DA6400: Introduction to Reinforcement Learning
Assignment 3

Jayagowtham J ME21B078, Lalit Jayanti ME21B096

The code for this assignment can be found at:
`https://github.com/DA6400-RL-JanMay2025/programming-assignment-03-me21b096_me21b078`

Algorithms and Training

Agent

This is the common code block for both of our SMDP and Intra-option Agents. It encapsulates initializing the agent and agent environment interaction parameters. Additionally, it also demonstrates action preference in an option and option preference at episode termination.

```python
class SMDPQLearningAgent/IntraOptionQLearningAgent:
    def __init__(self, state_space, action_space, options, seed):
        """
        The agent class which initialises the agent with state space and action space information, alongside the options that it has at its armoury and the agent seed. Also instantiates the discount factor for the problem. By default, the Q-values of each options are initialised to zero, the learning rate is set to None and the option_policy is epsilon-greedy.
        """

    def reset(self, seed=0):
        """Function which resets the agent parameters after episode ends
        """

    def update_hyperparameters(self, **kwargs):
        """This function updates hyperparameters overriding the
        default values.
        """

    def update_agent_parameters(self):
        """This function updates agent parameters to facilitate scheduling
        """

    def step(self, state, action, reward, next_state, done):
        """This function runs for every step of the episode, it is responsible
        for storing discounted rewards, Q-Table updates and
        choosing the next option, when the current option terminates
        """

    def act(self, state):
        """This function given a state acts according to the current option
        """
        return self.current_option.policy.act(state)

    def set_option(self, state):
        """At the beginning of an episode, this function chooses a
        new option based on our policy
        """
```

Options

This block of code is the implementation for options. It consists of 3 components, the first is the initiation set that determines if the given option can be initiated in a given state (implemented as a boolean array). The second component is the policy, this is essentially implemented as a map from state to actions (the details and implementation are in `scripts/policies.py`). The third component is the termination set (referred to as beta), representing the termination probability of the option in a given state.

```python
class Option:

    def __init__(self, index, initiation_set, policy, termination, label=None, seed=None):
        """
        Initialize an Option for SMDP Q-learning.

        Args:
            index (int): An index to identify and index the Qtables
            initiation_set (Any): An object that supports __getitem__ access.
                Determines whether the option can be initiated in a given state.

            policy (Callable): Policy to be used while executing option

            termination (Any): An object that supports __getitem__ access. Returns
                the probability of the option terminating at a given state.

            seed (int): Random seed for reproducibility.
        """

        self.index = index
        self.label = label

        self.initiation_set = initiation_set
        self.policy = policy
        self.termination = termination

        self.rng = np.random.default_rng(seed)

    def can_initiate(self, state):
        return self.initiation_set[state]

    def is_terminated(self, state):
        p = self.termination[state]
        return self.rng.uniform(0.0, 1.0) <= p

    def update_policy_parameters(self):
        self.policy.update()
```

Policies

The following code block illustrates our policy classes. We use ϵ-greedy policy as our global policy to choose our options. We have implemented the MoveTaxiPolicy which uses *go_to_color* directives as options. As our alternate set of options, we consider the directives of picking up and dropping off as our options.

```python
# Global Policy
class EpsilonGreedyPolicy:

    def __init__(self, options, eps_start, eps_end, eps_decay,
                 decay_type="exponential", seed: int = None):
        """ Initializes list of options, agent parameters and random number generator
        """

    def act(self, state, Qtable):
        """Select an option among all possible options in a given state using the epsilon-
    greedy strategy."""

    def update(self):
        """Update epsilon based on decay type."""

    def reset(self, eps_start: float, eps_end: float, eps_decay: float):
        """Reset the decay schedule."""

# Options policies
class MoveTaxiPolicy:

    def __init__(self, source):
        """ Compute shortest paths between destinations and each cell in the grid, alongside
      maintaining the path using Dijikstra algorithm
        """
        self.adj_list = tu.build_taxi_graph()
        self.dist, self.prev = tu.dijkstra(source, self.adj_list)

    def act(self, state):
        """ Given a state,
            - move along the shortest path to the option destination
            - at option destination,
                - if passenger not in taxi, pickup
                - else, drop
        """
        taxi_row, taxi_col, passenger_location, destination = (
            tu.decode_env_state(state)
        )

        curr_loc = (taxi_row, taxi_col)
        next_loc = self.prev[curr_loc]

        if next_loc is not None:
            move = (next_loc[0]-curr_loc[0], next_loc[1]-curr_loc[1])
            return tu.DIR_TO_ACTION[move]
        else:

            if passenger_location == tu.IN_TAXI:

                if tu.LOC_TO_COLOR[curr_loc] == destination:
                    return tu.DROPOFF
                else:
                    return tu.DROPOFF
            else:

                if tu.LOC_TO_COLOR[curr_loc] == passenger_location:
                    return tu.PICKUP
                else:
                    return tu.PICKUP


class PickUpPassenger:

    def __init__(self):
        """ Maintain directives to move to the correct location, once asked to.
        """

    def act(self, state):
        """ Given a state,
         - if passenger not in taxi, move along the shortest path to his location and at his
      location pickup
         - else, try pickup and incur a penalty
        """

class DropOffPassenger:

    def __init__(self):
        """ Maintain directives to move to the correct location, once asked to.
        """

    def act(self, state):
        """ Given a state,
         - move along the shortest path to his destination and drop-off
            - if passenger in taxi, task complete
            - else incur penalty
        """
```

SMDP Q-Learning

In this section we provide the snippet of code that performs the SMDP Q-Learning update and selects the next option when the current option terminates.

```
class SMDPQLearningAgent:

    ...

    def step(self, state, action, reward, next_state, done):

        Q = self.Qtable

        possible_options = []
        for option in self.options:
            if option.can_initiate(next_state):
                possible_options.append(option.index)

        next_Q_value = np.max(Q[next_state, possible_options])
        beta = self.current_option.termination[next_state]

        for option in self.options:
            if option.can_initiate(state) == False:
                continue
            if option.policy.act(state) != action:
                continue

            # SMDP Q-Learning Update
            curr_Q_value = Q[state, option.index]
            estimate_Q = (
                (1.0 - beta)*Q[next_state, option.index]
                + beta*next_Q_value
            )

            Q[state, option.index] = (
                curr_Q_value + self.LR *
                (reward + self.GAMMA*estimate_Q - curr_Q_value)
            )

        # Select next option (if current option terminates)
        if self.current_option.is_terminated(next_state):

            self.current_option = self.option_policy.act(
                next_state, self.Qtable
            )

    ...
```

Intra-Option Q-Learning

In this section we provide the snippet of code that performs the Intra-ption Q-Learning update and selects the next option when the current option terminates.

```
class IntraOptionQLearningAgent:

    ...

    def step(self, state, action, reward, next_state, done):

        Q = self.Qtable

        possible_options = []
        for option in self.options:
            if option.can_initiate(next_state):
                possible_options.append(option.index)

        next_Q_value = np.max(Q[next_state, possible_options])
        beta = self.current_option.termination[next_state]

        for option in self.options:
            if option.can_initiate(state) == False:
                continue
            if option.policy.act(state) != action:
                continue

            curr_Q_value = Q[state, option.index]

            # Estimating Q-Value values from next state and beta
            estimate_Q = (
                (1.0 - beta)*Q[next_state, option.index]
                + beta*next_Q_value
            )

            # Intra-Option Q-Learning update
            Q[state, option.index] = (
                curr_Q_value + self.LR *
                (reward + self.GAMMA*estimate_Q - curr_Q_value)
            )

        # Select next option (if current option terminates)
        if self.current_option.is_terminated(next_state):

            self.current_option = self.option_policy.act(
                next_state, self.Qtable
            )

    ...
```

Training

This code block consists of the main training loop for the agent, it handles the agent's actions, stepping the environment forward and also updating the agent's parameters for learning.

```
class Trainer:

    def training(self, env, agent, n_episodes=10000, process_training_info=lambda *args, **
    kwargs: (False, {})):
        """
        To train an agent in the given environment.

        Args:
            - env: The environment for training.
            - agent: An agent with `.step()`, `.act()`, and `.update_agent_parameters()`.
            - n_episodes (int, optional): Number of training episodes. Defaults to 10000.
            - process_training_info (function, optional): Runs after each episode.
                - First return value must be a `bool` for early stopping.
                - Second return value must be a `dict` to update the progress bar's postfix.

        Returns:
            - dict: Summary of the training process.
        """

        progress_bar = tqdm(range(1, n_episodes+1), desc="Training")

        #training loop
        for i_episode in progress_bar:
            state, _ = env.reset()
            agent.set_option(state)
            score = 0
            total_reward = 0
            terminated, truncated = False, False
            episode_history = []

            while not (terminated or truncated):
                action = agent.act(state)
                next_state, reward, terminated, truncated, _ = env.step(action)
                episode_history.append((state, action, reward, next_state))
                done = (terminated or truncated)
                agent.step(state, action, reward, next_state, done)
                state = next_state
                score += self.compute_score(reward)
                total_reward += reward

            agent.update_agent_parameters()
        return {
            "computation_time": end_time - begin_time,
            "scores": np.array(history_scores),
            "total_rewards": np.array(history_total_rewards)
        }

    def compute_score(self, reward):
        return reward
```

Metrics and Hyperparameters

To quantitatively compare the performance of the algorithms in an environment we have used the following two metrics:

- **Cumulative regret** : The sum of difference between return assuming optimal policy and the 5 run average return at each step. For this assignment we have tuned the hyperparameters to minimize the regret in all experiments.
- **Maximum rolling average score** : The maximum of the rolling average return over the past 100 episodes averaged over 5 runs

Further, based on the description in section 1, the agents are characterized by a few hyperparameters:

- **LR** : The learning rate used for updating the Q-Table (in both SMDP and Intra-Option Q-Learning).
- **eps_start** : The starting value of ϵ in the ϵ-greedy policy, for selecting options.
- **eps_end** : The minimum value ϵ can take while training.
- **decay_type** : Determines whether ϵ decays linearly or exponentially over time.
- **frac_episodes_to_decay** : The fraction of total episodes over which ϵ is decayed before being fixed at its minimum value.

Option Set 1 (GoTo Red, GoTo Green, GoTo Yellow, GoTo Blue)

The option set 1 consists of 4 Markov options (GoTo Red, GoTo Green, GoTo Yellow, GoTo Blue), that navigate the taxi to a specific location (Red, Green, Yellow and Blue). In particular, each option consists of 3 components:

Initiation set:

- Each option can be initiated when the taxi is not already at the destination location of the option.

Policy:

- The policy for each option follows a deterministic path to its target location, these paths are precomputed using Dijkstra's shortest path algorithm. This policy also wraps the logic for picking up and dropping.
- Using the precomputed paths, we map each state to a particular action that will lead it on the shortest path to its target location, making these options Markov in nature.
- When the taxi reaches the target location, depending on the state either pickup or drop actions are performed.

Termination set:

- The option terminates for all states where taxi has reached the target location except:
  - When a passenger is at the target location (to allow picking up the passenger).
  - When the passenger is in taxi and the taxi is at the destination (to allow dropping the passenger).

Figure 1 visualizes the action mapped to each state for each option.

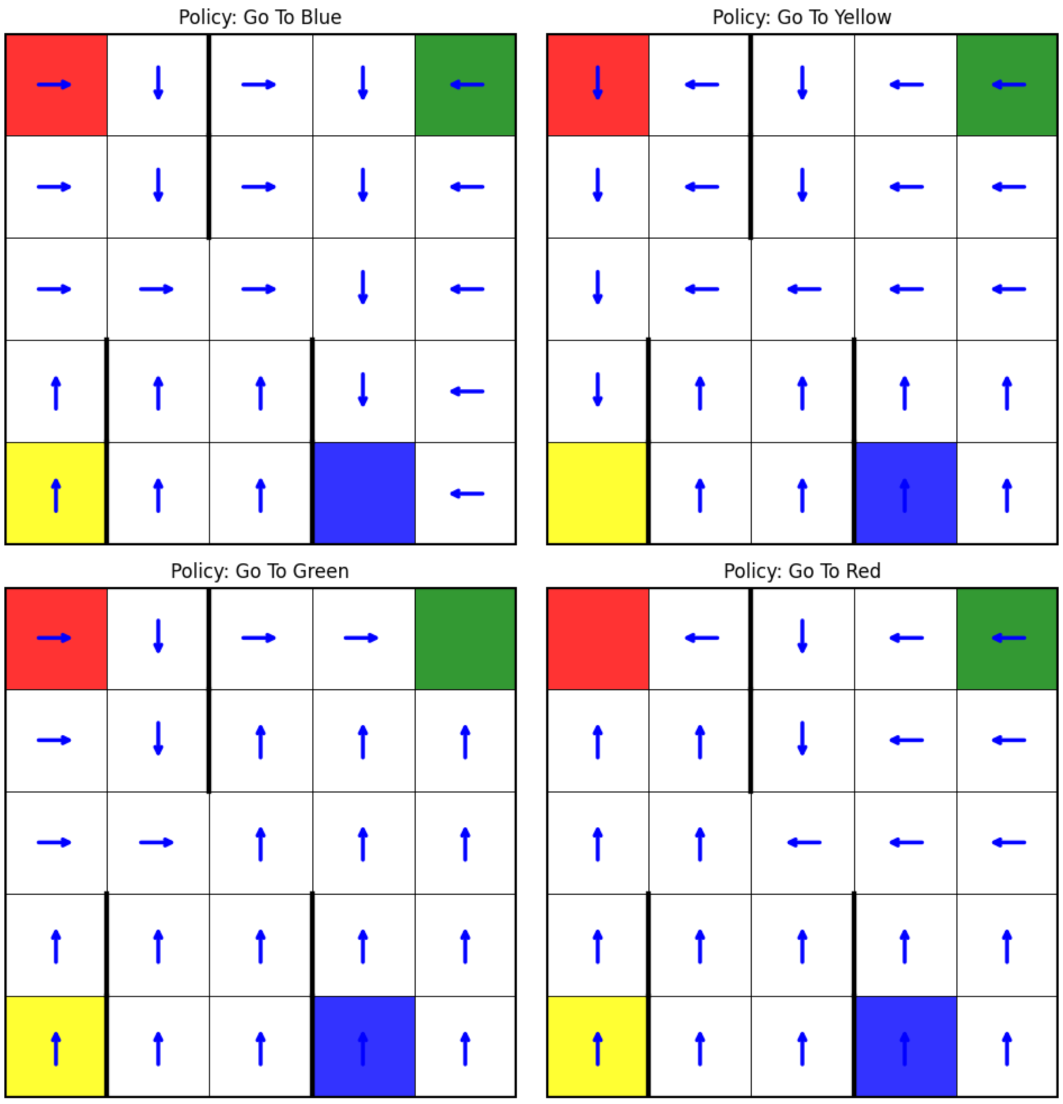


Figure 1: Visualization of the policies used in option set 1

Part (i): SMDP Q-Learning

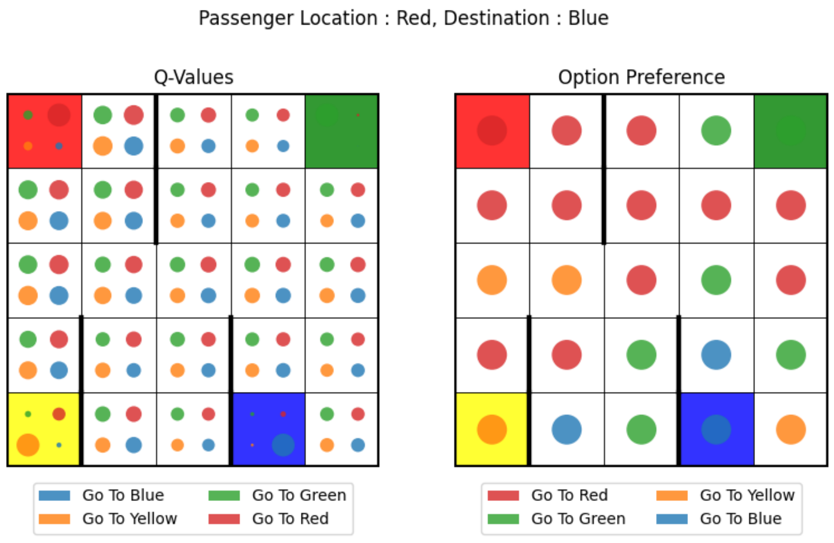
We train our SMDP Q-Learning agent according to the following set of hyperparameters and average our results over 5 experiments.

SMDP Option Set 1 Hyperparameters	
num_episodes	10000
LR	1×10^{-1}
eps_start	1
eps_end	0.005
decay_type	exponential
frac_episodes_to_decay	0.7

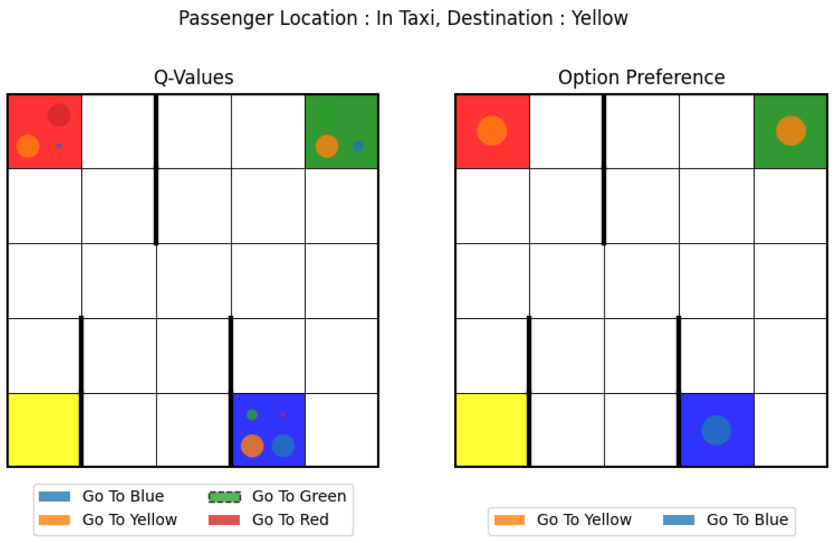
Table 1: Top Hyperparameter configuration for SMDP option set 1

We visualize the Q-values of each state with the aid of bubble plots. Each cell in the grid belongs to 20 states in the state space (5 passenger positions and 4 passenger destinations). We plot one sample each for when the passenger is not in taxi and when he is in taxi. The size of the bubbles is proportional to the ranking (the higher the ranking, the bigger the bubble) of Q-values for each cell given a passenger position and passenger destination.

Based on the visualized plots, we can infer that the value function most of the time (owing to insufficient training episodes) suggests the option that guides the taxi to the passenger location when he is not in the vehicle. Also, when he is in the taxi, the value function suggests the agent to move to the desired destination. We are able to see the preferences in only the destination cells, because the option is only instantiated there (except for the destination cell itself, because it is not a part of the initiation set). This happens because no option terminates in any cell except for the destination cells. We only decide about options in these other cells at the first time step when the taxi is starting at a random location in the grid. Since states with passenger in taxi are never the starting states, we don't learn the Q-values in these states.



(a) Learned Q-values in states before passenger is in taxi

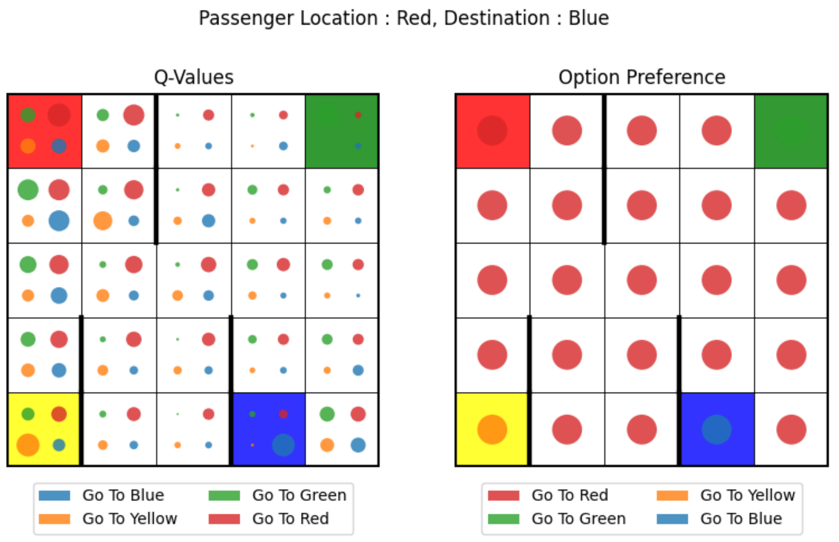


(b) Learned Q-values in states when passenger is in taxi

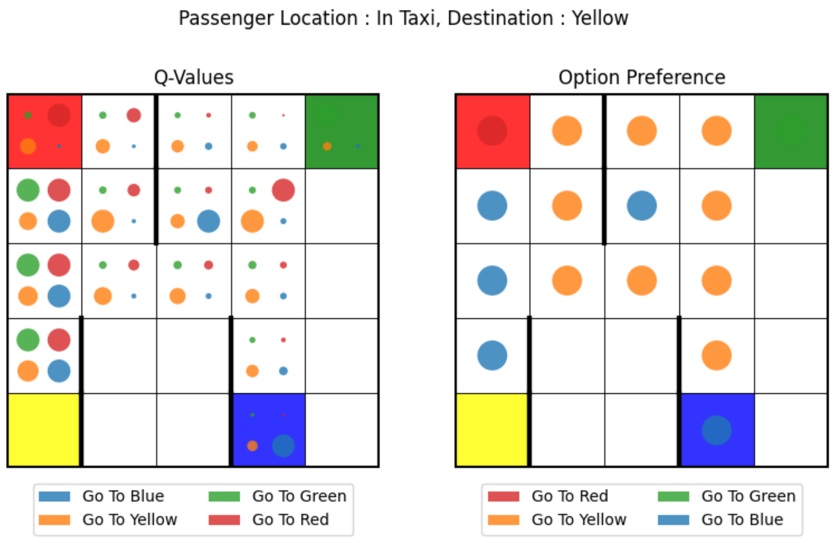
Figure 2: Visualization of learned Q-Values of SMDP Q-Learning for option set 1

Part (ii): Intra-Option Q-Learning

Similar to the SMDP Q-Learning case, we train the agent over 10000 episodes using the same hyperparameters as in Table 1 and average the performance over 5 runs. We have visualized the learned Q-Values and the option preferences for 2 particular cases (Case 1: Passenger location: Red, Destination: Blue) and (Case 2: Passenger location: In Taxi, Destination: Yellow) in Figure 3. From the plot for Case 1 we observe that when the passenger is not in the taxi, the agent prefers to choose to go to the location of the passenger as indicated by the larger Q-value bubble for that option (in this case GoTo Red). Meanwhile in Case 2, when the passenger is already in the taxi, the agent prefers to go to the destination, as indicated by the larger Q-Value bubble (in this case GoTo Yellow). Some option preferences deviate from the expected behavior, which can be attributed to insufficient training episodes, leading to a suboptimal policy. Nevertheless, this visual analysis supports that the agent has learned a generally effective policy.



(a) Learned Q-values in states before passenger is in taxi



(b) Learned Q-values in states when passenger is in taxi

Figure 3: Visualization of learned Q-Values of Intra-Option Q-Learning for option set 1

Thus the policy learned by both the agents can be stated as:

- If passenger is not in taxi, choose the option that leads to the passenger.
- If passenger is in the taxi, choose the option that leads to the destination.

We reason that the agent learns this policy because choosing options that do not contribute toward completing the task, such as moving to irrelevant locations, attempting to drop off a passenger who is not in the taxi, or picking up when no passenger is present, incurs negative rewards. These penalties represent wasted time steps and illegal actions. While in contrast, correctly choosing an option to go to a passenger and then successfully choosing to go to the destination for dropping them off results in a positive reward. As a result, the agent learns to avoid inefficient or invalid options and converges to the policy described above.

Part (iii): Comparison

Now we compare the performance of both the algorithms, Figure 4 contains the reward curves averaged over 5 runs for both the algorithms.

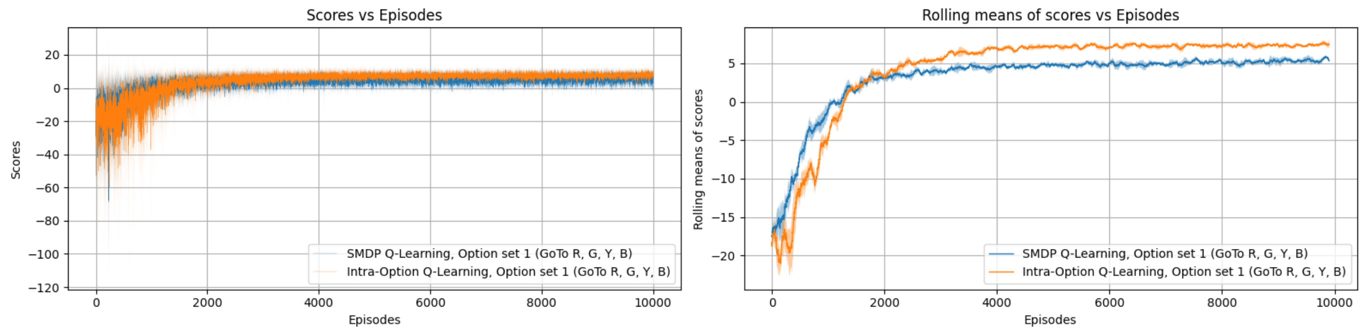


Figure 4: Comparing of performance of SMDP and Intra-Option Q-Learning for option set 1

From the Figure 4 it is evident that for approximately the first 1000 episodes the SMDP Q-Learning performs better however, afterwards Intra-Option Q-Learning performs better by a margin and reaches optimal performance first. We believe that the Intra-Option Q-Learning agent shows better performance quicker, because it uses samples more efficiently, i.e. it updates the Q-values of other states at every raw step of an episode while executing a single option. In particular we can illustrate this from the Case 1, when an agent is say going to pick up a passenger from a particular location, it updates the Q-Values of every state on that path, which are also perfectly valid starting points. This enables the agent to learn the optimal actions for even those states from which it has not currently started. As more Q-values have valid updates, the agent converges to an optimal policy faster.

Option Set 2 (PickUp Passenger, DropOff Passenger)

In this section we come up with a new option set that is mutually exclusive from the previous option set 1. The option set 2 consists of 2 Markov options (PickUp Passenger, DropOff Passenger), that navigate the taxi to the location of the passenger and destination respectively. These are more "informed" options and are expected to perform better. In particular, each option consists of 3 components:

Initiation set:

- Each option can be initiated in any state. We leave it to the Q-Learning algorithm to learn to pick these options in the correct states.

Policy:

- The policy for each option follows a deterministic path to the passenger or destination respectively, these paths are precomputed using Dijkstra's shortest path algorithm. This policy also wraps the logic for picking up and dropping.
- Using the precomputed paths, we map each state to a particular action that will lead it on the shortest path to its target location, making these options Markov in nature.
- When the taxi reaches the target location, depending on the option either pickup or drop actions are performed.

Termination set:

- The option to pick a passenger terminates once a passenger is in taxi.
- The option to drop a passenger terminates once the taxi reaches the destination (regardless of whether the passenger is in the taxi or not).

Figure 5 illustrates two sample cases demonstrating the behavior of the option set 2 policies, one for picking up a passenger and the other for dropping off. Each case visualizes how the option maps states to actions during execution.

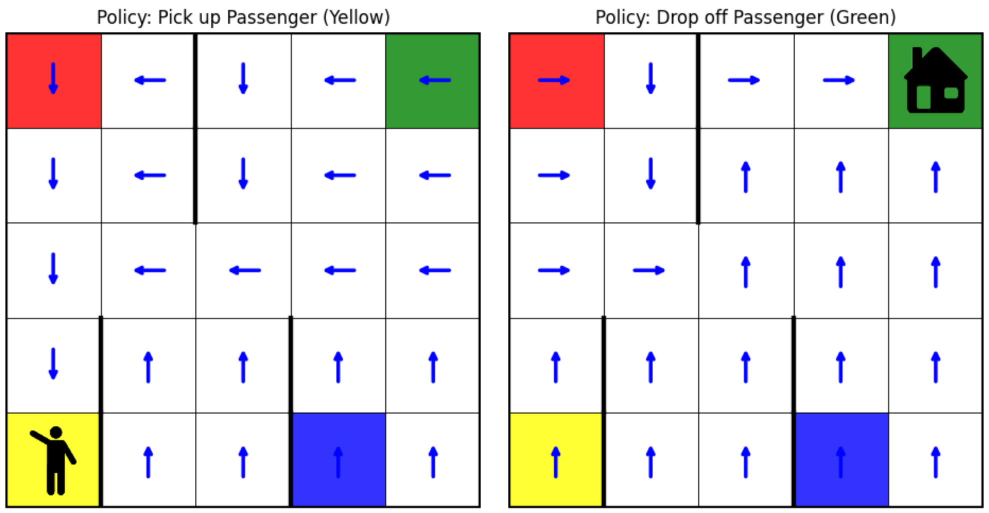

Policy: Pick up Passenger (Yellow)
Policy: Drop off Passenger (Green)

Figure 5: Visualization of the policies used in option set 2

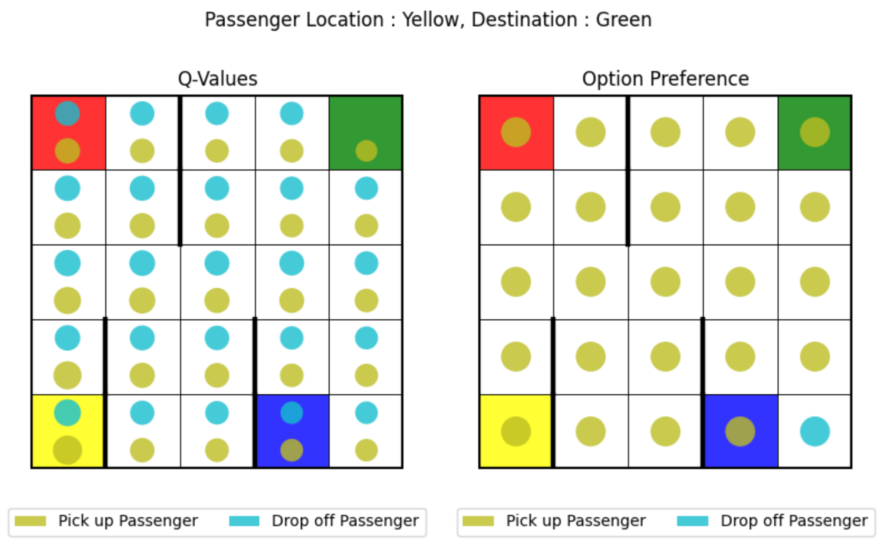
Part (i): SMDP Q-Learning

We train according to the following set of hyperparameters and average our results over 5 experiments.

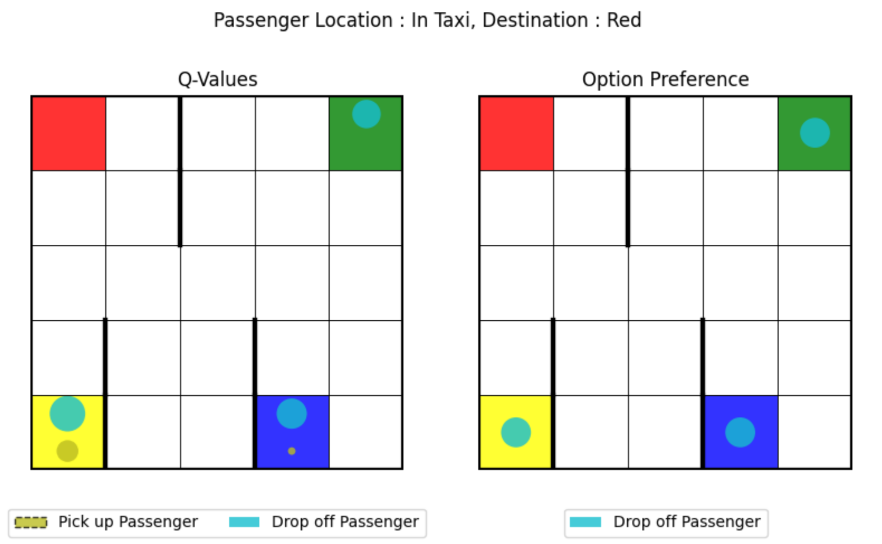
SMDP Option Set 2 Hyperparameters	
num_episodes	10000
LR	1×10^{-1}
eps_start	1
eps_end	0.005
decay_type	exponential
frac_episodes_to_decay	0.7

Table 2: Top Hyperparameter configuration for SMDP Option set 2

When we visualize the Q-values learned by our agent as in the previous sections, we observe that the agent learns to pick up the passenger when he is not in the taxi. When he is in taxi, the agent learns to drop him off to his destination. As pointed before, for the same reason, we only observe the learned Q-values for 3 destination states except the actual passenger destination.



(a) Learned Q-values in states before passenger is in taxi

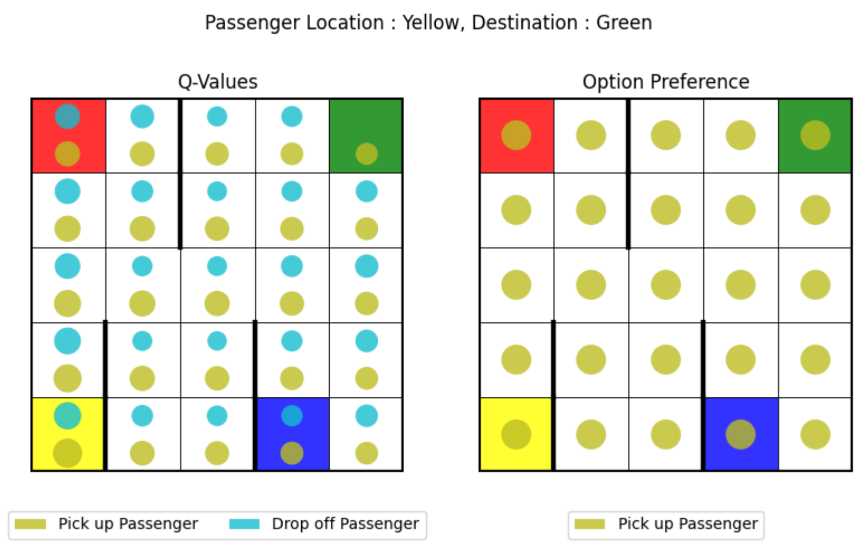


(b) Learned Q-values in states when passenger is in taxi

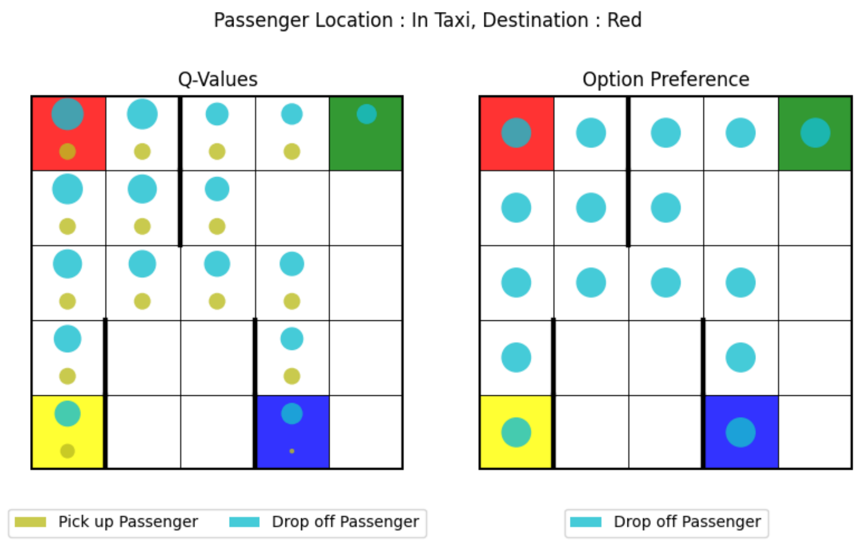
Figure 6: Visualization of learned Q-Values of SMDP Q-Learning for option set 2

Part (ii): Intra-Option Q-Learning

Similar to the previous case, we train the agent over 10000 episodes using the same hyperparameters as in Table 2 and average the performance over 5 runs. We have visualized the learned Q-Values and the option preferences for 2 particular cases (Case 1: Passenger location: Red, Destination: Blue) and (Case 2: Passenger location: In Taxi, Destination: Yellow) in Figure 3. From the plot for Case 1 we observe that when the passenger is not in the taxi, the agent prefers to pick up the passenger as indicated by the larger Q-value bubble for that option. Meanwhile in Case 2, when the passenger is already in the taxi, the agent prefers drop the passenger, as indicated by the larger Q-Value bubble. In this particular case, we have observed that all the learned option preferences are as expected, leading to the conclusion that the learned policy is optimal.



(a) Learned Q-values in states before passenger is in taxi



(b) Learned Q-values in states when passenger is in taxi

Figure 7: Visualization of learned Q-Values of Intra-Option Q-Learning for option set 2

Thus the policy learned by both the agents can be stated as:

- If passenger is not in taxi, choose the option to pick up passenger.
- If passenger is in the taxi, choose the option to drop the passenger.

We reason that the agent learns this policy because attempting to drop off a passenger before pickup results in invalid actions and incurs negative rewards due to time steps "wasted". While in contrast, successfully picking up a passenger and then executing a drop off yields positive rewards. Compared to option set 1, option set 2 offers fewer opportunities for the agent to make mistakes, allowing it to learn quickly. While Q-learning with option set 2 reduces the overhead of learning across 4 options to just 2, this reduction in overhead is in a sense balanced by the increased complexity involved in designing the two more sophisticated options, as opposed to the simpler 4 options in option set 1.

Part (iii): Comparison

In this section we compare the performance of SMDP and Intra-Option Q-Learning over both the option sets.

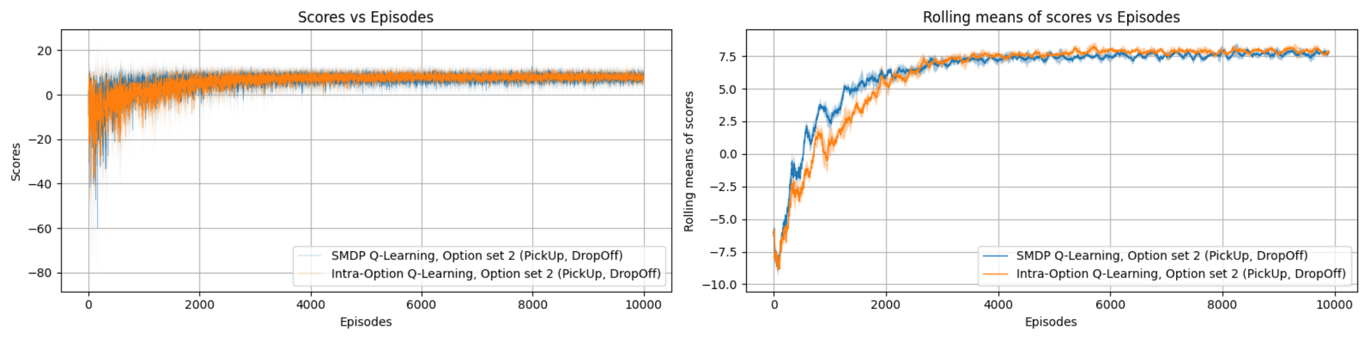


Figure 8: Comparing of performance of SMDP and Intra-Option Q-Learning for option set 2

From Figure 8, we observe that Intra-Option Q-Learning outperforms SMDP Q-Learning for option set 2, though the margin is relatively small. As discussed previously, this improvement is due to Intra-Option Q-Learning's more efficient use of samples through updates at every time step. However, the relatively small performance gap can be attributed to the more sophisticated design of the options in set 2, which reduces the room for making errors. As a result, even with less efficient sample usage, SMDP Q-Learning is still able to learn effective option preferences.

Comparison of Option sets

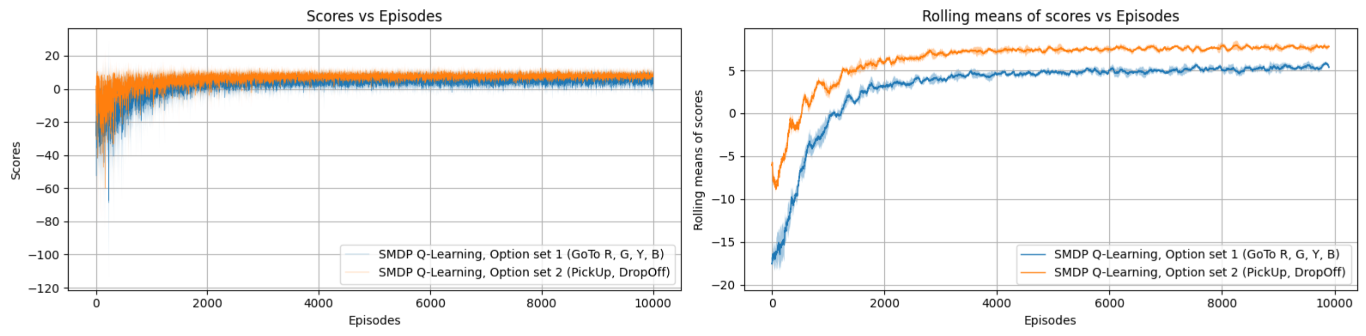


Figure 9: Comparing of performance of SMDP Q-Learning for option set 1 and 2

From Figure 9, we observe that SMDP Q-Learning with option set 2 significantly outperforms the same algorithm using option set 1. This aligns with our earlier discussion, that the more informed design of option set 2 allows the agent to learn faster due to fewer choices of options and also achieve a higher average reward.

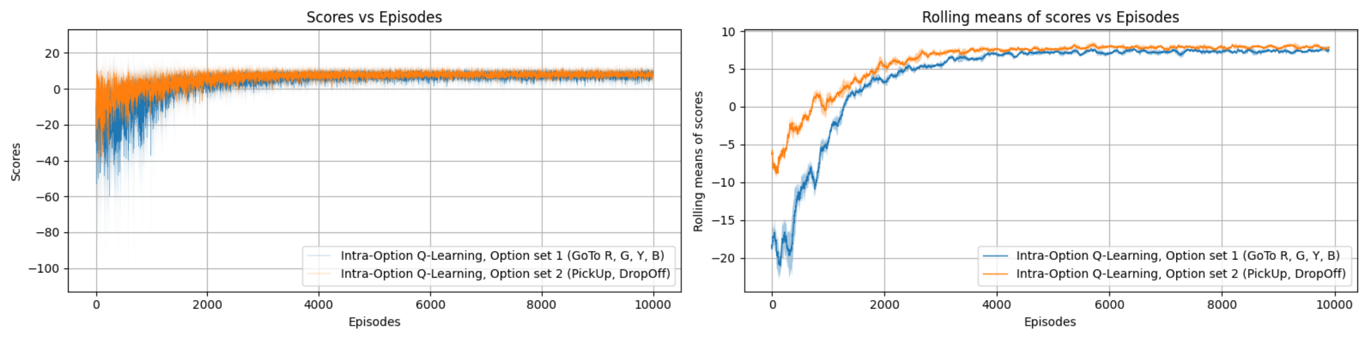


Figure 10: Comparing of performance of Intra-Option Q-Learning for option set 1 and 2

From Figure 10, we observe that Intra-Option Q-Learning with option set 2 outperforms the same algorithm using option set 1 by only a small margin. Although the same effect with the more informed choice of options is applicable here, we believe that the efficient sample usage inherent to Intra-Option Q-Learning enables the agent using option set 1 to also converge quickly to the optimal policy, narrowing the gap in performance.