0.7, 0.9], alpha decay =[0.95, 0.97, 0.99], epsilon = [0.5, 0.6, 0.7, 0.8, 0.9, 1.0], epsilon decay = [0.95, 0.97, 0.99]. This creates 5*5*3*6*3 = 1350 different cases. I run all of this cases and every time I do 50 simulations of gym to see whether it reaches the goal. I score the success as the number of wins out of 50 similar to what I did for policy and value iteration. The best one combination is gamma 0.5, alpha = 0.7, alpha decay = 0.97, epsilon = 0.6 and epsilon decay = 0.97. This one give 20 wins out of 50. This value will change at every simulation as it is stochastic.

I further vary these parameters and try to optimize further. I notice the stochasticity in the results because Q-learning and the step function are probabilistic. From Figure 5, I can see that the rewards declines sharply when epsilon is increased too much. This might be happening because too much exploration misses a better optimum. As the epsilon increases, exploration increases and as the epsilon decreases, exploitation increases. It seems anywhere between 0.55 and 0.63 is a good balance between exploration and exploitation. We can also see that wins decline sharply when epsilon is above 0.63.

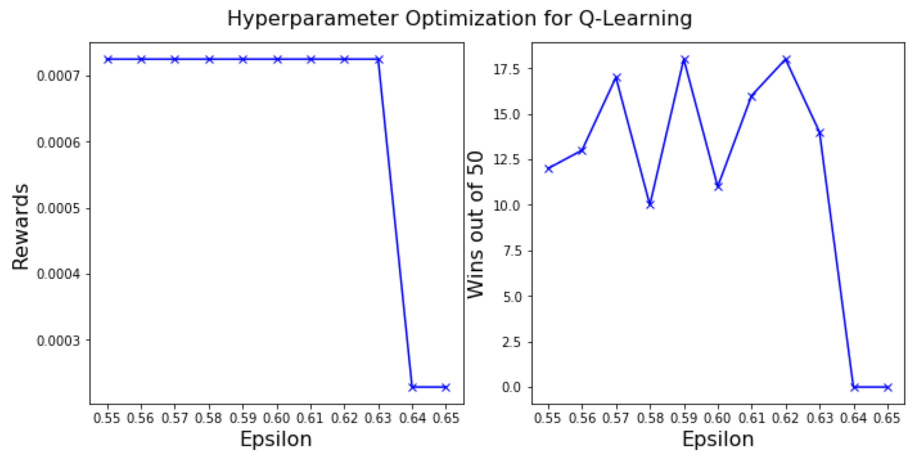


Figure 5. Left: Epsilon vs Rewards, Right: Epsilon vs Wins out of 50.

Figure 6 compares the rewards and time when the map size is increased from 4*4 to 8*8. We can see that rewards is decreasing. There are two possible reasons for that. 8*8 case is not tuned and the other reason is that the only reward is at the goal. Therefore, larger problem size will punish the value function because of the gamma value. I can also see that computational time is only slightly increasing for the 8*8 case. This is surprising because the computational complexity of Q-learning is $O(S^3)$. The reason for we don't see an exponential increase may be that the number of iterations are much fewer for the 8*8 case.

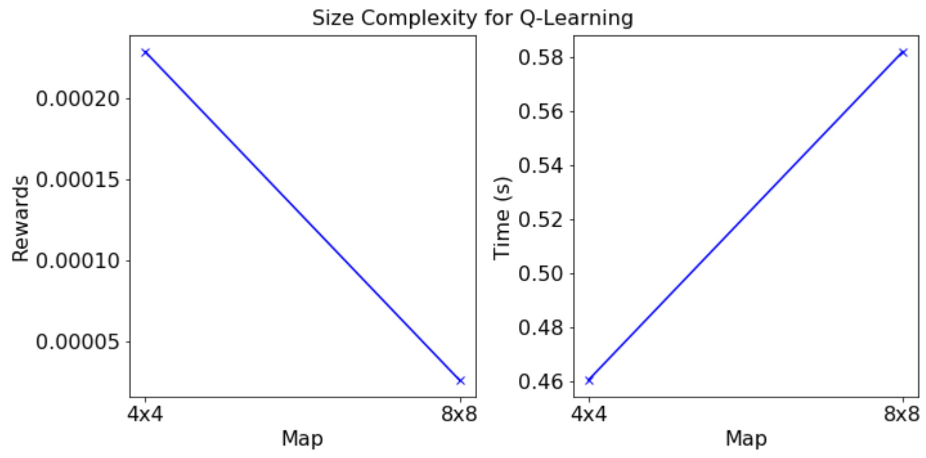
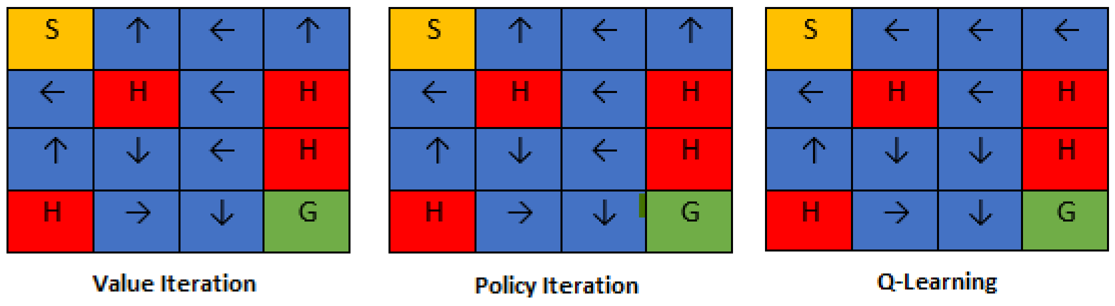


Figure 6. Left: Problem Size vs Rewards, Right: Problem Size vs Time

Comparison of Value Iteration, Policy Iteration and Q-Learning

Here, I compare value iteration (VI), policy iteration (PI) and Q-learning (QL) on Frozen lake data. I rerun the optimized algorithms with the optimized hyperparameters for the three algorithms. Figure 7 shows the policy map for VI, PI and QL. VI and PI policies are exactly same. These policies also make sense because I can see that they are trying to reach the goal while avoiding the holes. However, I can see that Q-learning policy is not as good as the other two.



Value Iteration

S	↑	←	↑
←	H	←	H
↑	↓	←	H
H	→	↓	G

Policy Iteration

S	↑	←	↑
←	H	←	H
↑	↓	←	H
H	→	↓	G

Q-Learning

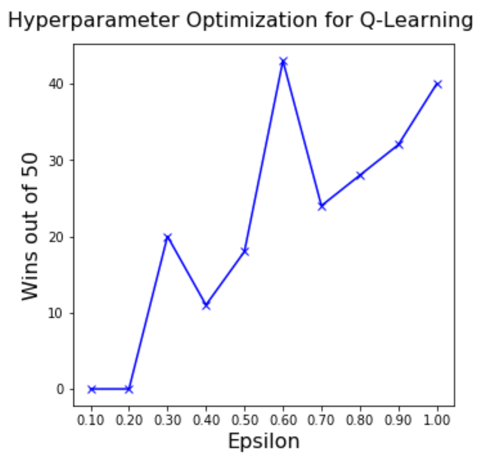
S	←	←	←
←	H	←	H
↑	↓	↓	H
H	→	↓	G

Figure 7. Left: Optimal policies for VI, PI and QL.

I observe from the table on the right that QL is much more costly than the others as it takes many more iterations. PI takes fewer iterations and less time than VI. This is expected because it is well known PI converges in fewer iterations. The reward for VI and PI are same as their value function calculations are same and they reach the same policy. However, QL has a very low reward probably because it was not able to find the most optimal policy.

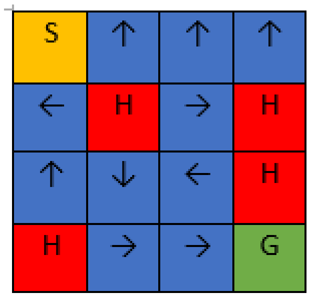
	VI	PI	QL
Iterations	26	5	-
Time (s)	0.003	0.001	0.46
Reward	0.133	0.136	0.0007

I think one reason for the suboptimal policy for QL is that the only reward is at the goal. If we add large penalties for the holes and small penalties for every step and increase the reward for the goal, this will probably give a better QL policy. Therefore, I modified the R matrix by hand and reused it on QLearning. The new R matrix rewards 300 at the goal, -300 at any hole and -1 for anything else. This encourages avoiding the holes and decreasing the number of the steps. I do another grid search over more than 1000 combinations, and I quickly find an optimal solution that gives 41 wins out of 50. The optimized hyperparameters are alpha = 0.5, alpha = 0.1, alpha decay = 0.95, epsilon = 0.6, epsilon decay = 0.97.



The image on the left shows that the wins for Q-learning tends to increase as the as epsilon (an exploratory force) is increasing. However, it seems that epsilon = 0.60 is the right balance between exploration and exploitation.

The scheme on the right shows the new optimal policy for Q-Learning. Even though, it is not the same as VI or PI, it is a much more improved version than earlier. It avoids the holes while aiming at the goal.



S	↑	↑	↑
←	H	→	H
↑	↓	←	H
H	→	→	G

Forest

This is an interesting problem because I made it large (size(S)=1000) and it is a non-grid problem. This is a problem where each state refers to a time period. For each time period, there is a p probability that the forest will burn and go to state 0, and 1-p probability that the forest will get older and go the next state i.e. s'=s+1. There are two actions wait encoded as 0, and cut encoded as 1. If you cut the forest, you sell it and make some money and the forest goes to state 0. If you wait, the forest get older and you may cut in the future with the hope getting a bigger reward. However, there is always a p probability that a fire will occur. More information can be found at [1].

I set up a problem where the forest is simulated for 600 years [0,599], where each year is a state. There is fire every 10 years. Therefore, probability of fire is 0.1. If you wait for 600 years, the reward is 1000. Otherwise, the reward for cutting the forest is 1.

In the plots below, I use meanV as the mean of the Value function of the optimal policy. I also simulate the game for 1000 years and collect the rewards which is called as Rewards below.

Policy Iteration: Gamma parameter has been modified between 0.1 and 0.9 to find the optimal policy as can be seen in Figure 8. As expected, a higher gamma is giving a higher Value function because they are proportionally correlated. It takes more iterations to converge at higher gamma and this make the training slower. Rewards, however, don't seem to change much and the values seem to be pretty high. The policy is basically 0 (wait) initially, then 1 (cut). Close to then end, policy becomes 0 again maybe with the hope of collecting the final reward. I would take 0.1 gamma as the optimal value as it is the fastest to train while giving similar number of rewards.

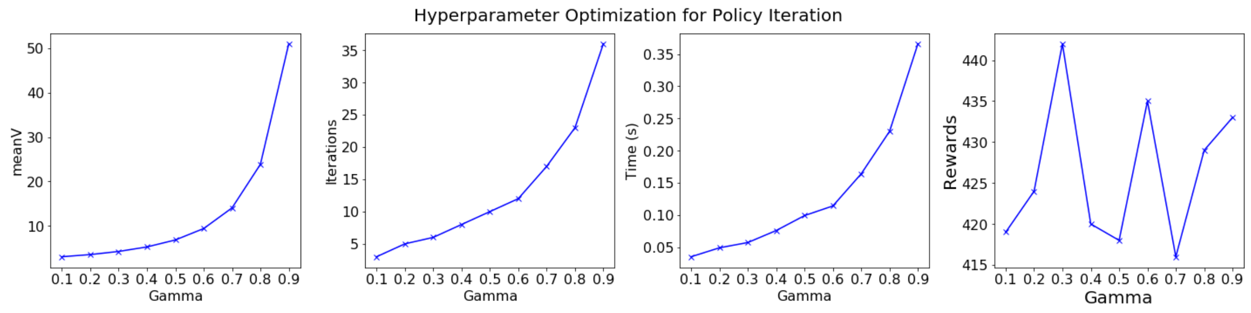


Figure 8: 1. Gamma vs mean(V) 2. Gamma vs Iterations 3. Gamma vs Time 4. Gamma vs Rewards.

Figure 9 looks at the size complexity of policy iteration for the forest problem. The number of states are varied between 1000 and 4000. As the problem size is increasing, the mean value is decreasing. This is something I observed in my previous analysis as well. This happens because there is only one reward for the wait option which is a constant. As this value is shared by more states, the mean value decreases. Iterations are flat as the problem size is increasing. Computational time is increasing more than linearly. I can see that as the problem size doubles, computational time increases by 5 times. This is contradictory to the computational complexity of of policy iteration, O(|A*S|). Rewards are pretty much the same for all of them as the simulation is run 1000 times independent of the problem size.

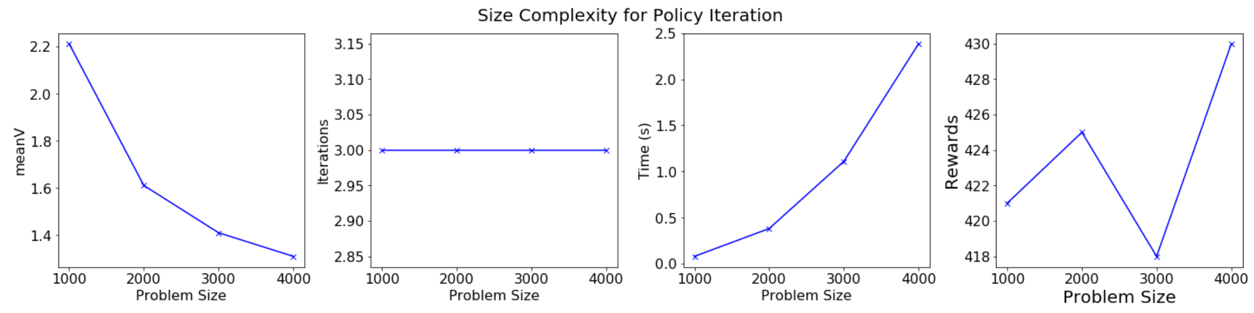


Figure 9: 1. Problem Size vs mean(V) 2. Problem Size vs Iterations 3. Problem Size vs Time 4. Problem Size vs Rewards.

Value Iteration: Figure 10 shows the gamma tuning for the Forest problem. Mean value increases as gamma increases. However, this is because gamma is proportional to the value funcction and is no indication of a better solution. The number of iterations are higher as gamma increases as observed before. This also increases the computational time. Rewards are pretty much the same as gamma is varied. Rewards are also similar to the one seen in policy iteration. The optimal policy is exactly the same as Policy iteration. Therefore, I thing gamma = 0.1 is the optimal parameter as it is the fastest.

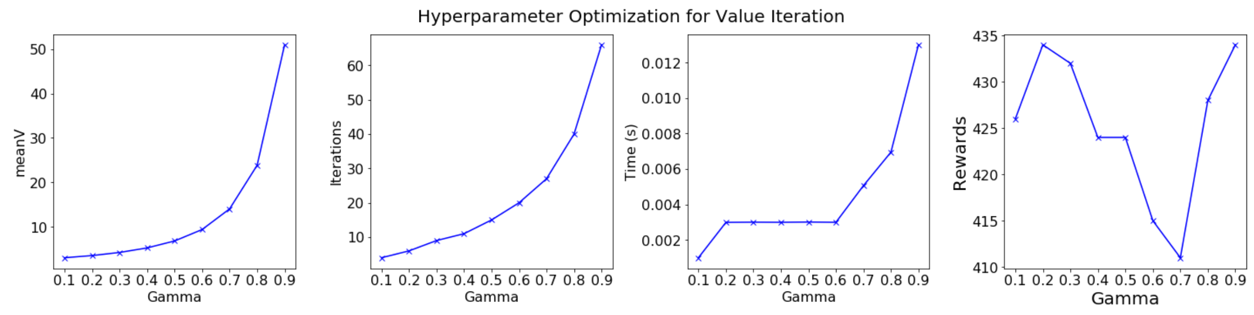


Figure 10: 1. Gamma vs mean(V) 2. Gamma vs Iterations 3. Gamma vs Time 4. Gamma vs Rewards.

Figure 11 varies the number of states between 1000 and 4000. Similar to policy iteration, mean value decreases because the number of rewards do not go as much as the number of states. The number of iterations are 4. The number of iterations for policy iteration were 3. It is well known that the number of iterations for policy iteration is less than that of policy iteration. Therefore, the figure confirms this statement. Computational time increases more or less linearly. This is in line with the time complexity of value iteration, O(|A*S|). Rewards, continue to be more or less same as a function of the prblem size. This is because the simulation is run for 1000 years independent of the problem size.

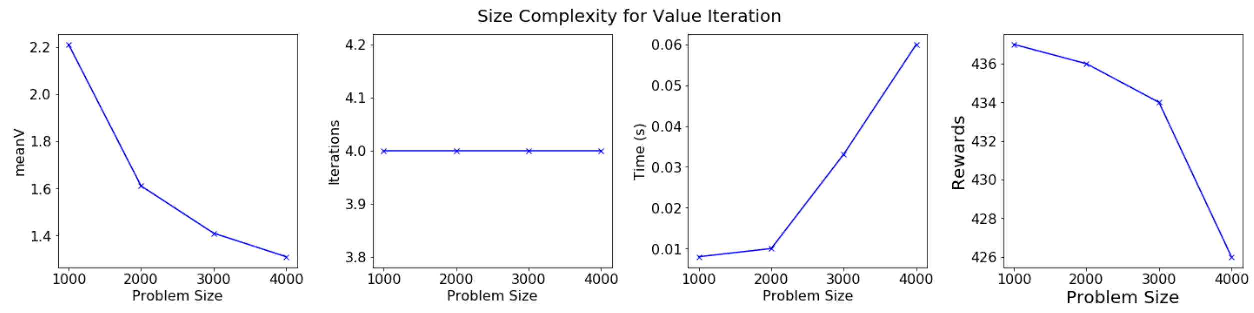


Figure 11: 1. Problem Size vs mean(V) 2. Problem Size vs Iterations 3. Problem Size vs Time 4. Problem Size vs Rewards.

Q-Learning: I do a grid search over gamma, alpha, alpha decay, epsilon and epsilon decay. The combinations are as follows: Gamma = [0.5, 0.6, 0.7, 0.8, 0.9], alpha = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9], alpha devay = [0.95, 0.97, 0.99], epsilon = [0.5, 0.6, 0.7, 0.8, 0.9, 1.0], epsilon decay = [0.95, 0.97, 0.99]. This makes 5*9*3*6*3 = 2430 combinations. The number of iterations is fixed to 10000. Q-learning gives the optimal solution for most of these combinations. However, I have found some combinations where the policy is worse. I have determined the quality of performance based on the rewards collected during 1000 years of simulation.

Finally, I fix gamma to 0.5, alpha to 0.1, alpha decay to 0.99 and epsilon to 0.5. I vary the epsilon decay as shown in Figure 12. Figure 12 shows that epsilon decay of 0.95 and 0.97 give lower rewards. This might be because they may be cause convergence in a local optimum. I can also see that mean value is lower within this interval which is more proof for local optimum. Computational time seems to be increasing slightly with increasing epsilon decay. I think an epsilon decay of 0.93 may create a good balance between exploration (trying random directions) and exploitation (following your wisdom).

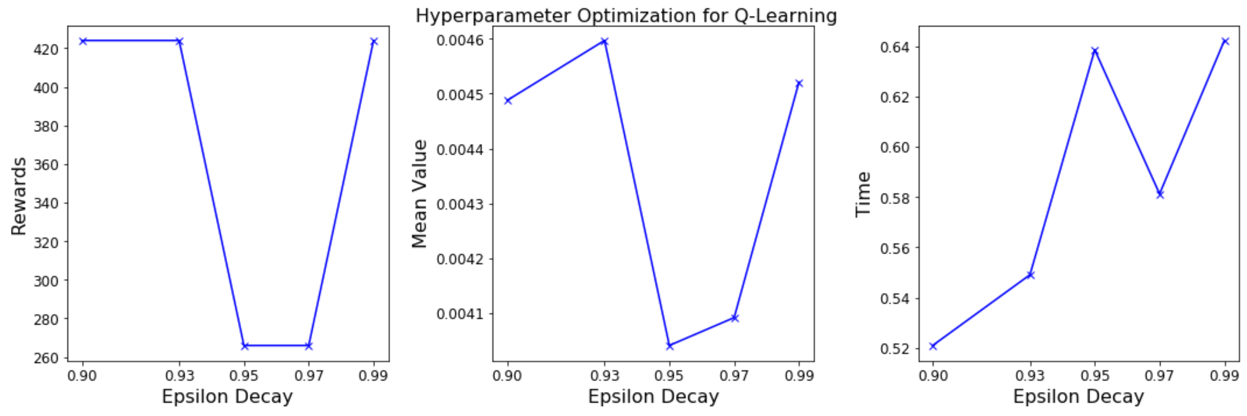


Figure 12: 1. Epsilon Decay vs Rewards 2. Epsilon Decay vs Mean Value 3. Epsilon Decay vs Time.

Figure 13 shows that the mean value decreases as a function of problem size as observed and explained before. Computational time is linear which is in contradiction with the time compllexity of Q-Learning, O(S^3). Rewards do not depend on the problem size as explained before.

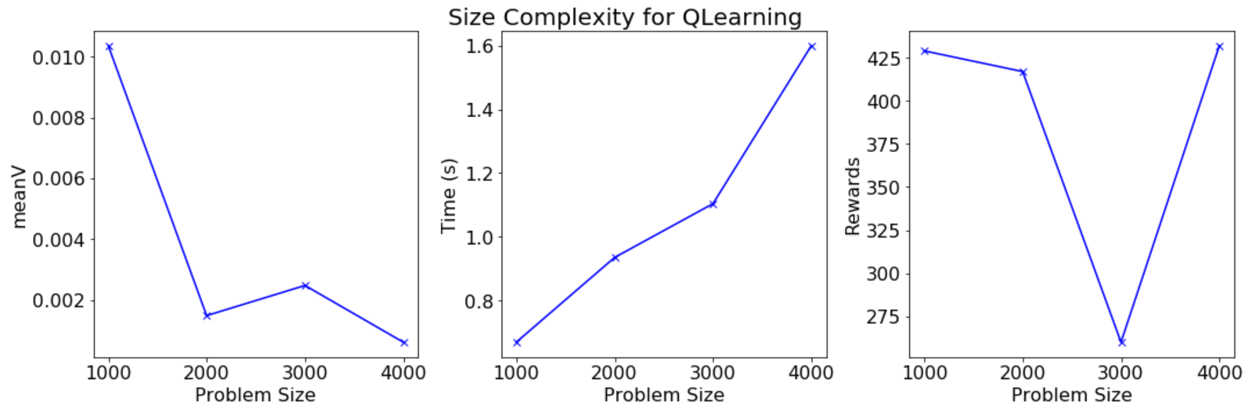


Figure 13: 1. Problem Size vs mean(V) 2. Problem Size vs Time 3. Problem Size vs Rewards

Comparison of Value Iteration, Policy Iteration and Q-Learning

The number of iterations for value iteration, policy iteration and q-learning are 4, 3 and 10000, respectively. Policy iteration converges in fewer iterations than the value iteration as it is supposed to be. Q-learning needs many iterations as it doesn't have any model to learn from.

Computational time for value iteration, policy iteration and q-learning is 0.001, 0.04 and 0.52 seconds. Typically, policy iteration should be faster than value iteration. They all reach a policy that gives similar number of rewards. Optimal policy for value iteration and policy iteration are same. However, the policy of Q-Learning is very different. This may be because there may be many optimal solutions.

Comparison of Frozen Lake and Forest Implementations

Frozen lake was a smaller implementation than Forest. Frozen lake has 16 states while Forest has 600 states. However, I can see that computational time is quite similar. This may be partly because the number of iterations required for Forest were lower.

Frozen lake reward matrix was not made in a way that could teach Q-Learning an optimal policy. Therefore, reward matrix had to be modified to find an optimal policy. On the other hand, forest problem almost always ended up in a good/optimal policy. At the end, I was able to make all the algorithms go to optimal policy for both of the examples.

Reference

1. https://miat.inrae.fr/MDPtoolbox/QuickStart.pdf