CS6700: Reinforcement Learning

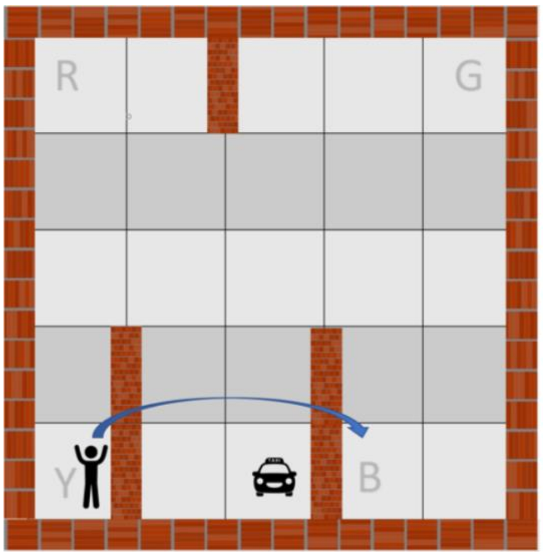
Assignment 3

Dhruv Bawa ME20B061
Pranav G S ME21B061

Problem statement:

Taxi-V3 is one of many environments available at OpenAI Gym. 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.

Solving : OpenAI Gym defines "solving" this task as getting average return of 9.7 over 100 consecutive trials.



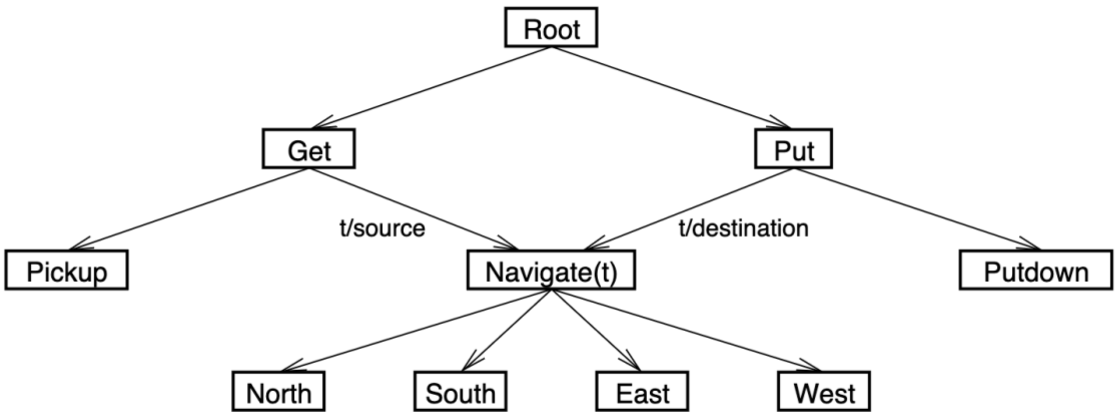


Figure 2: A task graph for the Taxi problem.

Algorithm 1 - SMDP Q-learning

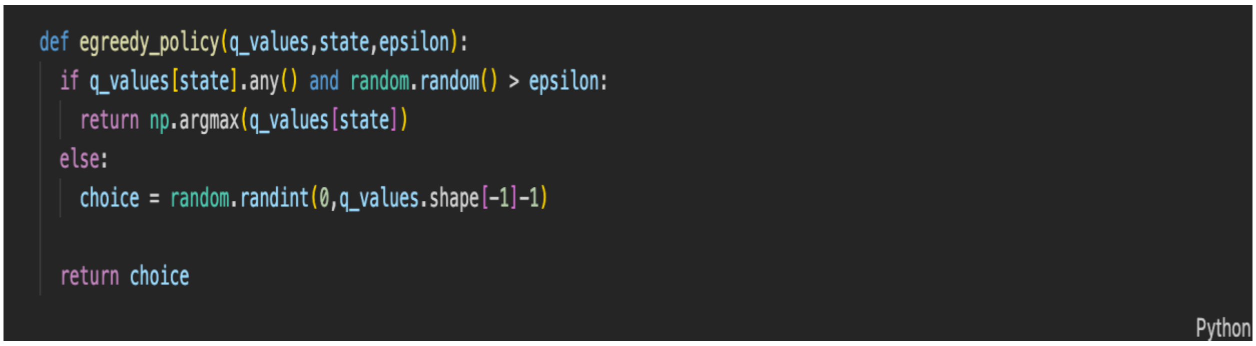
SMDP Q-learning extends the principles of traditional Q-learning to accommodate scenarios where actions do not result in immediate state changes. In these settings, actions may have variable durations, leading to non-fixed transitions between states. The adaptation lies in the Q-value function, which now incorporates the cumulative rewards over the course of an action and its subsequent transitions until the next decision point. By incorporating temporal abstraction and learning over multiple time scales, SMDP Q-learning can effectively navigate complex state spaces and optimize long-term rewards.

$$Q(s,o) \leftarrow Q(s,o) + \alpha \left(r + \gamma^k \max_{o' \in O_{s'}} Q(s',o') - Q(s,o) \right),$$

where k denotes the number of time steps elapsing between s and s',
r denotes the cumulative discounted reward over this time,
Learning parameters α

Epsilon greedy(Q- learning) Option Policy:

We define epsilon greedy policy to choose the action by selecting the action with highest Q- value or instead a random action is chosen.

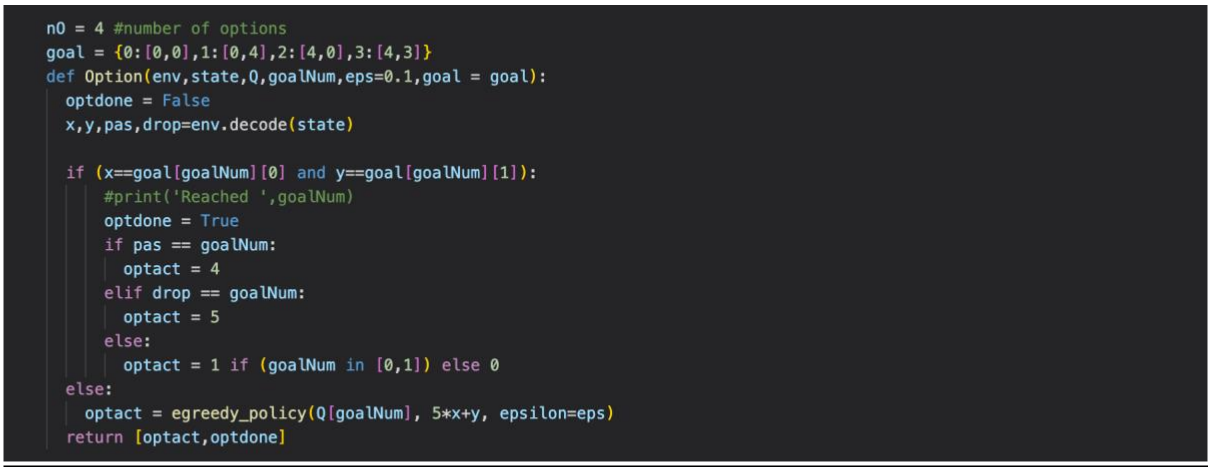


```python
def egreedy_policy(q_values,state,epsilon):
  if q_values[state].any() and random.random() > epsilon:
    return np.argmax(q_values[state])
  else:
    choice = random.randint(0,q_values.shape[-1]-1)

  return choice
```

Python

Original Option Policy:

- The “option policy” function optimizes the taxi’s behavior by strategically choosing actions based on its current state and the potential rewards associated with specific locations. The policy makes decision based on the current state and its proximity to target location: if it’s at its goal state then it initiates a “pickup” or a “drop”.
- When the taxi is not at a target location, the policy employs an epsilon-greedy strategy.Epsilon-greedy policies balance exploration (trying new actions) and exploitation (choosing known good actions).The taxi continues toward its destination using this exploration-exploitation trade-off.



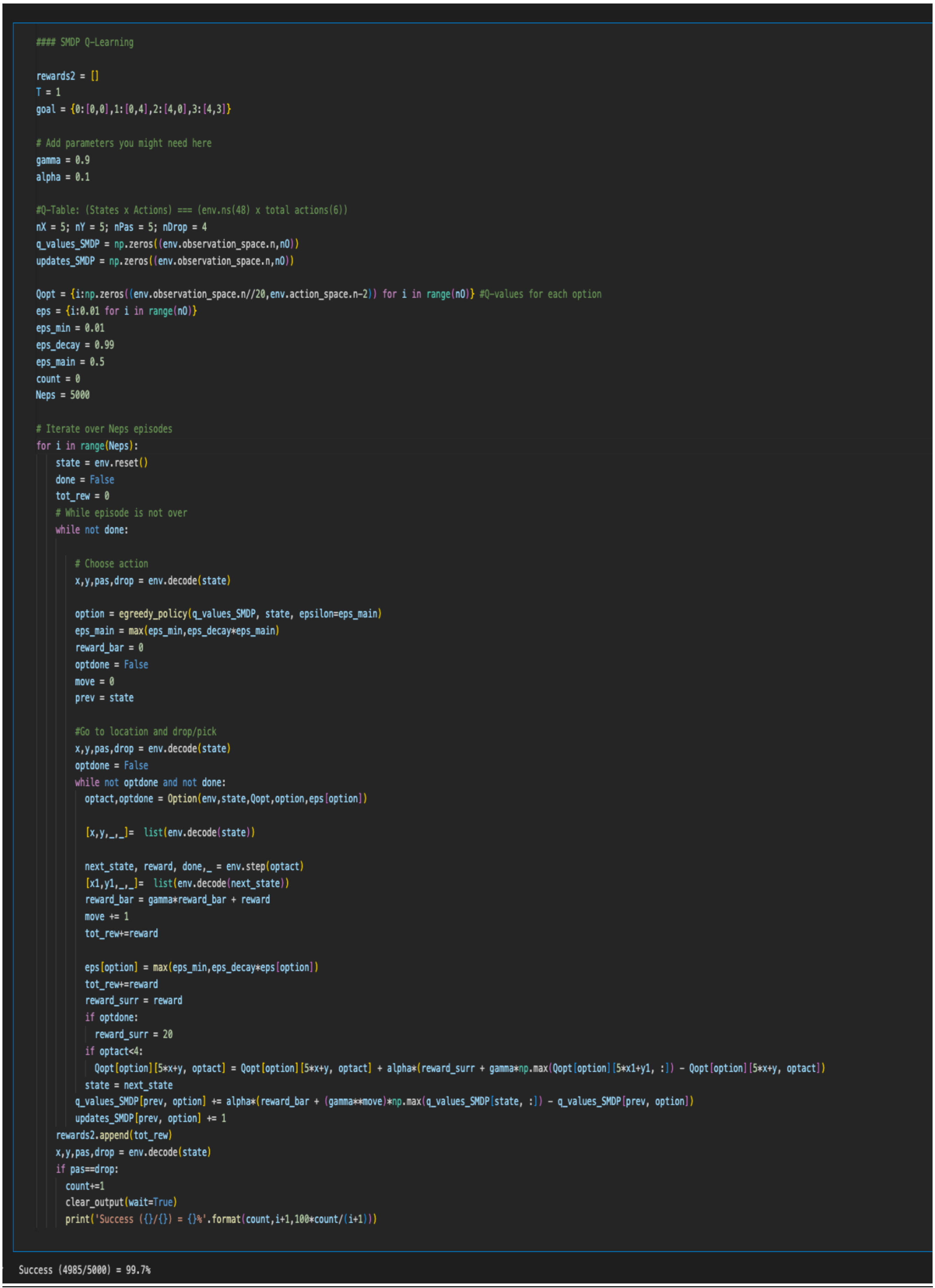
```
nO = 4 #number of options
goal = {0:[0,0],1:[0,4],2:[4,0],3:[4,3]}
def Option(env,state,Q,goalNum,eps=0.1,goal = goal):
  optdone = False
  x,y,pas,drop=env.decode(state)

  if (x==goal[goalNum][0] and y==goal[goalNum][1]):
      #print('Reached ',goalNum)
      optdone = True
      if pas == goalNum:
        optact = 4
      elif drop == goalNum:
        optact = 5
      else:
        optact = 1 if (goalNum in [0,1]) else 0
  else:
    optact = egreedy_policy(Q[goalNum], 5*x+y, epsilon=eps)
  return [optact,optdone]
```

SMDP Q – Learning:

Here, we only have 4 options that are :

1. Go to R, pick/drop according to the current state
2. Go to G, pick/drop according to the current state
3. Go to Y, pick/drop according to the current state
4. Go to B, pick/drop according to the current state

Each of the option policies was learnt through Q -Learning and is updated by the algorithm based on the rewards received. This process continues until the completion of each episode, with the total reward calculated at the end. (it is averaged over ever 100 runs- moving average) . We run this on the default 500 size state space.

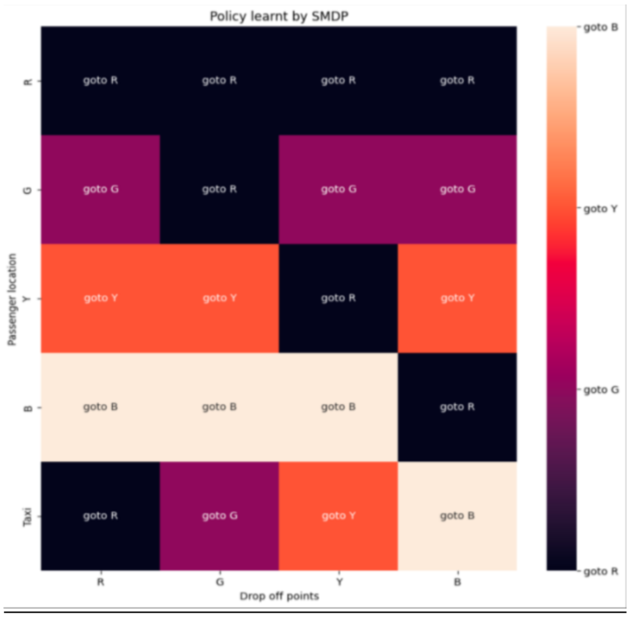


```python
#### SMDP Q-Learning

rewards2 = []
T = 1
goal = {0:[0,0],1:[0,4],2:[4,0],3:[4,3]}

# Add parameters you might need here
gamma = 0.9
alpha = 0.1

#Q-Table: (States x Actions) === (env.ns(48) x total actions(6))
nX = 5; nY = 5; nPas = 5; nDrop = 4
q_values_SMDP = np.zeros((env.observation_space.n,nO))
updates_SMDP = np.zeros((env.observation_space.n,nO))

Qopt = {i:np.zeros((env.observation_space.n//20,env.action_space.n-2)) for i in range(nO)} #Q-values for each option
eps = {i:0.01 for i in range(nO)}
eps_min = 0.01
eps_decay = 0.99
eps_main = 0.5
count = 0
Neps = 5000

# Iterate over Neps episodes
for i in range(Neps):
    state = env.reset()
    done = False
    tot_rew = 0
    # While episode is not over
    while not done:

        # Choose action
        x,y,pas,drop = env.decode(state)

        option = egreedy_policy(q_values_SMDP, state, epsilon=eps_main)
        eps_main = max(eps_min,eps_decay*eps_main)
        reward_bar = 0
        optdone = False
        move = 0
        prev = state

        #Go to location and drop/pick
        x,y,pas,drop = env.decode(state)
        optdone = False
        while not optdone and not done:
          optact,optdone = Option(env,state,Qopt,option,eps[option])

          [x,y,_,_]=  list(env.decode(state))

          next_state, reward, done,_ = env.step(optact)
          [x1,y1,_,_]=  list(env.decode(next_state))
          reward_bar = gamma*reward_bar + reward
          move += 1
          tot_rew+=reward

          eps[option] = max(eps_min,eps_decay*eps[option])
          tot_rew+=reward
          reward_surr = reward
          if optdone:
            reward_surr = 20
          if optact<4:
            Qopt[option][5*x+y, optact] = Qopt[option][5*x+y, optact] + alpha*(reward_surr + gamma*np.max(Qopt[option][5*x1+y1, :]) - Qopt[option][5*x+y, optact])
          state = next_state
        q_values_SMDP[prev, option] += alpha*(reward_bar + (gamma**move)*np.max(q_values_SMDP[state, :]) - q_values_SMDP[prev, option])
        updates_SMDP[prev, option] += 1
    rewards2.append(tot_rew)
    x,y,pas,drop = env.decode(state)
    if pas==drop:
      count+=1
      clear_output(wait=True)
      print('Success ({}/{}) = {}%'.format(count,i+1,100*count/(i+1)))
```

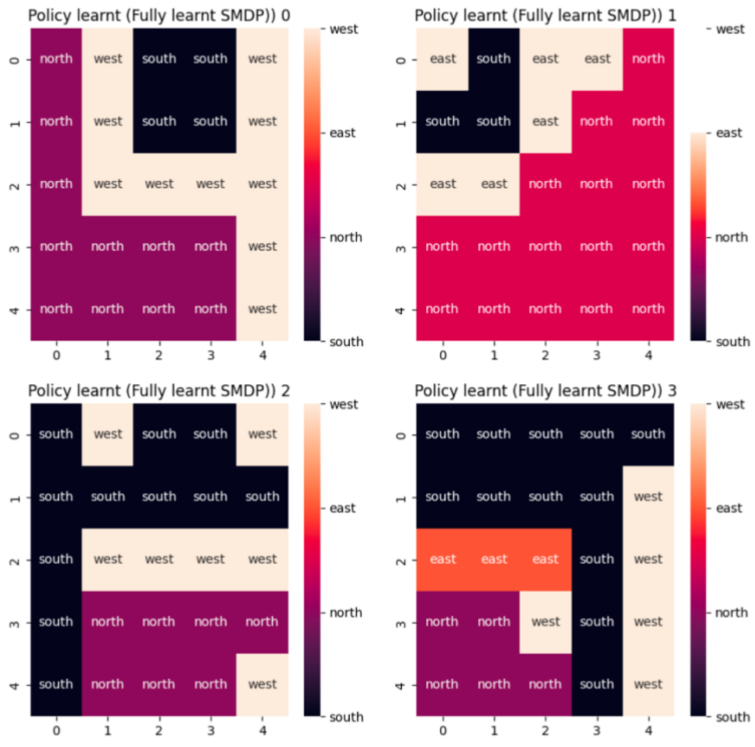
```
Success (4985/5000) = 99.7%
```

Visualization and results:

The generated heatplots, show the policy learned by SMDP in 2x2 grid fashion, and the below plot shows the directional policy actions taxi will take to move around in the environment.

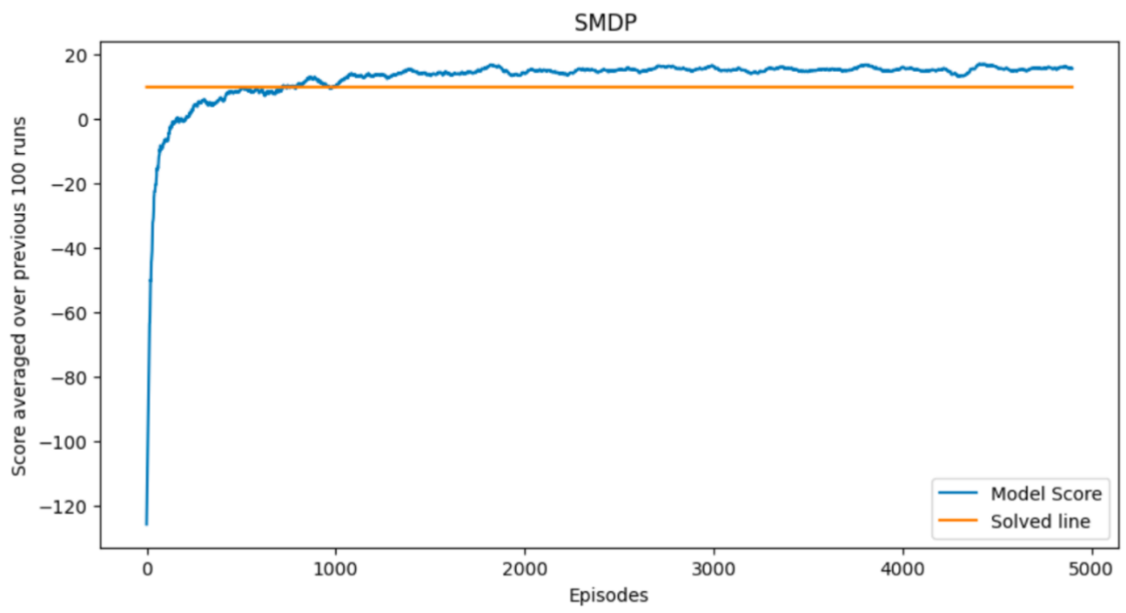


(option selection policy learnt via SMDP)



(SMDP Learnt policies for all 4 options)

The reward curve for SMDP Q-Learning represents the average across five different seeds over 5,000 episodes . 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.



Analysis:

- Early Exploration: During the initial stages of training, the algorithm exhibits high variability as it actively explores the environment and learns from a diverse range of experiences.
- Plateau phase: As the training progresses, the reward curve gradually smooths out, indicating the agent's improving proficiency in navigating the environment leading to more consistent performance over time.
- Policy Convergence: The stabilization of the reward curve implies that the agent is converging towards a stable policy. The hierarchical nature of the algorithm, facilitated by SMDP, enables the agent to learn robust policies that generalize well across the state space, effectively accounting for temporal dependencies in action sequences.
- Consistent performance(reduced exploration): Towards the later stages of training, the decrease in variance in the reward curve suggests a shift from exploration to exploitation(e- greedy’s nature). This shift allows the algorithm to exploit learned values more effectively, leading to diminished exploration and a more refined policy.

Algorithm 2 – IOQl

Options, also known as temporally extended actions, are higher-level actions that encompass a sequence of primitive actions. Intra-option Q-learning focuses on learning the Q-values associated with these options.

The core idea behind IOQL is to enable an agent to make decisions not only at the level of primitive actions but also at the level of options. This allows the agent to consider longer-term consequences of its actions and to learn more efficient policies for achieving its goals.

The Q-value function is updated using the Bellman equation, taking into account the Q-values of subsequent options or primitive actions.

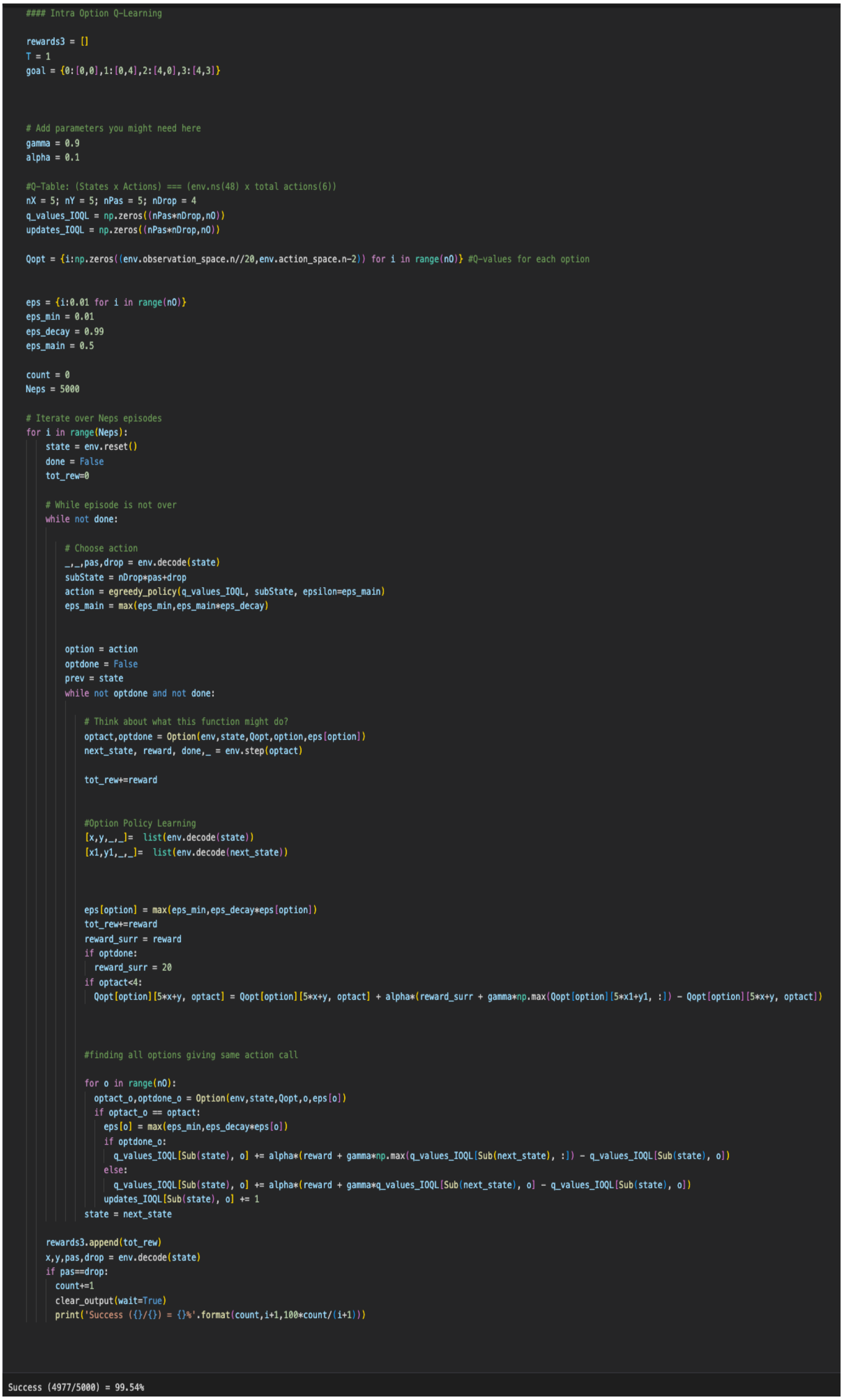
$$Q(s_t, o) \leftarrow Q(s_t, o) + \alpha\big[\ r_{t+1} + \gamma U(s_{t+1}, o)\big) \quad Q(s_t, o)\big],$$

where

$$U(s, o) = \ 1 \quad \beta(s)\big)Q(s, o) + \beta(s) \max_{o' \in \mathcal{O}} Q(s, o').$$

This equation updates the Q-value of the current option by combining the previously learned value with the newly acquired information from the immediate reward and the expected future rewards.

[Code for the same below]

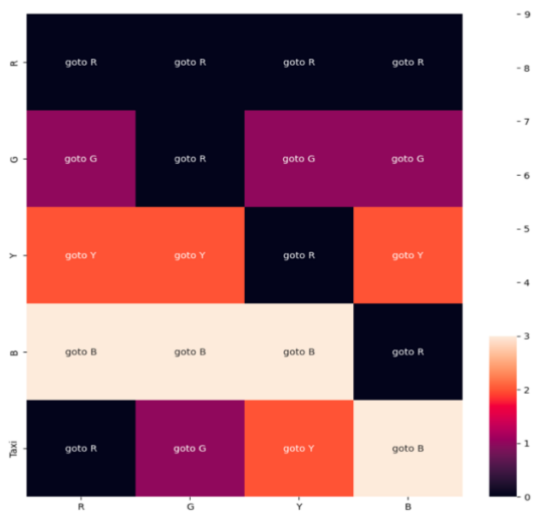


```
#### Intra Option Q-Learning

rewards3 = []
T = 1
goal = {0:[0,0],1:[0,4],2:[4,0],3:[4,3]}



# Add parameters you might need here
gamma = 0.9
alpha = 0.1

#Q-Table: (States x Actions) === (env.ns(48) x total actions(6))
nX = 5; nY = 5; nPas = 5; nDrop = 4
q_values_IOQL = np.zeros((nPas*nDrop,nO))
updates_IOQL = np.zeros((nPas*nDrop,nO))

Qopt = {i:np.zeros((env.observation_space.n//20,env.action_space.n-2)) for i in range(nO)} #Q-values for each option


eps = {i:0.01 for i in range(nO)}
eps_min = 0.01
eps_decay = 0.99
eps_main = 0.5

count = 0
Neps = 5000

# Iterate over Neps episodes
for i in range(Neps):
    state = env.reset()
    done = False
    tot_rew=0

    # While episode is not over
    while not done:

        # Choose action
        _,_,pas,drop = env.decode(state)
        subState = nDrop*pas+drop
        action = egreedy_policy(q_values_IOQL, subState, epsilon=eps_main)
        eps_main = max(eps_min,eps_main*eps_decay)


        option = action
        optdone = False
        prev = state
        while not optdone and not done:

            # Think about what this function might do?
            optact,optdone = Option(env,state,Qopt,option,eps[option])
            next_state, reward, done,_ = env.step(optact)

            tot_rew+=reward


            #Option Policy Learning
            [x,y,_,_]=  list(env.decode(state))
            [x1,y1,_,_]=  list(env.decode(next_state))



            eps[option] = max(eps_min,eps_decay*eps[option])
            tot_rew+=reward
            reward_surr = reward
            if optdone:
              reward_surr = 20
            if optact<4:
              Qopt[option][5*x+y, optact] = Qopt[option][5*x+y, optact] + alpha*(reward_surr + gamma*np.max(Qopt[option][5*x1+y1, :]) - Qopt[option][5*x+y, optact])



            #finding all options giving same action call

            for o in range(nO):
              optact_o,optdone_o = Option(env,state,Qopt,o,eps[o])
              if optact_o == optact:
                eps[o] = max(eps_min,eps_decay*eps[o])
                if optdone_o:
                  q_values_IOQL[Sub(state), o] += alpha*(reward + gamma*np.max(q_values_IOQL[Sub(next_state), :]) - q_values_IOQL[Sub(state), o])
                else:
                  q_values_IOQL[Sub(state), o] += alpha*(reward + gamma*q_values_IOQL[Sub(next_state), o] - q_values_IOQL[Sub(state), o])
                updates_IOQL[Sub(state), o] += 1
            state = next_state

    rewards3.append(tot_rew)
    x,y,pas,drop = env.decode(state)
    if pas==drop:
      count+=1
      clear_output(wait=True)
      print('Success ({}/{}) = {}%'.format(count,i+1,100*count/(i+1)))
```

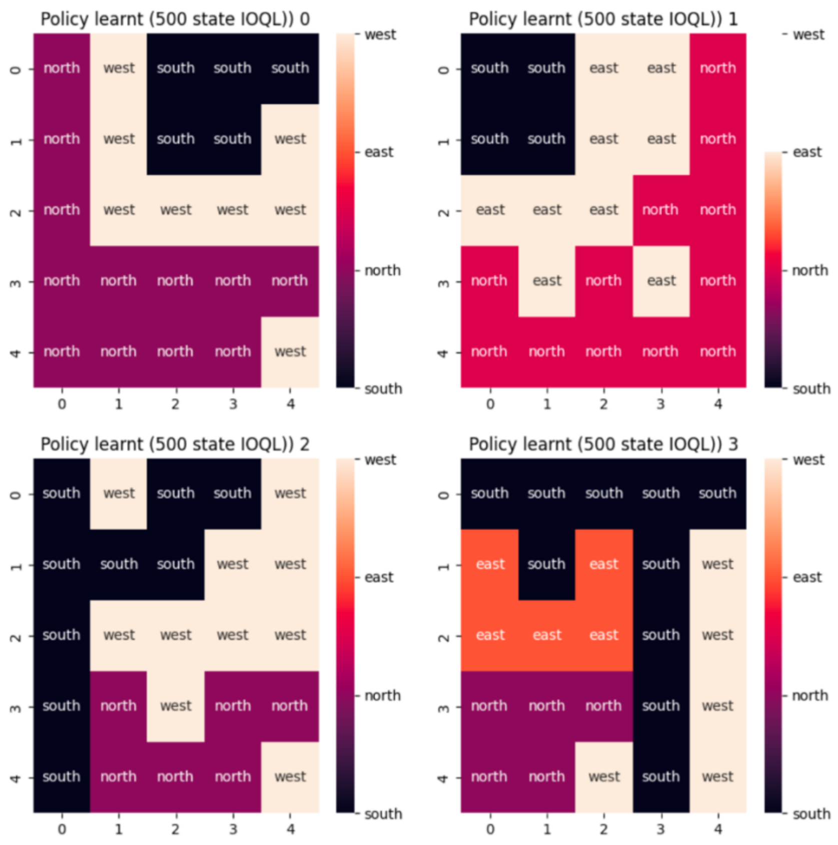
```
Success (4977/5000) = 99.54%
```

Visualization and results:

The generated heatplots, show the policy learned by IOQL in 2x2 grid fashion, and the below plot shows the directional policy actions taxi will take to move around in the environment.

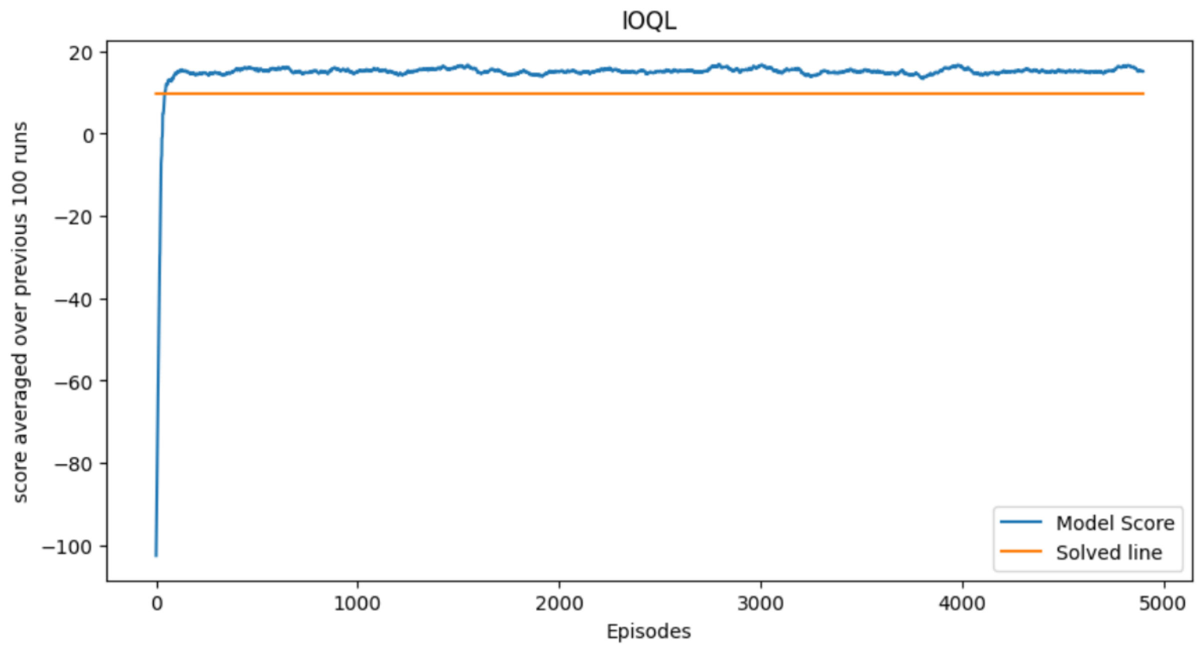


(option selection policy learnt via IOQL)



(IOQL Learnt policies for all 4 options)

The reward curve for SMDP Q-Learning represents the average across five different seeds over 5,000 episodes . 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.



Analysis:

- Early Exploration: During the initial stages of training, there is a sharp rise in the rewards and the algorithm exhibits high variability as it actively explores the environment and learns from a diverse range of experiences.

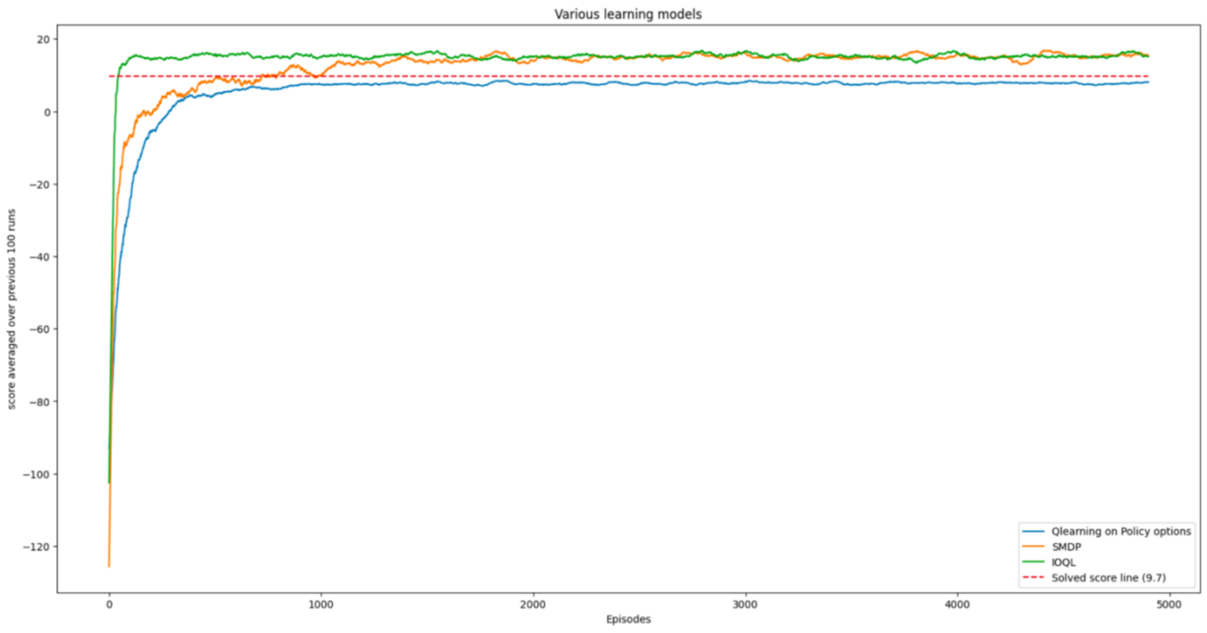
- Plateau phase: As the training progresses, the reward curve gradually smooths out, indicating the agent's improving proficiency in navigating the environment leading to more consistent performance over time.

- Policy Convergence: The stabilization of the reward curve implies that the agent is converging towards a stable policy. The hierarchical nature of the algorithm, facilitated by IOQL, enables the agent to learn robust policies that generalize well across the state space, effectively accounting for temporal dependencies in action sequences.

- Consistent performance(reduced exploration): Towards the later stages of training, the decrease in variance in the reward curve suggests a shift from exploration to exploitation(e- greedy’s nature). This shift allows the algorithm to exploit learned values more effectively, leading to diminished exploration and a more refined policy.

Comparison – SMDP, IOQL, Traditional Q learning

We see that both SMDP Q-learning and intra-option Q-learning methods learn quickly and efficiently. However, while comparing the reward versus episode plot shown below intra-option Q-learning (IOQL) outperforms SMDP Q-learning. IOQL not only converges faster but also achieves a higher convergence reward.



Rewards averaged over 100 episodes for different models

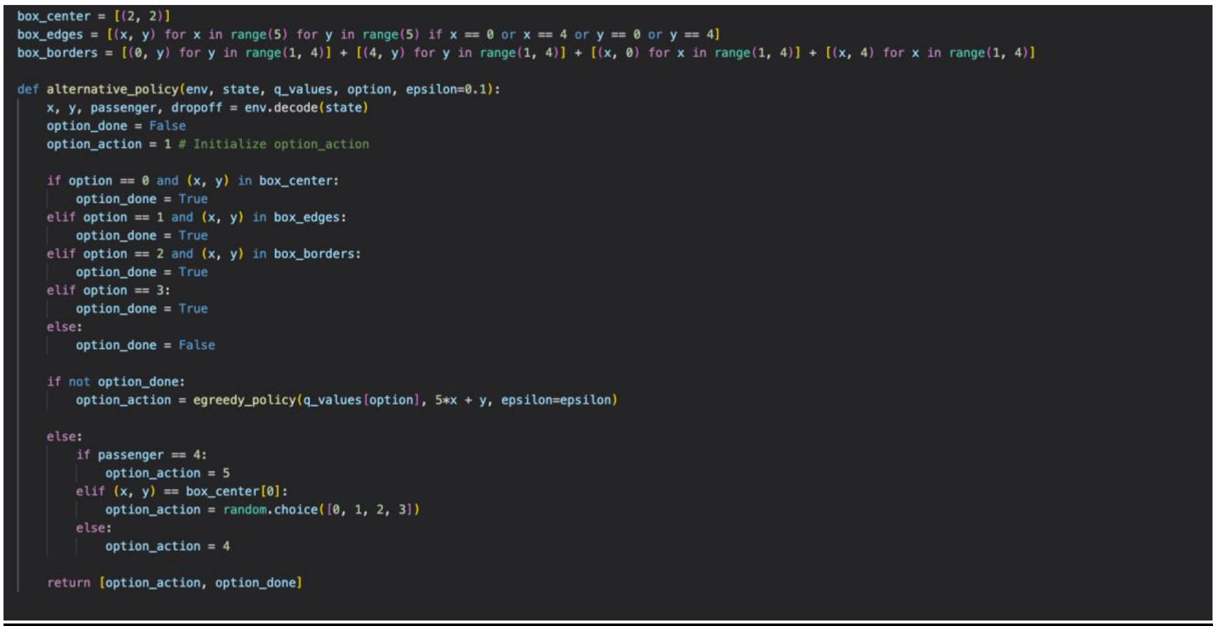
1. **SMDP (Semi-Markov Decision Process)**:
    - It models transitions between states as a sequence of steps, where each step can take different amounts of time.
    - One Q-Learning update is done to the state and option pair at the end of execution of an option.
    - SMDP is useful when actions have inherent sub-steps or when the environment dynamics are not fully Markovian.
2. **IOQL (Incremental Off-policy Q-learning)**:
    - It combines ideas from both Q-learning and experience replay.
    - An update is performed for each option that would have chosen the same action with its policy as the action chosen by the current executing option.
    - IOQL is particularly useful when dealing with sparse rewards or when the environment is stochastic.
3. **Traditional Q-learning**:
    - Traditional Q-learning is an off-policy reinforcement learning algorithm.
    - It learns an action-value function (Q-values) by iteratively updating estimates based on observed experiences.
    - Q-learning uses a greedy policy to select actions during exploration.
    - However, it can suffer from slow convergence and high variance in updates as seen above in the reward plot.

Alternate Options:

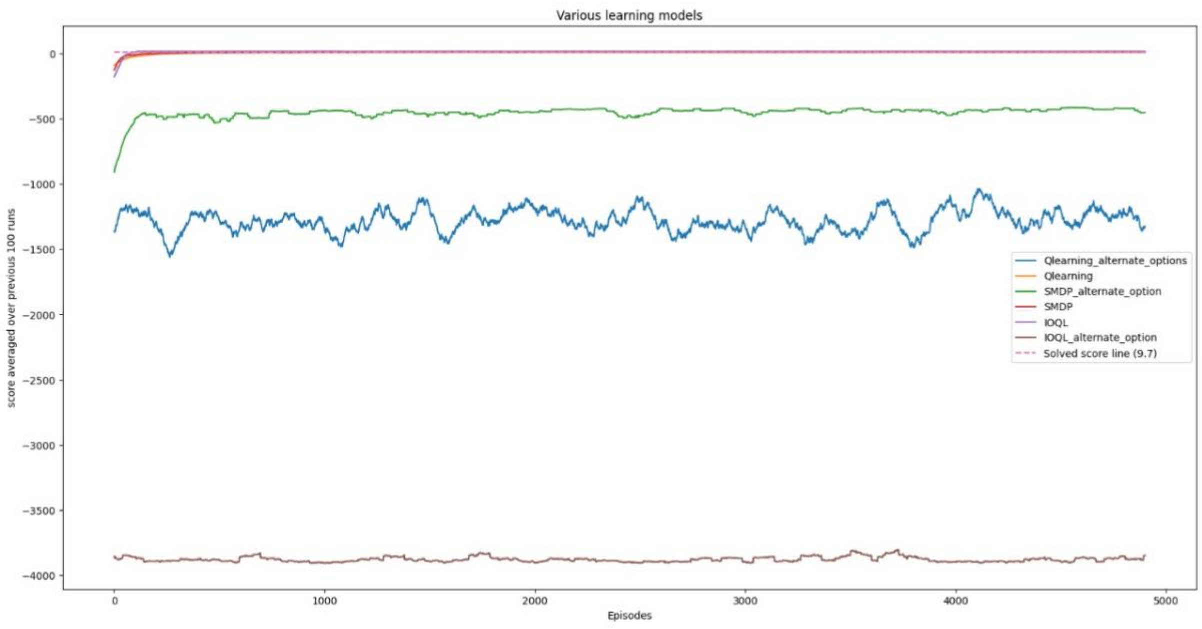
The "alternate option policy" are designed to position the taxi in a way , that are:

navigating to the centre of the grid for flexibility, moving around the edges to increase the chances of finding new passengers, moving to the nearest outer wall for quicker access to any corner, or no action needed: Sometimes, the taxi might already be en route to the passenger or destination. In such cases, no immediate action is required.

The idea behind designing these options is to max the taxi's availability and reduce the time for passenger pick-ups and drop-offs by strategically positioning it, rather than solely prioritizing the shortest path(greed y approach) to our predefined destinations(R, Y, B, G)



```python
box_center = [(2, 2)]
box_edges = [(x, y) for x in range(5) for y in range(5) if x == 0 or x == 4 or y == 0 or y == 4]
box_borders = [(0, y) for y in range(1, 4)] + [(4, y) for y in range(1, 4)] + [(x, 0) for x in range(1, 4)] + [(x, 4) for x in range(1, 4)]

def alternative_policy(env, state, q_values, option, epsilon=0.1):
    x, y, passenger, dropoff = env.decode(state)
    option_done = False
    option_action = 1 # Initialize option_action

    if option == 0 and (x, y) in box_center:
        option_done = True
    elif option == 1 and (x, y) in box_edges:
        option_done = True
    elif option == 2 and (x, y) in box_borders:
        option_done = True
    elif option == 3:
        option_done = True
    else:
        option_done = False

    if not option_done:
        option_action = egreedy_policy(q_values[option], 5*x + y, epsilon=epsilon)

    else:
        if passenger == 4:
            option_action = 5
        elif (x, y) == box_center[0]:
            option_action = random.choice([0, 1, 2, 3])
        else:
            option_action = 4

    return [option_action, option_done]
```



Conclusion:

1. In this Assignment, we explored two different Learning Algorithms on the OpenAI gym Taxi-V3 environment: namely SMDP Q-Learning and Intra Option Q-Learning.
2. It is certain from the last plot that a set of alternate options which are mutually exclusive to the current one is don't have high rewards and show low success rate.
3. Hence our algorithm with predefined set of options is the most optimum set.
4. Intra Option Q-Learning outperforms SMDP Q-Learning and both outperform Traditional Q learning.

GitHub link: https://github.com/good-doctor/CS6700--Reinforcement-Learning.git