Assignment 4 : Markov Decision Processes (CS7641)

Stella L. Soh (lsoh3@gatech.edu)

Abstract
This assignment explores the notion of an agent making decisions using Markov Decision Processes (MDP). In model-based MDP an agent could arrive at optimal policy using Policy Iteration and Value Iteration. We would explore using two interesting MDPs - Frozen Lake (grid-world) and Forest Management (non-grid-world). Besides running Policy Iteration and Value Iteration on these 2 MDP's, we would also be running model-free Q-Learning algorithm.

Introduction

The frozen lake problem - FrozenLake-v1 from OpenAI Gym provided me with a 4 x 4 grid-world environment. The example module in MDPtoolbox provided me with a non-grid world, 625-state-space environment. I made use of MDPtoolbox's PolicyIteration(), ValueIteration() and QLearning() to develop utility functions.

1.Frozen Lake

This is an interesting MDP problem as it illustrates the workings of Policy Iteration, Value Iteration and Q-learning on a small scale. In this small 4 x 4 grid world, there are 16 states. The agent is supposed to walk on this frozen lake from **S** (S for start) to **G**(G for Goal) amongst the **F** (Frozen surface) and **H** (Hole). The interesting thing about this Frozen Lake problem modeled in MDPtoolbox is that for each action or direction (←, ↓ , → , ↑) the agent chooses, there is a 0.333 chance that it would go in the other 3 directions if it is on a **F** grid. This enhances the stochastic nature of the problem. For example, if the agent is in state=9, and the action=1 (i.e. he wants to go down), the probability for going left, right and down direction would be 0.333 each. Fig. 1 shows the map of the Frozen lake, where the upper blue square is the starting point, the black square represents the holes, the white squares represent the ice, and the green square - the goal is where the Frisbee is located.

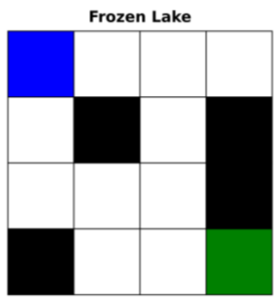


Fig. 1 – Map of the Frozen Lake

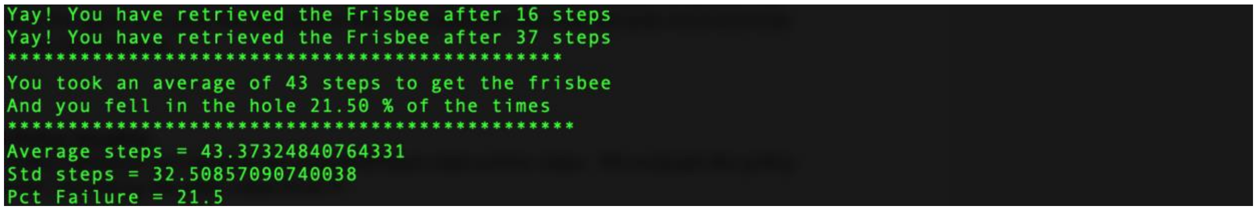
Given a deterministic environment such as this, the agent knows how to plan its action. The goal of the agent is to pick the best policy that will maximize the total rewards received from such an environment.

1.1 Policy Iteration

In Policy Iteration, the iterative algorithm evaluates the policy $\pi(s)$ by calculating the state value from V(s). Then we calculate the improved policy by using one-step lookahead to replace the initial policy $\pi(s)$.We do not care what the initial policy $\pi 0$ being optimal or not. During the execution, we concentrate on improving on every iteration by repeating policy evaluation and policy improvement steps.

By steps, we mean the number of steps required for the agent to get to the Frisbee. The iterations refer to the number of iterations required to converge to the optimal policy under Policy Iteration.

Here, we find that under Policy Iteration, the agent took an average of **43** steps to retrieve the Frisbee and fell into the hole **21.5%** of the time.



```
Yay! You have retrieved the Frisbee after 16 steps
Yay! You have retrieved the Frisbee after 37 steps
**********************************************
You took an average of 43 steps to get the frisbee
And you fell in the hole 21.50 % of the times
***********************************************
Average steps = 43.37324840764331
Std steps = 32.50857090740038
Pct Failure = 21.5
```

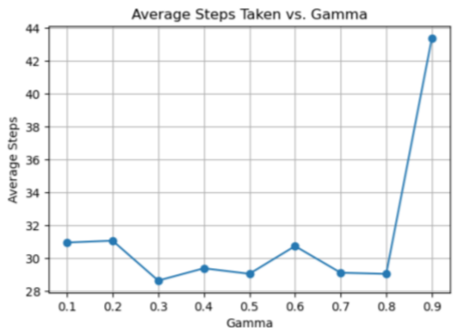


Fig. 2 - Average Steps Taken vs. Gamma

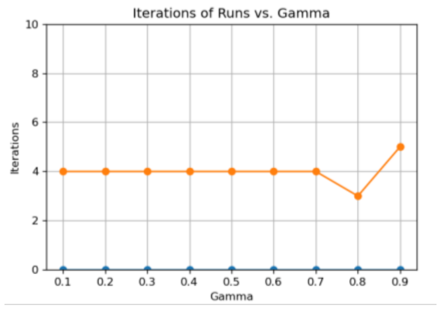


Fig. 3 - Iterations of Runs vs. Gamma

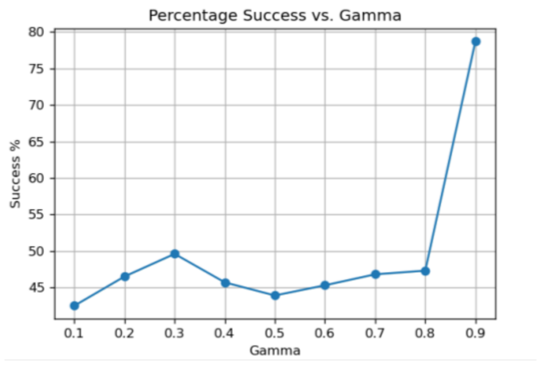
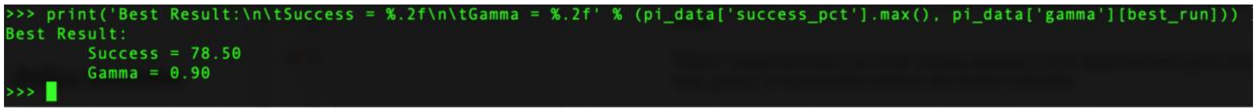


Fig. 4 - Percentage Success vs Gamma

From Fig. 2, we plot the average steps taken versus gamma fluctuates. Between gammas of 0.1 to 0.8, the average steps taken range between 28.5 to 31. Once gamma reaches 0.9, the average steps taken by the agent shot up to 43 steps. It's interesting to note that in Fig. 3, between gammas of 0.1 to 0.7, the iteration of runs stayed constant at 4. At gamma=0.8, the iteration of runs dipped to 3, and then made a jump to 5 at gamma=0.9. The probable reason is: the action space is finite, and in Policy Iteration, *γ (gamma)* in [0, 1] tunes the values of immediate next step to future rewards. As the *γ (gamma)* gets closer to 1 - in this case gamma=0.9, the probability of converging is very close, and thus we see the average steps jumped up to 43. Fig. 4 closely echoes this behavior. We see that the percentage jumped up from 48% at gamma=0.8 to 78.5% at gamma=0.9. This could be an indication that convergence is very close to the optimal policy.

When I found the best_run from taking argmax() of pi_data['success_pct'] and found the best_policy, it consistently reflects the result to be true.

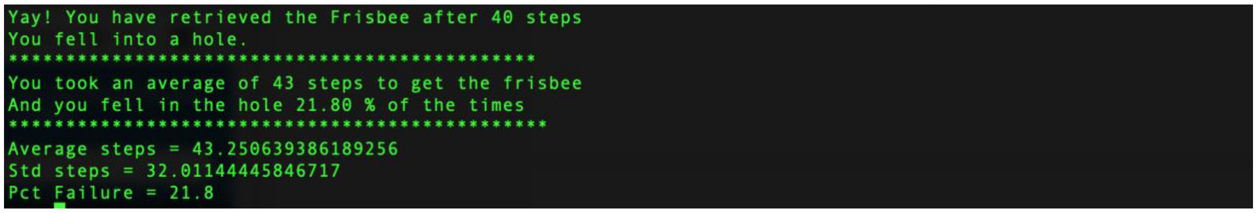


```
>>> print('Best Result:\n\tSuccess = %.2f\n\tGamma = %.2f' % (pi_data['success_pct'].max(), pi_data['gamma'][best_run]))
Best Result:
        Success = 78.50
        Gamma = 0.90
>>> 
```

The best performance came with the highest value of gamma at 78.50% success rate in an average of 43 steps.

1.2 Value Iteration

In Value Iteration, instead of evaluating and improving on the policy, this iterative algorithm updates the Bellman's equation. It learns the values of all the states, and iteration ends when an epsilon-optimal policy is found or after a specified number (max_iter) of iterations.



```
Yay! You have retrieved the Frisbee after 40 steps
You fell into a hole.
***********************************************
You took an average of 43 steps to get the frisbee
And you fell in the hole 21.80 % of the times
************************************************
Average steps = 43.250639386189256
Std steps = 32.01144445846717
Pct Failure = 21.8
```

In running Value Iteration, I found that this algorithm also took 43 steps, and the agent fell into the hole 21.8% of the time.

I defined a set of γ ***(gamma)*** values and epsilon values and plotted the bar charts for average steps versus gamma, and the success percentage versus gamma.

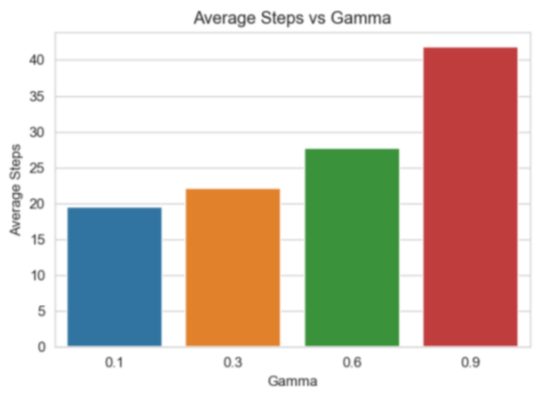


Fig. 5 - Average Steps vs Gamma

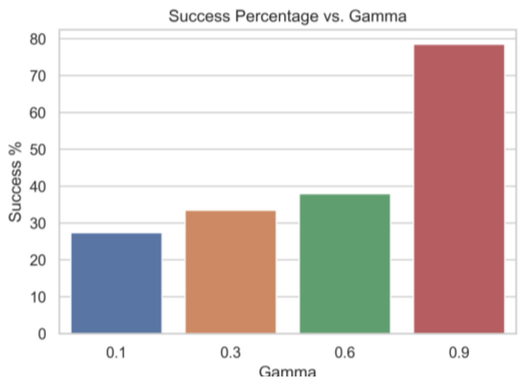


Fig. 6 - Success Percentage vs Gamma

As we know, γ *(gamma)* is a reflection of how much one values the future reward. Epsilon is the stopping criterion. Lower gamma values will put more weight on short-term gains while higher gamma values will put more weight towards long-term gains. Fig. 6 shows that as we increase γ (more weight on long-term gain), the success percentage is higher. I selected $\gamma = 0.9$, and plotted the iterations versus epsilons for that gamma value. Fig. 7 is the result:

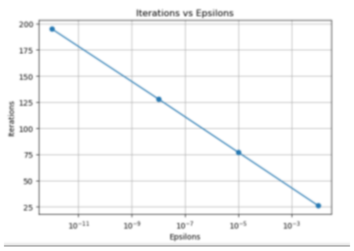


Fig. 7 - Iterations vs Epsilons

We can see in Fig. 7, the smaller the epsilon values, the smaller would be the update to the Bellman equation, and hence a larger number of iterations is required for convergence. Compared to the other gamma values [0.1, 0.3, 0.6, 0.9] that I experimented with, $\gamma = 0.9$ gives the biggest positive reward at the goal, although it takes a longer time.

1.3 Q-Learning

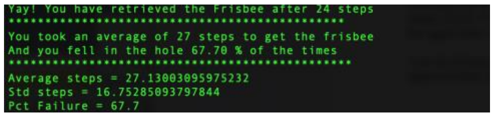
For this assignment, I have chosen the Q-Learning algorithm. It is a model-free approach whereby the agent does not spend time going through transition states and rewards. Rather, from interacting with the environment, it learns through an Epsilon-greedy strategy which randomly chooses between exploration and exploitation.

In Q-learning, epsilon corresponds to how much exploration the agent is taking. If the agent has epsilon at 0, the agent is not exploring; while an epsilon at 1 means the agent is exploring "greedily" all paths to find an optimal path. By default, the hiive mdptoolbox's Q-learning module has epsilon set at 1.0. In my Q-learning code, therefore, I defined the epsilon_decays (corresponding to the exploration rate) to be in the range of [0.9, 0.999].

Discount factor γ gives weight to future rewards. I defined γ(gammas) to be in the range of [0.8, 0.9, 0.99].

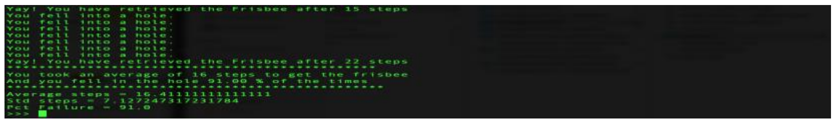
Learning rate $\underline{\alpha}$ (alpha) gives weight to past experiences. I defined alpha_decays to be in the range of [0.9, 0.999]. This alpha_decays parameter determines how large or how small of a leap the agent takes in the search for an optimal policy.

I ran the Q-learning algorithm for frozen-lake on 2 machines (Windows and Macbook) for approximately 4 hours each. The results I obtained on the Macbook run:



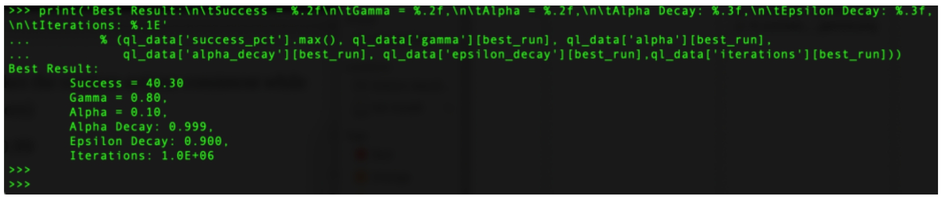
```
Yay! You have retrieved the Frisbee after 24 steps
*************************************************
You took an average of 27 steps to get the frisbee
And you fell in the hole 67.70 % of the times
**************************************************
Average steps = 27.13003095975232
Std steps = 16.75285093797844
Pct Failure = 67.7
```

tells me that under Q-learning, with a success rate of 32.3% (100 - 67.7 %), the agent retrieved the Frisbee in an average of 27 steps. The results I obtained on the Windows run:



```
Yay! You have retrieved the Frisbee after 15 steps
You fell into a hole.
You fell into a hole.
You fell into a hole.
You fell into a hole.
You fell into a hole.
You fell into a hole.
You fell into a hole.
Yay! You have retrieved the Frisbee after 22 steps
*************************************************
You took an average of 16 steps to get the frisbee
And you fell in the hole 91.00 % of the times
**************************************************
Average steps = 16.41111111111111
Std steps = 7.127247317231784
Pct Failure = 91.0
>>> 
```

showed that the agent retrieved the Frisbee in an average of 16 steps but the success rate was 9% (100 - 91.0). This was a little puzzling to me. However, they both yielded the same best_run and best_policy values:



```
>>> print('Best Result:\n\tSuccess = %.2f\n\tGamma = %.2f,\n\tAlpha = %.2f,\n\tAlpha Decay: %.3f,\n\tEpsilon Decay: %.3f,
\n\tIterations: %.1E'
...          % (ql_data['success_pct'].max(), ql_data['gamma'][best_run], ql_data['alpha'][best_run],
...             ql_data['alpha_decay'][best_run], ql_data['epsilon_decay'][best_run],ql_data['iterations'][best_run]))
Best Result:
        Success = 40.30
        Gamma = 0.80,
        Alpha = 0.10,
        Alpha Decay: 0.999,
        Epsilon Decay: 0.900,
        Iterations: 1.0E+06
>>>
>>>
```

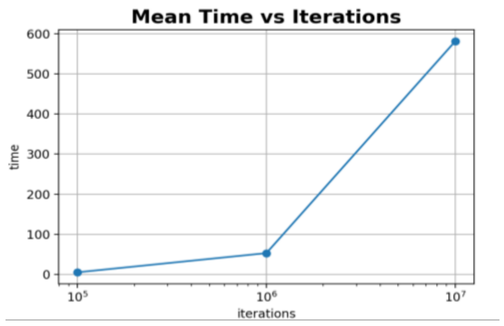


Fig. 8 - Mean Time vs Iterations

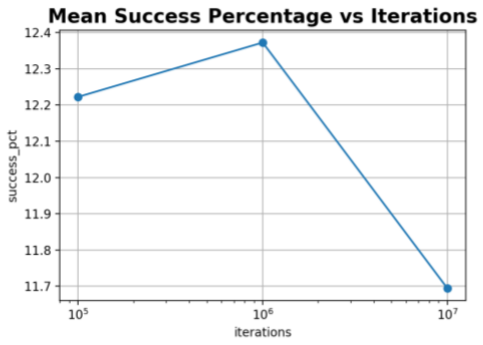


Fig. 9 - Mean Success Percentage vs Iterations

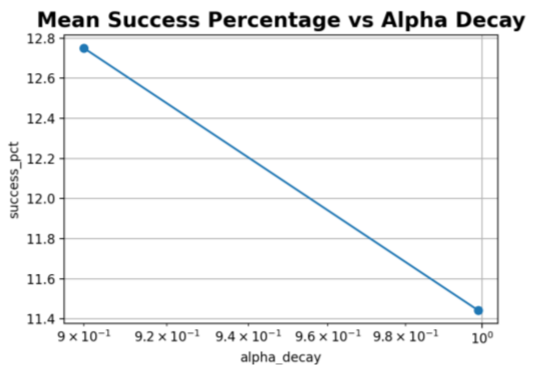


Fig. 10 - Mean Success Percentage vs. Alpha Decay

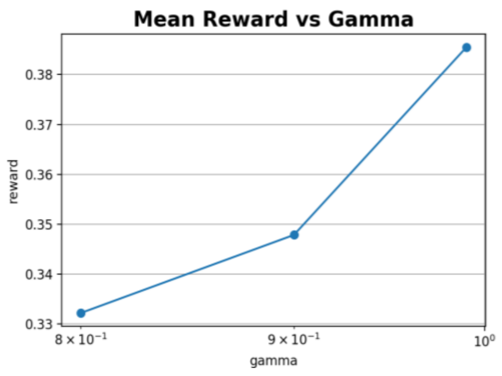


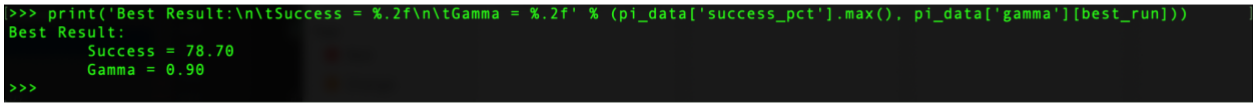
Fig. 11 - Mean Reward vs. Gamma

Both runs yielded the same graphs as shown in Figs. 8 - 11. Interesting point to note is Fig. 9. At iterations = 1000,000 (or 1 million), the mean percentage of success is highest, at around 12.38%. After that 1-million-th iteration, the mean percentage of success starts to decline. This gives me a clue that in future, there is no real benefit in running more than 1 million iterations. The optimal path is reached once the experiment runs for 1 million iterations.

Another interesting point to note is Fig. 10. Fig. 10 shows that as alpha decay gets closer to 1, the percentage success is lower. Initially, I was wondering if this observation is accurate. Pondering about it further, I found that it makes sense. As alpha decay gets closer to 1, there is not much change in alpha, this means that finding the optimal path is taking a longer time. Thus the rate of success is lower.

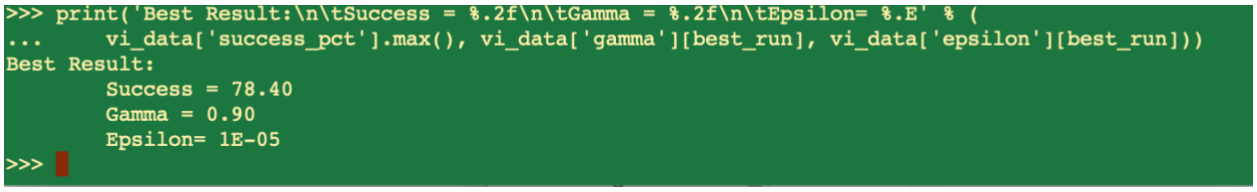
1.4 Comparison of the 3 Algorithms

The Q-learning algorithm, in exploring all possible state space, and choosing randomly between exploitation and exploration, translates to it taking the longest time (~4 hours) for it to find the optimal path to reach the goal. On the other hand, Policy Iteration and Value Iteration took much shorter time to converge to the optimal path to reach the goal. The Policy Iteration reported a success of ~78.70 % with gamma=0.90:



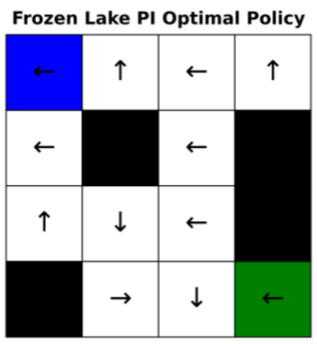
```
>>> print('Best Result:\n\tSuccess = %.2f\n\tGamma = %.2f' % (pi_data['success_pct'].max(), pi_data['gamma'][best_run]))
Best Result:
        Success = 78.70
        Gamma = 0.90
>>>
```

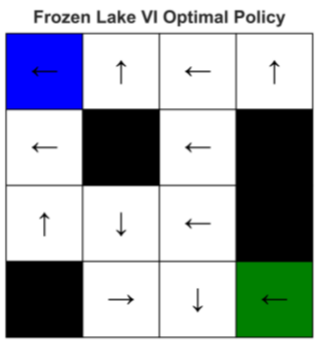
The Value Iteration reported a success of



```
>>> print('Best Result:\n\tSuccess = %.2f\n\tGamma = %.2f\n\tEpsilon= %.E' % (
...     vi_data['success_pct'].max(), vi_data['gamma'][best_run], vi_data['epsilon'][best_run]))
Best Result:
        Success = 78.40
        Gamma = 0.90
        Epsilon= 1E-05
>>>
```

78.40% with gamma=0.90. Both reported roughly similar success rates with the same gamma value. The Policy Iteration, however, converged faster from my observation. I believe this is due to the fact that during execution, the algorithm is repeating the policy evaluation and policy improvement in each iteration. That is to say, in each iteration, the Policy Iteration algorithm goes through 2 phases - one to evaluate the policy and one to improve on it. On the other hand, in Value Iteration, the algorithm starts with a random value or utility function, V(s) and covers these 2 phases by taking a maximum over the utility function for all possible actions. The Value Iteration algorithm, would therefore, take a slightly longer time than the Policy Iteration.





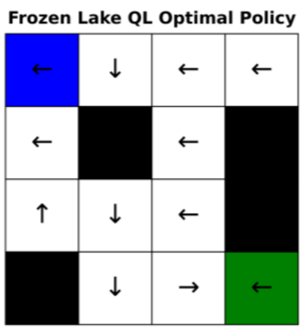


Fig. 12 – PI Optimal Policy *Fig. 13 - VI Optimal Policy* *Fig. 14 - QL Optimal Policy*

As we can see in Figs. 12 and 13, the optimal paths to reach the goal for Policy Iteration and Value Iteration look almost the same while the Q-Learning's optimal path looks different.

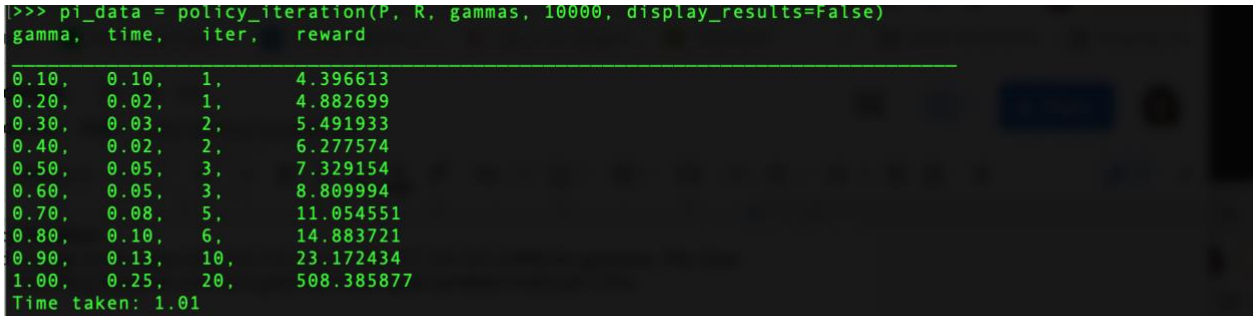
Q-learning makes an exhaustive exploration to check every possible path to find the globally optimal policy. Looking at Fig. 14, except for the down arrow [see top row, second column]into the black square (a Hole), all the other arrows in dodging the black squares make sense. I would have expected that cell [top row, second column] to have had an up arrow to avoid the black square. I do not have an explanation yet for why Q-learning algorithm advocates this move.

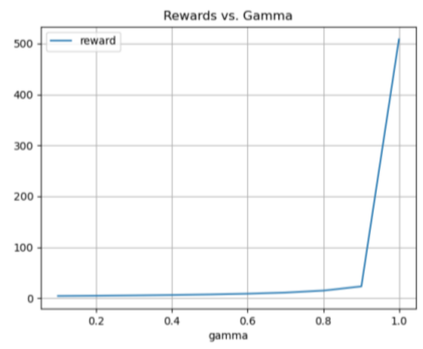
2. Forest Management

For the non-grid world problem, I chose Forest Management and tapped into hiive.mdptoolbox.example's forest() class. I defined a state space of 625. This problem has real relevance and benefit to the wildfire management of California, where I reside. In simulating this problem and presenting to a layman later on, the actions of WAIT and CUT are easy to understand and the rewards - having the profit from cutting and selling the wood or the probability of a wildfire that burns down the wood - easy to visualize. So, I could well foresee such a problem resolution bringing great relief and benefit to the state of California if I were to continue enhancing the project after this class.

2.1 Policy Iteration

I defined a range of 10 values [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.999] for gammas. In just 20 iterations, running Policy Iteration on this non-grid, 625 state-space problem took just 1.01s :


```
>>> pi_data = policy_iteration(P, R, gammas, 10000, display_results=False)
gamma,  time,   iter,   reward
________________________________________________________________________________
0.10,   0.10,   1,      4.396613
0.20,   0.02,   1,      4.882699
0.30,   0.03,   2,      5.491933
0.40,   0.02,   2,      6.277574
0.50,   0.05,   3,      7.329154
0.60,   0.05,   3,      8.809994
0.70,   0.08,   5,      11.054551
0.80,   0.10,   6,      14.883721
0.90,   0.13,   10,     23.172434
1.00,   0.25,   20,     508.385877
Time taken: 1.01
```



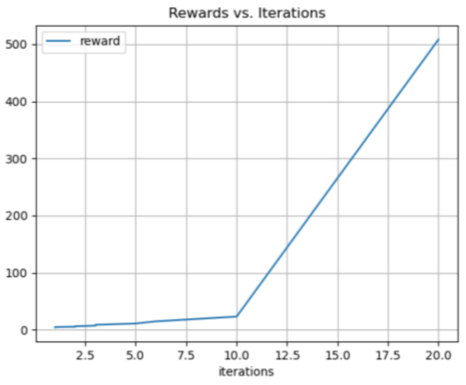


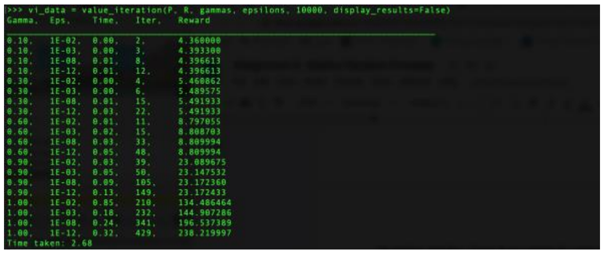
Fig. 15 – Rewards vs. Gamma

Fig. 16 - Rewards vs. Iterations

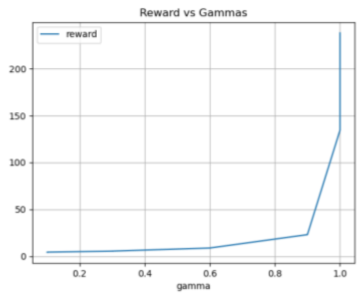
The findings from Figs. 15 and 16 also confirm the screen-shot capture, which is - at gamma close to 1.0 (say 0.999), the reward is around 500, but it takes 20 iterations to get to this greatest reward.

2.2 Value Iteration

For Value Iteration, I defined gammas = [0.1, 0.3, 0.6, 0.9, 0.999] and epsilons = [1e-2, 1e-3, 1e-8, 1e-12] for the experiment. It took 2.68s to run Value Iteration on this non-grid, 625-state space problem. This is twice as long as the time it takes to run Policy Iteration.



```
>>> vi_data = value_iteration(P, R, gammas, epsilons, 10000, display_results=False)
Gamma,  Eps.,    Time,   Iter,    Reward
______________________________________________________________________________
0.10,   1E-02,  0.00,   2,       4.360000
0.10,   1E-03,  0.00,   3,       4.393300
0.10,   1E-08,  0.01,   8,       4.396613
0.10,   1E-12,  0.01,   12,      4.396613
0.30,   1E-02,  0.00,   4,       5.460862
0.30,   1E-03,  0.00,   6,       5.489575
0.30,   1E-08,  0.01,   15,      5.491933
0.30,   1E-12,  0.03,   22,      5.491933
0.60,   1E-02,  0.01,   11,      8.797055
0.60,   1E-03,  0.02,   15,      8.808703
0.60,   1E-08,  0.03,   33,      8.809994
0.60,   1E-12,  0.05,   48,      8.809994
0.90,   1E-02,  0.03,   39,      23.089675
0.90,   1E-03,  0.05,   50,      23.147532
0.90,   1E-08,  0.09,   105,     23.172360
0.90,   1E-12,  0.13,   149,     23.172433
1.00,   1E-02,  0.85,   210,     134.486464
1.00,   1E-03,  0.18,   232,     144.907286
1.00,   1E-08,  0.24,   341,     196.537389
1.00,   1E-12,  0.32,   429,     238.219997
Time taken: 2.68
```



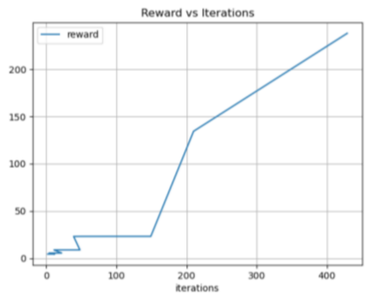


Fig. 17 - Reward vs Gammas

Fig. 18 - Reward vs. Iterations

What is interesting in the screen-shot capture is: at 1.0 gamma, the largest reward occurs when the epsilon is at its smallest. In Value Iteration, epsilon is the stopping criterion. When epsilon is at its smallest, this means the update to the optimal utility value, V(s) is smaller. This takes a longer time for the algorithm to converge to an optimal value, and thus the number of iterations is largest.

2.3 Q-Learning

Looking at the chart of Fig. 19, initially, I was puzzled how the reward would be the smallest when the alpha decay is at 1.00. When I mulled over this, I reasoned that when alpha decay is 1.00, it means there is not too much of a change in alpha. If there isn't too much of an update in the Q-table update, this means finding the optimal path is going to take longer. The percentage success is lower, and hence the reward is lower. So, Fig. 19 makes sense.

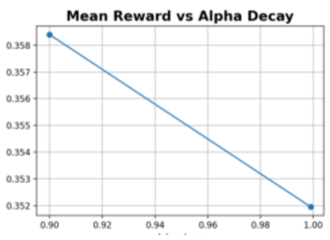


Fig. 19 – Mean Reward vs. Alpha Decay

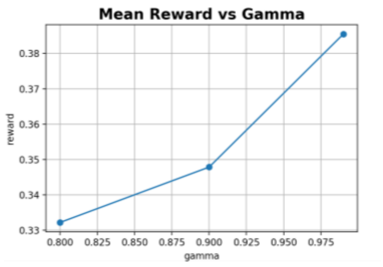


Fig. 20 – Mean Reward vs. Gamma

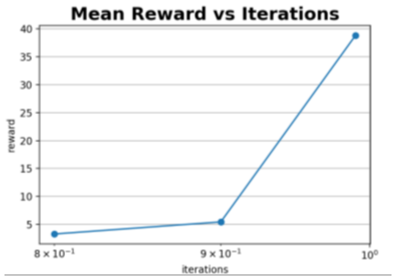


Fig. 21 – Mean Reward vs. Iterations

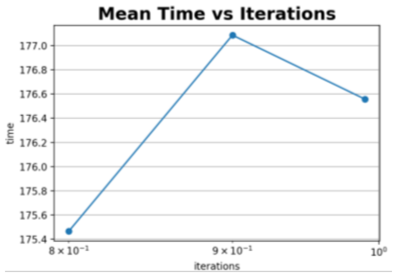


Fig. 22 – Mean Time vs. Iterations

Unlike Policy Iteration and Value Iteration, the Q-learning algorithm took approximately 4 hours to run. The best_run and best_policy yield the following results in 10,000,000 iterations:

- Reward = 47.68
- Gamma = 0.99
- Alpha = 0.20
- Alpha Decay = 0.999
- Epsilon Decay = 0.999

2.4 Comparison

As can be seen in the output of plot_forest_management(), Policy Iteration and Value Iteration algorithms both converged to the same optimal policy results as shown in Figs. 23 and 24. For the sake of presenting the 1-dimensional policy array, I have propagated the result into a 2-dimensional array. The topmost left cell corresponds to the youngest tree, while the bottom row's rightmost cell corresponds to the oldest tree. A "W" in a green cell denotes a "Wait" action, while a "C" in a black box denotes a "Cut" action.

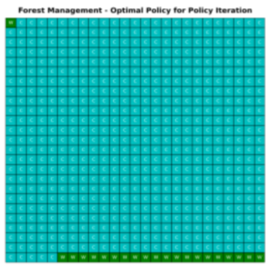


Fig. 23 – Optimal Policy for Policy Iteration

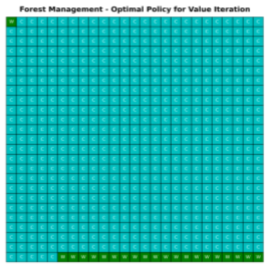


Fig. 24 – Optimal Policy for Value Iteration

What the 2 policies above advocate is to cut a tree every year, and then as we get closer to the end, with the oldest trees in the forest, we would wait.

For Policy Iteration, the best_run policy is when gamma=0.999 and the reward is 508.39. For the Value Iteration, the best_run policy is when gamma=0.999 and the reward is 238.22. On the other hand, for Q-learning at gamma=0.99, the reward is only 47.68. As Q-learning algorithm exhaustively searches every cell, we can see in Fig. 25 that it advocates trees in certain age-groups to be **CUT**, and **WAIT** on the trees in certain age-groups.

2.5 Experiments

p = 0.9

As I mentioned before, this forest management problem is highly applicable and relevant to the management of the wildfire situation in California. I set out to vary p, the probability of wildfire occurrence , r1 - the reward when the action is WAIT, and r2 - the reward when the action is CUT.

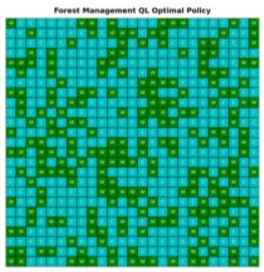
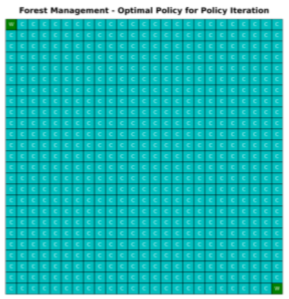


Fig. 25 - Forest Management QL Optimal Policy

The default value for p is 0.1. In my first experiment, I set p to be 0.9. In light of a great probability of wildfire (p=0.9), both Policy Iteration (Fig.26) and Value Iteration (Fig. 27) advocate the same policy - i.e. WAIT on the youngest-aged (topmost left) trees and the oldest-aged (bottom rightmost) trees, and CUT the rest and sell them for



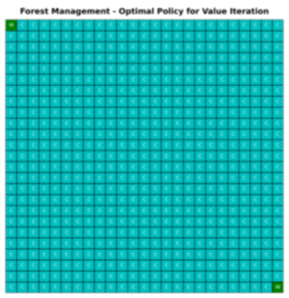


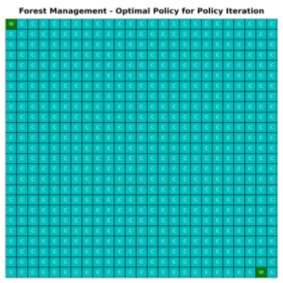
Fig. 26 - Optimal Policy for Policy Iteration

Fig. 27 - Optimal Policy for Value Iteration

profit.

r1 = 0.4

The default value of r1 is set to 4 in mdptoolbox. This is the reward when the action is WAIT. The default value of r2 is set to 2 in mdptoolbox. This is the reward when the action is CUT.
I decided to let the value of r2 stay at 2, while I changed the value of r1 to 0.4. This means the reward for CUTTING the trees is greater than WAITING. I ran Policy Iteration and Value Iteration on the forest management MDP. The optimal policy for both are as shown:



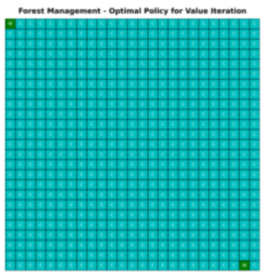


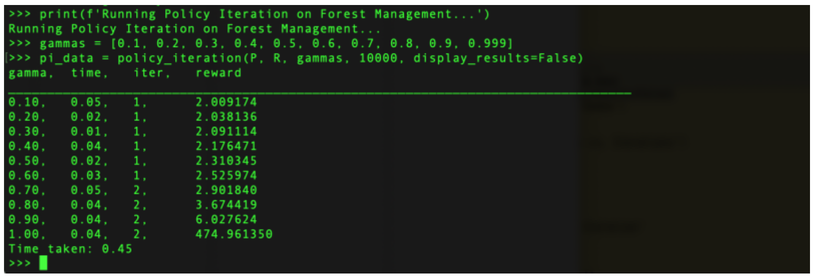
Fig. 28 - Optimal Policy for Policy Iteration

Fig. 29 - Optimal Policy for Value Iteration

No surprise there on Figs. 28 and 29, since r2, the reward for CUTTING is greater than r1, the reward for WAITING.

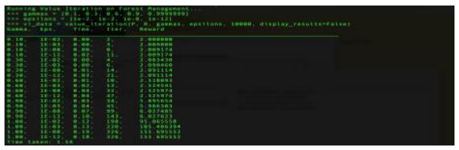
We see in Figs. 28 and 29, the advocate is for most of the trees to be CUT while we WAIT on the youngest-aged tree and the second-to-the last oldest-aged trees.

Something to point out though is the true power of policy iteration: with only two iterations, one arrived at the optimal policy for forest management:



```
>>> print(f'Running Policy Iteration on Forest Management...')
Running Policy Iteration on Forest Management...
>>> gammas = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.999]
>>> pi_data = policy_iteration(P, R, gammas, 10000, display_results=False)
gamma,  time,   iter,   reward
________________________________________________________________________________
0.10,   0.05,   1,      2.009174
0.20,   0.02,   1,      2.038136
0.30,   0.01,   1,      2.091114
0.40,   0.04,   1,      2.176471
0.50,   0.02,   1,      2.310345
0.60,   0.03,   1,      2.525974
0.70,   0.05,   2,      2.901840
0.80,   0.04,   2,      3.674419
0.90,   0.04,   2,      6.027624
1.00,   0.04,   2,      474.961350
Time taken: 0.45
>>> 
```

For value iteration on this same forest management MDP, it took about 326 iterations:



```
Running Value Iteration on Forest Management...
>>> gammas = [0.1, 0.3, 0.6, 0.9, 0.9999999]
>>> epsilons = [1e-2, 1e-3, 1e-8, 1e-12]
>>> vi_data = value_iteration(P, R, gammas, epsilons, 10000, display_results=False)
Gamma,  Eps,    Time,   Iter,   Reward
________________________________________________________________________________
0.10,   1E-02,  0.00,   2,      2.000000
0.10,   1E-03,  0.00,   3,      2.009000
0.10,   1E-08,  0.00,   8,      2.009174
0.10,   1E-12,  0.02,   11,     2.009174
0.30,   1E-02,  0.00,   4,      2.083430
0.30,   1E-03,  0.00,   6,      2.090460
0.30,   1E-08,  0.01,   14,     2.091114
0.30,   1E-12,  0.02,   21,     2.091114
0.60,   1E-02,  0.01,   10,     2.510093
0.60,   1E-03,  0.02,   13,     2.524541
0.60,   1E-08,  0.03,   32,     2.525974
0.60,   1E-12,  0.04,   47,     2.525974
0.90,   1E-02,  0.03,   34,     5.895654
0.90,   1E-03,  0.04,   45,     5.986303
0.90,   1E-08,  0.07,   99,     6.027485
0.90,   1E-12,  0.10,   143,    6.027623
1.00,   1E-02,  0.12,   190,    95.065558
1.00,   1E-03,  0.13,   220,    105.406394
1.00,   1E-08,  0.19,   326,    133.695532
1.00,   1E-12,  0.18,   326,    133.695532
Time taken: 1.56
```

r2 = 0.2
I let the value of r1 stay at 4, while I changed r2 to 0.2. This means the reward of WAITING is greater than the reward of CUTTING. Again, I ran Policy Iteration and Value Iteration on the forest management MDP. The optimal policy for both are as shown:

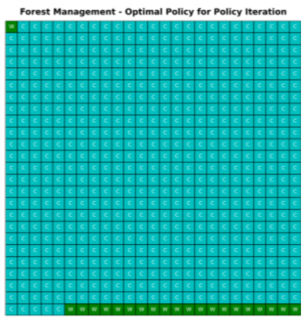


Fig. 30 - Optimal Policy for Policy Iteration

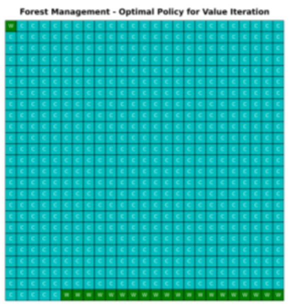


Fig. 31 - Optimal Policy for Value Iteration

The advocate is for initial WAIT on the youngest-age trees, CUT for the subsequent intermediate-age trees, and finally a WAIT on the oldest-age trees.

Again, policy iteration took about 20 iterations to converge, and the maximum reward is 508. The value iteration took about 429 iterations to converge, and the maximum reward is 238. The power of policy iteration shows itself again.

California has worrisome wildfire seasons in recent years. The solutions presented here, though simplified, is a good simulation and gives good insight if California were to adopt MDP for its forest management.