Problem 1

March 20, 2024

1 1.a

State space In a finite MPD, a **state space** is defined as all of the possible configurations the agent can be in given its environment. To create a state space we need to define all the pieces of important information to represent the MPD problem. I took into account the elevator, elevator floor, door status, call floors, exit floors, and location of the person. So we can say:

$State\_Space = ((E_A, F_{A_i}, D_{A_k}), (E_B, F_{B_j}, D_{B_l}), ((F_{Call_m}, F_{Exit_o}, F_{Location_p}), (F_{Call_n}, F_{Exit_p}, F_{Location_r})))$

for all $i, j, k, l, m, n, o, p, q, r$

Where:

- E is one of the elevators
- F is the floor of one of the elevators
- $Door$ is the status of the door (OPEN or CLOSED)
- F_{Call} is the floor a person is calling from (0 if no passenger)
- F_{Exit} is the floor a person wants to exit (0 if no passenger)
- $F_{Location}$ is where the person is located (IN_A, IN_B, or WAITING)
- $i, j \epsilon [1, 2, 3, 4, 5, 6]$
- $m, n, o, p \epsilon [0, 1, 2, 3, 4, 5, 6]$
- $k, l \epsilon [open, closed]$
- $q, r \epsilon [in\_a, in\_b, waiting]$

The state space would be of size:

$State\_Space| = |((1, 6, 2), (1, 6, 2), ((7, 7, 3), (7, 7, 3)))| =$

$= |1 * 6 * 2 * 1 * 6 * 2 * 7 * 7 * 3 * 7 * 7 * 3| =$

$= 3111696$ states

Action space Simarly we define the **action space** as the unique actions that can be taken at a time step. Given there are 2 elevators, each elevator moves independently, and we have the action set for elevator as $UP, DOWN, HOLD, DOORS$ we can define:

$Action\_Set = ['UP', 'DOWN', 'HOLD', 'DOORS']$

$Action\_Space = ((Action\_Set_i, E_A), (Action\_Set_j, E_B))$

for all i, j

Where:

- E_A is elevator A
- E_B is elevator B

The size of the action space is $4 * 1 * 4 * 1) = 16$

Time step Lastly, we need to define a timestep. Since the elevator can only make decisions every 5 seconds, we can decide our timestep to be 5 seconds. And this timestep still holds even given the arrival rates, as we can model the arrivals rate by the number of seconds in the timestep.

2 1.b

Reward Function For the reward function we care about the wait times. We need to devise a way to reward the agent when it minimizes the wait time of an action and penalize it when it lengthens the wait time of a passenger. We can define the reward psuedocode function as follows:

```
def reward(s, a, s')

    reward = 0
    m_r = movment reward
    m_p = movement penalty

    d_r = door reward
    d_p = door penalty

    h_r = hold reward
    h_p = hold penalty

    given action a, current state, and new state:

        if closer to exit floor:
            reward += m_r
        else:
            penalize -= m_p

        if closer to call floor
            reward += m_r
        else:
            penalize -= m_p

        if closer doors opened and departed passenger:
            reward += d_r
        else:
            penalize -= d_p
        if hold doors and departed passengers
            reward += h_r
        else:
            penalize -= h_p

    return reward
```

This function would go through the action pairs for elevator A and B and aggregate the rewards for the corresponding action.

Utility Additionally, generally, we typically define **Utility** as the expected sum of future rewards:

$Utility = reward(s, a, s')_t + \gamma_k * reward(s, a, s')_{t+1} + ... + \gamma^k * reward(s, a, s')_{t+n}$

Where gamma is the discount factor.

In terms of actual implementation of the **Utility function** in our MPD problem, we implicity define the utility of the state-actions inside a **Q table**, which is the expected future rewards of each state action pairs:

$Q[state][action] = Utility(s, a)$

Where $Utility(s, a)$ is a method to determine the expected future/expected rewards. The implementation of $Utility(s, a)$ would cahnge given on which RL algorithm being implemented.

3 1.c

To define a simulator for the elevator system, we need to define:

- state transition rules for each step
- a simulation for passengers
- a reward function

You can view the environment model / simulator implementation HERE

4 1.d

If we wanted to model impatience, we would have to incorporate **elapsed time** to our **state space** and **reward function** (the action space, given my implementation should be unchanged).

- For our state space, we would liekly have to keep track of the time elapsed. Thw downside to this approach is that time is continous, and would likely exponentially increase the state space. Even if you discretized time, the state space would still be large.
- For our reward function, we would have to use the elapsed time we keep track of in the state space and use that to add to the aggregate rewards for a single step. we would likely change the time to a negative number and add that to the rewards to incentivize shorter times: $reward = -T_{time} + reward$
- For our action space, it should remain unchanged.

5 MPD

Putting all our definition together we define our elevator MPD as :

$MPD(S, A, T, R, S_0) =$ -

- $s_a \,\epsilon\, S \, where \, s_a = ((E_A, F_{A_i}, D_{A_k}), (E_B, F_{B_j}, D_{B_l}), ((F_{Call_m}, F_{Exit_o}, F_{Location_p}), (F_{Call_n}, F_{Exit_p}, F_{Location_r})))$
- $a_a \,\epsilon\, A \, where \, a_a = ((Action\_Set_i, E_A), (Action\_Set_j, E_B))$
- $T(s, a, s') = P(f_e | f_c)$
- $S_0 = s_{a_0} = ((A, 1, 1), (B, 1, 1), ((0, 0, waiting), (0, 0, waiting)))$
- $R = R(s, a, a')$

Problem 2

March 20, 2024

6 2.a.i - 2.a.iii

Q-learning has been implemented and the graphs for 2.i - 2.iii can be viewed HERE.

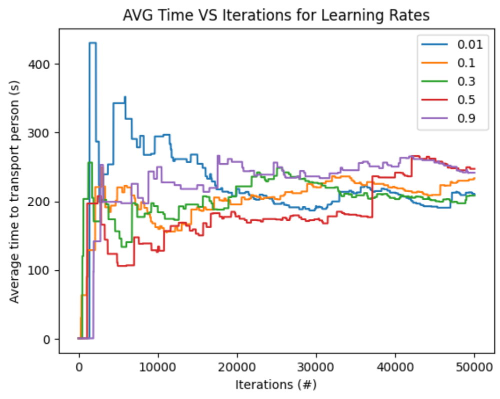
7 2.b.i - 2.b.iii

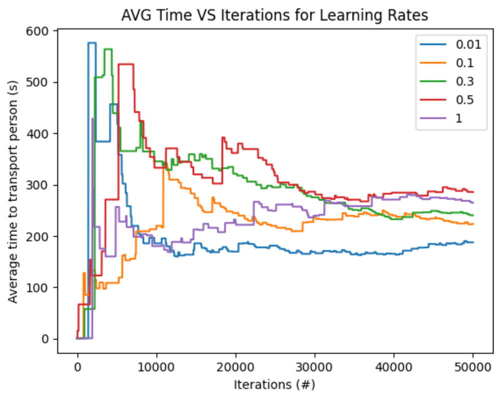
SARSA-learning has been implemented and the graphs for 2.i - 2.iii can be viewed HERE.

7.1 2.c.i Q Learning VS SARSA Learning

The main difference between Q learning and SARSA are how the Q table is updated after each action. So we dont notice too many differences. In terms of average rewards accross all alphas, gammas, and epsilons for both Q learning and SARSA, there was not much of a difference. However, in terms of average passenger wait times, there was a difference. In SARSA, there was much less variance between the average wait times across the changing alpha, gamma, and epsilon values. But in Q learning, there was a great variance between the avergae wait times for the changing variables.

For example, notice how different in the variance in values for Q Learning and SARSA for alpha values, respectively:

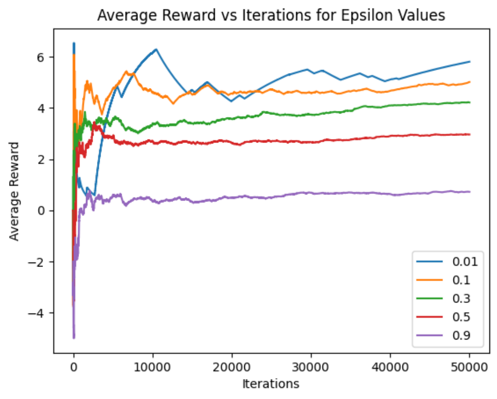




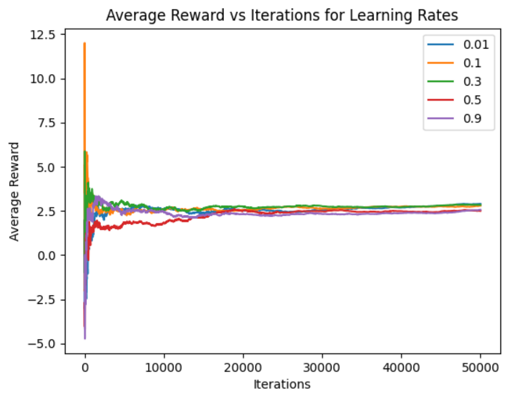
In SARSA, the values are closer to each other than in Q Learning, and this is reflected for both gamma and epsilon. Overall, there was not much of a difference between the end result of the best combinations of values for Q learning and SARSA learning.

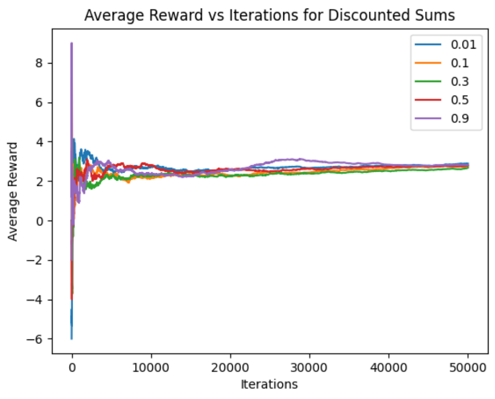
7.2 2.c.i Alpha VS Exploration VS GAMMA

For both Q learning and SARSA learning, changing the exploration rate had yielded the highest avergae rewards, with an averge reward of 5. And there was also a high variance among the average rewards when changing the exploration rates. Here is an the exploration rate graph for avg rewards for Q learning:



For the gamma and alpha values, they eventually all seemed to converge to the same value for both SARSA and Q learning. Here are the average rewards for changing gamma and alpha in Q learning:





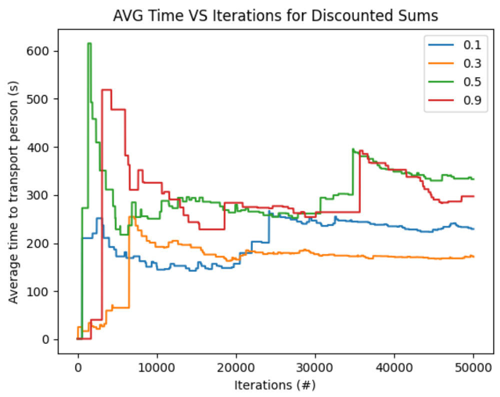
Overall, it seemed the best learning rate, discount factor, and exploration rate for both Q learning and SARSA was learning rate = .01, discount factor = .01, and exploration rate = .5 over 50,000 iterations. This means our agent needs to learn slow, care only about immediate rewards, and choose other actions 50% of the time, over 13.89 hours for it to take an average of about 3.25 minutes for the elevator to arrive to a person's call floor and take them to their exit floor.

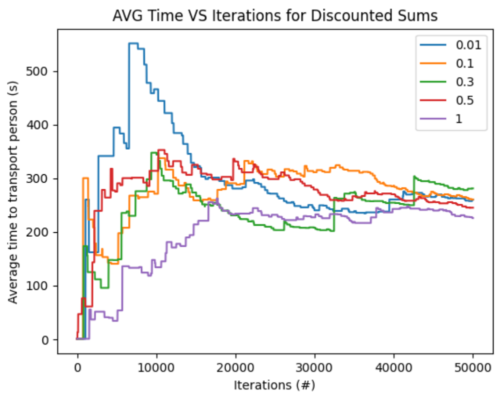
7.3 2.c.ii Q Learning VS SARSA Learning

There was not much of a difference between rewards. Results were similar to 2.i; SARSA had a higher variance in wait times than Q learning.

7.4 2.c.ii Alpha VS Exploration VS GAMMA

Changing the exploration rate still yielded the highest rewards and their is still not much of a difference between the avg rewards for changing the gamma and changing the alpha. However, now, having a higher discount factor lowers the wait time. Here is what the avg wait time graph looks like for Q learning and SARSA with discount factors, respectively.





Overall, we see that the system at best preferred an alpha = .5 for Q, .01 for SARSA, gamma = .5 for Q, 1 for SARSA, and epsilon = .3 for Q, .5 for SARSA. This means we preffered an agent that learned at a modereate rate, cared about future actions, and stuck with actions that yielded high rewards over 13.89 hours for it to take an average of about 4 minutes for the elevator to arrive to a person's call floor and take them to their exit floor.

7.5 2.c.iii Q Learning VS SARSA Learning

There was not much of a difference between avg rewards for Q learning over SARSA across alphas, gammas, and epsilons. Sometimes, one did worse or better than the other, but the outcomes seem the same. Even in terms of average wait times, they were much more similar to each other compared to 2.i and 2.ii.

7.6 2.c.iii Alpha VS Exploration VS GAMMA

For 2.iii, SARSA learning seemed much less prone to changing values. When controlling the alpha, gamma, and epsilon values, the avg wait times and rewards did not change much. However for Q learning, wehn we changed the values, we saw there was either a noticable increase or decrease in wait time.

Overall, we see that the system at best preferred an agent that learned slowly , cared about immediate rewards, and chose other action 50% of the time over 13.89 hours for it to take an average of about 4 minutes for the elevator to arrive to a person's call floor and take them to their exit floor.

Problem 3

March 20, 2024

8 3.a.i - 3.a.iii

Q-learning-lambda has been implemented and the outputs for 2.i - 2.iii can be viewed HERE

The lambda algorithms take a very long time to complete. You can modify the number of iterations or modify the state space size to shorten the time in globals.py

Modify state space

```
# globals.py
NFLOORS = 3
START_FLOORS = [1]
START_PROB = [1]
EXIT_FLOORS = [2,3]
EXIT_PROB = [.5, .5]
FLOORS = [1, 2, 3]
```

OR

Modify Iterations

```
# globals.py
ITERATIONS = 1
```

9 3.b.i - 3.b.iii

SARSA-lambda has been implemented and the outputs for 2.i - 2.iii can be viewed HERE

The lambda algorithms take a very long time to complete. You can modify the number of iterations or modify the state space size to shorten the time in globals.py

Modify state space

```
# globals.py
NFLOORS = 3
START_FLOORS = [1]
START_PROB = [1]
EXIT_FLOORS = [2,3]
EXIT_PROB = [.5, .5]
FLOORS = [1, 2, 3]
```

OR

Modify Iterations

```
# globals.py
ITERATIONS = 1
```

10 3.c.i - 3.c.iii

The issue with Eligibility traces for this problem The eligbility traces absolutely tanked the learning performance. To elucidate, for $Sarsa(\lambda)$ or $Q(\lambda)$ learning, we initialize an eligibility trace that is the same size as the Q table to keep a record of the occurence of the state-action pairs:

$|e(s, a)| = |Q(s, a)|$

And recall, given our state space and action space, we have a Q table with a large amount of state-action pairs:

$|Q\_table| = |State\_space| * |Action\_space| = 3111696 * 16 = 49,787,136$

So this means we have two tables of size 49,787,136. This isnt a major problem alone, but, the bigger issue lies in the latter half of the λ algorithms:

```
for all s
    for all a
        update Q
        update E
```

This requires us to loop through the entire set of state-action pairs, which is of size 49,787,136, for every step just to update our Q and Eligibility tables. For my PC, this took about 30 seconds each step, making my agent learning, given the state space size, practically impossisble.

Possible Solutions?

One way would be to reduce the size fo the state space. I made it possible for the user to modify the problem desctiption so they could use less floors if they wanted to for problems 2.i and 2.ii in the globals.py file:

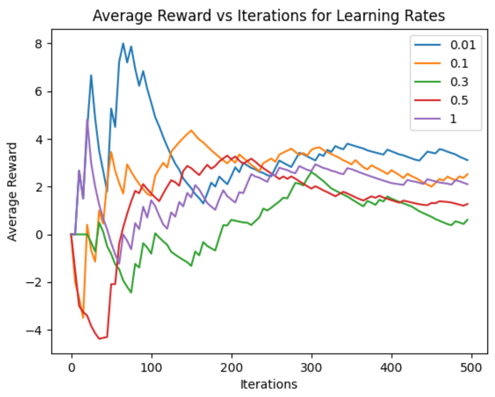
```
NFLOORS = 3
START_FLOORS = [1]
START_PROB = [1]
EXIT_FLOORS = [2,3]
EXIT_PROB = [.5, .5]
FLOORS = [1, 2, 3]
```

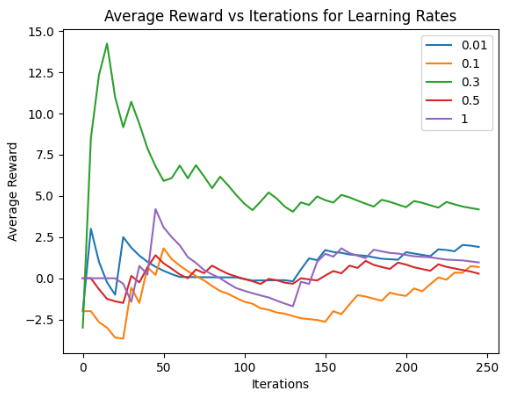
This would reduce the problem to 3 floors instead of 6, which would make the learning 37 x faster.

You could also use helper classes to keep track of state information rather than using the state space. For example, instead of keeping track of passengers in the satte space, you could just make a class, which is just as valid, as long as you define this class mathematically to be apart of you state space.

Algorithm Learning Curves

To get any useful inforamtion, I chose to reduce the state space size and reduce the number of iterations. Here is the alpha rewards graph for 2.i for Q lambda and SARSA lambda, respectively:





In conclusion, I wouldnt recommend using the lambda algorithms for large states spaces unless you have used methods to reduce the effect of the curse of dimensionality, otherwise, your learning time will be substantionally increased.