Project 3

SMDP and Intra Option Q-Learning

Kemal Mudie Tosora
M.Tech In Data Science
Indian Institute of Technology Madras
Chennai, India
`ge22m010@smail.iitm.ac.in`
Professor Balaraman Ravindran
Head School of Data Science and AI, Robert Bosch Data Science and AI Lab, Center for Responsible AI
Indian Institute of Technology Madras
Chennai, India
`ravi@cse..iitm.ac.in`

Reinforcement Learning

Department of Data Science and Artificial Intelligence

Indian Institute Of Technology Madras

Contents

1 Introduction

In this programming assignment specifically focuses on the application of two advanced RL techniques: Semi-Markov Decision Processes (SMDP) and Intra-Option Q-Learning. Our primary goal is to implement and compare the performance of these techniques in the context of the Taxi-v3 environment provided by OpenAI's Gymnasium.

The Taxi-v3 environment presents a simple yet challenging task: a taxi navigating a grid world to pick up and drop off passengers at designated locations. By implementing SMDP and Intra-Option Q-Learning, we aim to enhance the taxi's decision-making process by enabling it to learn high-level actions (options) that can span multiple time steps, thereby improving its efficiency and effectiveness in achieving its objectives.

The significance of this work lies in its potential to showcase the effectiveness of advanced RL techniques in solving complex problems. By breaking down the task into smaller, more manageable sub-goals, the agent can learn to make more informed decisions that lead to higher rewards, demonstrating the power and versatility of RL in real-world applications [1].

Through this assignment, we will not only gain insights into the practical implementation of SMDP and Intra-Option Q-Learning but also explore their impact on the performance of an RL agent in a simulated environment [4, 3].

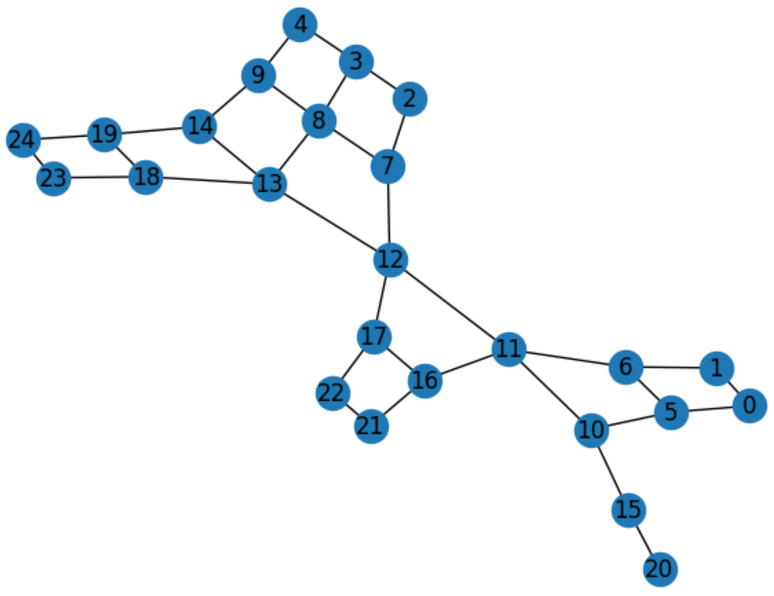


Figure 1: Visualizing Taxi-v3 environment as a graph

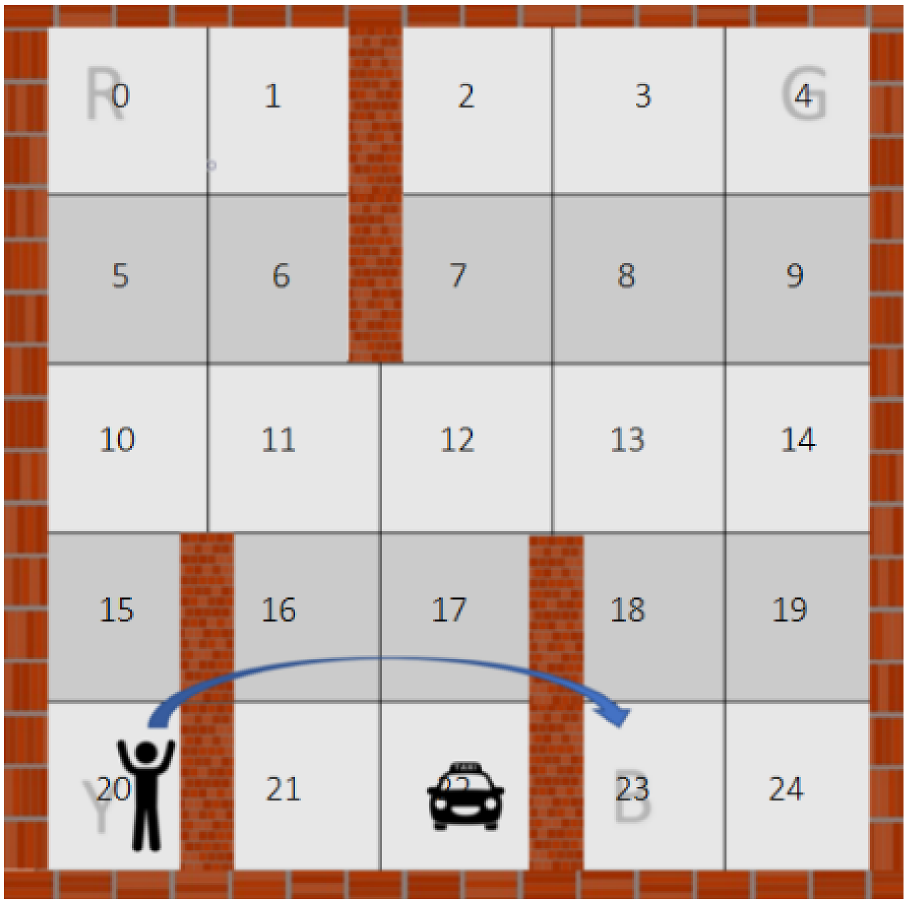


Figure 2: Visualizing Taxi-v3 environment with positions annotated

2 Experimental Setup

2.1 Environment Description

The environment for this task is the taxi domain, illustrated in Fig. 2. It is a 5x5 matrix, where each cell is a position your taxi can occupy. A single passenger can be picked up, dropped off, or in transit. There are four designated locations indicated by R(ed), G(reen), Y(ellow), and B(lue). At the start of each episode, the taxi and passenger are placed randomly. The taxi navigates to the passenger, picks them up, goes to the destination, and drops off the passenger, ending the episode. There are 500 discrete states: 25 taxi positions, 5 passenger locations (including in the taxi), and 4 destinations. However, only 400 states are reachable during an episode due to the end state when the passenger is at their destination. After a successful episode, four additional states are possible when both the taxi and passenger are at the destination, totaling 404 reachable states.
Passenger locations: 0: R(ed), 1: G(reen), 2: Y(ellow), 3: B(lue), 4: in taxi

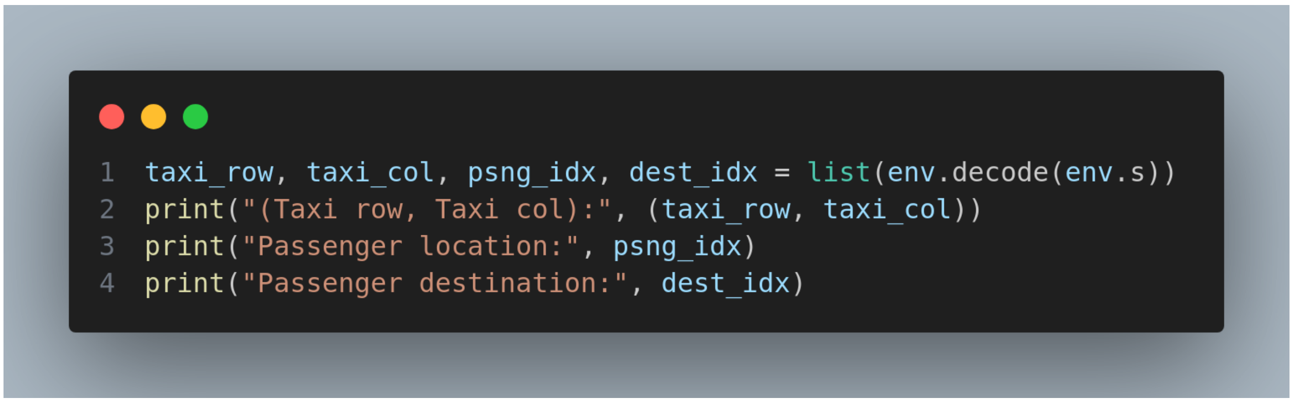
Destinations: 0: R(ed), 1: G(reen), 2: Y(ellow), 3: B(lue)
The reward structure is as follows:

- -1 per step unless another reward is triggered.
- +20 for delivering a passenger.
- -10 for executing illegal ”pickup” and ”drop-off” actions.

We use the ”Taxi-v3” environment provided by Gym, which models the problem as a 5x5 grid world with 6 actions:

- Move South
- Move North
- Move East
- Move West
- Pick-Up
- Drop-Off

An important method is the environment’s decode function, which converts integer-indexed states to tangible quantities.



```
taxi_row, taxi_col, psng_idx, dest_idx = list(env.decode(env.s))
print("(Taxi row, Taxi col):", (taxi_row, taxi_col))
print("Passenger location:", psng_idx)
print("Passenger destination:", dest_idx)
```

Figure 3: Decoding the State of the Environment

The `decode` method returns the current taxi row and column, passenger index, and destination index.

2.2 Option Definition

We start by defining the options given in the problem statement. The problem statement asks us to define 4 options; one each for moving the taxi to one of the four designated locations (R, G, B, Y), executable when the taxi is not already there. For defining an option we require three components: a policy, a termination condition, and an initiation set. A description of these three for our case:

- **Policy:** The objective of the option is to move the taxi to a designated location. Since we are penalized with a -1 reward for every timestep, it makes sense to define the option such that it takes the least amount of time to reach the required location. We will keep this in consideration while designing policies for each option.
- **Termination condition:** In our case, this is rather straightforward as all the options terminate deterministically only on reaching the required location.
- **Initiation set:** The initiation set consists of all states apart from the target location. Therefore we use an if condition on the current state to check if the current state is a valid location to perform the option.

Next, we define the policy and termination conditions for the four options. We relegate the discussion of initiation sets to a later subsection. For the following discussion, taxi location (x, y) corresponds to the taxi being at row x and column y.

- **Move to R:**

  We use the following conditions to define the policy (for the shortest path):

  - Note that wherever possible, it is optimal to move left (west).
  - Once you reach column 0, it is optimal to move up (north).
  - If you are at (0, 2) or (1, 2), it is optimal to move down (south).
  - If you are at (3, 1), (3, 3), (4, 1) or (4, 3), it is optimal to move up (north).

  This option terminates on reaching (0, 0).

- **Move to Y:**

  The policy is identical to the case of R except for the fact that you have to move down (south) in column 0 instead of up (north).

  This option terminates on reaching (4, 0).

- **Move to G:**

  We use the following conditions to define the policy:

  - Once you reach column 4, it is optimal to move up (north).
  - If you are at (0, 1) or (1, 1), it is optimal to move down (south).
  - If you are at (3, 0), (3, 2), (4, 0) or (4, 2), it is optimal to move up (north).
  - In all other cases, it is optimal to move right (east).

  This option terminates on reaching (0, 4).

- **Move to B:**

  We use the following conditions to define the policy:

  - If you are at (4, 4), it is optimal to move left (west).
  - Otherwise, once you reach column 3 or column 4, it is optimal to move down (south).
  - If you are at (0, 1) or (1, 1), it is optimal to move down (south).
  - If you are at (3, 0), (3, 2), (4, 0) or (4, 2), it is optimal to move up.
  - In all other cases, it is optimal to move right (east).

  This option terminates on reaching (4, 3).

  We proceed to assign an integer value to each option as follows:

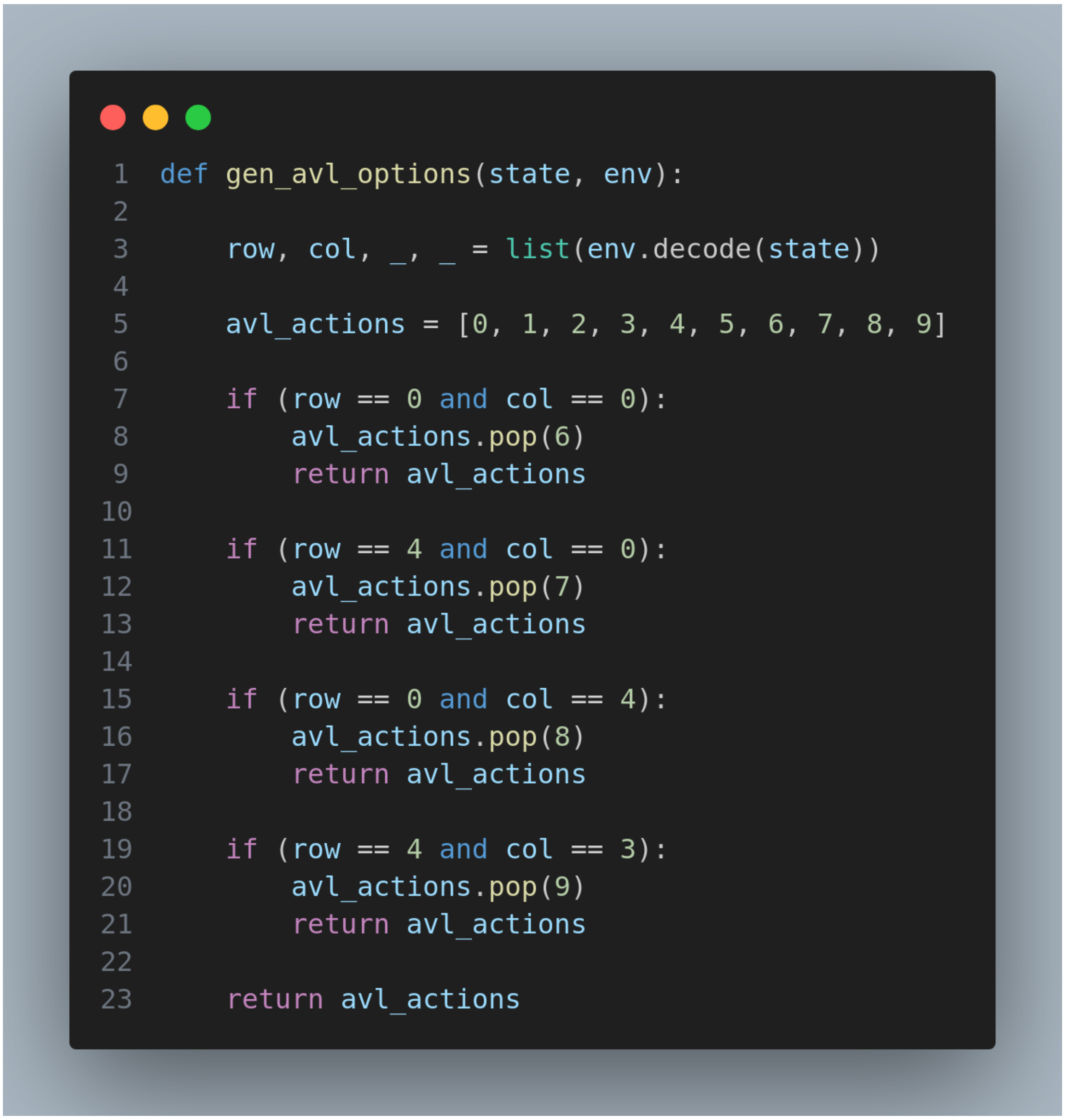
  - 6: Move to R
  - 7: Move to Y
  - 8: Move to G
  - 9: Move to B

  Therefore, including both primitive actions and options, we have a total of 10 possible actions.

- **Initiation sets**:

  We note that for each option, the initiation set is the set of all cells except for the target grid cell. For instance, for the option 'Move to R', the initiation set consists of all cells except (0, 0).

  We write a function *gen_avl_options()* which takes in a state as an argument and outputs all the possible options that can be executed from that state:

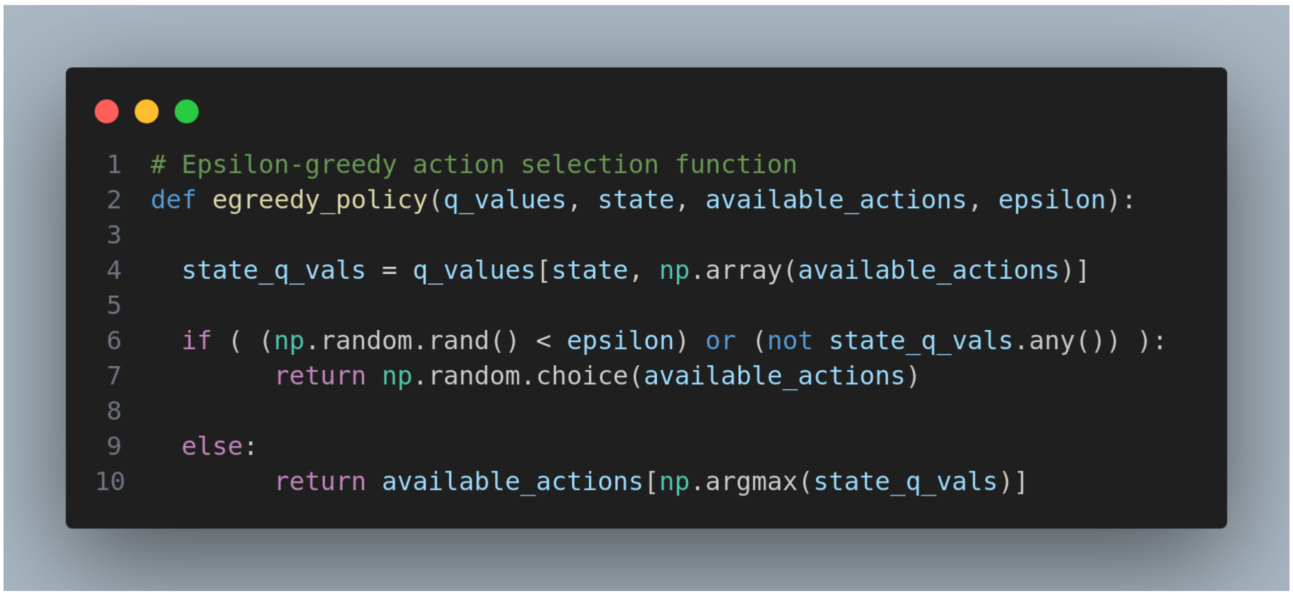


```python
def gen_avl_options(state, env):

    row, col, _, _ = list(env.decode(state))

    avl_actions = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

    if (row == 0 and col == 0):
        avl_actions.pop(6)
        return avl_actions

    if (row == 4 and col == 0):
        avl_actions.pop(7)
        return avl_actions

    if (row == 0 and col == 4):
        avl_actions.pop(8)
        return avl_actions

    if (row == 4 and col == 3):
        avl_actions.pop(9)
        return avl_actions

    return avl_actions
```

Figure 4: Generating Available Options for each state

2.3 Exploration Strategy

We use an ϵ-greedy strategy for exploration. A slight difference from the usual approach is that we ensure the selection is made only from the list of available actions for each state. This is achieved by passing the list of available options, obtained as the output of the *gen_avl_options()* function, as an input to the ϵ-greedy strategy and performing selection on this subset.

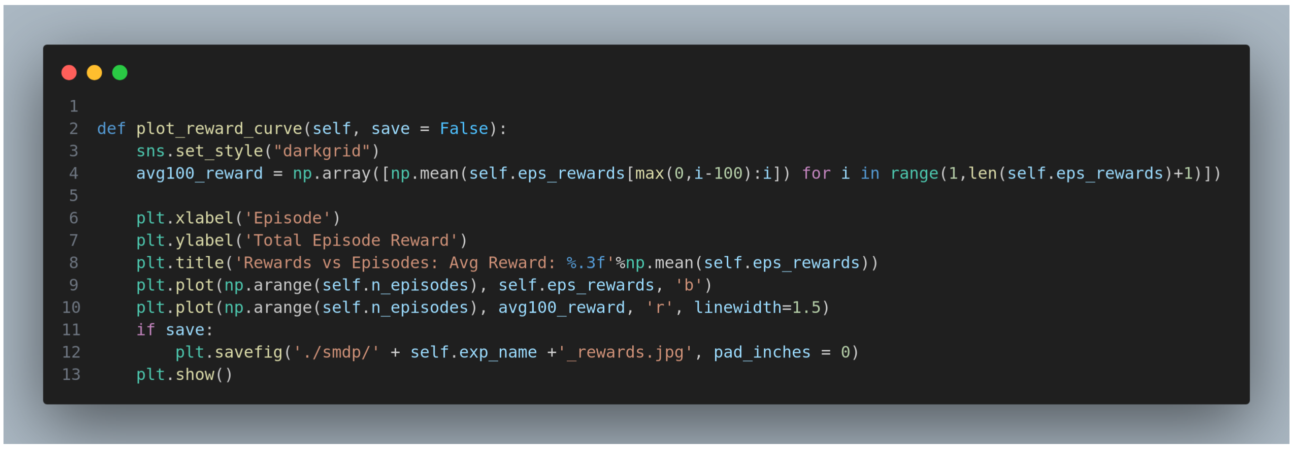


```python
# Epsilon-greedy action selection function
def egreedy_policy(q_values, state, available_actions, epsilon):

  state_q_vals = q_values[state, np.array(available_actions)]

  if ( (np.random.rand() < epsilon) or (not state_q_vals.any()) ):
        return np.random.choice(available_actions)

  else:
        return available_actions[np.argmax(state_q_vals)]
```

Figure 5: ϵ-greedy Action Selection

2.4 Plotting Reward Curve

The reward curve is plotted across 10 runs and averaged with the number of episodes bound at 3000 for both SMDP and Intra-Option Q-Learning. The moving average across a window of 100 episodes is also plotted to give an idea of the average trends.
The code block for plotting the reward curve is given below:

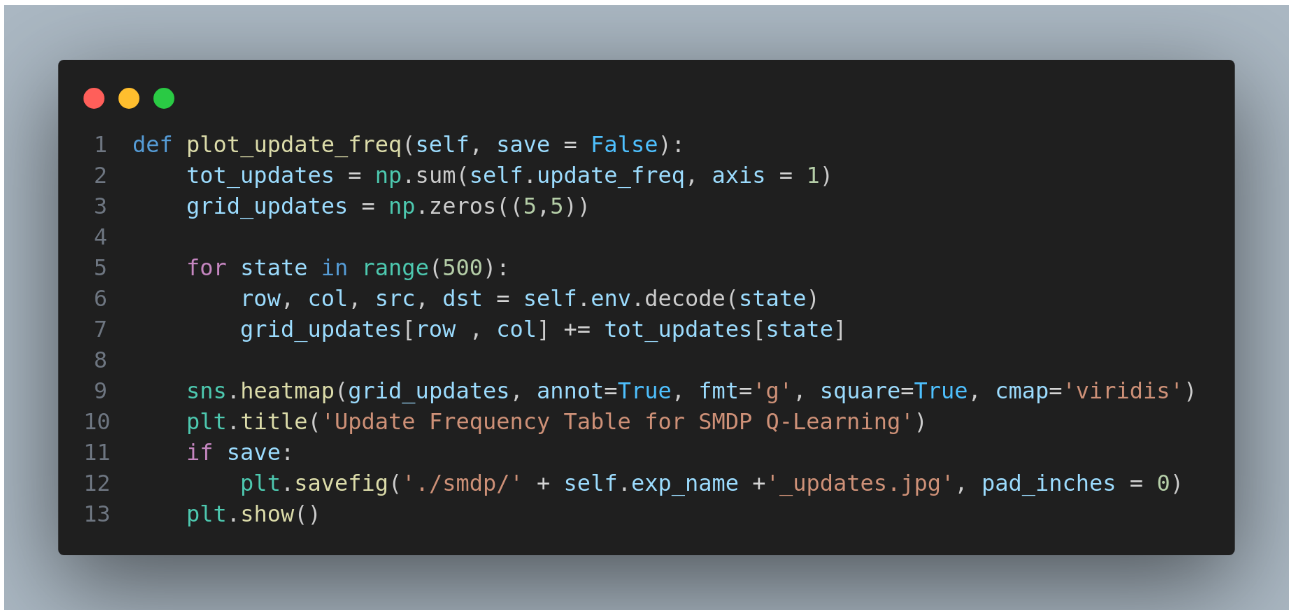


```python

def plot_reward_curve(self, save = False):
    sns.set_style("darkgrid")
    avg100_reward = np.array([np.mean(self.eps_rewards[max(0,i-100):i]) for i in range(1,len(self.eps_rewards)+1)])

    plt.xlabel('Episode')
    plt.ylabel('Total Episode Reward')
    plt.title('Rewards vs Episodes: Avg Reward: %.3f'%np.mean(self.eps_rewards))
    plt.plot(np.arange(self.n_episodes), self.eps_rewards, 'b')
    plt.plot(np.arange(self.n_episodes), avg100_reward, 'r', linewidth=1.5)
    if save:
        plt.savefig('./smdp/' + self.exp_name +'_rewards.jpg', pad_inches = 0)
    plt.show()
```

Figure 6: Plotting the Reward Curve

2.5 Plotting Update Frequency Tables

The update frequencies are stored in a 500 × 10 array with one value for every action and state combination. However, to visualize them better we aggregate the update across all actions and simply plot the update frequency in the form of a table. The code block for plotting the reward curve is given below:



```python
def plot_update_freq(self, save = False):
    tot_updates = np.sum(self.update_freq, axis = 1)
    grid_updates = np.zeros((5,5))

    for state in range(500):
        row, col, src, dst = self.env.decode(state)
        grid_updates[row , col] += tot_updates[state]

    sns.heatmap(grid_updates, annot=True, fmt='g', square=True, cmap='viridis')
    plt.title('Update Frequency Table for SMDP Q-Learning')
    if save:
        plt.savefig('./smdp/' + self.exp_name +'_updates.jpg', pad_inches = 0)
    plt.show()
```

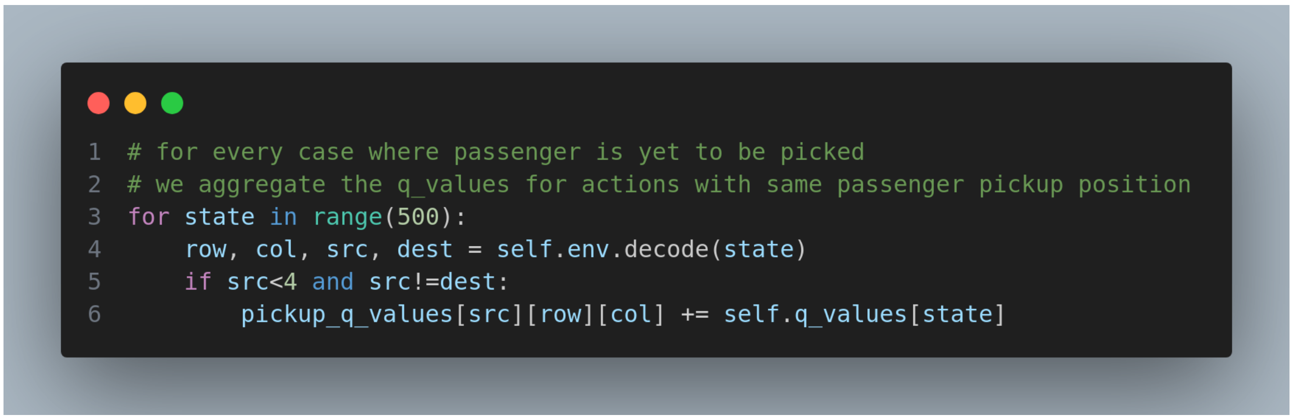
Figure 7: Plotting the Update Frequency Table

2.6 Plotting Q-Value Tables

The Q-values are stored in a 500×10 array with one value for every action and state combination. However, to visualize them better, we split the state set into 2:

- **Pick Up:** When $psng\_idx < 4$, the passenger is not inside the taxi yet, and the taxi must journey to the passenger location (psng idx) and pick up the passenger.
- **Drop Off:** When $psng\_idx = 4$, the passenger is inside the taxi, and the taxi must journey to the destination location dest idx and drop off the passenger.

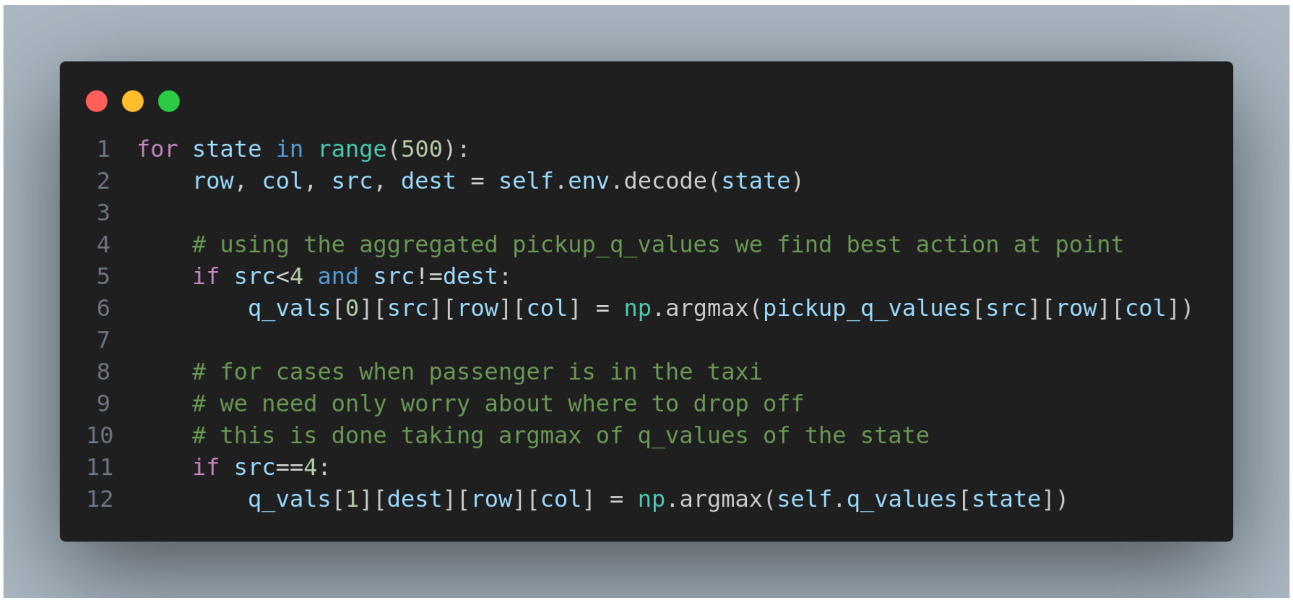
In order to form a cohesive understanding of how the taxi moves in the ”pickup” phase, we aggregate the Q-values for each action and final destination belonging to a particular taxi location and pickup location of the passenger.
This is done using the following code block:



```python
# for every case where passenger is yet to be picked
# we aggregate the q_values for actions with same passenger pickup position
for state in range(500):
    row, col, src, dest = self.env.decode(state)
    if src<4 and src!=dest:
        pickup_q_values[src][row][col] += self.q_values[state]
```

Figure 8: Calculating Aggregate Q-Values for Pickup

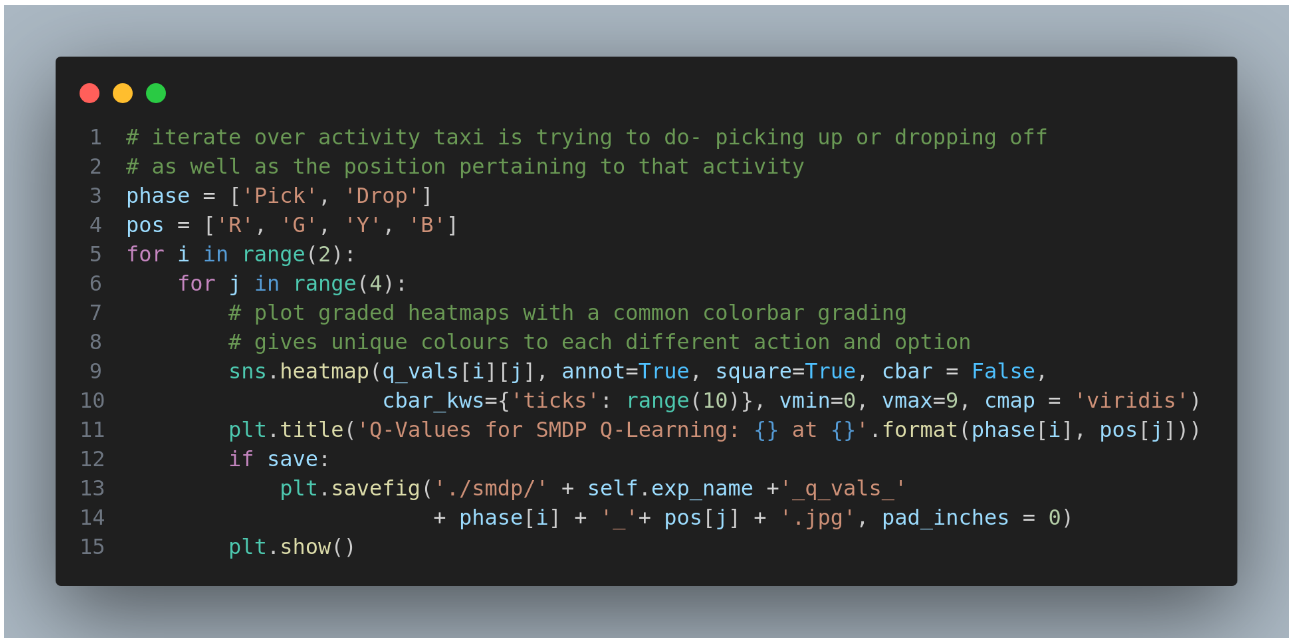
In order to find the favoured action of the taxi, we calculate the action with the maximum Q-value amongst the available actions for each phase of the taxi movement as well as for various pickup and drop-off locations.



```python
for state in range(500):
    row, col, src, dest = self.env.decode(state)

    # using the aggregated pickup_q_values we find best action at point
    if src<4 and src!=dest:
        q_vals[0][src][row][col] = np.argmax(pickup_q_values[src][row][col])

    # for cases when passenger is in the taxi
    # we need only worry about where to drop off
    # this is done taking argmax of q_values of the state
    if src==4:
        q_vals[1][dest][row][col] = np.argmax(self.q_values[state])
```

Figure 9: Calculating Action with highest Q-value

Once we have the required “best” actions, we can now plot the actions chosen in the form of heatmaps for each of the 8 cases, with the phase being either pick up or drop off and the location of pickup/dropoff coming from the list [R, G, Y, B].



```python
# iterate over activity taxi is trying to do- picking up or dropping off
# as well as the position pertaining to that activity
phase = ['Pick', 'Drop']
pos = ['R', 'G', 'Y', 'B']
for i in range(2):
    for j in range(4):
        # plot graded heatmaps with a common colorbar grading
        # gives unique colours to each different action and option
        sns.heatmap(q_vals[i][j], annot=True, square=True, cbar = False,
                    cbar_kws={'ticks': range(10)}, vmin=0, vmax=9, cmap = 'viridis')
        plt.title('Q-Values for SMDP Q-Learning: {} at {}'.format(phase[i], pos[j]))
        if save:
            plt.savefig('./smdp/' + self.exp_name +'_q_vals_'
                        + phase[i] + '_'+ pos[j] + '.jpg', pad_inches = 0)
        plt.show()
```

Figure 10: Generating Best Action Heatmaps

3 SMDP Q-Learning

Semi-Markov Decision Processes (SMDPs) extend Markov Decision Processes (MDPs) by introducing actions that can take multiple time steps to complete, known as options[2]. An option is defined as a sequence of actions that terminates in a specific state, referred to as the ”goal” state.

SMDP Q-Learning extends the classic Q-Learning algorithm to incorporate options, allowing for semi-markov processes. Unlike vanilla Q-Learning, which assumes an MDP, SMDP Q-Learning enables the agent to learn faster by generalizing primitive actions into more elaborate options. At each time step, the agent selects an action based on its Q-values for the current state. When an option is selected, the agent executes the option until it reaches the goal state. Upon reaching the goal state, the Q-value for the option is updated based on the reward obtained and the Q-values of the options in the goal state.

Key points of SMDP Q-Learning:

- Introduces options as sequences of actions terminating in goal states.
- Allows actions to take multiple time steps to complete.
- Enables faster learning by generalizing primitive actions into options.
- Updates Q-values for options based on rewards and the Q-values of options in goal states.

SMDP Q-Learning is particularly useful in tasks where actions have long-term effects or where certain actions are better suited for achieving long-term goals. By allowing the agent to learn options, it can more effectively navigate complex environments and achieve its objectives efficiently.

3.1 Update Equations

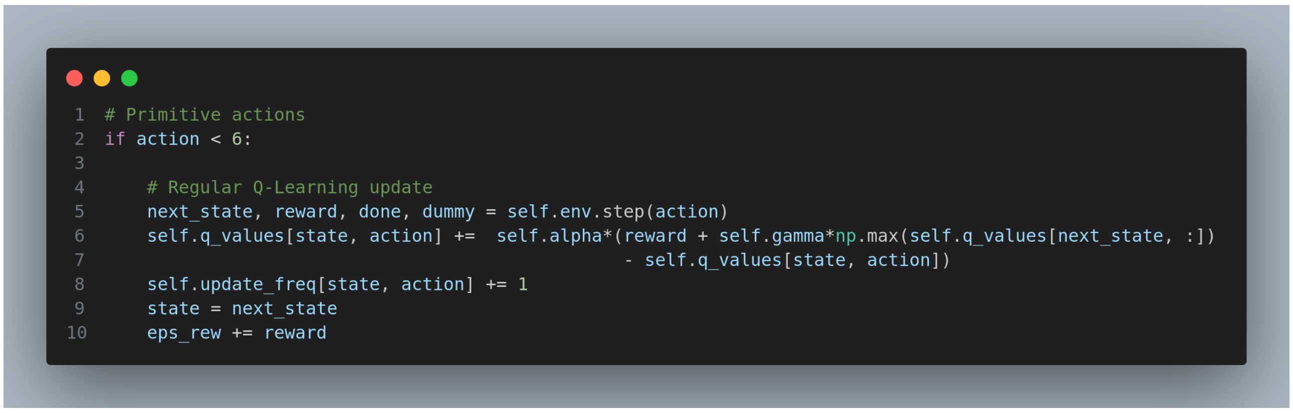
When a primitive action a is selected, the Q-value is updated according to the classic Q-Learning update equation: When a primitive action a is selected, the Q-value is updated according to the classic Q-Learning update equation:

$$Q'(s,a) = Q(s,a) + \alpha \cdot [r + \gamma \cdot \max_{o'} Q(s',o') - Q(s,a)] \tag{1}$$

where,

- $Q(o,s,a)$: Q-value of state s when action a is taken
- α: Learning Rate
- R: Reward for choosing action a at state s
- γ: Discount Factor
- o': Next action/option
- s': Next State

The code blocks for updates in case of primitive actions are shown below:



```python
# Primitive actions
if action < 6:

    # Regular Q-Learning update
    next_state, reward, done, dummy = self.env.step(action)
    self.q_values[state, action] +=  self.alpha*(reward + self.gamma*np.max(self.q_values[next_state, :])
                                                 - self.q_values[state, action])
    self.update_freq[state, action] += 1
    state = next_state
    eps_rew += reward
```

Figure 11: SMDP update for primitive actions

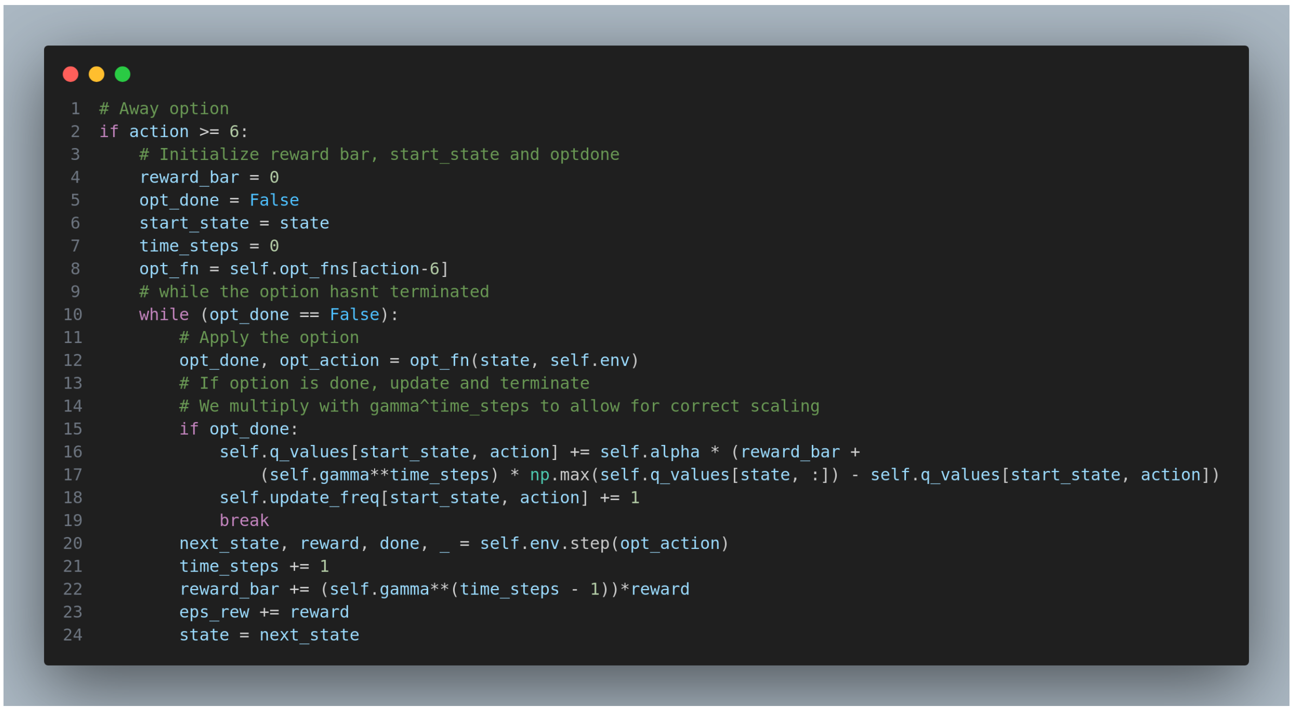
However, the updated equation for the Q-value of an option is as follows:

$$Q'(o,s,a) = Q(o,s,a) + \alpha \cdot [R + \gamma^{\tau} \cdot \max_{o'} Q(o,s',o') - Q(o,s,a)] \tag{2}$$

where,

- $Q(o,s,a)$: Q-value of option o in state s when action a is taken
- α: Learning Rate
- R: cumulative discounted reward across all actions taken as part of the option
- γ: discount factor while calculating return
- τ: Number of time steps as part of an option
- o': Next action/option

The code blocks for updates in case of option selection are shown below:



```python
# Away option
if action >= 6:
    # Initialize reward bar, start_state and optdone
    reward_bar = 0
    opt_done = False
    start_state = state
    time_steps = 0
    opt_fn = self.opt_fns[action-6]
    # while the option hasnt terminated
    while (opt_done == False):
        # Apply the option
        opt_done, opt_action = opt_fn(state, self.env)
        # If option is done, update and terminate
        # We multiply with gamma^time_steps to allow for correct scaling
        if opt_done:
            self.q_values[start_state, action] += self.alpha * (reward_bar +
                (self.gamma**time_steps) * np.max(self.q_values[state, :]) - self.q_values[start_state, action])
            self.update_freq[start_state, action] += 1
            break
        next_state, reward, done, _ = self.env.step(opt_action)
        time_steps += 1
        reward_bar += (self.gamma**(time_steps - 1))*reward
        eps_rew += reward
        state = next_state
```

Figure 12: SMDP update for option selection

3.2 Hyperparameter Tuning

We carry out hyperparameter tuning on the hyperparameters like α (learning rate), and ϵ the control parameter for the ϵ-greedy action selection rule.

This metric used to evaluate the performance of a combination of hyperparameters is the average reward.
We test the following combinations of hyperparameters (a total of 20 configurations):

- α: 0.5, 0.1, 0.05, 0.01
- ϵ: 0.1, 0.05, 0.01, 0.005, 0.001

Note: It was observed during initial experimentation that when $\gamma = 0.99$ is selected the convergence of SMDP as well as the average rewards are comparable if not better than Intra-Option, however for the sake of the assignment the given value of $\gamma = 0.9$ was used.

3.3 Results

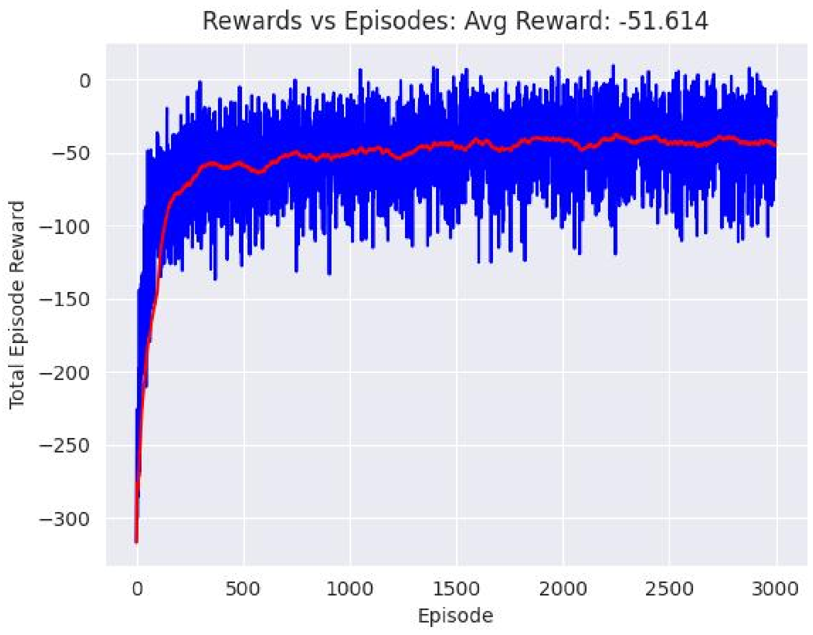


Figure 13: SMDP: Average reward plot

- From the average reward plot (averaged over 10 runs), we observe that the **rewards per episode increases as we expect**. This indicates that the agent is learning to solve the environment more efficiently. The red line indicates the running average over the past 100 episodes, which increases as it should.

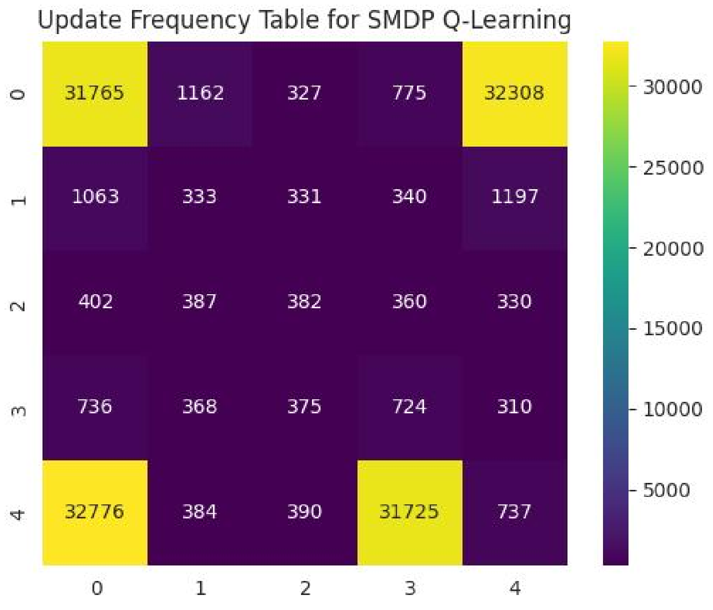
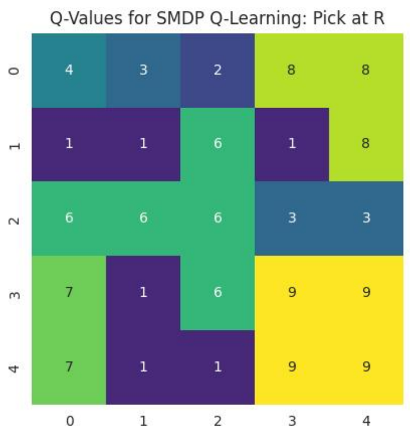
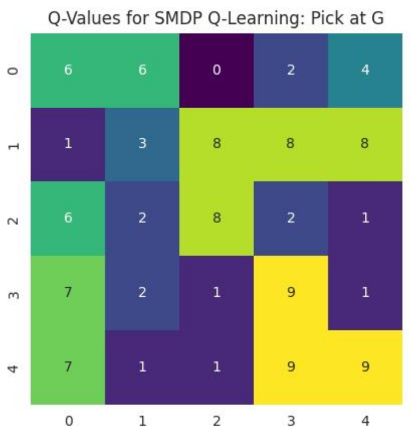


Figure 14: SMDP: Update frequency

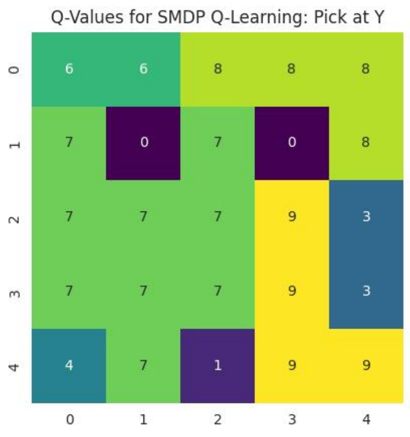
- From the update frequency plot, we observe that the **agent largely makes use of options**. Since the defined options are **very precise and goal-oriented**, taking them would have led to higher Q-values and consequently higher update frequencies. Since primitive actions do not get updated when options are chosen, their update frequency remains low.



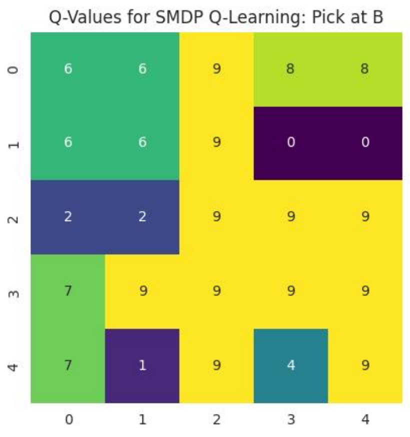
(a) Pick at location R



(b) Pick at location G



(c) Pick at location Y



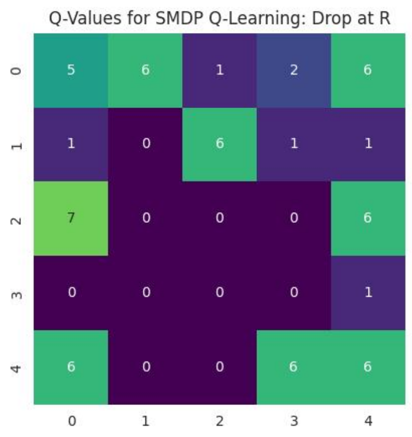
(d) Pick at location B

Figure 15: SMDP Q-learning: Actions/Options with largest Q-value for each grid cell

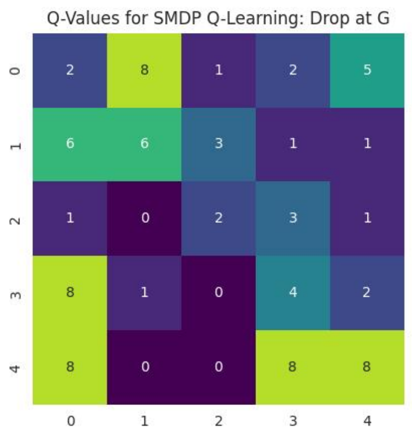
The four heatmaps displayed above indicate the preferred action for each grid cell based on the Q-value for the cases where the taxi has yet to pick up the passenger. These heatmaps provide insight into the policy learnt by the agent.

- It can be seen that the taxi **picks up the passenger at the specified location in all cases** from the fact that action 4 (pick-up) is performed at the passenger locations.
- In other grid cells, the policy learnt is such that the **taxi moves closer to the passenger location** as can be seen from the fact that in many cells, the option to move to the passenger location is chosen (6 for R, 8 for G, 7 for Y and 9 for B).

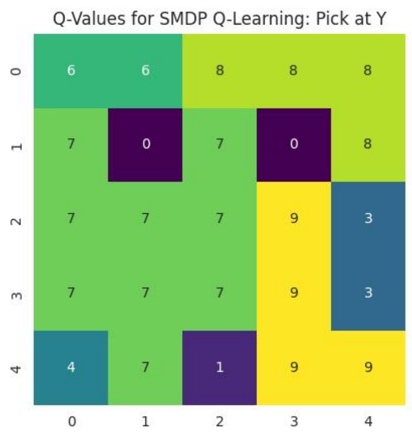
- Even when other primitive actions/options are chosen, they are generally such that they lead to the passenger location.



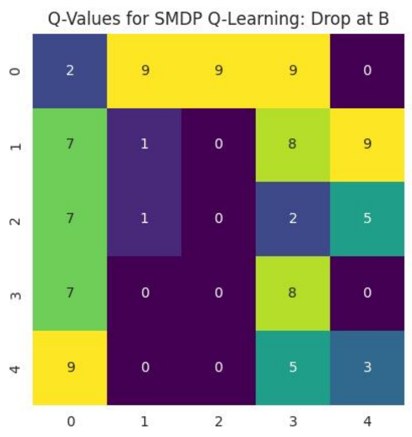
(a) Drop at location R



(b) Drop at location G



(c) Drop at location Y



(d) Drop at location B

Figure 16: SMDP Q-learning: Actions/Options with largest Q-value for each grid cell

The four heatmaps displayed above indicate the preferred action for each grid cell based on the Q-value for the cases where the taxi has picked up the passenger and is ready to drop the passenger off at the destination. These heatmaps provide insight into the policy learnt by the agent.

- It can be seen that the taxi **drops off the passenger at the specified location in all cases** from the fact that action 5 (drop-off) is performed at the passenger locations. However, we note that the **performance in other grid cells is poorer when compared to the pick-up plots**. Though the required option is chosen in some cells, wrong actions/options are chosen in several cases. This is probably because of the **slow convergence of the SMDP Q-learning algorithm**.

4 Intra-Option Q-Learning

Intra-Option Q-Learning is an extension of SMDP Q-Learning developed to provide a more efficient alternative. The key idea behind Intra-Option Q-Learning is to address the relatively slow nature of SMDP methods, which require an option to be executed until termination.

- Unlike SMDP Q-Learning, where only the chosen option's Q-values are updated, Intra-Option Q-Learning updates the Q-values for all compatible options. Options that result in the same primitive action are termed compatible.
- This means that even if an option is chosen, the Q-value for the corresponding primitive action is updated. This results in a much higher frequency of updates compared to SMDP Q-Learning.
- Due to its more frequent updates, Intra-Option Q-Learning is expected to converge faster than conventional SMDP Q-Learning. This efficiency makes it a promising approach for learning in complex environments where long-term dependencies exist.

4.1 Update Equations

When a primitive action a is selected, like the SMDP case, we make the following update:

$$Q'(s,a) = Q(s,a) + \alpha \cdot [r + \gamma \cdot \max_{o'} Q(s',o') - Q(s,a)] \tag{3}$$

where,

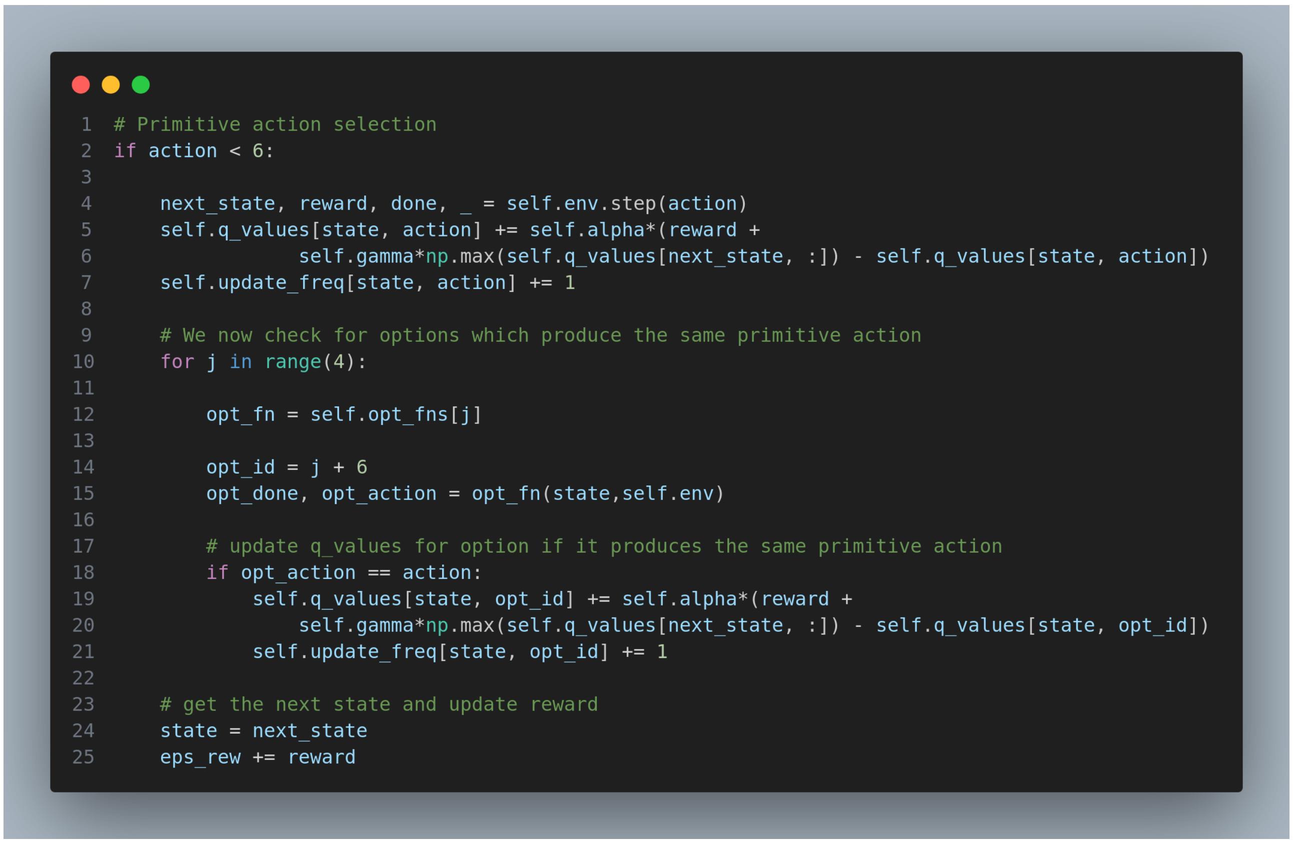
- $Q(o,s,a)$: Q-value of state s when action a is taken
- α: Learning Rate
- R: Reward for choosing action a at state s
- γ: Discount Factor
- o': Next action/option

- s': Next State

However, we now also perform the update:

$$Q'(s,o) = Q(s,o) + \alpha \cdot [r + \gamma \cdot \max_{o'} Q(s',o') - Q(s,o)] \tag{4}$$

$\forall o \in \mathcal{O}'$ where $\mathcal{O}'$ is the set of all options compatible with action a.
The code blocks for updates in case of action selection are shown below:



```python
# Primitive action selection
if action < 6:

    next_state, reward, done, _ = self.env.step(action)
    self.q_values[state, action] += self.alpha*(reward +
                self.gamma*np.max(self.q_values[next_state, :]) - self.q_values[state, action])
    self.update_freq[state, action] += 1

    # We now check for options which produce the same primitive action
    for j in range(4):

        opt_fn = self.opt_fns[j]

        opt_id = j + 6
        opt_done, opt_action = opt_fn(state,self.env)

        # update q_values for option if it produces the same primitive action
        if opt_action == action:
            self.q_values[state, opt_id] += self.alpha*(reward +
                self.gamma*np.max(self.q_values[next_state, :]) - self.q_values[state, opt_id])
            self.update_freq[state, opt_id] += 1

    # get the next state and update reward
    state = next_state
    eps_rew += reward
```

Figure 17: Intra-Option update for primitive actions

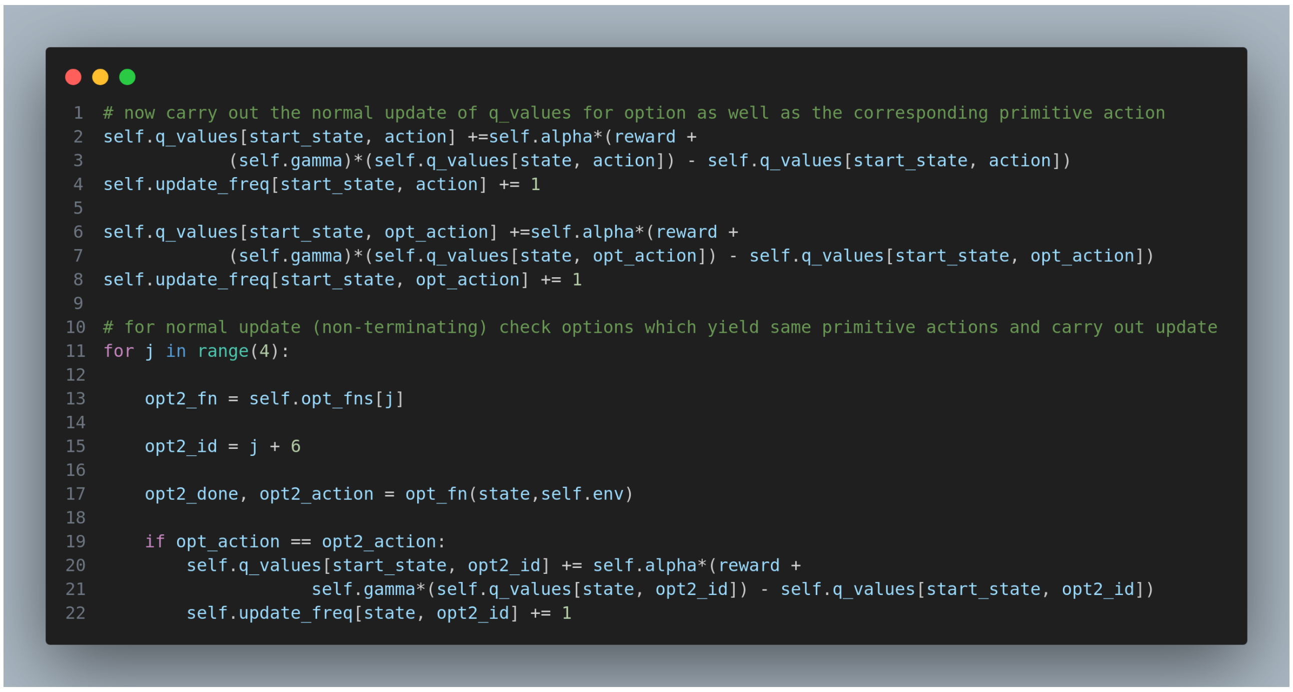
When an option o is selected, we perform the following updates **for all compatible options and primitive actions**:

$$Q'(s,o) = Q(s,o) + \alpha \cdot [r + \gamma \cdot Q(s',o) - Q(s,o)] \tag{5}$$

if the option does not terminate at s' and

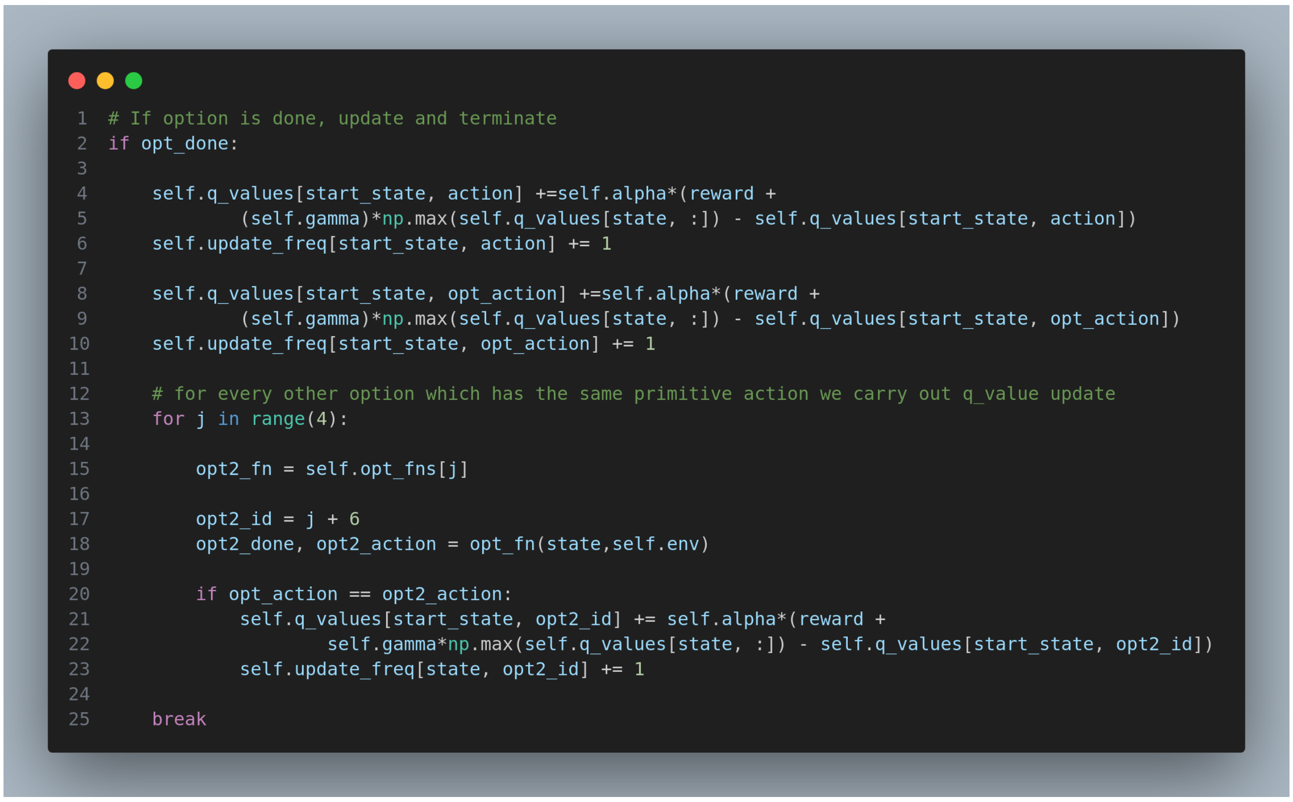
$$Q'(s,o) = Q(s,o) + \alpha \cdot [r + \gamma \cdot \max_{o'} Q(s',o') - Q(s,o)] \tag{6}$$

if the option terminates at s'.
The code blocks for updates in case of option selection are shown below:



```python
# now carry out the normal update of q_values for option as well as the corresponding primitive action
self.q_values[start_state, action] +=self.alpha*(reward +
            (self.gamma)*(self.q_values[state, action]) - self.q_values[start_state, action])
self.update_freq[start_state, action] += 1

self.q_values[start_state, opt_action] +=self.alpha*(reward +
            (self.gamma)*(self.q_values[state, opt_action]) - self.q_values[start_state, opt_action])
self.update_freq[start_state, opt_action] += 1

# for normal update (non-terminating) check options which yield same primitive actions and carry out update
for j in range(4):

    opt2_fn = self.opt_fns[j]

    opt2_id = j + 6

    opt2_done, opt2_action = opt_fn(state,self.env)

    if opt_action == opt2_action:
        self.q_values[start_state, opt2_id] += self.alpha*(reward +
                    self.gamma*(self.q_values[state, opt2_id]) - self.q_values[start_state, opt2_id])
        self.update_freq[state, opt2_id] += 1
```

Figure 18: Intra-Option update for options (Non-Terminating)



```python
# If option is done, update and terminate
if opt_done:

    self.q_values[start_state, action] +=self.alpha*(reward +
            (self.gamma)*np.max(self.q_values[state, :]) - self.q_values[start_state, action])
    self.update_freq[start_state, action] += 1

    self.q_values[start_state, opt_action] +=self.alpha*(reward +
            (self.gamma)*np.max(self.q_values[state, :]) - self.q_values[start_state, opt_action])
    self.update_freq[start_state, opt_action] += 1

    # for every other option which has the same primitive action we carry out q_value update
    for j in range(4):

        opt2_fn = self.opt_fns[j]

        opt2_id = j + 6
        opt2_done, opt2_action = opt_fn(state,self.env)

        if opt_action == opt2_action:
            self.q_values[start_state, opt2_id] += self.alpha*(reward +
                    self.gamma*np.max(self.q_values[state, :]) - self.q_values[start_state, opt2_id])
            self.update_freq[state, opt2_id] += 1

    break
```

Figure 19: Intra-Option update for options (Terminating)

4.2 Hyper-parameter Tuning

We carry out hyper-parameter tuning on the hyper-parameters like α (learning rate), and ϵ the control parameter for the ϵ-greedy action selection rule. This metric used to evaluate the performance of a combination of hyper-parameters is the average reward. We test the following combinations of hyper parameters (a total of 20 configurations):

- α: 0.5, 0.1, 0.05, 0.01
- ϵ: 0.1, 0.05, 0.01, 0.005, 0.001

The log files generated while carrying out this training are saved and present in the code folder.

4.3 Results

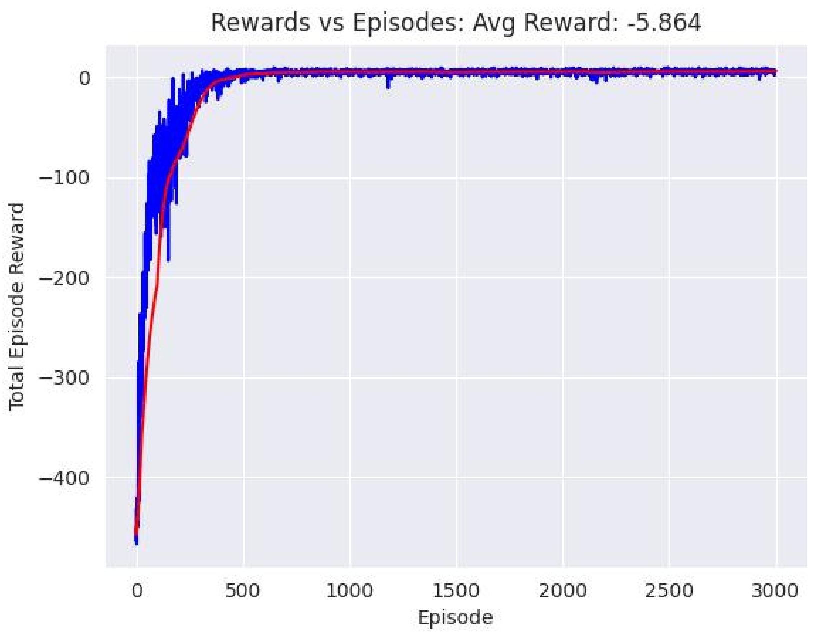


Figure 20: Intra-Option: Average reward plot

- From the average reward plot (averaged over 10 runs), we observe that the rewards per episode increase as expected. The red line indicates the running average over the past 100 episodes, which increases as it should. The average reward is also much higher than the SMDP case.

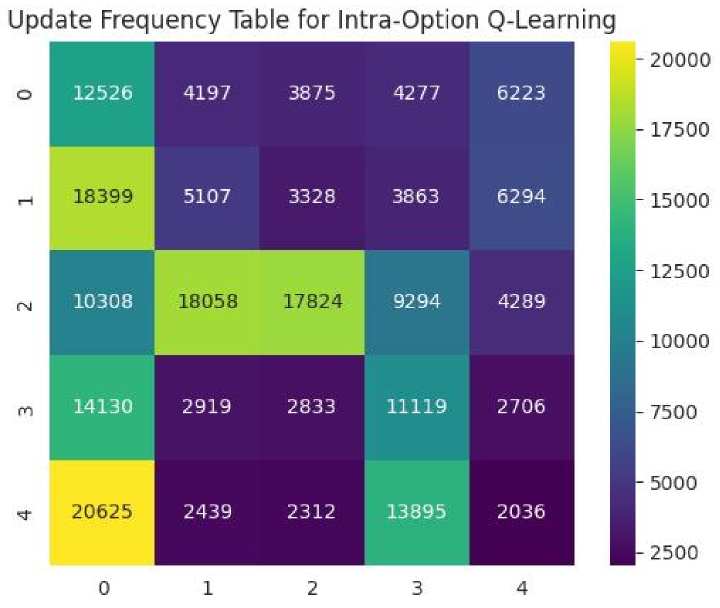
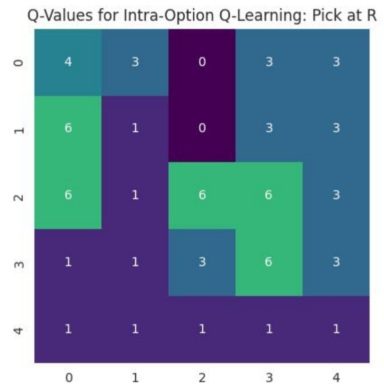
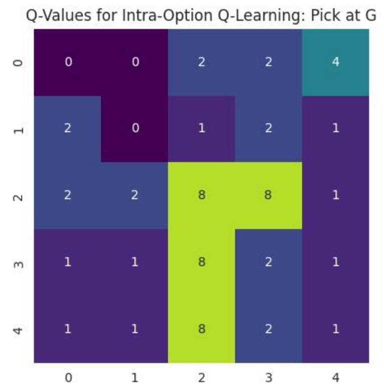


Figure 21: Intra-Option: Update frequency

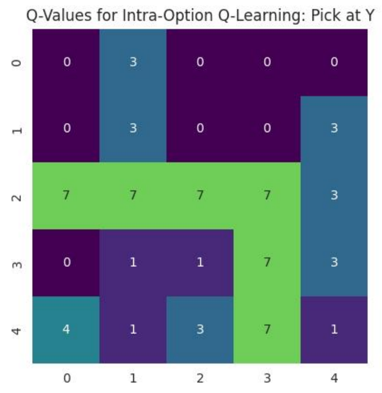
- From the update frequency plot, we observe that the agent updates all grid cells rather uniformly. This is in stark contrast with the case of SMDP Q-Learning where options were dominant. This difference could be attributed to the fact that in Intra-Option Q-Learning, all compatible actions/options are updated, leading to a more uniform distribution of updates.



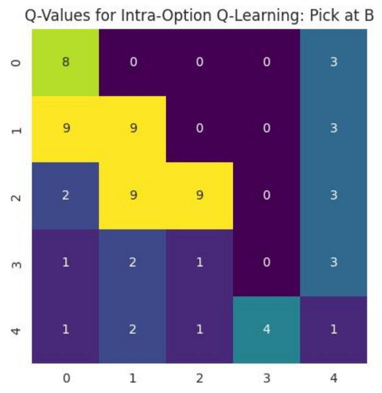
(a) Pick at location R



(b) Pick at location G



(c) Pick at location Y

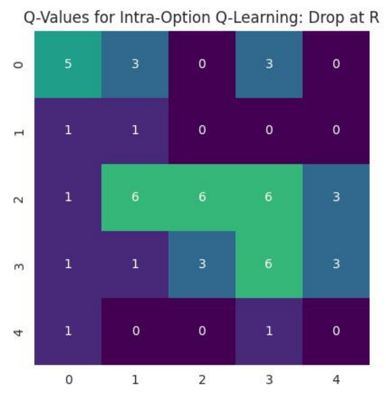


(d) Pick at location B

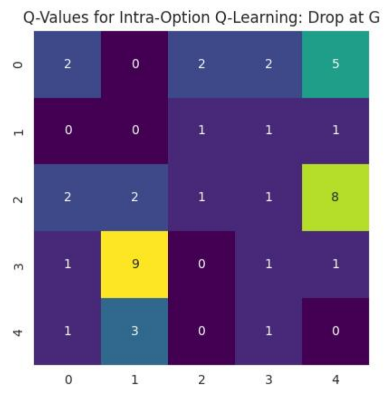
Figure 22: Intra-Option Q-learning: Actions/Options with largest Q-value for each grid cell

The four heatmaps displayed above indicate the preferred action for each grid cell based on the Q-value for the cases where the taxi is yet to pick up the passenger. These heatmaps provide insight into the policy learned by the agent.

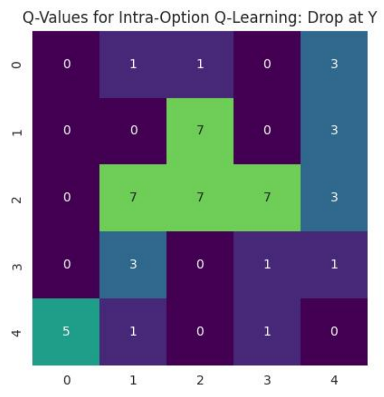
- It can be seen that the taxi **picks up the passenger at the specified location in all cases**, as action 4 (pick-up) is performed at the passenger locations.
- In other grid cells, the learned policy is such that the **taxi moves closer to the passenger location**.
- However, the use of options directly is not as prevalent as in the SMDP case. This could be attributed to the fact that **all compatible actions are updated here**. Though options may not be used directly, compatible actions also lead to optimal results.



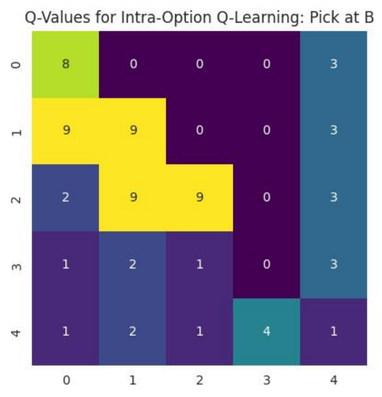
(a) Pick at location R



(b) Pick at location G



(c) Pick at location Y



(d) Pick at location B

Figure 23: Intra-Option Q-learning: Actions/Options with largest Q-value for each grid cell

The four heatmaps displayed above indicate the preferred action for each grid cell based on the Q-value for the cases where the taxi has picked up the passenger and is ready to drop the passenger off at the destination. These heatmaps provide insight into the policy learned by the agent.

- It can be seen that the taxi **drops off the passenger at the specified location in all cases**, as action 5 (drop-off) is performed at the passenger locations.
- We also note that the **performance in other grid cells is as good as the performance in the pick-up plots**. This is in contrast to the SMDP case where the performance was much poorer compared to the pick-up case. This alludes to the fact that intra-option q-learning exhibits better performance thanks to the higher update frequency.

5 Comparing SMDP and Intra-Option Q-Learning

We first compare the rewards obtained in the case of both SMDP and Intra-Option Q-Learning. The plots for the rewards are taken with a moving average across a window of 10 episodes. This smoothens the irregularities of the plot enough to be able to make good observations about the general trends.

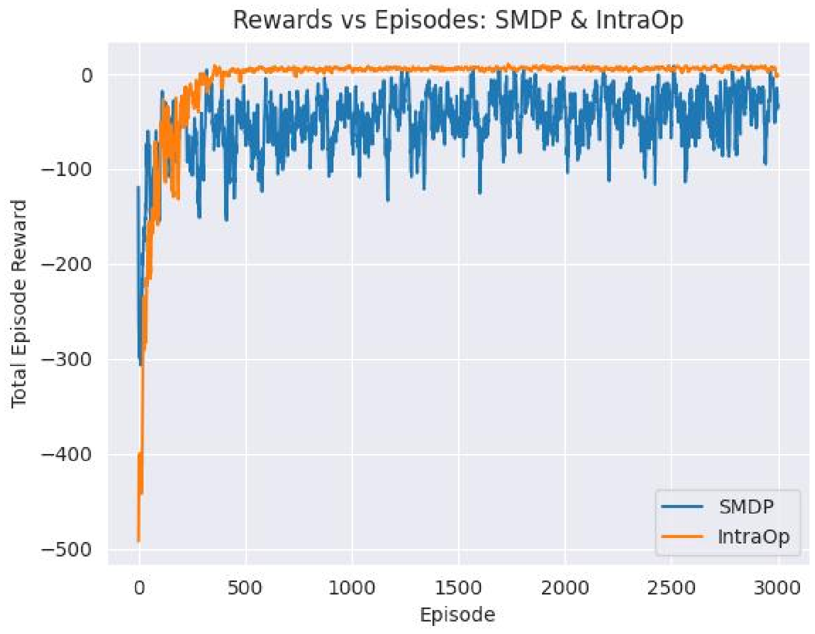


Figure 24: Reward Curve: SMDP vs Intra-Option Q-Learning

As we can see:

- Intra-Option Q-Learning significantly outperforms even the best-tuned version of SMDP Q-Learning.
- This is attributed to the frequent intra-option updates in Intra-Option Q-Learning, where both primitive actions and options' Q-values are updated at every step.
- The frequent updates allow for greater flexibility in learning and faster convergence.
- Intra-Option Q-Learning can learn sub-policies with different durations, enhancing its flexibility in the decision-making process.

Now we can compare the tables of update frequencies.

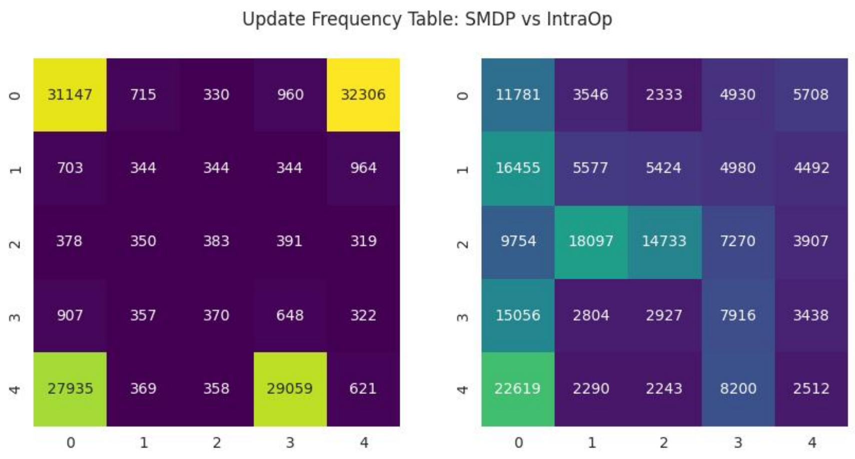


Figure 25: Update Frequency: SMDP vs Intra-Option Q-Learning

As we can see:

- As we can clearly see, there is a heavy tendency for options to be directly selected in the case of SMDP Q-Learning.
- This is because the option update equations do not change the Q-values of the primitive actions, which causes them to be neglected as a result.
- The option definition is thus very important for SMDP.

6 Alternate Options

6.1 Defining Alternate Options

We have been asked to define an alternate set of options mutually exclusive of the given set of options and compare the performance of both sets. Towards this, we define the options as described below:

- **Keep South**
  The policy in this case is to keep moving south till termination is reached.
  Termination/Initiation set:
  - All cells in row 4

  Note that this definition is possible as we do not have any horizontal walls to block southward movement.

- **Keep North**
  The policy in this case is to keep moving north till termination is reached.
  Termination/Initiation set:
  - All cells in row 0

  This definition is possible as there are no horizontal walls to block northward movement.
- **Keep East**
  The policy in this case is to keep moving east till termination is reached.
  Termination/Initiation set:
  - All cells in column 4
  - Cells (0, 1), (1, 1), (3, 0), (3, 2), (4, 0), (4, 2)

  Note that the set is more involved here due to the presence of vertical walls.
- **Keep West**
  The policy in this case is to keep moving west till termination is reached.
  Termination/Initiation set:
  - All cells in column 0
  - Cells (0, 2), (1, 2), (3, 1), (3, 3), (4, 1), (4, 3)

6.2 Comparing Results for Intra-Option Q-Learning

We carry out hyperparameter tuning on the hyperparameters like α (learning rate), and ϵ using average reward. We test the following combinations of hyperparameters (total 12 configurations): α: [0.5, 0.1, 0.05, 0.01] and ϵ: [0.1, 0.01, 0.001] The average reward curve obtained on the best hyperparameters by averaging rewards over 10 runs is plotted below:

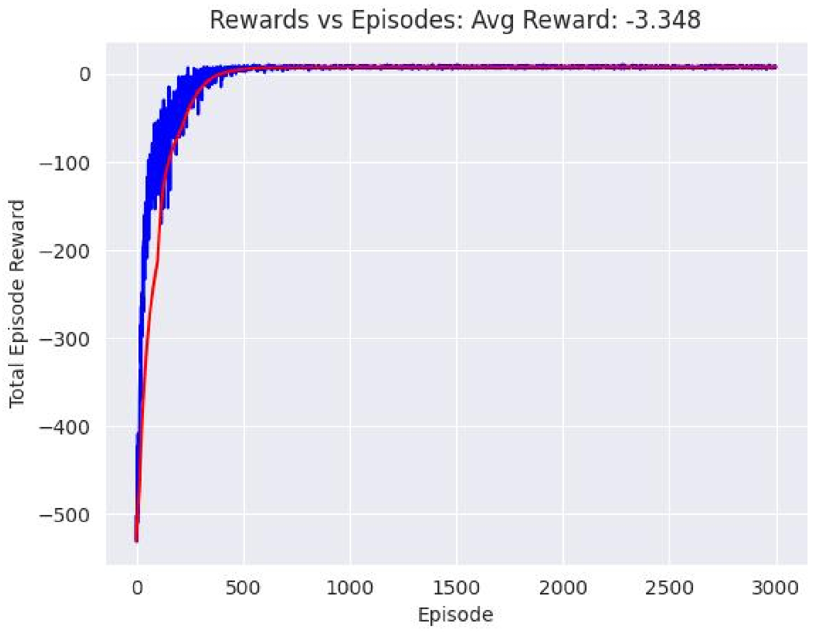


Figure 26: Intra-Option Reward Curve: New Options

We can also compare the plots for the old and new rewards as given below.

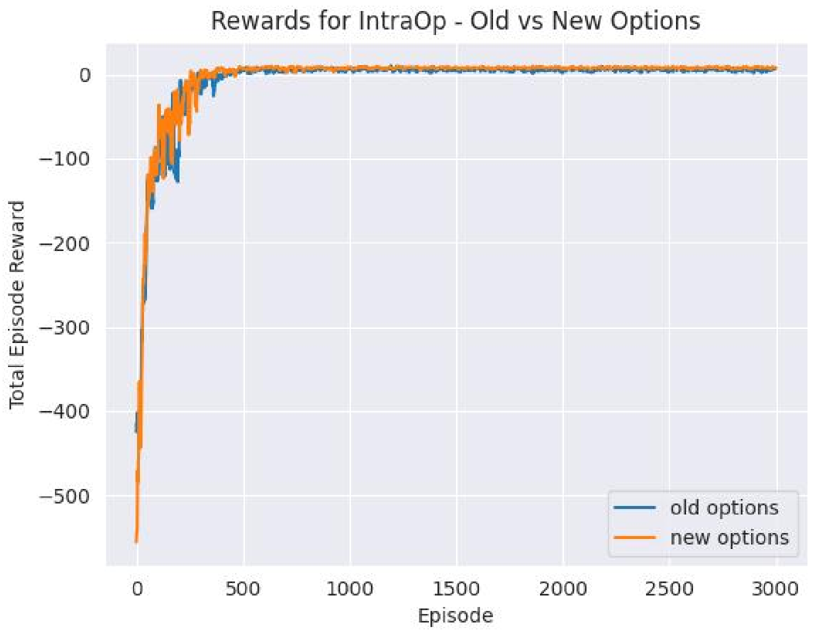


Figure 27: Intra Option Rewards - Old vs New Options

If we notice it carefully, we can see that the new options seem to be performing slightly better.

6.3 Comparing Results for SMDP Q-Learning

We carry out hyperparameter tuning on the hyperparameter α (learning rate) We test the following combinations of hyperparameters (total 4 configurations): α: [0.5, 0.1, 0.05, 0.01] The average reward curve obtained on the best hyperparameters by averaging rewards over 10 runs is plotted below:

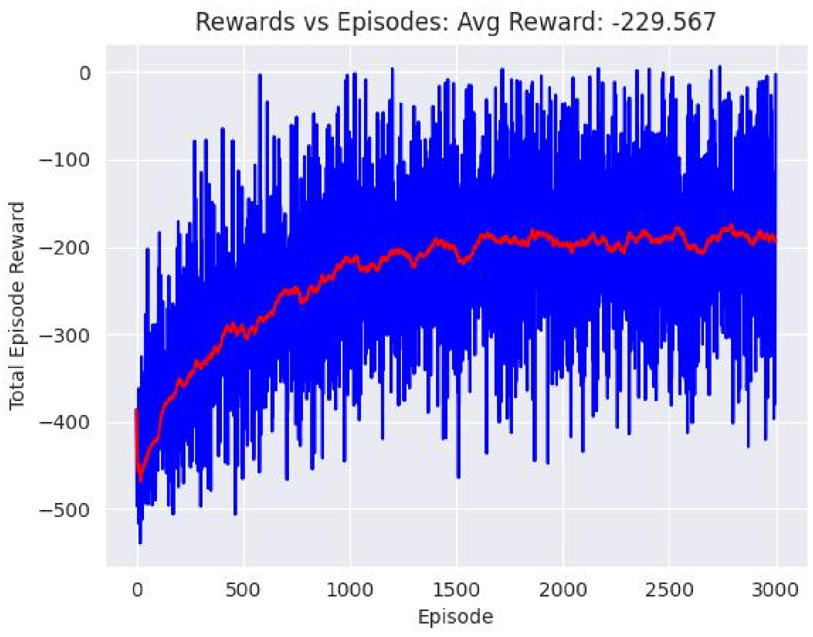


Figure 28: SMDP Reward Curve: New Options

We can also compare the plots for the old and new rewards as given below.

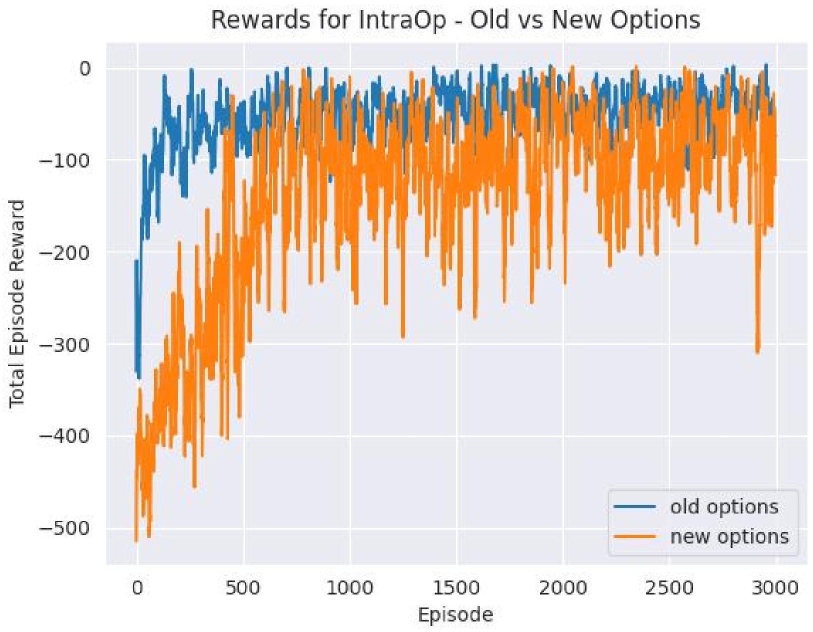


Figure 29: SMDP Rewards - Old vs New Options

As we can see:

- As we can see, SMDP performs very poorly for the newly defined options.
- This can be attributed to the fact that most of the value updates take place for options in SMDP.
- The new options being ill-defined for the problem at hand thus greatly affects the performance of SMDP, almost completely preventing good convergence.

7 Conclusion

In this assignment, we have looked at two popular option-based methods: SMDP Q-learning and Intra-Option Q-learning. We extensively analyzed their performance on the 'Taxi-v3' environment, empirically arriving at the following conclusions:

- Both SMDP and Intra-Option Q-learning improve over classical Q-learning by making use of options.
- Empirically, Intra-Option Q-learning outperforms SMDP Q-learning. This could be attributed to the higher update frequency.

- Due to the lower update frequency of primitive actions, SMDP relies heavily on the availability of precise and goal-oriented options, as can be seen from the effect of using less precise options.
- Intra-Option Q-learning is relatively more robust to the quality of options, as can be seen from the fact that the performance practically remains unchanged even when lower quality options are used.

References

[1] GoCoder. Rl tutorial with openai gym. `https://www.gocoder.one/blog/rl-tutorial-with-openai-gym/`. Accessed: 2024-04-19.

[2] Matthew Hoffman and Nando de Freitas. Inference strategies for solving semi-markov decision processes. In *Decision Theory Models for Applications in Artificial Intelligence: Concepts and Solutions*, pages 82–96. IGI Global, 2012.

[3] Martin Stolle and Doina Precup. Learning options in reinforcement learning. In *Abstraction, Reformulation, and Approximation: 5th International Symposium, SARA 2002 Kananaskis, Alberta, Canada August 2–4, 2002 Proceedings 5*, pages 212–223. Springer, 2002.

[4] Richard S Sutton, Doina Precup, and Satinder Singh. Intra-option learning about temporally abstract actions. In *ICML*, volume 98, pages 556–564, 1998.