CS6700 - Reinforcement Learning
Programming Assignment-3
CS23S024 and NS24Z074

GitHub Repo link:
https://github.com/gokull1998/CS6700_PA3_CS23S024_NS24Z074

SMDP and Intra-option Q learning for Taxi-v3 environment

Environment overview:

The environment for this task is the taxi domain, illustrated in Fig. 1. It is a 5x5 matrix, where each cell is a position your taxi can stay at. There is a single passenger who can be either picked up or dropped off, or is being transported.

There are four designated locations in the grid world indicated by R(ed), G(reen), Y(ellow), and B(lue). When the episode starts, the taxi starts off at a random square and the passenger is at a random location. The taxi drives to the passenger's location, picks up the passenger, drives to the passenger's destination (another one of the four specified locations), and then drops off the passenger. Once the passenger is dropped off, the episode ends.

There are 500 discrete states since there are 25 taxi positions, 5 possible locations of the passenger (including the case when the passenger is in the taxi), and 4 destination locations.

Note that there are 400 states that can actually be reached during an episode. The missing states correspond to situations in which the passenger is at the same location as their destination, as this typically signals the end of an episode. Four additional states can be observed right after a successful episodes, when both the passenger and the taxi are at the destination. This gives a total of 404 reachable discrete states.

Passenger locations: 0: R(ed); 1: G(reen); 2: Y(ellow); 3: B(lue); 4: in taxi
Destinations: 0: R(ed); 1: G(reen); 2: Y(ellow); 3: B(lue)
Rewards:
• -1 per step unless other reward is triggered.
• +20 delivering passenger.
• -10 executing "pickup" and "drop-off" actions illegally.
The discount factor is taken to be $\gamma = 0.9$.

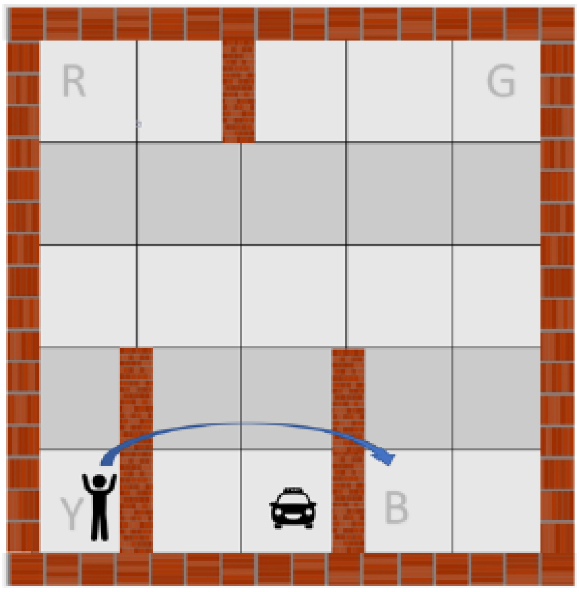


Figure 1: Taxi Domain

Actions and options for this environment -
Actions: There are 6 discrete deterministic actions: 0: move south; 1: move north; 2: move east; 3: move west; 4: pick passenger up; and 5: drop passenger off.
Options: Options to move the taxi to each of the four designated locations, executable when the taxi is not already there.

SMDP Q Learning

The SMDP algorithm is as follows:

- Initialize Q(s, o) arbitrarily for all s ∈ S, o ∈ O.
- Repeat until convergence:
    - For each episode:
        - Initialize the state s.
        - Choose an option o using the current policy π.
        - Repeat until termination:
            - Take action a according to o.
            - Observe reward r and next state s'.

Update the Q-value of the current state-action-option triplet:

- $Q(s, o) \leftarrow Q(s, o) + \alpha * (r + \gamma * Q(s', \pi(s')) - Q(s, o))$
- Update the state: s <- s'.
- Update the policy π based on the learned Q-values.
- Check convergence by monitoring the changes in Q-values or the performance of the policy over episodes.

In addition to the primitive actions, there are 4 options defined for this environment such as Red(), Green(), Yellow() and Blue() which move the taxi to the corresponding locations in the grid to pickup/drop off the passenger.

These options have deterministic policies and are defined as shown below:

```
def Red(env,state):

   optdone = False
   taxi_row, taxi_col, _, _ = env.decode(state)

   red_grid = np.array([
                        [1, 3, 0, 0, 0],
                        [1, 3, 3, 3, 3],
                        [1, 3, 3, 3, 3],
                        [1, 1, 1, 1, 1],
                        [1, 1, 1, 1, 1]
                        ])

   optact = red_grid[taxi_row,taxi_col]

   if (taxi_row, taxi_col) == (0,0):
       optdone = True

   return [optact, optdone]
```

```
def Green(env,state):

   optdone = False
   taxi_row, taxi_col, _, _ = env.decode(state)

   green_grid = np.array([
                        [0, 0, 2, 2, 1],
                        [2, 2, 2, 2, 1],
                        [2, 2, 2, 2, 1],
                        [1, 1, 1, 1, 1],
                        [1, 1, 1, 1, 1]
                        ])

   optact = green_grid[taxi_row,taxi_col]

   if (taxi_row, taxi_col) == (0,4):
       optdone = True

   return [optact, optdone]


def Yellow(env,state):

   optdone = False
   taxi_row, taxi_col, _, _ = env.decode(state)

   yellow_grid = np.array([
                        [0, 0, 0, 0, 0],
                        [0, 3, 3, 3, 3],
                        [0, 3, 3, 3, 3],
                        [0, 1, 1, 1, 1],
                        [0, 1, 1, 1, 1]
                        ])

   optact = yellow_grid[taxi_row,taxi_col]

   if (taxi_row, taxi_col) == (4,0):
       optdone = True

   return [optact, optdone]


def Blue(env,state):
   optdone = False
   taxi_row, taxi_col, _, _ = env.decode(state)

   blue_grid = np.array([
                       [0, 0, 2, 0, 3],
                       [2, 2, 2, 0, 3],
                       [2, 2, 2, 0, 3],
                       [1, 1, 1, 0, 3],
                       [1, 1, 1, 0, 3]
                       ])

   optact = blue_grid[taxi_row,taxi_col]

   if (taxi_row, taxi_col) == (4,3):
       optdone = True

   return [optact, optdone]

all_options = ["south", "north", "east", "west", "pick", "drop",
"Red","Green","Yellow","Blue"]
```

The SMDP Q learning algorithm code for primitive actions and an option (Red) is given below:

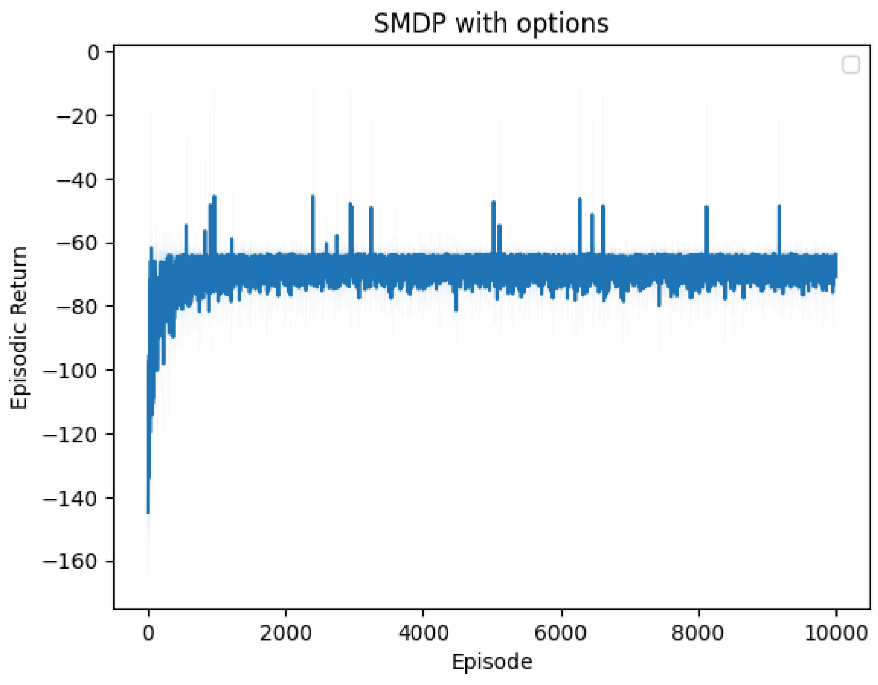
```
while ts < 100:

           # Choose action
           action = egreedy_policy(q_values_SMDP, state,
epsilon=epsilon)

           taxi_row, taxi_col, pass_loc, dest_loc =
env.decode(state)

           # Checking if primitive action
           if action < 6:

               # Perform regular Q-Learning update for
state-action pair
               next_state, reward, done, _, _ = env.step(action)
               q_values_SMDP[state, action] += alpha * (reward +
gamma * np.max(q_values_SMDP[next_state]) - q_values_SMDP[state,
action])
```

```
            state = next_state
            episode_reward += reward


        # Checking if action chosen is an option


        if (taxi_row, taxi_col) != (0,0) and action == 6: #
action => Red option
            reward_bar = 0
            opt_ts = 0

            optdone = False
            while (optdone == False):

                optact,optdone = Red(env,state)
                """print('Optact:',all_options[optact])
                print('Optdone:',optdone)"""
                next_state, reward, done, _, _ =
env.step(optact)


                reward_bar = gamma*reward_bar + reward

                state = next_state

                opt_ts += 1
                ts += 1

                if opt_ts > 8:
                    break

            # Complete SMDP Q-Learning Update
            q_values_SMDP[state, action] += alpha *
(reward_bar + gamma * np.max(q_values_SMDP[next_state]) -
q_values_SMDP[state, action])


            episode_reward += reward_bar
```

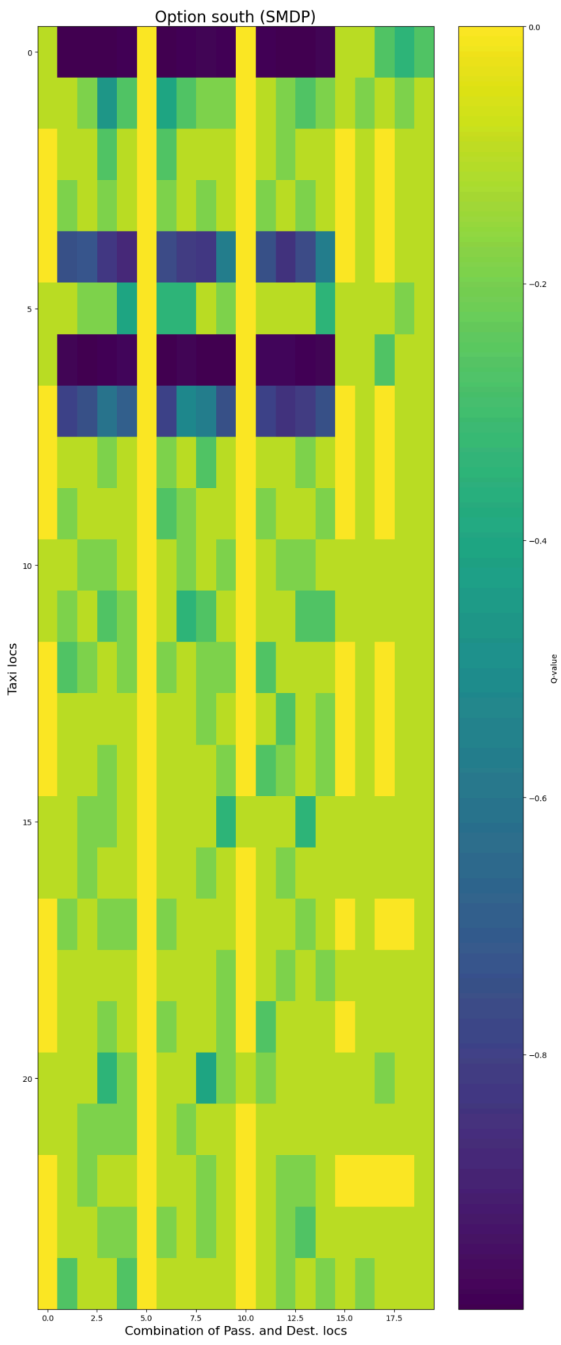
Results -

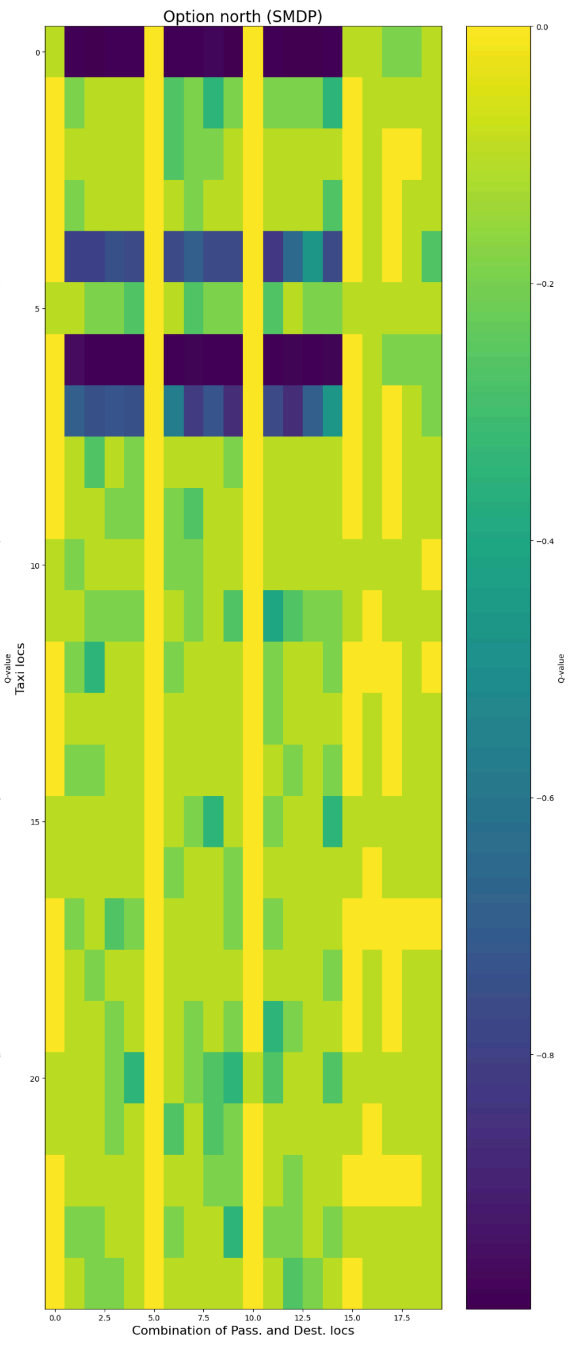
Best set of hyperparameters -
Alpha = 0.1
Gamma = 0.9
Epsilon = 0.1

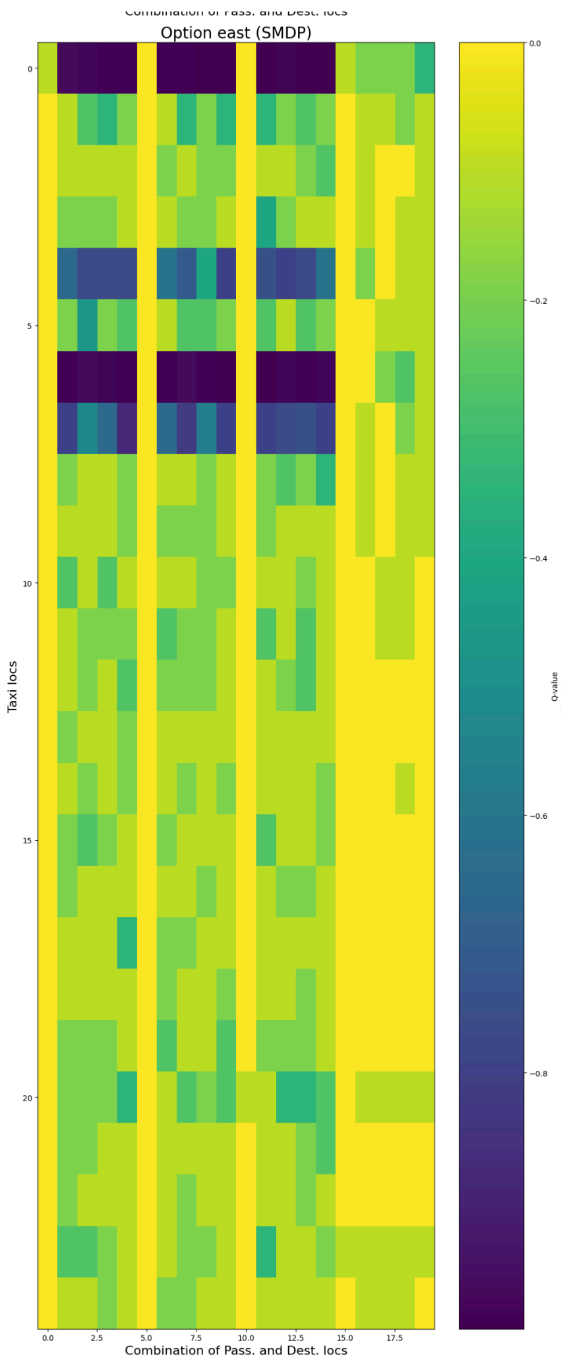
The reward curve for the given options is shown below.

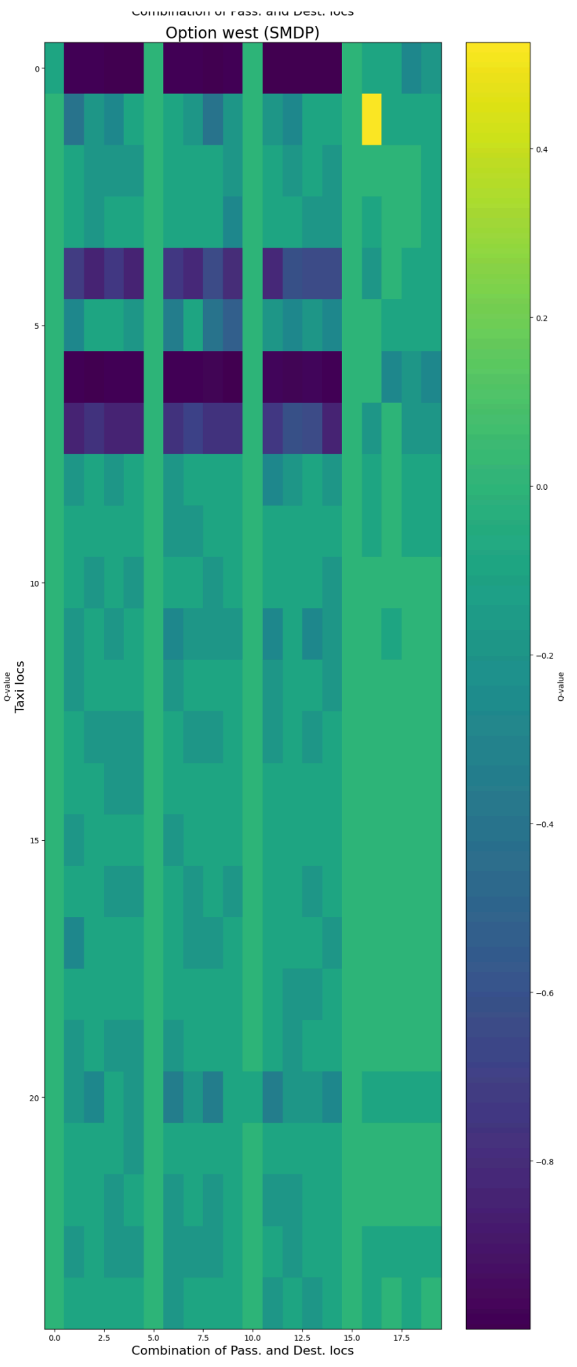


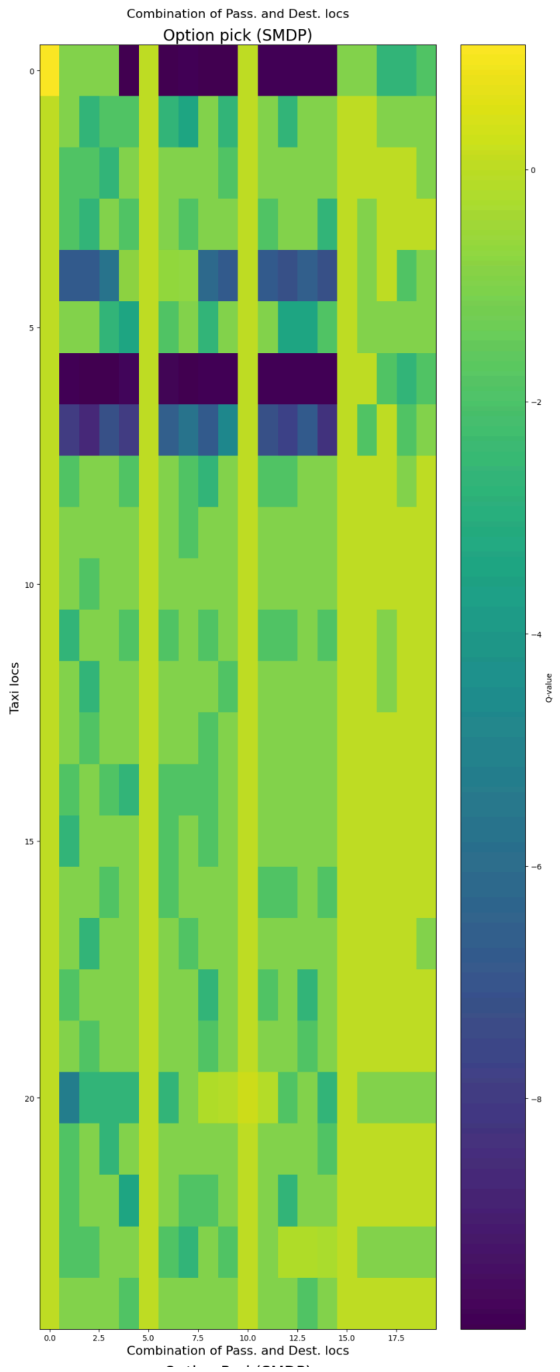
The Q tables for all options (including primitive actions) is shown below. The rows are the taxi locations in the grid and the columns are all combinations of passenger and destination locations.


Option south (SMDP)
Taxi locs
Combination of Pass. and Dest. locs
Q-value
0
5
10
15
20
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0.0
−0.2
−0.4
−0.6
−0.8

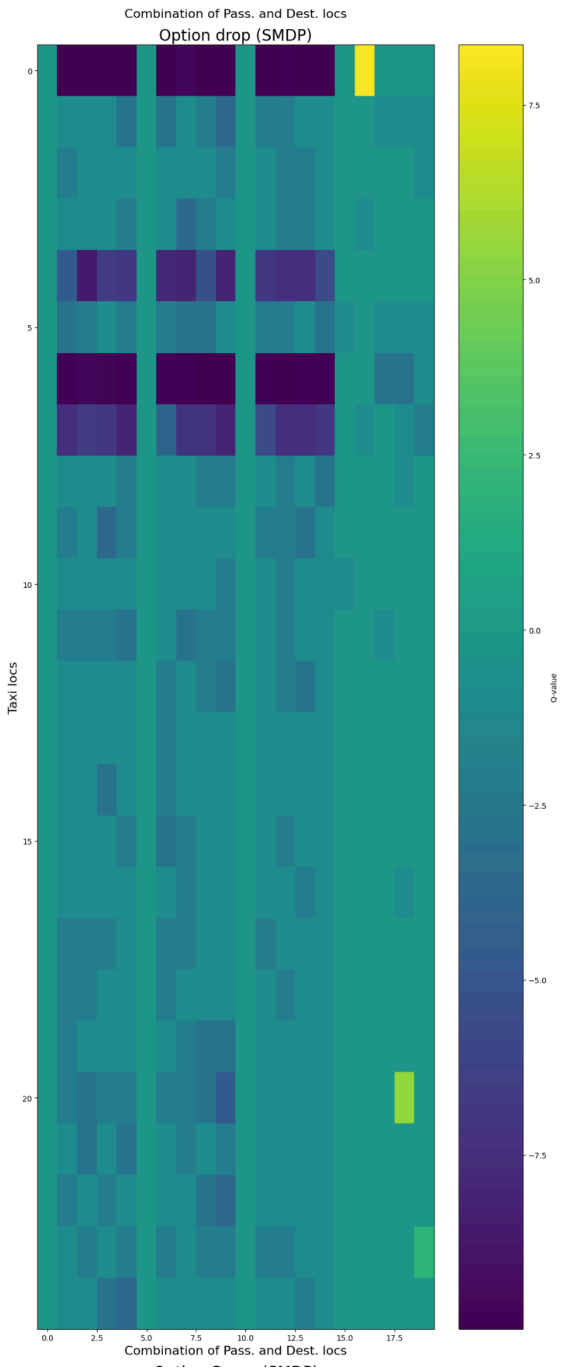

Option north (SMDP)
Taxi locs
Combination of Pass. and Dest. locs
Q-value
0
5
10
15
20
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0.0
−0.2
−0.4
−0.6
−0.8


Combination of Pass. and Dest. locs
Option east (SMDP)
Taxi locs
0
5
10
15
20
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Combination of Pass. and Dest. locs
Q-value
0.0
−0.2
−0.4
−0.6
−0.8

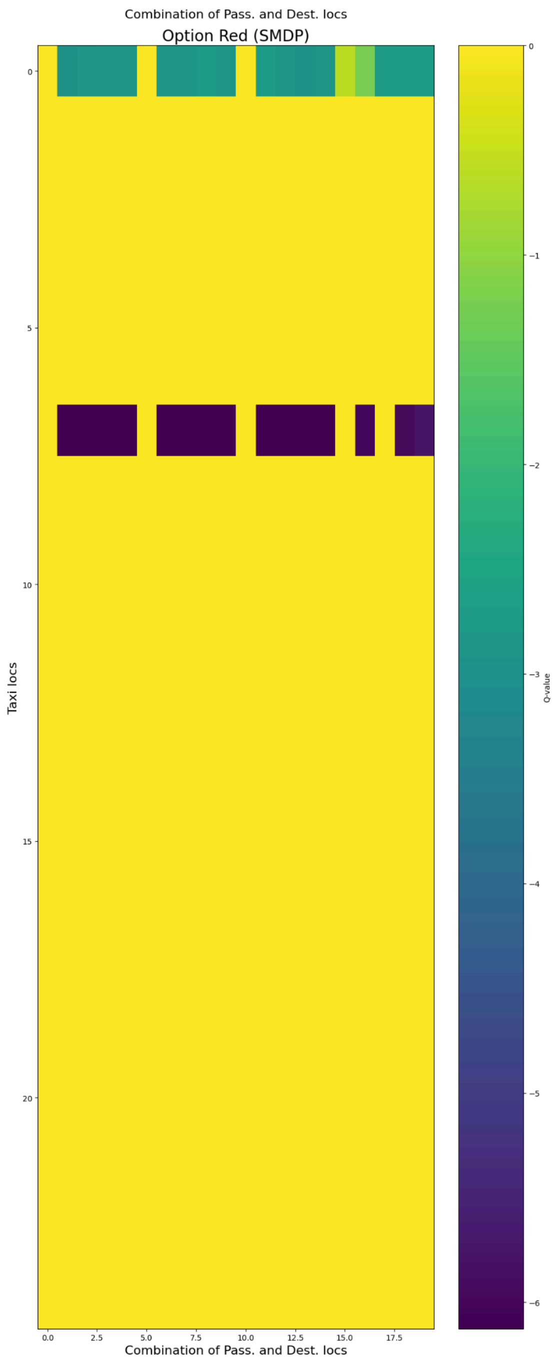

Combination of Pass. and Dest. locs
Option west (SMDP)
Taxi locs
0
5
10
15
20
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Combination of Pass. and Dest. locs
Q-value
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8

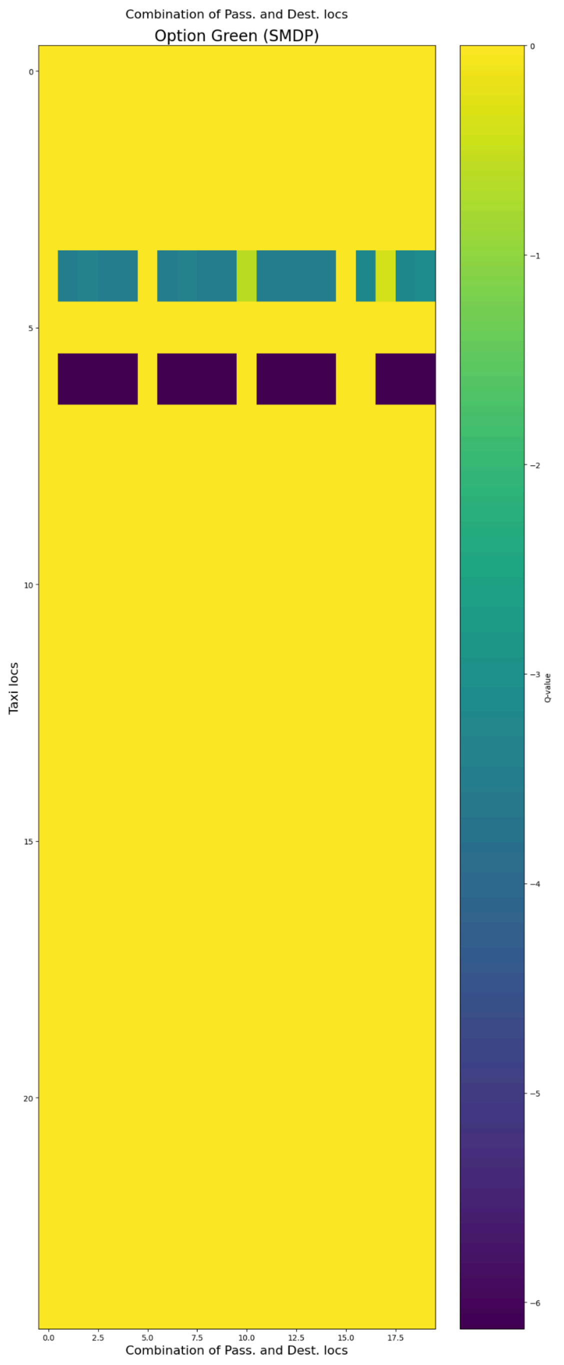


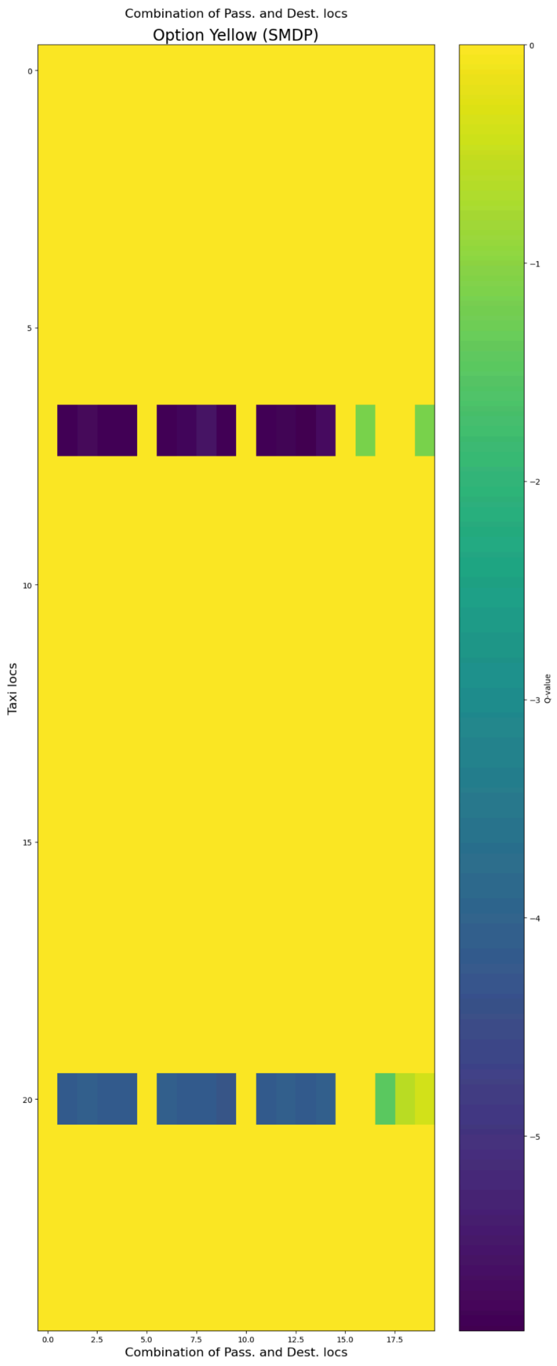
Combination of Pass. and Dest. locs
Option pick (SMDP)
Taxi locs
0
5
10
15
20
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Combination of Pass. and Dest. locs
Q-value
0
−2
−4
−6
−8

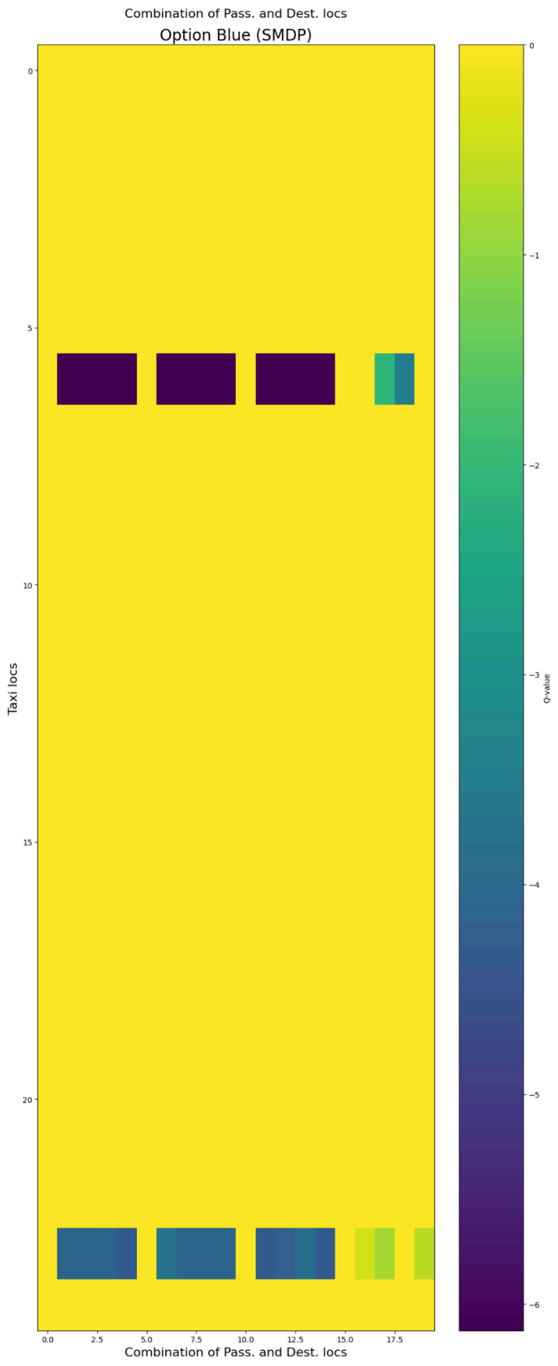


Combination of Pass. and Dest. locs
Option drop (SMDP)
Taxi locs
0
5
10
15
20
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Combination of Pass. and Dest. locs
Q-value
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5


Combination of Pass. and Dest. locs
Option Red (SMDP)
Taxi locs
Combination of Pass. and Dest. locs
Q-value
0
5
10
15
20
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0
−1
−2
−3
−4
−5
−6


Combination of Pass. and Dest. locs
Option Green (SMDP)
Taxi locs
Combination of Pass. and Dest. locs
Q-value
0
5
10
15
20
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0
−1
−2
−3
−4
−5
−6


Combination of Pass. and Dest. locs
Option Yellow (SMDP)
Taxi locs
Combination of Pass. and Dest. locs
Q-value

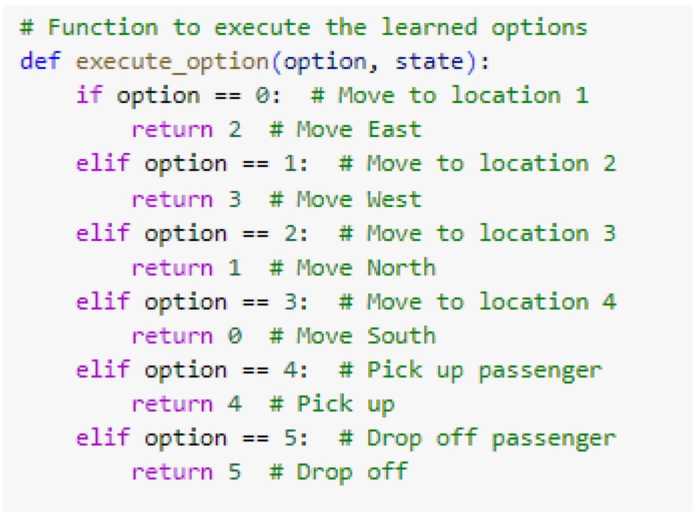

Combination of Pass. and Dest. locs
Option Blue (SMDP)
Taxi locs
Combination of Pass. and Dest. locs
Q-value

Intra Option Q Learning

Important snippets from the code

Deterministic Options - we tried 2 ways of defining the deterministic options.

In general, options could be defined for this problem as follows:



```python
# Function to execute the learned options
def execute_option(option, state):
    if option == 0:  # Move to location 1
        return 2  # Move East
    elif option == 1:  # Move to location 2
        return 3  # Move West
    elif option == 2:  # Move to location 3
        return 1  # Move North
    elif option == 3:  # Move to location 4
        return 0  # Move South
    elif option == 4:  # Pick up passenger
        return 4  # Pick up
    elif option == 5:  # Drop off passenger
        return 5  # Drop off
```

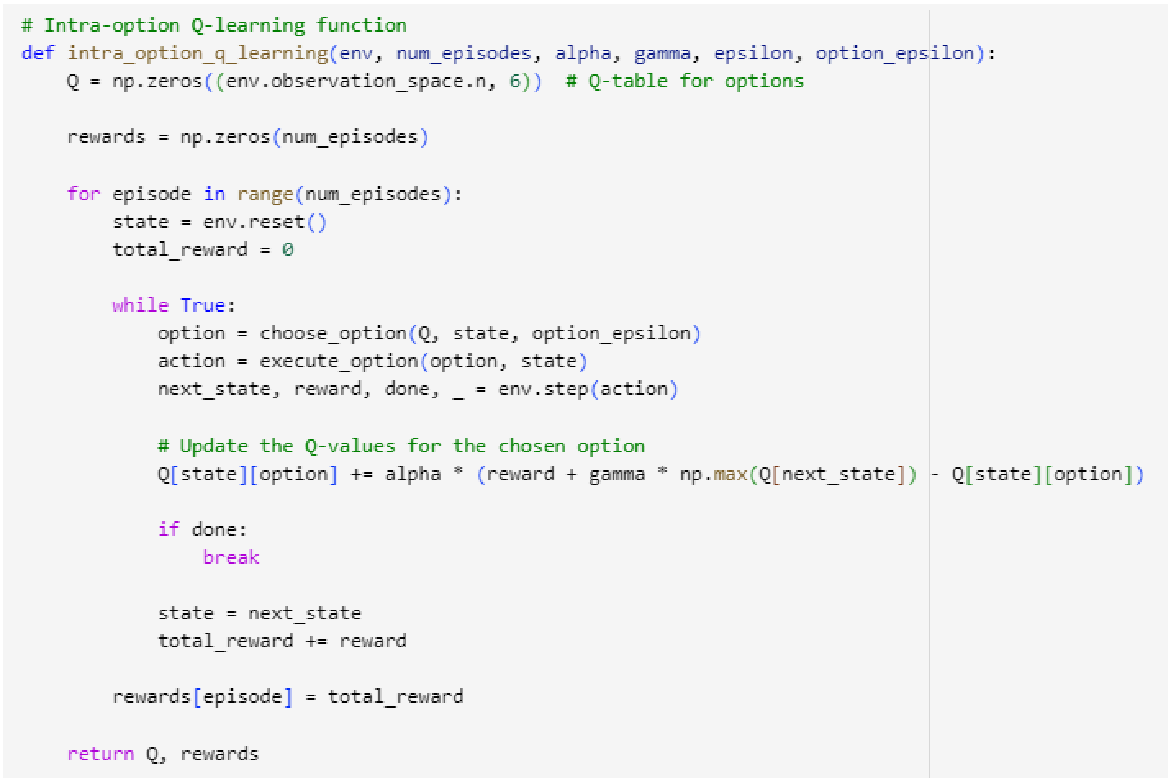
However, for this problem statement - we tried approach 2 of defining options as stated in the question:

```
policyOptionR = np.array([[1,3,0,0,0],
                 [1,3,0,0,0],
                 [1,3,3,3,3],
                 [1,1,1,1,1],
                 [1,1,1,1,1]
                 ])
policyOptionG = np.array([[0,0,2,2,1],
                 [0,0,2,2,1],
                 [2,2,2,1,1],
                 [1,2,1,1,1],
                 [1,2,1,1,1]
                 ])
policyOptionY = np.array([[0,3,0,0,0],
                          [0,3,0,0,0],
                          [0,3,3,3,3],
                          [0,1,1,1,3],
                          [0,1,3,1,3]
                          ])
policyOptionB = np.array([[0,0,0,0,3],
                          [0,0,0,0,3],
                          [2,2,2,0,3],
                          [1,1,1,0,3],
                          [1,1,1,0,3]
                          ])
policyOpt = [policyOptionR,policyOptionG,policyOptionY,policyOptionB]
```

Epsilon greedy policy for exploration of options -

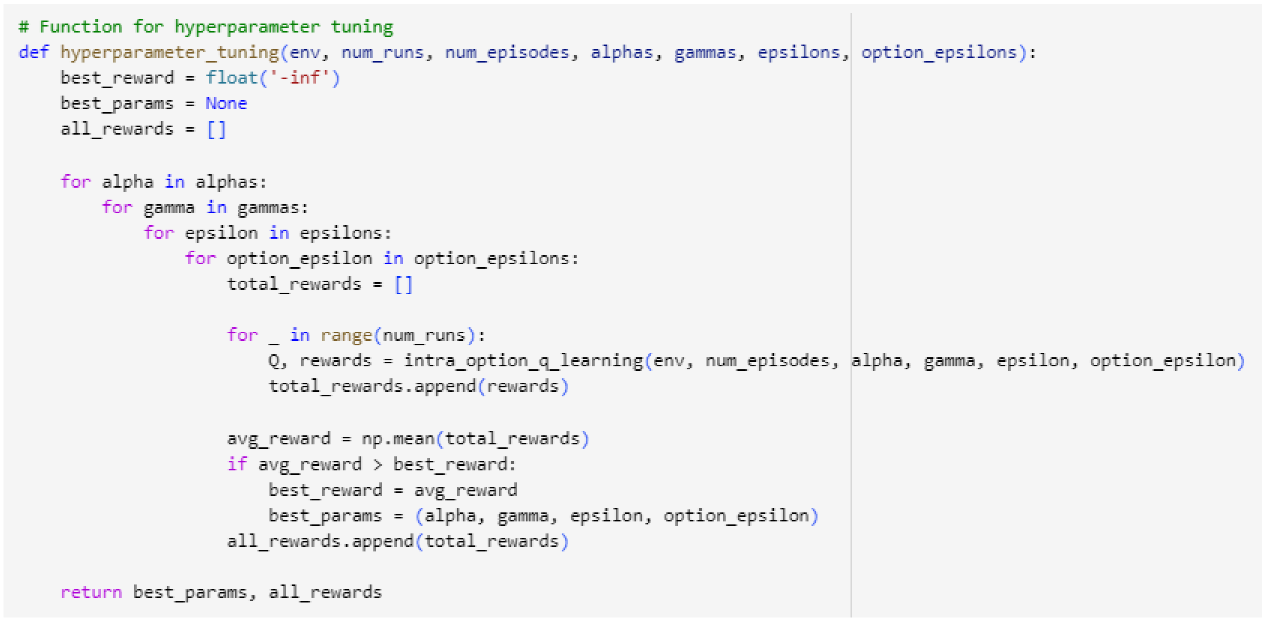
```
# Function to choose an option using epsilon-greedy policy
def choose_option(Q, state, epsilon):
    if np.random.random() < epsilon:
        return np.random.randint(6)  # Choose a random option
    else:
        return np.argmax(Q[state])  # Choose the option with the highest Q-value
```

Intra options q learning function definition -



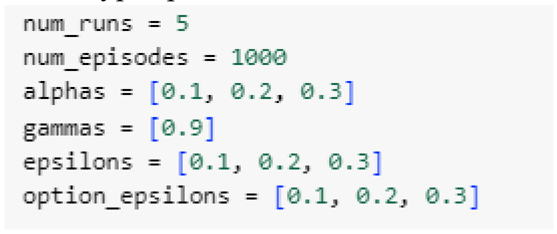
```python
# Intra-option Q-learning function
def intra_option_q_learning(env, num_episodes, alpha, gamma, epsilon, option_epsilon):
    Q = np.zeros((env.observation_space.n, 6))  # Q-table for options

    rewards = np.zeros(num_episodes)

    for episode in range(num_episodes):
        state = env.reset()
        total_reward = 0

        while True:
            option = choose_option(Q, state, option_epsilon)
            action = execute_option(option, state)
            next_state, reward, done, _ = env.step(action)

            # Update the Q-values for the chosen option
            Q[state][option] += alpha * (reward + gamma * np.max(Q[next_state]) - Q[state][option])

            if done:
                break

            state = next_state
            total_reward += reward

        rewards[episode] = total_reward

    return Q, rewards
```

Function for hyper-parameter tuning -



```python
# Function for hyperparameter tuning
def hyperparameter_tuning(env, num_runs, num_episodes, alphas, gammas, epsilons, option_epsilons):
    best_reward = float('-inf')
    best_params = None
    all_rewards = []

    for alpha in alphas:
        for gamma in gammas:
            for epsilon in epsilons:
                for option_epsilon in option_epsilons:
                    total_rewards = []

                    for _ in range(num_runs):
                        Q, rewards = intra_option_q_learning(env, num_episodes, alpha, gamma, epsilon, option_epsilon)
                        total_rewards.append(rewards)

                    avg_reward = np.mean(total_rewards)
                    if avg_reward > best_reward:
                        best_reward = avg_reward
                        best_params = (alpha, gamma, epsilon, option_epsilon)
                    all_rewards.append(total_rewards)

    return best_params, all_rewards
```

Set of hyper-parameters chosen -



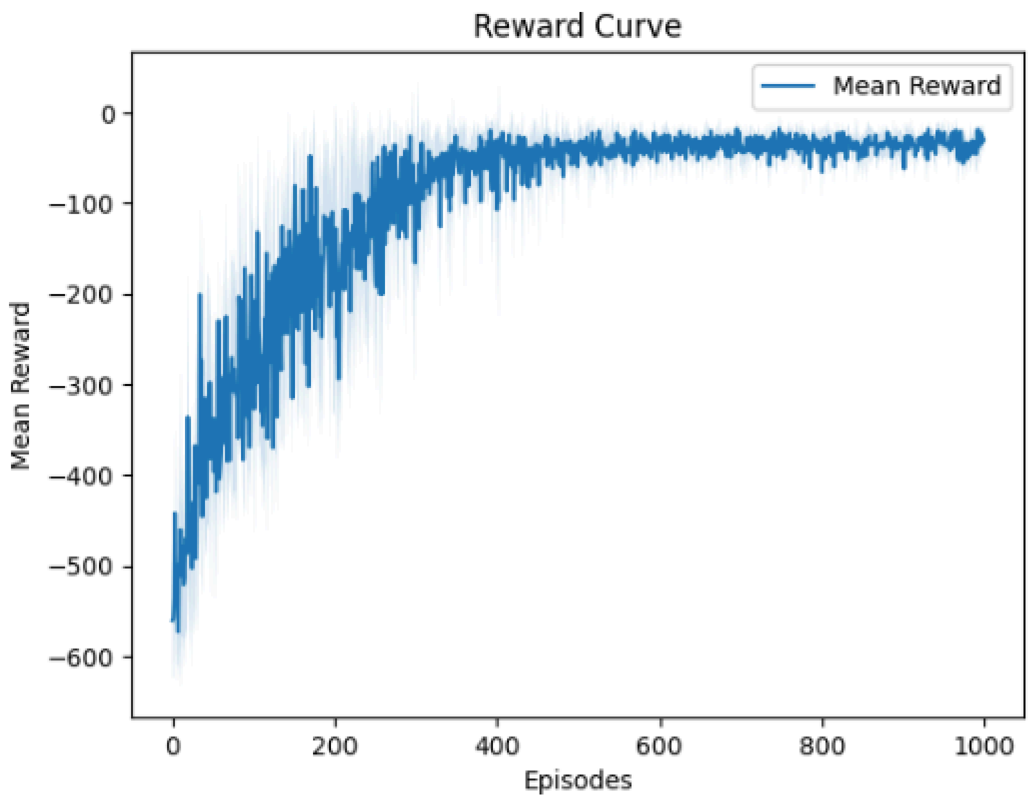
```
num_runs = 5
num_episodes = 1000
alphas = [0.1, 0.2, 0.3]
gammas = [0.9]
epsilons = [0.1, 0.2, 0.3]
option_epsilons = [0.1, 0.2, 0.3]
```

Results -

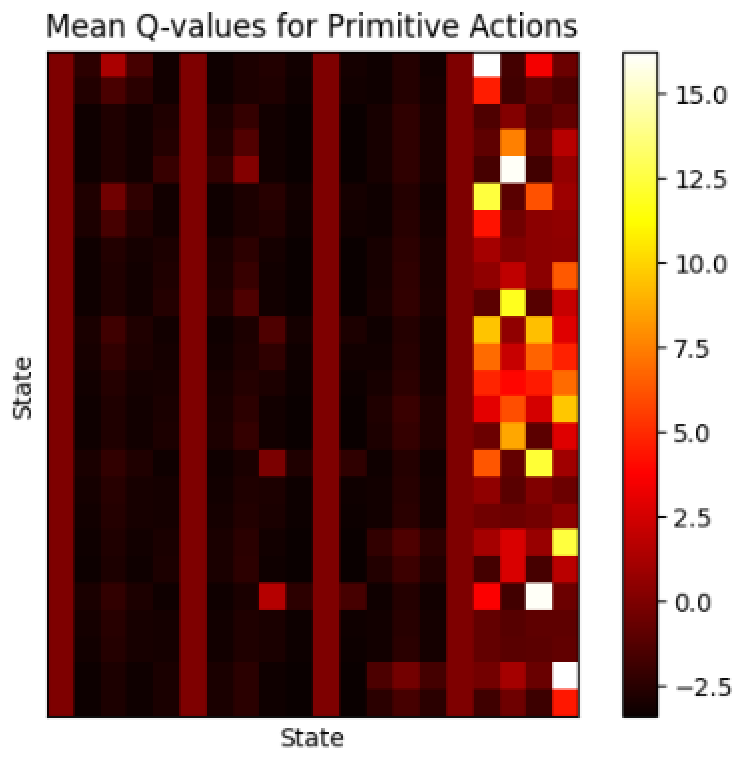
1. Best set of hyperparameters -
Alpha = 0.3
Gamma = 0.9
Epsilon = 0.2
Options_epsilons = 0.1

Q1: Plot reward curves and visualize the learned Q-values.

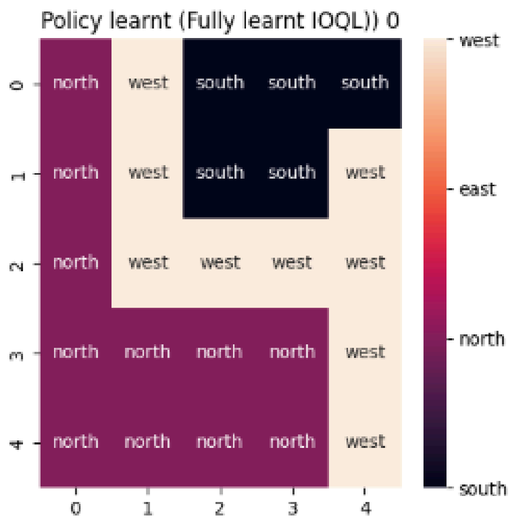
2. Reward curve for the best set of hyperparameters -

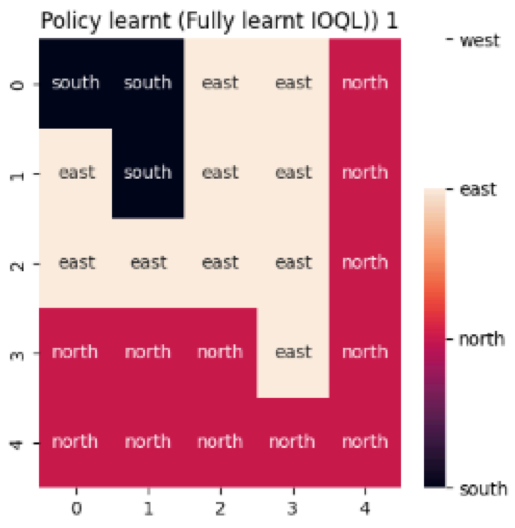


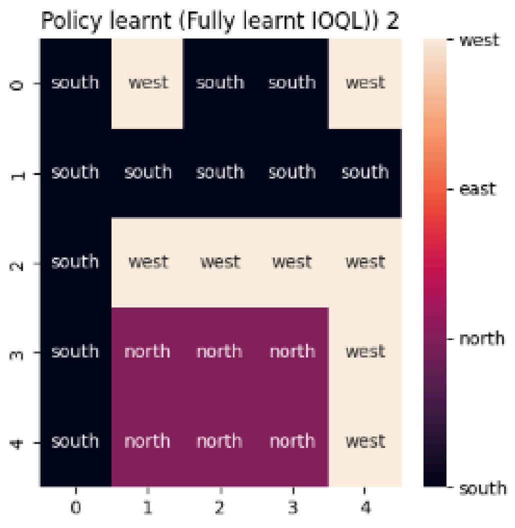
3. Learnt Q table for states and the primitive actions -

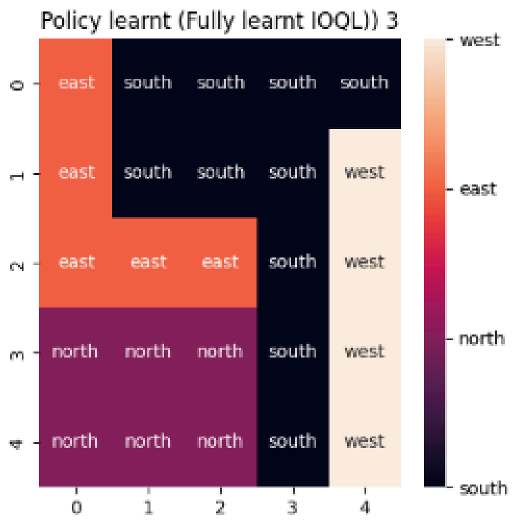


Q-table for each of the policy options

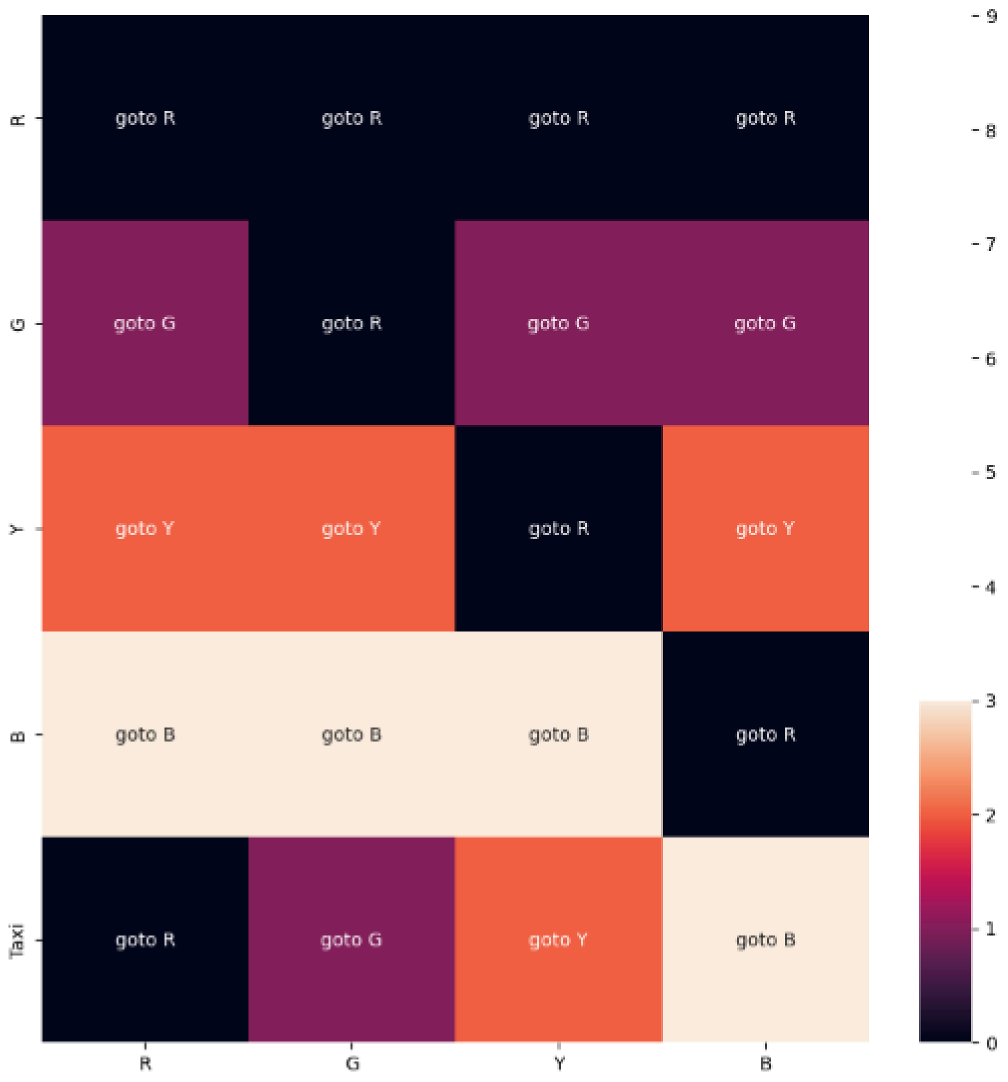








Final Q table - corresponding to the policy learnt



Q2: Provide a written description of the policies learnt and your reasoning behind why the respective algorithm learns the policy.

Answer :
The policies learned by the intra-option Q-learning algorithm in the Taxi-v3 environment can be understood by analyzing the Q-values obtained for each state-action pair. Here's a written description of the policies learned and the reasoning behind why the algorithm learns these policies:

Option Policies:
The algorithm learns option policies for moving the taxi to each of the four designated locations (locations 1, 2, 3, and 4) and for picking up and dropping off passengers.

For each option, the algorithm learns the best sequence of primitive actions (move south, move north, move east, move west, pick up passenger, drop off passenger) to reach the desired location or perform the required action.
The Q-values associated with each option and primitive action indicate the expected future rewards for taking that action in a particular state.
For example, for the option to move to location 1, the algorithm learns that moving east (action 2) is the most rewarding action in certain states because it leads the taxi closer to location 1. Similarly, picking up the passenger (action 4) becomes rewarding when the taxi is near the passenger's location.

Primitive Action Policies:
The algorithm also learns policies for primitive actions, i.e., individual movements and actions of the taxi within the environment.
The learned policies for primitive actions are influenced by the options' policies. Primitive actions are chosen within the context of executing an option or as part of a larger sequence of actions to achieve a particular goal.
For instance, the algorithm learns that moving in a direction towards the passenger's location or the drop-off location tends to yield higher rewards, as it progresses towards completing the task.
Similarly, the algorithm learns to avoid illegal moves or actions that do not contribute to reaching the goal, which may lead to negative rewards.
The intra-option Q-learning algorithm learns these policies through trial and error, iterative updating the Q-values based on the rewards received during interactions with the environment. By exploring different actions and options and observing their outcomes, the algorithm gradually learns the optimal policy for maximizing rewards in the Taxi-v3 environment. The algorithm's ability to learn these policies is driven by its reinforcement learning framework, where it seeks to maximize cumulative rewards over time.

Q3: Is there an alternate set of options that you can use to solve this problem, such that this set and the given options to move the taxi are mutually exclusive? If so, run both algorithms with this alternate set of options and compare performance with the algorithms run on the options to move the taxi.

Mutually exclusive options would mean that each option serves a distinct purpose or action that cannot be achieved by the other options.

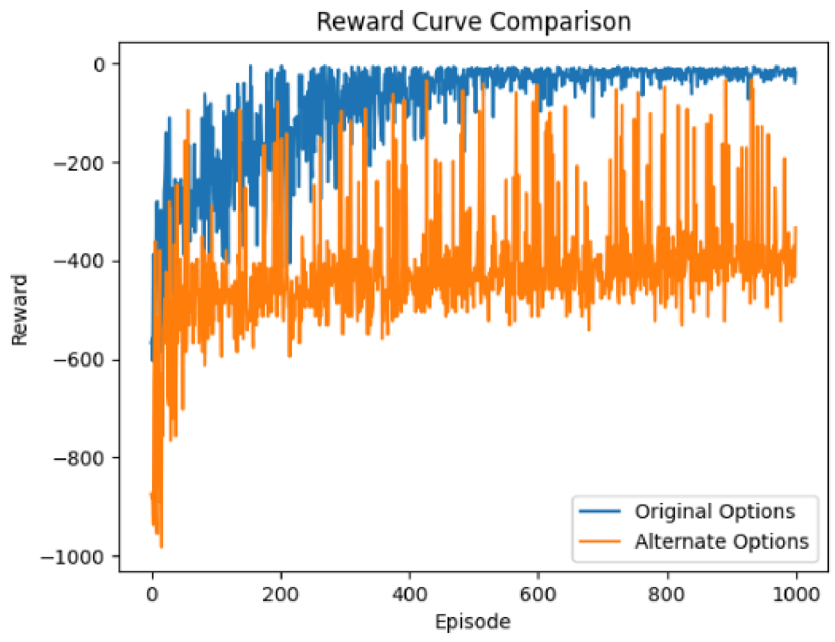
Original options were defined as follows:

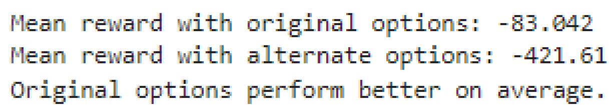
```python
# Function to execute the original options
def execute_option_original(option, state):
    if option == 0:  # Move south
        return 0  # Move taxi south
    elif option == 1:  # Move north
        return 1  # Move taxi north
    elif option == 2:  # Move east
        return 2  # Move taxi east
    elif option == 3:  # Move west
        return 3  # Move taxi west
    elif option == 4:  # Pick up passenger
        return 4  # Pick up passenger
    elif option == 5:  # Drop off passenger
        return 5  # Drop off passenger
```

One possible set of alternate options which are mutually exclusive to the original set of options are as follows:

```python
# Function to execute the alternate options
def execute_option_alternate(option, state):
    if option == 0:  # Go to random location
        return np.random.randint(4)  # Choose a random direction to move
    elif option == 1:  # Wait for a certain number of steps
        return 4  # Action to wait for a certain number of steps
    elif option == 2:  # Explore new area
        return 5  # Action to explore a new area
    elif option == 3:  # Navigate through obstacles
        return np.random.randint(4)  # Randomly choose an action to navigate through obstacles
    elif option == 4:  # Avoid obstacles
        return 4  # Action to avoid obstacles
    elif option == 5:  # Perform a specific task
        return 5  # Action to perform a specific task
```

The reward curve for both the options are as follows:





```
Mean reward with original options: -83.042
Mean reward with alternate options: -421.61
Original options perform better on average.
```

<u>Reasoning -</u>

The original options, which involve basic actions such as moving in different directions, picking up, and dropping off passengers, are more closely aligned with the task of navigating a taxi in a grid world environment to pick up and drop off passengers. These options directly address the primary objectives of the task and provide the taxi agent with clear and effective actions to achieve its goals.

On the other hand, the alternate options introduced actions such as waiting, exploring new areas, and navigating through obstacles. While these actions may seem useful in certain scenarios, they might not be as relevant or effective in the specific task of taxi navigation. For example, waiting for a certain number of steps or exploring new areas might not contribute significantly to completing the task of picking up and dropping off passengers. Similarly, navigating through obstacles or avoiding obstacles may not be directly applicable in the context of a grid world where obstacles are not explicitly defined.

As a result, the original options, which are more tailored to the task requirements, are likely to yield better performance in terms of achieving higher rewards over time compared to the alternate options. The original options provide the taxi agent with a more focused and efficient set of actions to navigate the environment and accomplish its objectives.

Difference between Intra-options Q-learning and SMDP Q-learning:

1. State Representation:
SMDP Q-learning operates directly on the state-action space, where each state-action pair has an associated Q-value.
Intra-option Q-learning operates on a higher-level space where options are considered. Each state has Q-values associated with different options.

2. Action Selection:
In SMDP Q-learning, actions are selected directly based on the Q-values of individual actions in the state-action space.
In intra-option Q-learning, options are first chosen based on their Q-values, and then actions are chosen within the options.

3. **Temporal Abstraction**:
SMDP Q-learning does not explicitly handle temporal abstraction. Each action is considered as a single step in the environment.
Intra-option Q-learning exploits temporal abstraction by learning options, which are sequences of actions executed until the termination condition is met.

4. Learning Structure:
SMDP Q-learning learns the value of each action in each state directly from experience.
Intra-option Q-learning learns the value of options, which are sequences of actions, and thus learns a hierarchical structure over actions.

5. Efficiency:
Intra-option Q-learning can potentially be more sample-efficient in environments with long-horizon tasks or where actions can be grouped into meaningful sequences.
SMDP Q-learning may require more samples to learn optimal policies in such environments due to the lack of explicit consideration of temporal abstraction.

6. Generalization:
Intra-option Q-learning allows for generalization across states within an option, which can be advantageous in environments with similar sub-tasks that occur in different states.
SMDP Q-learning does not inherently provide this level of generalization, as each state-action pair is considered independently.

7. Complexity:
Intra-option Q-learning introduces additional complexity in learning and implementation due to the need to learn options and their associated policies.
SMDP Q-learning is conceptually simpler and more straightforward to implement, as it operates directly on the state-action space without considering higher-level abstractions.

In summary, SMDP Q-learning and intra-option Q-learning differ in their approach to temporal abstraction, state representation, action selection, and learning structure. While SMDP Q-learning is more straightforward and suitable for simple environments, intra-option Q-learning is advantageous in more complex environments with long-horizon tasks or where actions can be grouped into meaningful sequences.