CS6700: Reinforcement Learning

Programming Assignment - II Report

HARSHAVARDHAN SRINIVAS PENTAKOTA [ME20B083]

PUTREVU SAI SATWIK [ME20B138]

GITHUB LINK.

April 22, 2024

Table of Contents

1 Settings of Environment

The environment for this task is the taxi domain.

Goal: To pick up the passenger and Drop him at the Destination.
Primitive Actions:

- 0: move south
- 1: move north
- 2: move east
- 3: move west
- 4: pickup passenger
- 5: drop off passenger

No of Passenger locations: 5
No of Destinations: 4
Map: 5 x 5 Grid

So for each (Passanger location, Destination) state we have another 25 possible states

Options: Options to move the taxi to each of the four designated locations, executable when the taxi is not already there.

So, No of options = 4;

Options: The taxi goes to a specific goal location (R, G, Y, B). After reaching there, it has the option of picking up or dropping the passenger. If it's not at the goal location, it uses epsilon-greedy on option qvalues to choose any primitive action (up,down,right,left).

Q values of SMDP or IOQL is 20 sub-states x N options

Code snippets for Option Definition:

```
'''
We have 4 goals and an option to go to each goal.
'''

num_options = 4

# Position of goals
goal_pos = env.unwrapped.locs


'''
Goal is R, G, Y, B
'''

def Deliver_policy(Q_option, goal_pos, goal, state, epsilon):
    # Decode the state into taxi position, passenger location, and drop location
    taxi_X, taxi_Y, Passenger, DropLoc = env.decode(state)

    # Initialize the option termination flag
    optdone = False

    # Check if the taxi is at the goal location
    if (taxi_X == goal_pos[goal][0] and taxi_Y == goal_pos[goal][1]):
        optdone =  True

        # If the passenger is at the goal location, pick them up
        if (Passenger == goal):
            optact = 4  # Pick up the passenger at the goal
        # If the drop location is at the goal location, drop the passenger
        elif (DropLoc == goal):
            optact = 5  # Drop the passenger at the goal
        else:  # If it's just a (R, G, Y, B) location but not a pickup or drop location
            # Choose the action based on epsilon-greedy policy
            optact = epsilon_policy(Q_option[goal], taxi_X * 5 + taxi_Y, epsilon)
    else:
        # If not at the goal location, choose the action based on epsilon-greedy policy
        optact = epsilon_policy(Q_option[goal], taxi_X * 5 + taxi_Y, epsilon)

    return [optact, optdone]

```

2 SMDP Q-Learning:

For SMDP Q learning, we have 4 options as we mentioned before.

Each Option consists of Q values to decide the primitive action needed to move towards the Goal for each given substate (Grid Position of Taxi[25 states])

Q values option = 4 options * (25 taxi loc * 4 Primitve actions)

```
num_of_options = 4
N_options  = 4
N_goals = 4
N_passenger_location  = 5
N_row = 5
N_col  =5
Q_values_SMDP = np.ones((N_passenger_location*N_goals,N_options))
Q_values_options = np.ones((N_goals,5*5,no_of_actions-2))
alpha = 0.4
gamma = 0.99
```

SMDP ALGO:

```
def SMDP(seed, Deliver_policy=Deliver_policy):
    # Load the environment and set the state and action spaces
    env, state_space, action_space = LoadingEnv(seed=seed)

    # Initialize rewards and other variables
    rewards = []
    episodes = 1000
    eps = 0.1
    eps_options = {i: 0.1 for i in range(N_options)}  # Epsilon for each option
    count_success = 0

    # Loop through episodes
    for i in tqdm(range(episodes)):
        state = env.reset()
        done = False
        total_reward = 0

        # Loop until the episode is done
        while not done:
            # Decode the current state to get taxi location, passenger location, and drop location
            taxi_X, taxi_Y, Passenger, DropLoc = env.decode(state)

            # Determine the sub-state for transforming 500 states to fit in N_passenger_location * N_goals
```

```
        sub_state = Passenger * N_goals + DropLoc

        # Choose an option using epsilon-greedy policy
        option = epsilon_policy(Q_values_SMDP, sub_state, epsilon=eps)

        # Initialize variables for option execution
        reward_bar = 0
        opt_done = False
        steps = 0
        prev_state = state

        # Execute the option until it is done or the episode ends
        while not opt_done and not done:
            # Choose an action for the option
            Taxi_curr_X, Taxi_curr_Y, _, _ = env.decode(state)
            opt_act, opt_done = Deliver_policy(Q_values_options, goal_pos, option, state, eps_options[option])

            # Execute the action and observe the next state and reward
            next_state, reward, done, _ = env.step(opt_act)
            Taxi_next_X, Taxi_next_Y, _, _ = env.decode(next_state)
            # Update the surrogate reward based on whether the option is completed
            if opt_done:
                reward_surr = 20  # Surrogate reward for completing the option
            else:
                reward_surr = reward

            # Update Q-values if the action is primitive
            if opt_act < 4:
                Q_values_options[option][5 * Taxi_curr_X + Taxi_curr_Y][opt_act] += alpha * (
                        reward_surr + gamma * np.max(Q_values_options[option][5 * Taxi_next_X + Taxi_next_Y, :]) -
                        Q_values_options[option][5 * Taxi_curr_X + Taxi_curr_Y, opt_act])

            # Update variables
            steps += 1
            reward_bar = gamma * reward_bar + reward
            total_reward += reward
            state = next_state

        # Update SMDP Q-value
        _, _, passenger, DropLoc = env.decode(state)
```

```
            sub_state = N_goals * passenger + DropLoc

            _, _, passenger, DropLoc = env.decode(prev_state)
            prev_sub_state = N_goals * passenger + DropLoc

            Q_values_SMDP[prev_sub_state, option] += alpha * (
                        reward_bar + (gamma ** steps) * np.max(Q_values_SMDP[sub_state, :]) -
                        Q_values_SMDP[prev_sub_state, option])

        # Append total reward for the episode
        rewards.append(total_reward)

    return rewards

```

SMDP Reward Plot:

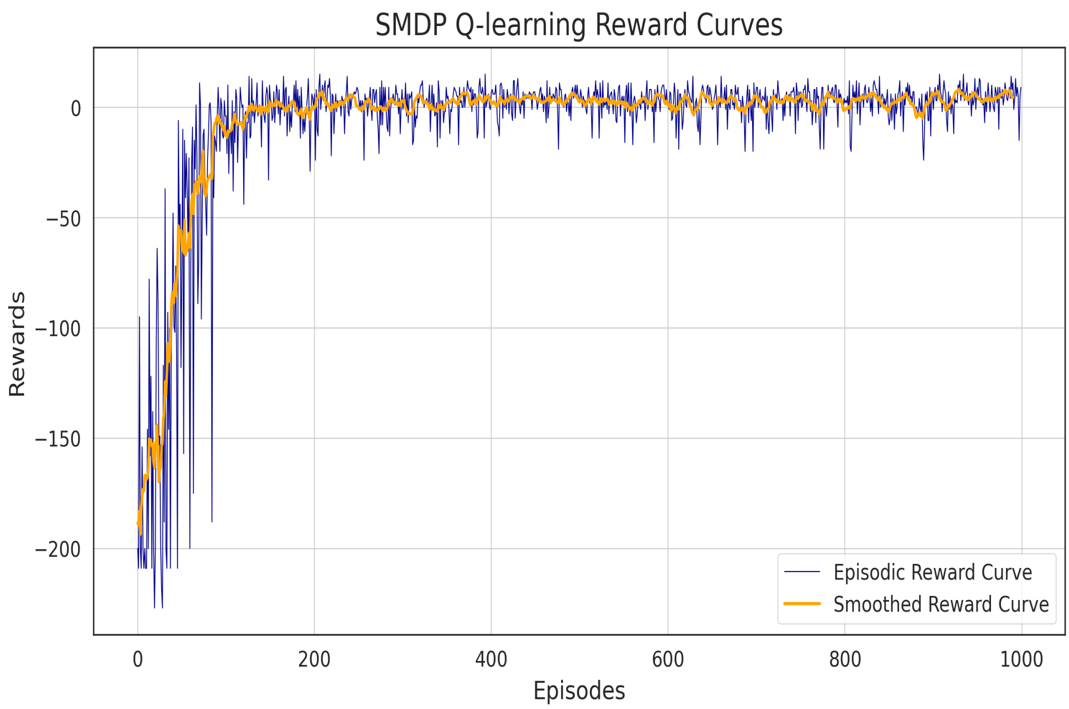


Figure 1: SMDP Q learning Reward Curve

Q values option[Move Red (option 0) is 25 x 4 Qmap denoting the Qmap to choose primitive action to move the taxi towards that option goal.

Visulaization of Polciy Qmap:

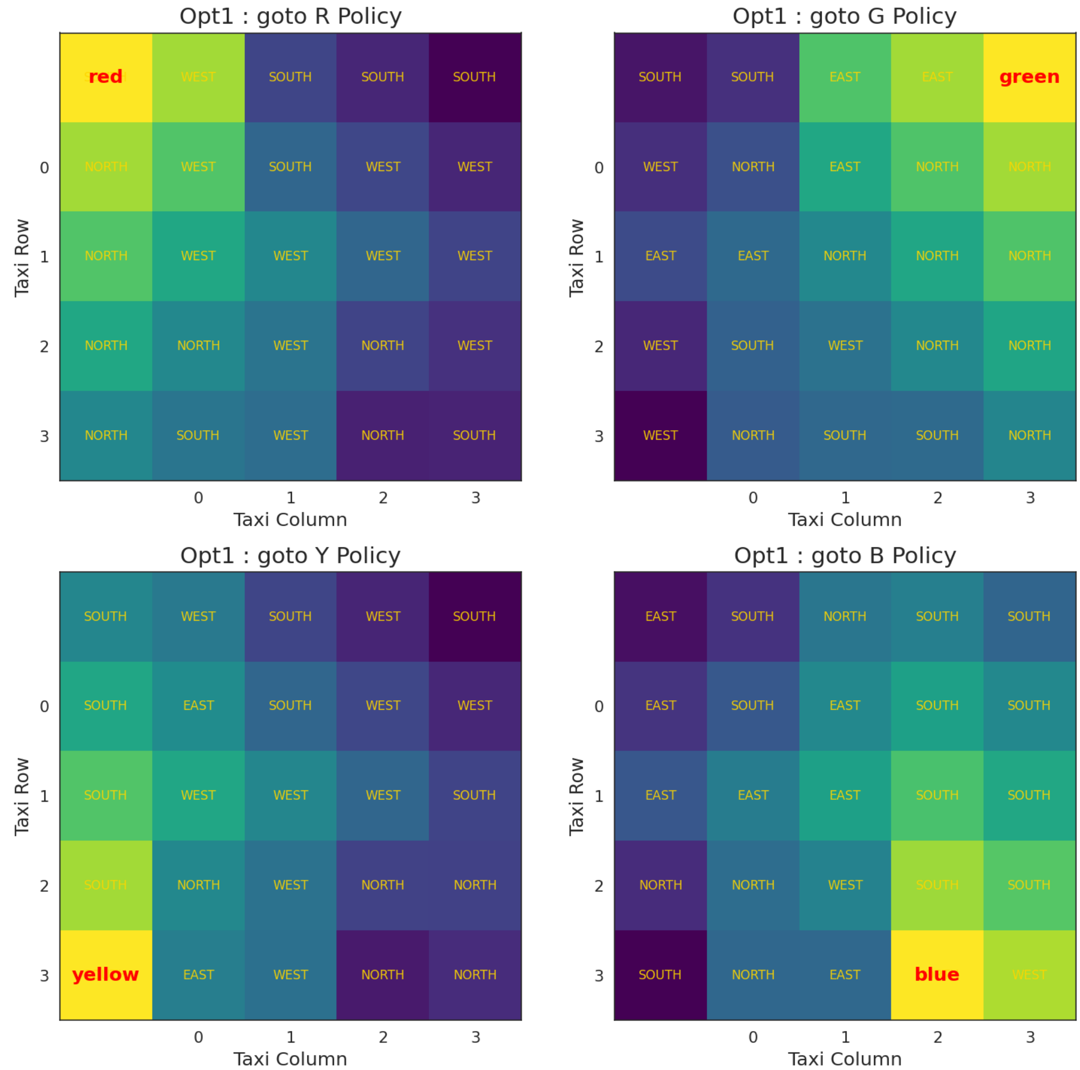


Figure 2: Qmap of each policy

Q values of SMDP or IOQL is 20 sub-states x N options. For each sub-state we choose which one of the 4 options is best.

Visulaization of SMDP Polciy Qmap:

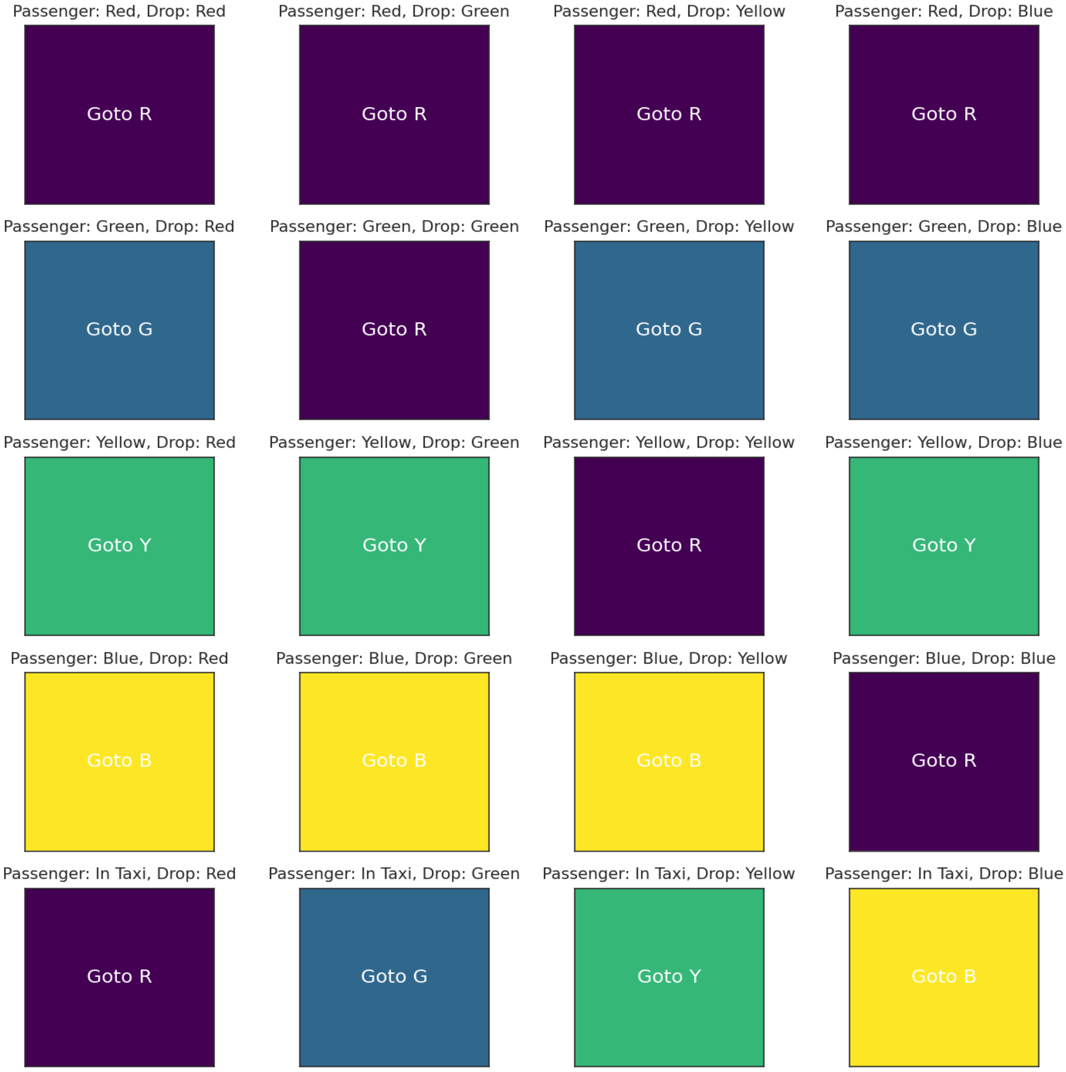


Figure 3: SMDP Qmap

3 IOQL Q-Learning:

For IOQL Q learning, we have 4 options as we mentioned before.

Each Option consists of Q values to decide the primitive action needed to move towards the Goal for each given substrate (Grid Position of Taxi[25 states])

Q values option = 4 options * (25 taxi loc * 4 Primitve actions)

```
num_of_options = 4
N_options  = 4
N_goals = 4
N_passenger_location  = 5
N_row = 5
N_col  =5
Q_values_IOQL = np.ones((N_passenger_location*N_goals,N_options))
Q_values_options = np.ones((N_goals,5*5,no_of_actions-2))
alpha = 0.1
gamma = 0.99
```

IOQL ALGO:

```
def IOQL(seed,Deliver_policy = Deliver_policy):
    # Load the environment and set the state and action spaces
    env, state_space, action_space = LoadingEnv(seed=seed)

    # Initialize rewards and other variables
    rewards = []
    episodes = 1000
    eps = 0.1
    eps_options = {i: 0.1 for i in range(N_options)}  # Epsilon for each option
    count_success = 0

    # Loop through episodes
    for i in tqdm(range(episodes)):
        state = env.reset()
        done = False
        total_reward = 0

        # Loop until the episode is done
        while not done:
            # Decode the current state to get taxi location, passenger location, and drop location
            taxi_X, taxi_Y, Passenger, DropLoc = env.decode(state)

            # Calculate the sub-state to fit in N_passenger_location * N_goals by considering only passenger_loc, DropLoc
```

```
        sub_state = Passenger * N_goals + DropLoc

        # Choose an option using epsilon-greedy policy
        option = epsilon_policy(Q_values_IOQL, sub_state, epsilon=eps)

        # Initialize variables for option execution
        opt_done = False
        steps = 0
        prev_state = state

        # Execute the option until it is done or the episode ends
        while not opt_done and not done:
            # Decode the current state to get the taxi location, passenger location, and drop location
            Taxi_curr_X, Taxi_curr_Y, passenger, DropLoc = env.decode(state)

            # Calculate the sub-state
            sub_state = N_goals * passenger + DropLoc

            # Choose an action for the option
            opt_act, opt_done = Deliver_policy(Q_values_options, goal_pos, option, state, eps_options[option])

            # Execute the action and observe the next state and reward
            next_state, reward, done, _ = env.step(opt_act)

            # Decode the next state to get the taxi location, passenger location, and drop location
            Taxi_next_X, Taxi_next_Y, passenger1, DropLoc1 = env.decode(next_state)

            # Calculate the next sub-state
            next_sub_state = N_goals * passenger1 + DropLoc1

            # Update the surrogate reward based on whether the option is completed
            if opt_done:
                reward_surr = 100  # Surrogate reward for completing the option
            else:
                reward_surr = reward

            # Update Q-values if the action is primitive
            if opt_act < 4:
                Q_values_options[option][5 * Taxi_curr_X + Taxi_curr_Y][opt_act] += alpha * (reward_surr + gamma * np.max(Q_values_options[option][5 * Taxi_next_X + Taxi_next_Y]) -Q_values_options[option][5 * Taxi_curr_X + Taxi_curr_Y, opt_act])
```

```python

                # Update total reward
                total_reward += reward

                # Update Q-values for each option
                for O in range(N_options):
                    opt_act_option, opt_done_option = Deliver_policy(Q_values_options, goal_pos, O, state,eps_options[O])
                    if opt_act_option == opt_act:
                        if opt_done_option:
                            Q_values_IOQL[sub_state, O] += alpha * (reward + gamma * np.max(Q_values_IOQL[next_sub_state]) - Q_values_IOQL[sub_state, O])
                        else:
                            Q_values_IOQL[sub_state, O] += alpha * ( reward + gamma * (Q_values_IOQL[next_sub_state, O]) -Q_values_IOQL[sub_state, O])

                # Update the current state
                state = next_state

        # Append total reward for the episode
        rewards.append(total_reward)

    return rewards

```

IOQL Reward Plot:

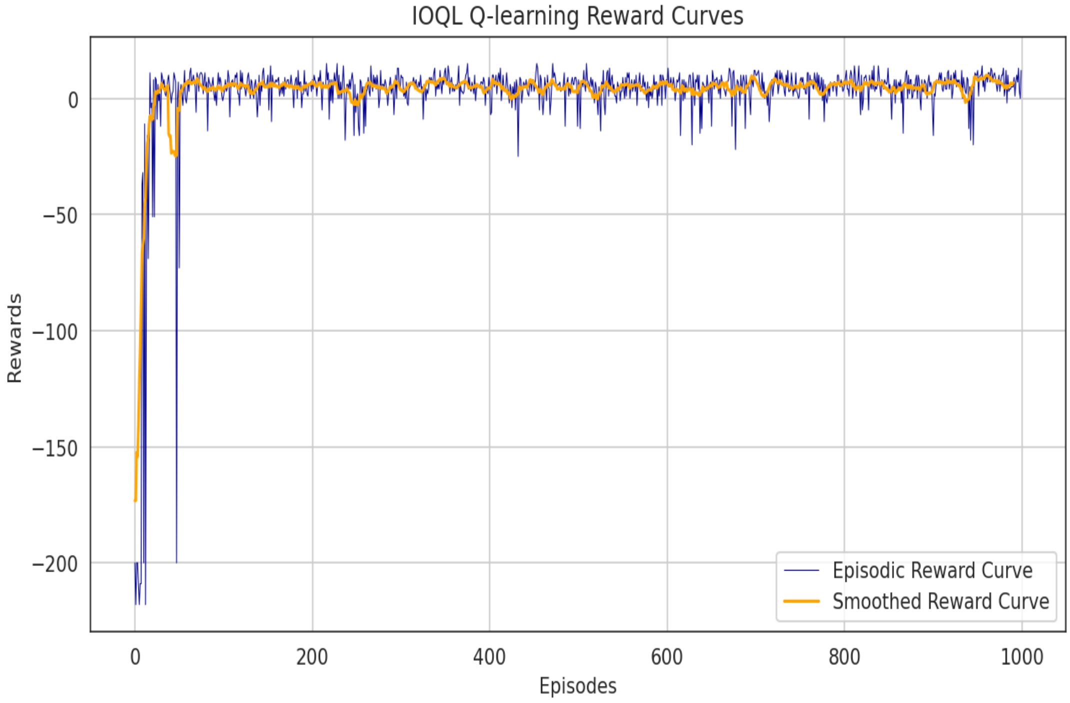


Figure 4: IOQL Q learning Reward Curve

Q values option[Move Red (option 0) is 25 x 4 Qmap denoting the Qmap to choose primitive action to move the taxi towards that option goal.

Visulaization of Polciy Qmap -IOQL:

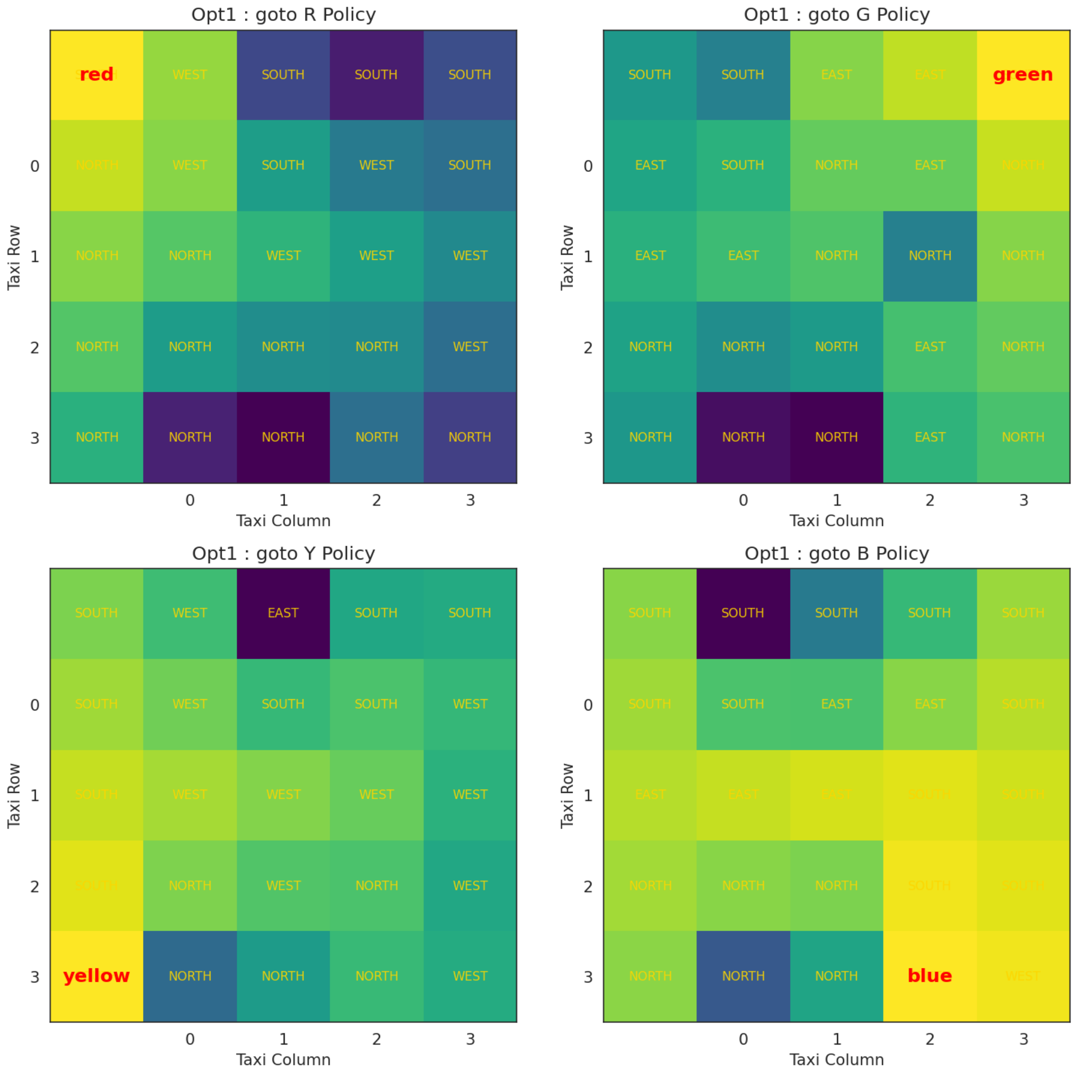


Figure 5: Qmap of each policy

For each sub-state we choose which one of the 4 options is best.

Visulaization of IOQL Polciy Qmap:

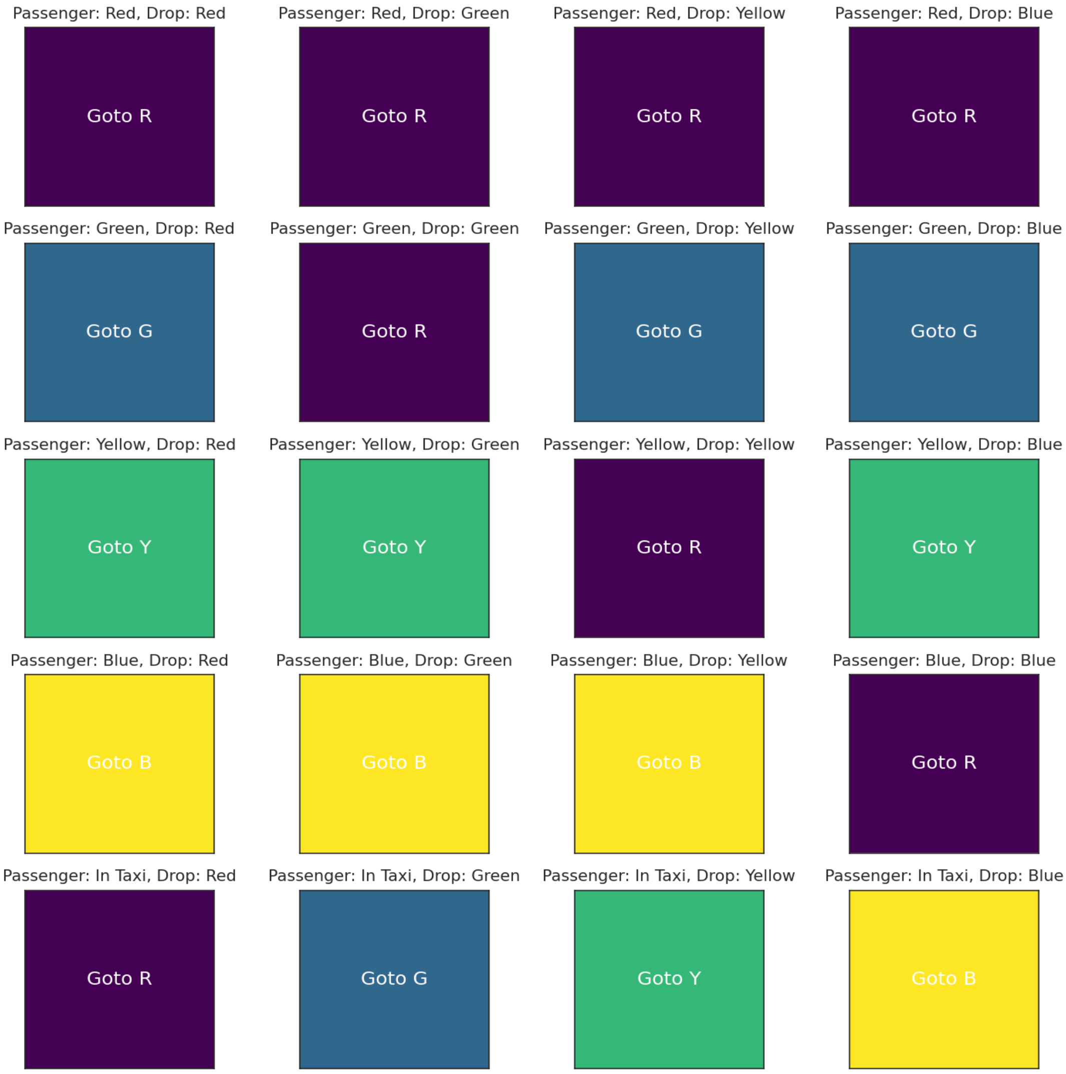


Figure 6: IOQL Qmap

4 Inferences:

4.1 Reasons for the Working of Hierarchical Learning Algorithms:

Hierarchical learning enables agents to think in terms of larger actions, such as "go to the pickup location in Red" or "drop off the passenger in Blue," rather than focusing on individual steps. By considering these higher-level actions, agents can learn and plan more efficiently, ultimately leading to better performance in complex tasks like taxi navigation.

SMDP Q-learning Policy Reasoning:

SMDP Q-learning involves two levels of policies: the main policy for choosing options and the option policies for completing the option goal.

Main Policy to Choose Option:

1. Figure 3 illustrates the decoding of 500 states into 20 sub-states, representing pickup and drop locations.
2. Each sub-state corresponds to selecting an option to reach the goal, simplifying the learning process by focusing on a single action rather than multiple movements.
3. Figure 3 depicts the optimal option for each sub-state.

Option Policy:

1. Within the main policy, option policies dictate primitive actions to complete a taxi maneuver.
2. Figure 2 displays the Q-map indicating the shortest path for each option goal.

SMDP Q-learning learns this policy by allowing the agent to operate at a higher level, focusing on actions like "go to location Y" or "go to location Blue." By simplifying the decision-making process to selecting one option for each sub-state, the agent can efficiently achieve its objectives.

IOQL (Intra-Option Q-learning) Policy:

The main and option policies used in IOQL are analogous to those in SMDP (see Figure 6 and Figure 5).

IOQL optimizes Q-values within each option using an intra-option learning approach. By emphasizing the value of staying within an option until completion, IOQL enhances decision-making efficiency. Incorporating 20 sub-states enables IOQL to effectively explore and exploit the environment, resulting in improved performance compared to traditional Q-learning methods.

5 Comparsion of IOQL vs SMDP:

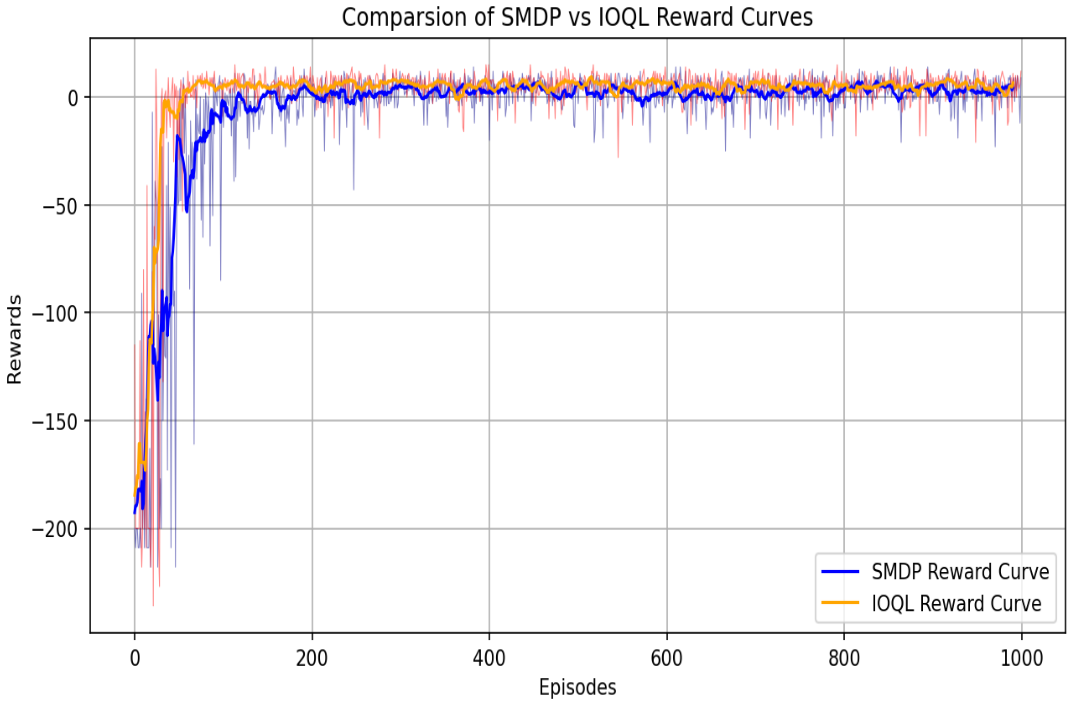


Figure 7: IOQL vs SMDP Reward Curve

IOQL performs better than SMDP Q learning (see Fig 7)

IOQL performs better because it allows for simultaneous updates across multiple options for a single action. This means that when taking a certain action, IOQL can update the Q-values for multiple options if those options were followed. This simultaneous updating enables IOQL to make more efficient use of experience, leading to faster learning and improved performance compared to SMDP Q-learning.

6 New Set of Options:

The set of options are taken such that the taxi tries to move towards the corners of the grid provided the taxi is already not present. So the **total number of actions taken are 8 including the 4 primitive actions (UP, DOWN, RIGHT, LEFT).**

No of Options : 4

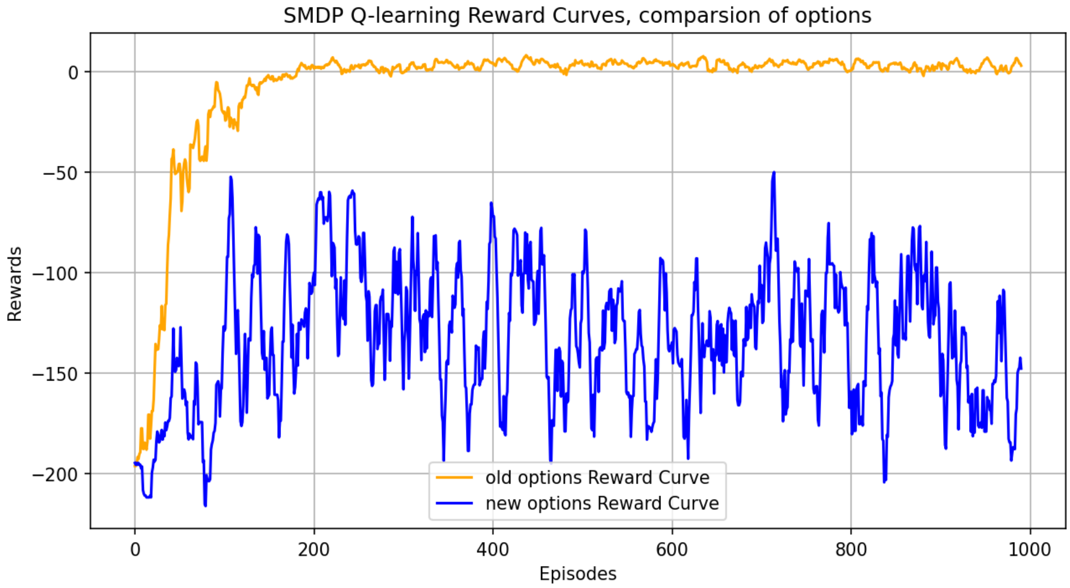
Total No of options : 8 (4 Primitive actions + 4 options

Corners : (0, 0), (0, 4), (4, 0), (4, 4)

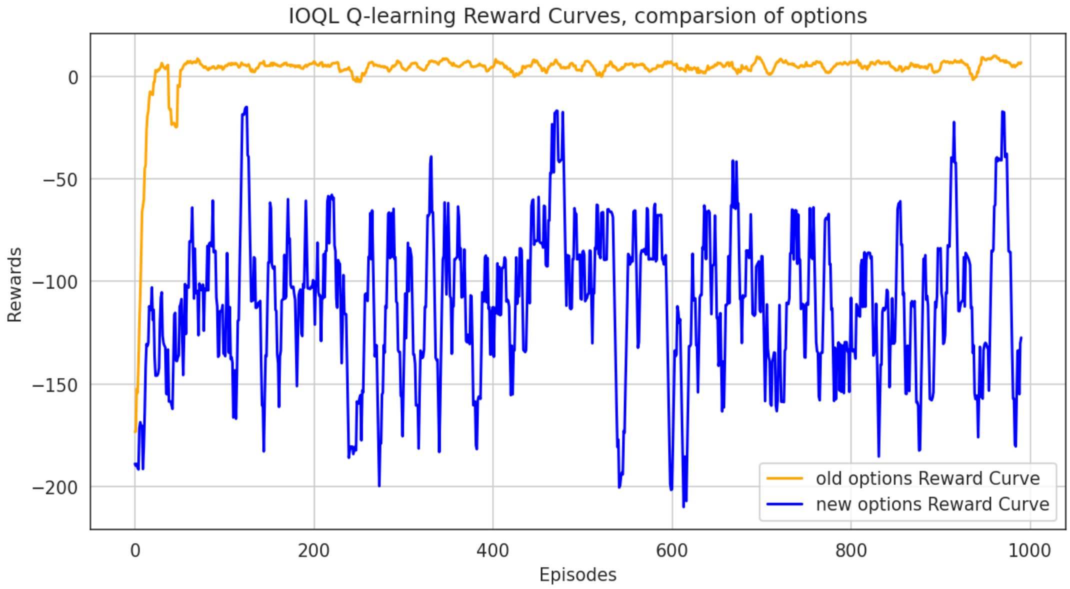
New Option Policy Code:

```
num_options = 4

goal_pos = [(0,0), (0,1),(2,0), (4,4)]

def new_policy(Q_option, goal_pos, goal, state,epsilon):

  taxi_X, taxi_Y, Passenger, DropLoc  =  env.decode(state)
  optdone = False
  if(goal >3): # 4, 5, 6, 7 are chosen for UP, DOWN, RIGHT, and LEFT respectively as the input
    optact = goal - 4
    optdone  = True
    return [optact,optdone]
  if (taxi_X == goal_pos[goal][0] and taxi_Y == goal_pos[goal][1]):
    optdone =  True
    if (Passenger == goal):
      optact = 4 #pick the passenger at goal
    elif (DropLoc == goal):
      optact = 5 # Drop the passenger at goal
    else:
      optact = epsilon_policy(Q_option[goal],taxi_X*5+taxi_Y,epsilon)
  else:
    optact = epsilon_policy(Q_option[goal],taxi_X*5+taxi_Y,epsilon) ## epsilon_greedy(Q_option[goal][state]) will choose one among UP, DOWN, RIGHT, LEFT


  return [optact,optdone]
```

Comparision of Old options vs New options for SMDP-QLearning



Comparision of Old options vs New options for IOQL-QLearning



It is evident from both plots that old options are better

GITHUB REPO LINK:
GITHUB USERNAMES:
ME20B083 : harshavardhan379
ME20B138 : SATWIK-ME20B138

GITHUB REPO LINK : https://github.com/harshavardhan379/CS6700$_PA3$.