CS6700: Reinforcement Learning

Programming Assignment - 3 Report

Siddhagavali Shital [ME20B166]

Srinivas Chowdary [ME20B174]

GitHub Repository Link

April 20, 2024

Table of Contents

1 Algorithm 1 - SMDP Q-learning

SMDP Q-learning is a reinforcement learning algorithm designed for solving problems with semi-Markov decision processes (SMDPs). It extends traditional Q-learning to handle variable-length sequences of actions, allowing for more flexible and efficient decision-making in dynamic environments. 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

1.1 Code snippets and Outline :

1.1.1 Epsilon Greedy:

Defining epsilon-greedy policy function for selecting actions, balancing the exploration and exploitation by either selecting the action with the highest Q-value based on the epsilon parameter or choosing a random action

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

Listing 1: Epsilon Greedy policy

1.1.2 Original Option Policy:

The ‘option policy‘ function encapsulates strategies that direct the taxi towards predefined corner locations in the grid. The policy decides actions based on whether the taxi is already at a target location; if so, it triggers passenger pick-up or drop-off if applicable. Otherwise, the taxi is directed using an epsilon-greedy policy to continue towards the destination. This reflects a focused strategy to reduce travel time by heading straight to specific points of interest, namely the locations where passengers are likely to be or want to go, thereby directly maximizing reward potential.

```
num_options = 4
LOCATIONS = {0:[0,0], 1:[0,4], 2:[4,0], 3:[4,3]}

def option_policy(env, state, q_values, action, epsilon=0.1, locations=LOCATIONS):
    option_done = False
    taxi_x, taxi_y, passenger, dropoff = env.decode(state)

    if (taxi_x == locations[action][0] and taxi_y == locations[action][1]):
        option_done = True
        if passenger == action:
            option_action = 4
        elif dropoff == action:
            option_action = 5
        else:
            option_action = 1 if (action in [0, 1]) else 0
    else:
```

```python
        option_action = egreedy_policy(q_values[action], 5 * taxi_x + taxi_y, epsilon=
            epsilon)

    return [option_action, option_done]
```

Listing 2: Original Option Policy

1.1.3 SMDP Q-learning:

Provided code implementing the SMDP (Semi-Markov Decision Process) learning algorithm. It iterates through a specified number of episodes, during each of which it resets the environment and executes actions based on epsilon-greedy policy. The algorithm learns a set of options, represented by Q-values, and updates them based on the rewards received. This process continues until the completion of each episode, with the average reward calculated at the end. The purpose of this algorithm is to train an agent to make sequential decisions in a semi-Markovian environment, optimizing its actions to maximize the cumulative reward over time.

```python
    def SMDP_learning(gamma=0.9, learning_rate=0.1, epsilon_decay=0.99, seed=0,RENDER=
        False):
    main_epsilon = 0.5
    min_epsilon = 0.01
    count = 0
    q_values_SMDP = np.zeros((num_passengers * num_dropoffs, num_options))
    updates_SMDP = np.zeros((num_passengers * num_dropoffs, num_options))
    option_Q_values = {i: np.zeros((state_space // 20, action_space - 2)) for i in range
        (num_options)}
    option_eps = {i: 0.01 for i in range(num_options)}
    smdp_rewards = []
    for i in tqdm(range(num_episodes)):
        state = env.reset()
        env.seed(seed)
        done = False
        total_reward = 0

        while not done:
            taxi_x, taxi_y, passenger, dropoff = env.decode(state)
            sub_state = num_dropoffs * passenger + dropoff
            option = egreedy_policy(q_values_SMDP, sub_state, epsilon=main_epsilon)
            main_epsilon = max(min_epsilon, epsilon_decay * main_epsilon)
            reward_sum = 0
            option_done = False
            num_moves = 0
            prev_state = state

            taxi_x, taxi_y, passenger, dropoff = env.decode(state)
            option_done = False

            while not option_done and not done:
              option_action, option_done = option_policy(env, state, option_Q_values,
                  option, option_eps[option])

              taxi_x, taxi_y, passenger, dropoff = env.decode(state)

              next_state, reward, done,_ = env.step(option_action)
              taxi_x_next, taxi_y_next, passenger, dropoff = env.decode(next_state)
              reward_sum = gamma * reward_sum + reward
              num_moves += 1
              total_reward += reward
              if RENDER:
                  clear_output(wait=True)
                  print(env.render())
                  time.sleep(T)

              option_eps[option] = max(min_epsilon, epsilon_decay * option_eps[option])

              reward_surrogate = reward if not option_done else 20
              if option_action < 4:
                  option_Q_values[option][5 * taxi_x + taxi_y, option_action] += learning_rate * (
                              reward_surrogate + gamma * np.max(option_Q_values[option][5 * taxi_x_next + taxi_y_next, :]) -
                              option_Q_values[option][5 * taxi_x + taxi_y, option_action])
              state = next_state


            _, _, prev_passenger, prev_dropoff = env.decode(prev_state)
            prev_sub_state = num_dropoffs * prev_passenger + prev_dropoff

            _, _, next_passenger, next_dropoff = env.decode(state)
            sub_state = num_dropoffs * next_passenger + next_dropoff

            q_values_SMDP[prev_sub_state, option] += learning_rate * (
                        reward_sum + (gamma ** num_moves) * np.max(q_values_SMDP[sub_state, :]) -
                        q_values_SMDP[prev_sub_state, option])
            updates_SMDP[prev_sub_state, option] += 1
        smdp_rewards.append(total_reward)
        taxi_x, taxi_y, passenger, dropoff = env.decode(state)
        if passenger == dropoff:
            count += 1
    return np.sum(smdp_rewards)/num_episodes, q_values_SMDP, updates_SMDP, option_Q_values, smdp_rewards, count
```

Listing 3: SMDP Q-Learning

1.2 Visulization:

1.2.1 SMDP Q-Learning Reward Plot:

The reward curve for SMDP Q-Learning represents the average across five different seeds over 10,000 episodes. 1:

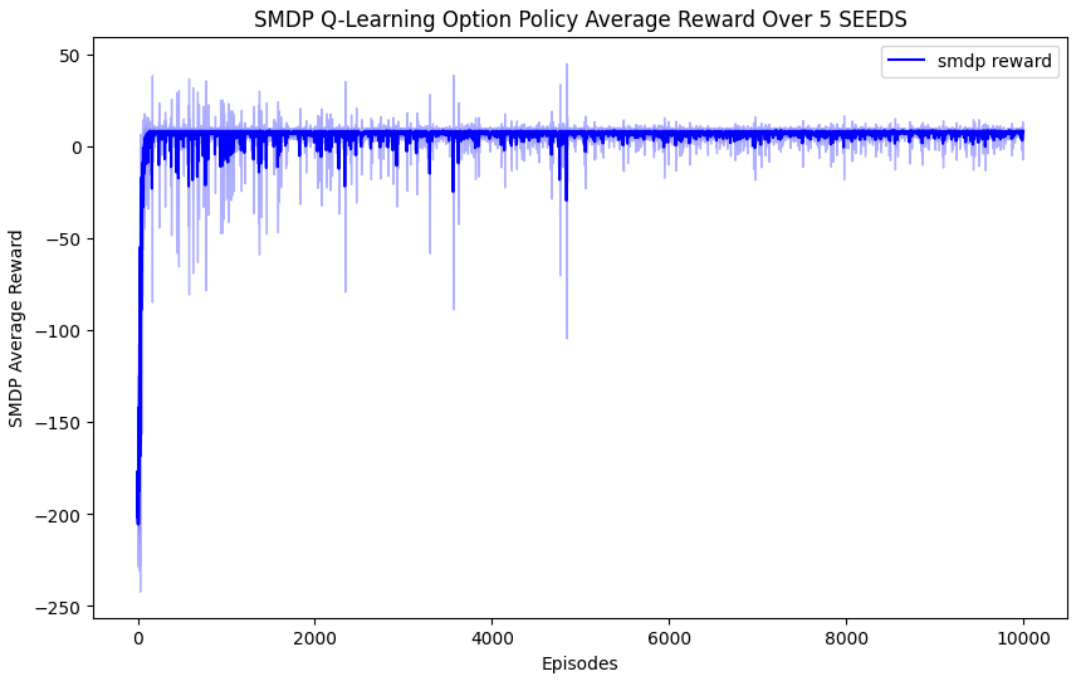

Figure 1: SMDP Reward Curve

- **Early Learning and Variability:** The sharp increase in rewards at the beginning followed by high variability is indicative of the algorithm rapidly exploring and learning from a diverse set of experiences. This initial phase is where the agent encounters a wide variety of states and updates its Q-values significantly based on the received rewards.

- **Smoothing Over Time:** As learning progresses, the reward curve smooths out, reflecting the agent's growing competence in the environment. The SMDP's ability to encapsulate sequences of actions into options allows the agent to make more informed and cohesive decisions, leading to more consistent performance.
- **Convergence of Policy:** The leveling of the curve suggests that the agent is converging on a stable policy. The SMDP algorithm, through its hierarchical structure, enables the agent to more effectively learn policies that are robust across the state space, accounting for the temporal dependencies in sequences of actions.
- **Diminished Exploration:** Towards the end of the training episodes, the reduced variance in the reward curve is likely due to a decrease in exploration as the ϵ-greedy policy shifts more towards exploitation of the learned values, and the algorithm takes advantage of the accumulated knowledge about the environment's dynamics.

1.2.2 SMDP Q-table:

Visualizing the maximum Q-values for each state-action pair in an SMDP Q-table using a heatmap, with labels indicating the optimal action to take from each state. 2:

- **Direct Routes to Goals:** The policy prioritizes direct options to passenger pick-up or drop-off points, aligning with the most significant immediate reward for reducing steps per episode.
- **Avoidance of Redundant Actions:** The heatmap shows no policy to remain in the same cell when the passenger is already there, suggesting learned efficiency by avoiding redundant pick-up actions.
- **Temporal Abstraction:** By utilizing options in the SMDP framework, the agent can plan over extended time horizons, enabling it to learn policies that consider the future consequences of current actions, such as the benefits of driving towards a goal location.
- **Discount Factor Influence:** The chosen discount factor (γ) helps the algorithm prioritize long-term rewards, which guides the policy towards sequences of actions that result in the passenger being picked up and dropped off efficiently over immediate but less optimal rewards.

1.2.3 SMDP Learned Policy:

Generated the subplots, each displaying the learned policy for different SMDP options using arrows to represent actions, arranged in a 2x2 grid. 3:

- **Directional Guidance:** The heatmaps display coherent directional policies to guide the taxi around the grid, indicative of the algorithm learning to navigate effectively.

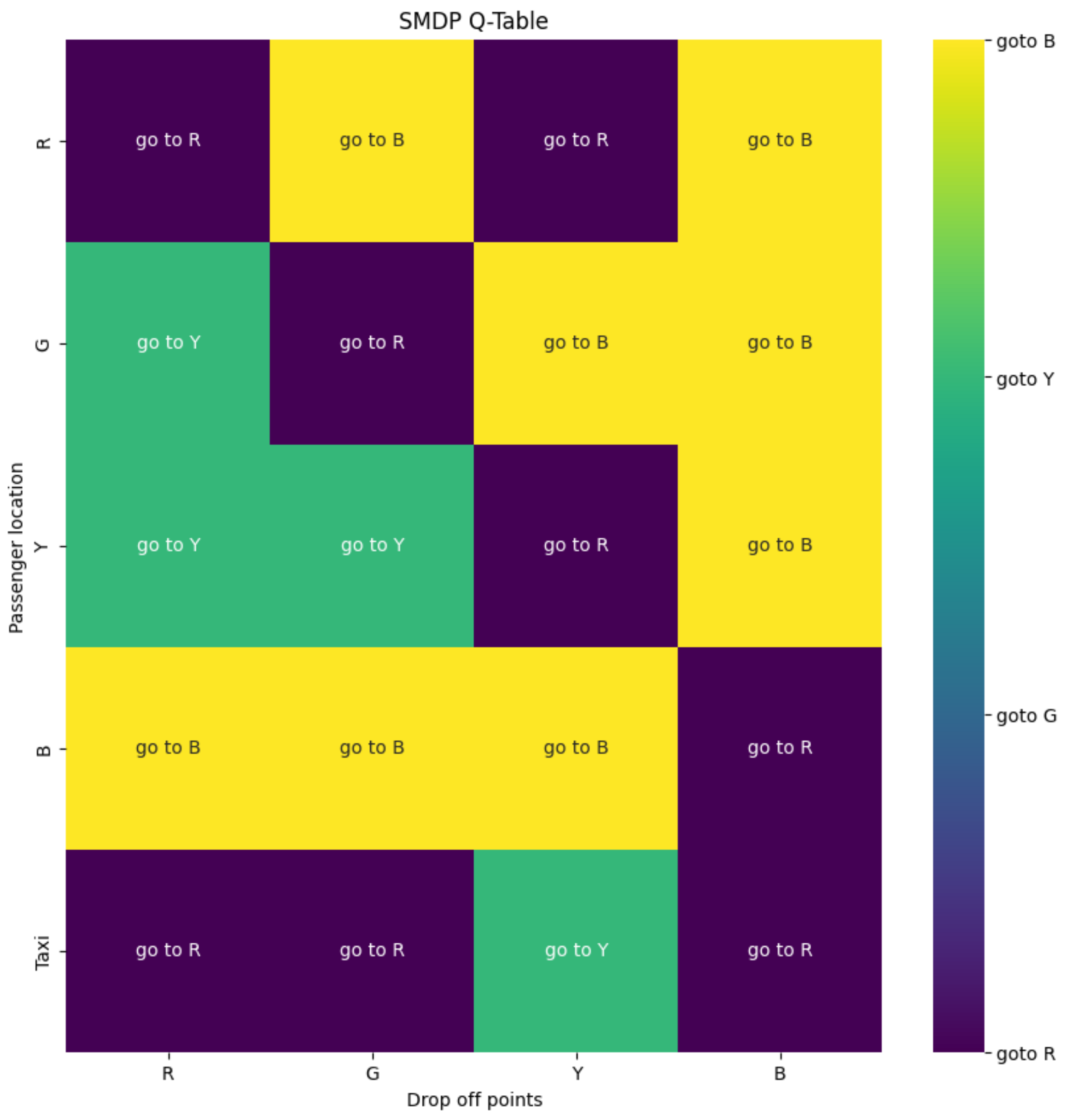

Figure 2: SMDP Q-Table Heatmap

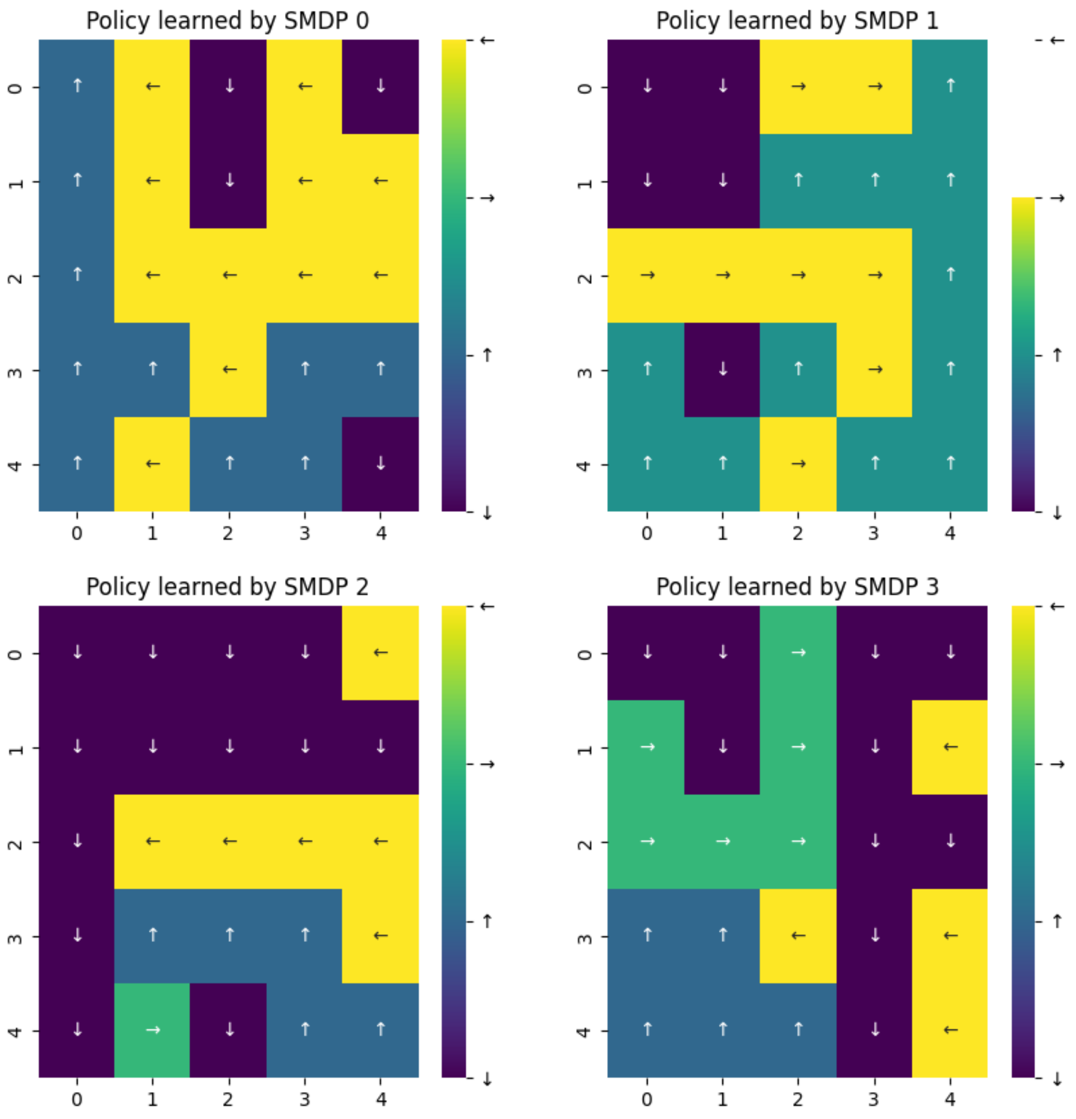

Figure 3: SMDP Policies learnt

- **Strategic Movement:** Different patterns across the options suggest that the algorithm develops specialized strategies for each option to move towards specific goals.
- **Consistency Across Options:** The presence of clear paths toward goals in each SMDP suggests that the algorithm consistently learns to optimize routes from various starting points.
- **Adaptation to Environment Layout:** The varied directions in different sectors of the grid reflect the algorithm's adaptation to the environment's layout, learning to circumnavigate obstacles effectively.

2 Algorithm 2 - Intra option Q learning

2.1 Intra-Option Q Learning:

Intra-option Q-learning is an extension of Q-learning specifically designed to handle hierarchical reinforcement learning tasks where actions can have a longer temporal scope, called options. Options represent temporally extended actions that are composed of primitive actions.

The basic idea behind intra-option Q-learning is to learn a Q-value function not just for primitive actions, but also for options. This allows the agent to decide both which primitive action to take and which option to execute. The Q-value function is updated using the Bellman equation, taking into account the Q-values of subsequent options or primitive actions.

$$Q(s, o) = (1 - \alpha) \cdot Q(s, o) + \alpha \cdot (r + \gamma \cdot Q(s', o'))$$

This equation essentially updates the Q-value of the current option o based on the immediate reward and the discounted value of the next option or primitive action.

The option policy and epsilon greedy functions remain are the same as used for SMDP as mentioned above.

Code for IOQL:

```
for i in range(episodes):
      state = env.reset()
      done = False
      t=0
      while not done:
          _,_,passenger,dropoffs = env.decode(state)
          subState = num_dropoffs*passenger+dropoffs
          action = egreedy_policy(q_values_IOQL, subState, epsilon=eps_main)
          eps_main = max(eps_min,eps_main*eps_decay)
          if RENDER:
            clear_output(wait=True)
            print(env.render())
            time.sleep(T)

          option = action
          optdone = False
          prev = state
          while not optdone and not done:
              optact,optdone = option_policy(env,state,Qopt,option,eps[option])
              next_state, reward, done,_ = env.step(optact)

              t+=reward

              [taxi_x,taxi_y,prev_passenger, prev_dropoff]=  list(env.decode(state))
              prev_sub_state = num_dropoffs * prev_passenger + prev_dropoff
              [taxi_x1,taxi_y1,next_passenger, next_dropoff]=  list(env.decode(
                  next_state))
              sub_state = num_dropoffs * next_passenger + next_dropoff

              eps[option] = max(eps_min,eps_decay*eps[option])
              reward_surr = reward
              if optdone:
                reward_surr = 20
              if optact<4:
                Qopt[option][5*taxi_x+taxi_y, optact] = Qopt[option][5*taxi_x+taxi_y,
                    optact] + alpha*(reward_surr + gamma*np.max(Qopt[option][5*taxi_x1+
                    taxi_y1, :]) - Qopt[option][5*taxi_x+taxi_y, optact])


              for o in range(num_options):
                optact_o,optdone_o = option_policy(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[prev_sub_state, o] += alpha*(reward + gamma*np.max(
                        q_values_IOQL[sub_state, :]) - q_values_IOQL[prev_sub_state, o])
                  else:
                    q_values_IOQL[prev_sub_state, o] += alpha*(reward + gamma*
                        q_values_IOQL[sub_state, o] - q_values_IOQL[prev_sub_state, o])

                  updates_IOQL[prev_sub_state, o] += 1
              state = next_state

      rewards.append(t)
      taxi_x,taxi_y,passenger,dropoffs = env.decode(state)
```

2.2 Results

The reward curve for IOQL Q-Learning represents the average across five different seeds over 10,000 episodes

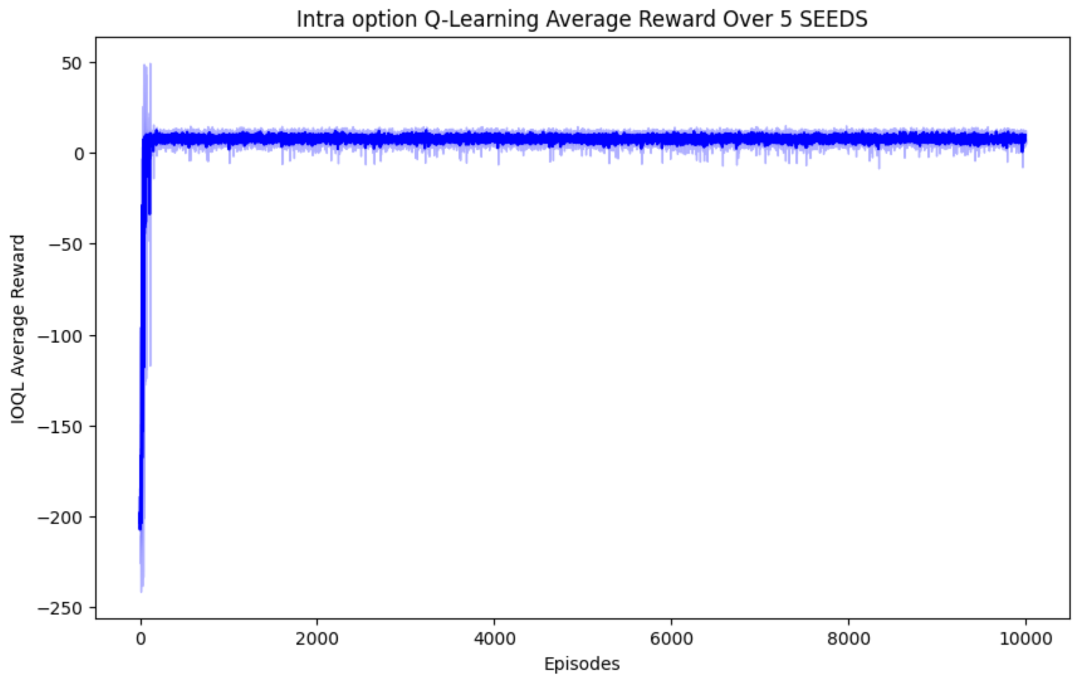

Figure 4: IOQL Q-Table Heatmap

- **Early Learning and Variability:** The initial sharp rise in rewards, followed by high variability, indicates rapid exploration and learning from diverse experiences. During this phase, the agent encounters a wide range of states and updates its Q-values significantly based on received rewards.
- **Smoothing Over Time:** As learning progresses, the reward curve becomes smoother, reflecting the agent's increasing competence in the environment. The option-based approach allows the agent to make more informed decisions, resulting in more consistent performance.
- **Convergence of Policy:** The leveling of the curve suggests that the agent is converging on a stable policy. The hierarchical structure of the algorithm enables the

agent to learn robust policies across the state space, considering temporal dependencies in action sequences.

- **Diminished Exploration:** Towards the end of training episodes, the reduced variance in the reward curve indicates decreased exploration. This is due to the ϵ-greedy policy shifting more towards exploitation of learned values, as the algorithm capitalizes on accumulated knowledge about the environment.

Visualizing the maximum Q-values for each state-action pair in an IOQL Q-table using a heatmap, with labels indicating the optimal action to take from each state. 5:

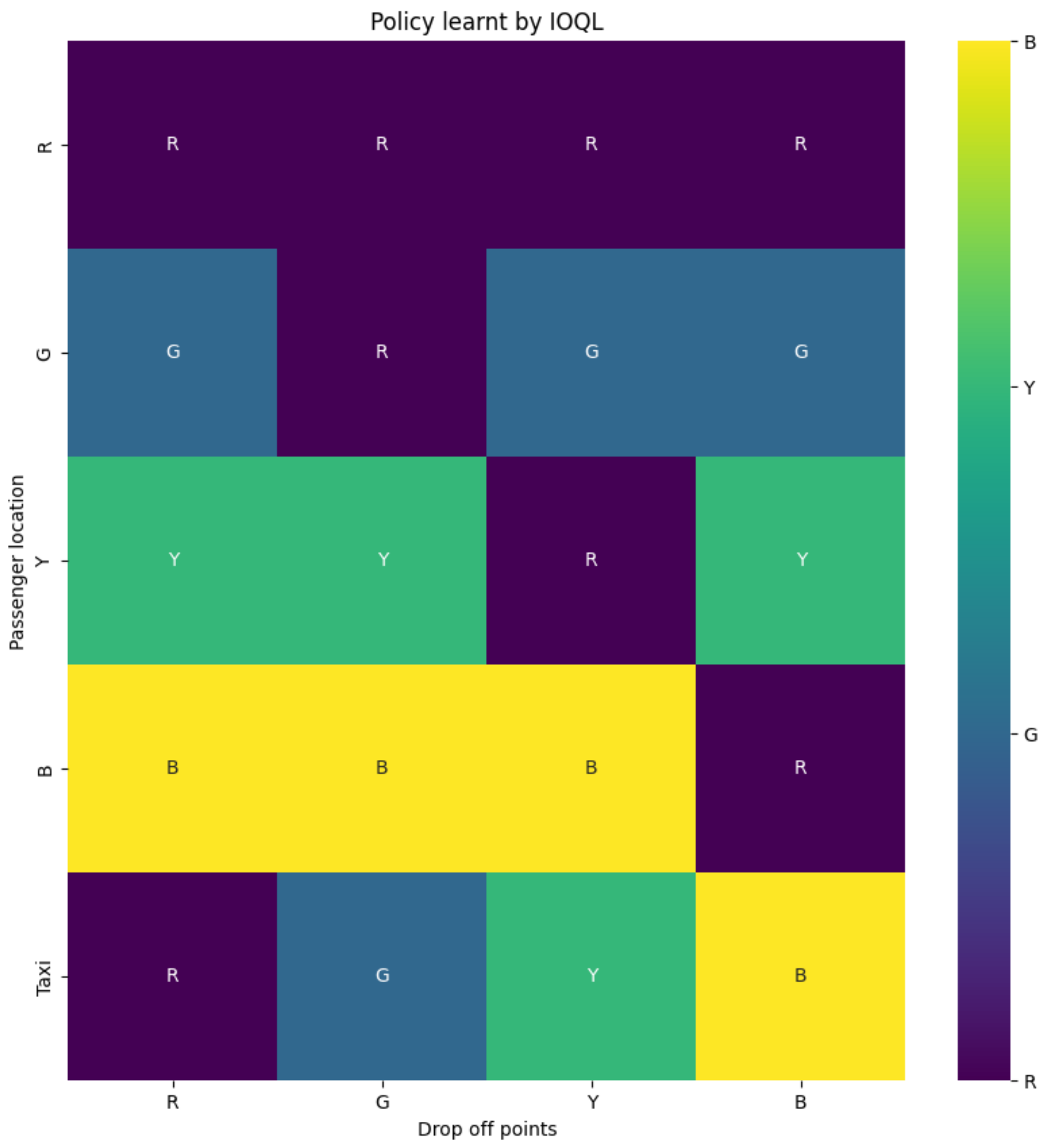

Figure 5: IOQL Q-Table Heatmap

- **Direct Routes to Goals:** The policy prioritizes direct options to passenger pick-up or drop-off points, optimizing for reducing steps per episode.
- **Avoidance of Redundant Actions:** The heatmap reveals efficient learning as there are no policies to remain in the same cell when the passenger is already there, indicating avoidance of redundant pick-up actions.

- **Temporal Abstraction:** Utilizing options in the SMDP framework allows the agent to plan over extended time horizons, facilitating the learning of policies that consider future consequences, such as driving towards goal locations.
- **Discount Factor Influence:** The chosen discount factor (γ) prioritizes long-term rewards, guiding the policy towards sequences of actions that efficiently pick up and drop off passengers over immediate but less optimal rewards.

Generated the subplots, each displaying the learned policy for different SMDP options using arrows to represent actions, arranged in a 2x2 grid. 6:

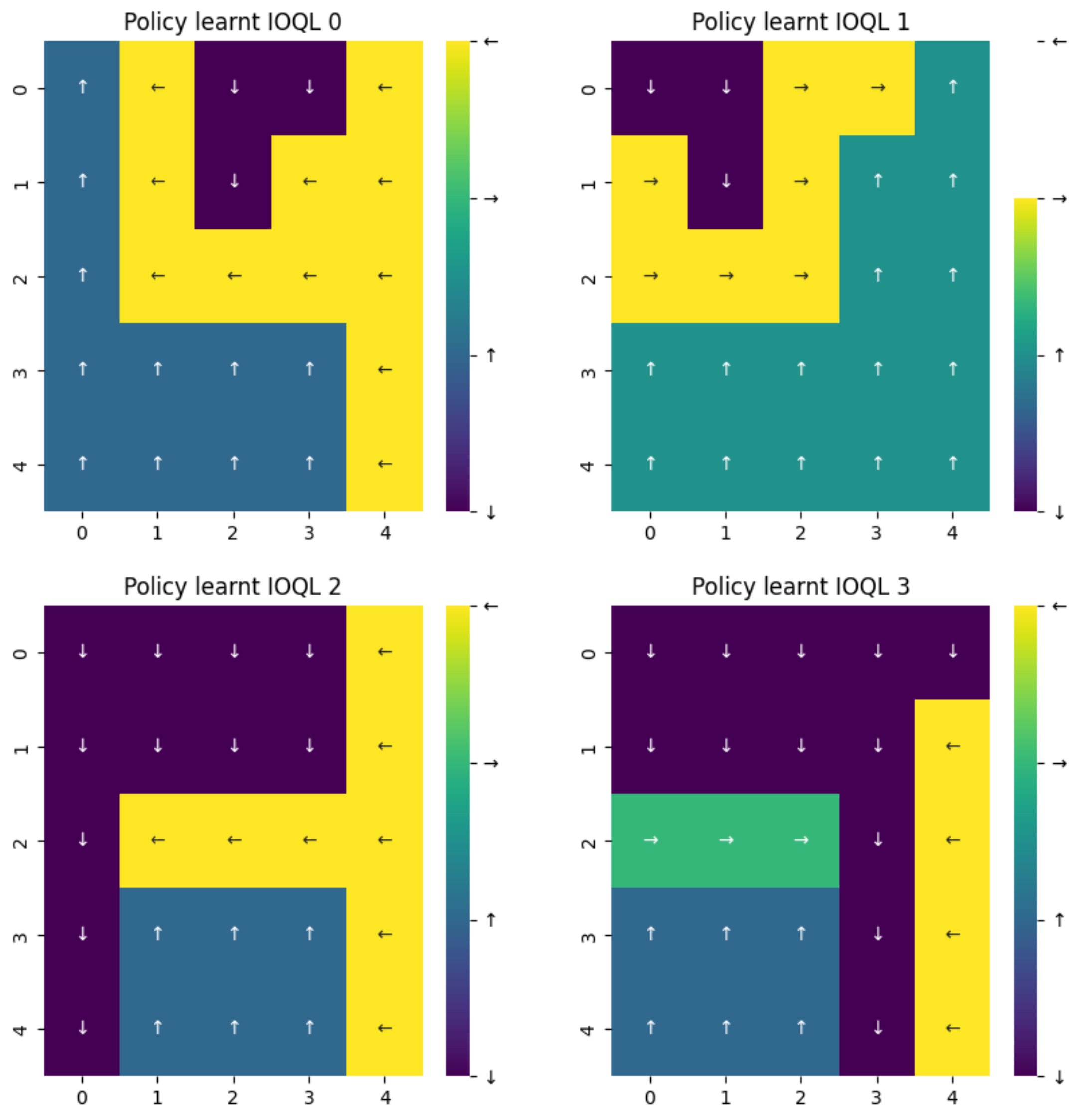

Figure 6: IOQL Policies learnt

- **Directional Guidance:** The heatmaps display coherent directional policies to guide the taxi around the grid, indicative of the algorithm learning to navigate effectively.
- **Strategic Movement:** Different patterns across the options suggest that the algorithm develops specialized strategies for each option to move towards specific goals.

- **Consistency Across Options:** The presence of clear paths toward goals in each SMDP suggests that the algorithm consistently learns to optimize routes from various starting points.
- **Adaptation to Environment Layout:** The varied directions in different sectors of the grid reflect the algorithm's adaptation to the environment's layout, learning to circumnavigate obstacles effectively.

3 Alternate set of options

3.1 Alternative Option Policy:

The ‘alternate option policy‘ introduces strategic options that are distinct from direct movement to the grid’s corners. These options aim to position the taxi in advantageous spots: navigating to the grid’s center for flexibility, patrolling the edges to potentially encounter passengers, moving to the nearest border for quick access to any corner, or staying put, potentially in a high-traffic zone. The logic behind these options is to maximize the taxi’s availability and response time to passenger pick-ups and drop-offs by being in strategic positions, rather than just focusing on the shortest path to the predefined destinations.

```python
CENTRAL_AREA = [(2, 2)]
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]
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 alternate_option_policy(env, state, q_values, option, epsilon=0.1):
    taxi_x, taxi_y, passenger, dropoff = env.decode(state)

    if option == 0 and (taxi_x, taxi_y) in CENTRAL_AREA:
        option_done = True
    elif option == 1 and (taxi_x, taxi_y) in EDGES:
        option_done = True
    elif option == 2 and (taxi_x, taxi_y) in 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 * taxi_x + taxi_y, epsilon=
            epsilon)
    else:
        if passenger == 4:
            option_action = 5
        elif (taxi_x, taxi_y) == CENTRAL_AREA[0]:
            option_action = random.choice([0, 1, 2, 3])
        else:
            option_action = 4

    return [option_action, option_done]
```

Listing 4: AlternativeOption Policy

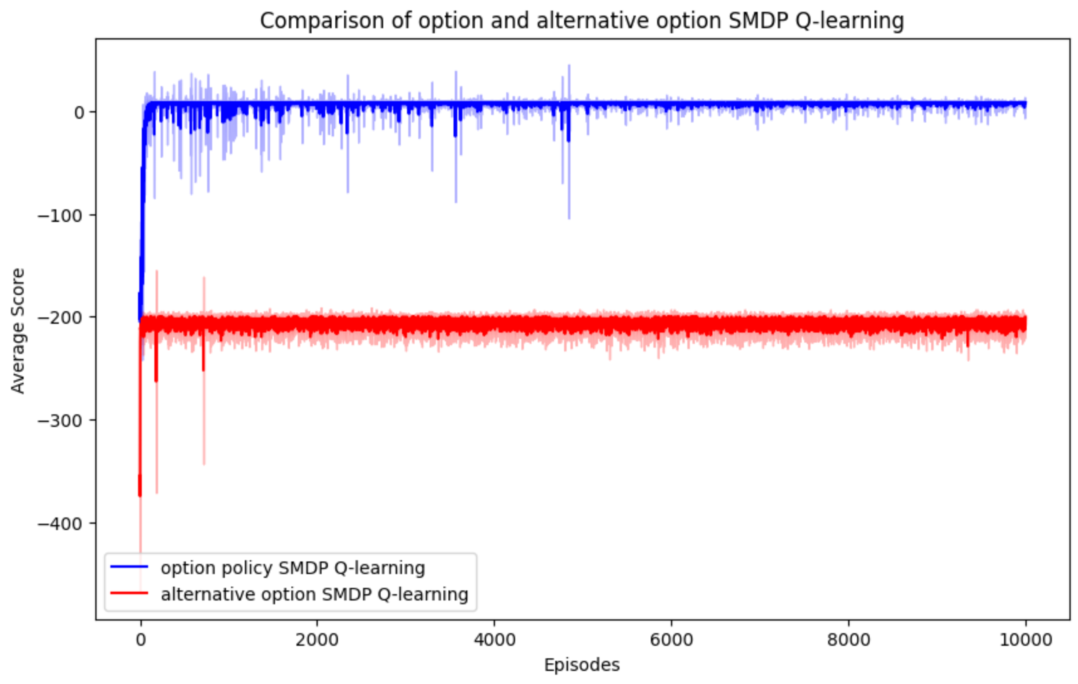

Figure 7: SMDP Original and Alternative Option Policies Comparison

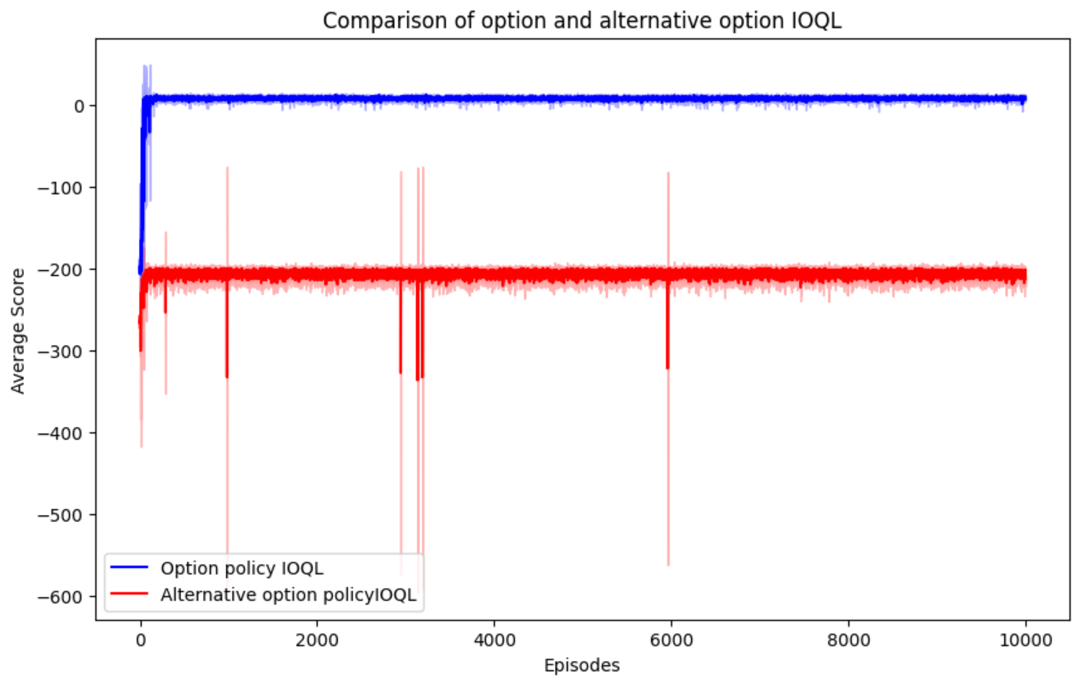

Figure 8: IOQL Original and Alternative Option Policies Comparison

Comparison of alternate options with original options.

- **Higher Efficiency of Original Options**: The original option policy shows higher average scores, indicating that direct movement to specific locations aligns well with the taxi domain's goal of passenger pick-up and drop-off.
- **Struggle with Alternative Options**: The alternative options result in lower average scores, suggesting these strategies may not be as effective in navigating the environment or may require more episodes to learn.
- **Goal-Oriented Options**: The original option policy is specifically designed with the taxi domain's objectives in mind—picking up and dropping off passengers at fixed locations—making it more effective in this context.
- **Inherent Structure Advantage**: The structured nature of the original options aligns closely with the environment's discrete states, which likely facilitates more straightforward learning and decision-making.
- **Direct Path Efficiency**: The original options create policies that focus on direct paths to the goals, reducing the number of steps and thus penalties per episode, enhancing overall performance.
- **Complexity of Alternative Strategies**: The alternative options introduce a level of complexity in the policy that may not directly translate to the primary objectives of the task, leading to less optimal performance compared to the original options.

4 Comparison of SMDP vs Intra option Q learning

Both SMDP Q-learning and intra-option Q-learning methods learn quickly and effectively. However, when comparing the reward versus episode curve 9, intra-option Q-learning (IOQL) outperforms SMDP Q-learning. It not only reaches convergence faster but also achieves a higher convergence reward.

This improvement with intra-option Q-learning can be attributed to its ability to incorporate higher-level actions, known as options, which allows for more efficient exploration and learning. By enabling the agent to plan over extended time horizons and consider sequences of actions, intra-option Q-learning facilitates more informed decision-making, resulting in faster convergence and higher rewards.

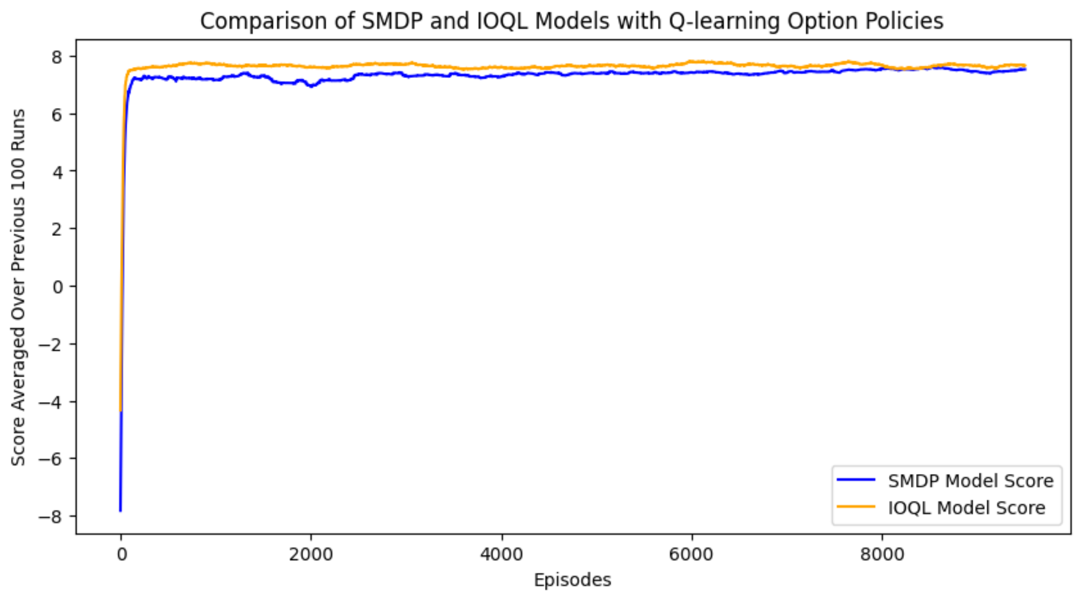

Figure 9: SMDP vs IOQL