DA6400: Reinforcement Learning
Programming Assignment 3 Report

KUSH RUSHABH SHAH | CH21B053 M S SAI TEJA | ME21B171

May 3, 2025

Abstract

This report presents the implementation and evaluation of two temporally-extended reinforcement learning algorithms—1-step SMDP Q-learning and Intra-option Q-learning—on the Taxi-v3 environment using two different option sets. The results are for the best hyperparameters

1 Introduction

In hierarchical reinforcement learning, an *option* is a temporally-extended course of action characterized by an initiation set, an internal policy over primitive actions, and a termination condition. Sutton *et al.* (1999) introduced the Semi-Markov Decision Process (SMDP) framework to support learning over options.

- **SMDP Q-learning** treats each option as an atomic action: when an option o is selected in state s, the agent follows the option's policy for k steps, receives a cumulative discounted reward $R = \sum_{t=0}^{k-1} \gamma^t r_t$, lands in state s', and updates

$$Q(s,o) \leftarrow Q(s,o) + \alpha\Big(R + \gamma^k \max_{o'} Q(s',o') - Q(s,o)\Big).$$

- **Intra-option Q-learning** performs additional updates at each primitive step: for every option o_j whose policy would have chosen the executed action in the current state, the algorithm updates $Q(s,o_j)$ towards the one-step reward plus the value of continuing o_j (or the greedy value if o_j would terminate).

2 Environment

We use Gymnasium's `Taxi-v3` environment, where a taxi must navigate a 5×5 grid to:

1. Navigate to one of four landmark locations (Red, Green, Yellow, Blue) to pick up a passenger,
2. Execute the `PICKUP` action,
3. Navigate to the passenger's destination landmark,
4. Execute the `DROPOFF` action.

State space is 500 discrete states, and the primitive action set is {South, East, North, West, PICKUP, DROPOFF

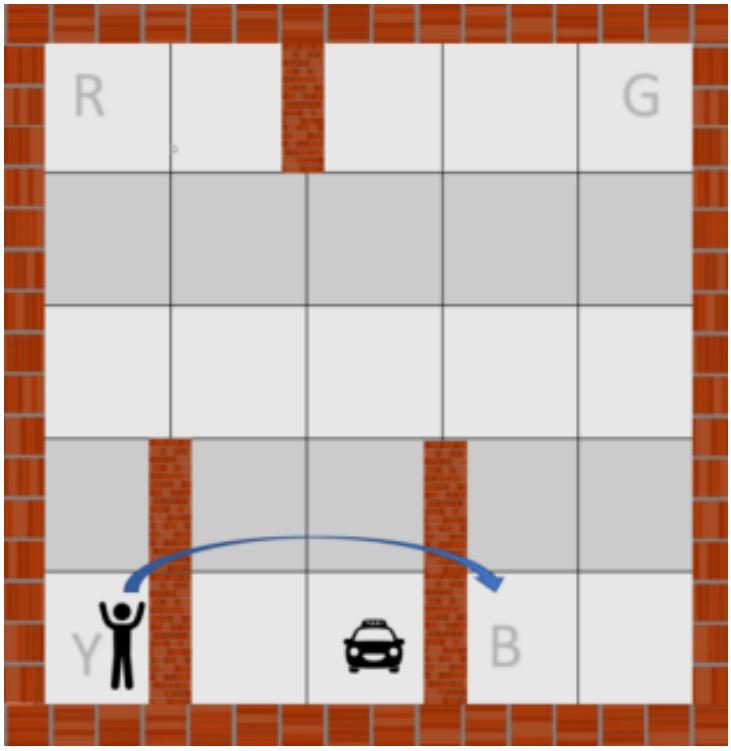

Figure 1: Taxi-v3 grid world environment.

3 Algorithm Implementation

3.1 Option Definitions

We compare two option sets:

Original option set: Four navigation options (`nav_to_R,G,Y,B`). Each option follows a greedy movement policy (vertical then horizontal) and is only available when the taxi is not at its target position.

```
class TaxiOption:
    def __init__(self, target_pos, env):
        self.target_pos = target_pos
        self.env = env

    def get_action(self, state):
        taxi_row, taxi_col, _, _ = self.env.unwrapped.decode(state)
        t_row, t_col = self.target_pos

        if taxi_row > t_row:
            return 1  # North
        elif taxi_row < t_row:
            return 0  # South
        if taxi_col > t_col:
            return 3  # West
        else:
            return 2  # East

    def is_terminated(self, state):
        taxi_row, taxi_col, _, _ = self.env.unwrapped.decode(state)
        return (taxi_row, taxi_col) == self.target_pos
```

Listing 1: Navigation Option to Target

Passenger-based option set: Dynamic navigation options that first navigate to the passenger's pickup location, then to the destination, with one `pickup` option, and one `dropoff` option.

```python
class PassengerOption:
    def __init__(self, env):
        self.env = env

    def get_action(self, state):
        taxi_row, taxi_col, pass_loc, dest_idx = self.env.unwrapped.decode(
            state)

        # Determine target based on passenger location
        if pass_loc == 4:  # Passenger in taxi
            target = self.env.unwrapped.locs[dest_idx]
        else:
            target = self.env.unwrapped.locs[pass_loc]

        # Greedy navigation
        if taxi_row > target[0]:
            return 1  # North
        elif taxi_row < target[0]:
            return 0  # South
        if taxi_col > target[1]:
            return 3  # West
        else:
            return 2  # East

    def is_terminated(self, state):
        _, _, pass_loc, dest_idx = self.env.unwrapped.decode(state)
        return pass_loc == dest_idx  # Passenger at destination
```

Listing 2: Passenger-Based Navigation Option

3.2 SMDP Q-Learning Agent

- ϵ-greedy selection over options.
- One-step update as in the SMDP Q-learning equation above.
- No additional intra-option updates.

Below are some important code snippets in the implementation of SMDP with options

Available Options Filter:

```python
def get_available(self, state):
    available = list(range(self.num_primitives))
    taxi_row, taxi_col, _, _ = self.env.unwrapped.decode(state)

    for i, opt in enumerate(self.options):
        if hasattr(opt, 'target_pos'):
            if (taxi_row, taxi_col) != opt.target_pos:
                available.append(self.num_primitives + i)
        elif hasattr(opt, 'is_available'):
            if opt.is_available(state):
                available.append(self.num_primitives + i)

    return available
```

Listing 3: get_available function

This method builds the set of all legal actions (primitive + options) in the current state. It introspects each option—checking `target_pos` for navigation options and `is_available` for passenger-focused ones. The returned list of indices drives both exploration and exploitation in the agent's policy.

Option Execution Logic:

```
def execute_option(self, action_idx, state):
    if action_idx < self.num_primitives:
        next_state, reward, terminated, truncated, _ = self.env.step(action_idx
            )
        return reward, next_state, 1, terminated or truncated
    else:
        option = self.options[action_idx - self.num_primitives]
        total_reward = 0
        discount = 1.0
        steps = 0
        current_state = state
        terminated = False

        while True:
            if option.is_terminated(current_state):
                break
            action = option.get_action(current_state)
            next_state, reward, terminated, truncated, _ = self.env.step(action
                )
            total_reward += discount * reward
            discount *= self.gamma
            steps += 1
            current_state = next_state
            if terminated or truncated:
                break
        return total_reward, current_state, steps, terminated or truncated
```

Listing 4: execute_option function

This routine dispatches either a single primitive or an entire option until its termination condition. It accumulates the discounted reward over `steps` and tracks when the episode ends via `terminated or truncated`. The tuple `(total_reward, next_state, steps, done)` is used to perform the SMDP-style update.

Q-Value Update Rule:

```
def update(self, state, action_idx, reward, next_state, steps):
    future_max = np.max(self.Q[next_state, self.get_available(next_state)])
    target = reward + (self.gamma ** steps) * future_max
    self.Q[state, action_idx] += self.alpha * (target - self.Q[state,
        action_idx])
```

Listing 5: update function

Here we form the SMDP Q-learning target by adding the discounted future max-value after `steps`. The temporal-difference error `(target - Q[state, action])` is scaled by the learning rate `alpha`. This single-line update integrates both primitive-step and option-step learning seamlessly.

3.3 Intra-option Q-Learning Agent (Bonus)

- ϵ-greedy selection over both primitive actions and options
- Dual updates at each timestep:

  - Standard Q-learning update for executed primitive action
  - Intra-option updates for all compatible options
- Option continuation value calculation during updates

Intra-option Update Logic:

```
def intra_option_update(self, state, action, next_state, reward):
    # Primitive action update
    future_max = np.max(self.Q[next_state])
    self.Q[state, action] += self.alpha * (
        reward + self.gamma * future_max - self.Q[state, action]
    )

    # Option value updates
    for i, option in enumerate(self.options):
        if option.get_action(state) == action:
            option_idx = self.num_primitives + i
            continuation_value = (
                self.Q[next_state, option_idx]
                if not option.is_terminated(next_state)
                else np.max(self.Q[next_state])
            )
            self.Q[state, option_idx] += self.alpha * (
                reward + self.gamma * continuation_value -
                self.Q[state, option_idx]
            )
```

Listing 6: intra_option_update function

This method implements the core intra-option learning mechanism. For each option that would have selected the executed primitive action ($\pi_o(s) = a$), it computes:

$$Q(s,o) \leftarrow Q(s,o) + \alpha \left[r + \gamma \left(I_{\neg\beta} Q(s',o) + I_\beta \max_{a'} Q(s',a') \right) - Q(s,o) \right]$$

where β indicates option termination in s'. The `continuation_value` term handles both continuing and terminating cases.

Training Loop with Stepwise Updates:

```
def train(self, episodes):
    for _ in range(episodes):
        state = env.reset()
        while not done:
            action_idx = self.choose_action(state)

            if action_idx < 6:  # Primitive action
                next_state, reward, done, _ = env.step(action_idx)
            else:  # Option execution
                option = self.options[action_idx - 6]
                while not option.is_terminated(state):
                    action = option.get_action(state)
                    next_state, reward, done, _ = env.step(action)
                    self.intra_option_update(state, action,
                                             next_state, reward)
                    state = next_state
```

Listing 7: train function

The training loop demonstrates three key features:

1. Seamless integration of primitive and option execution

2. Nested option execution with termination checking

3. Intra-option updates at *every* primitive step during option execution

Option Availability Handling:

```
def get_available(self, state):
    available = list(range(self.num_primitives))
    for i, opt in enumerate(self.options):
        if opt.is_available(state):
            available.append(self.num_primitives + i)
    return available
```

Listing 8: get_available function

This simplified availability check assumes options self-report their availability through `is_available()`. The passenger option remains always available, while navigation options become unavailable when at their target position.

4 Results and Discussions

We report the reward plots, visualized q values, options and other plots for the 2 different option sets learnt using SMDP Q-learning.

4.1 SMDP with the Original Option Set

Here we show results for the original option set.

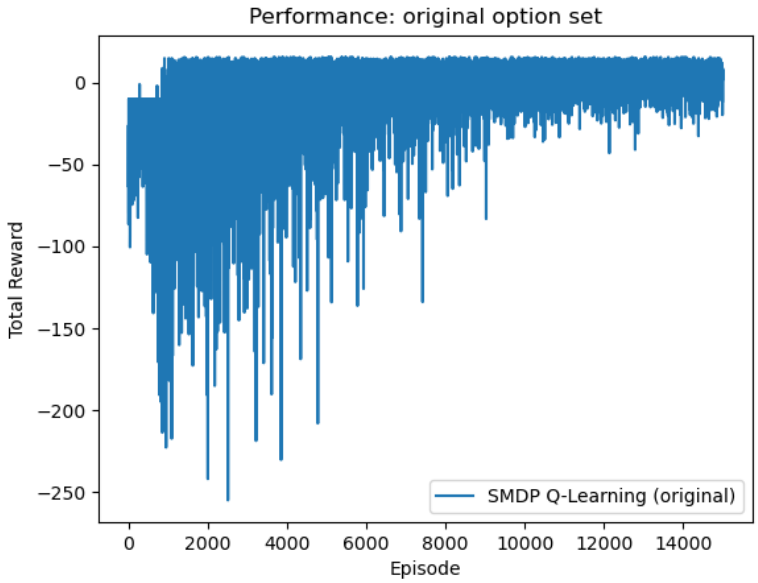

Figure 2: Reward plot for SMDP Q-learning with original option set

The learning curve demonstrates that SMDP Q-Learning with landmark navigation options achieves successful convergence, starting with poor performance before rapidly improving around episode 2000. Performance stability gradually increases over training, with the reward converging above 6.

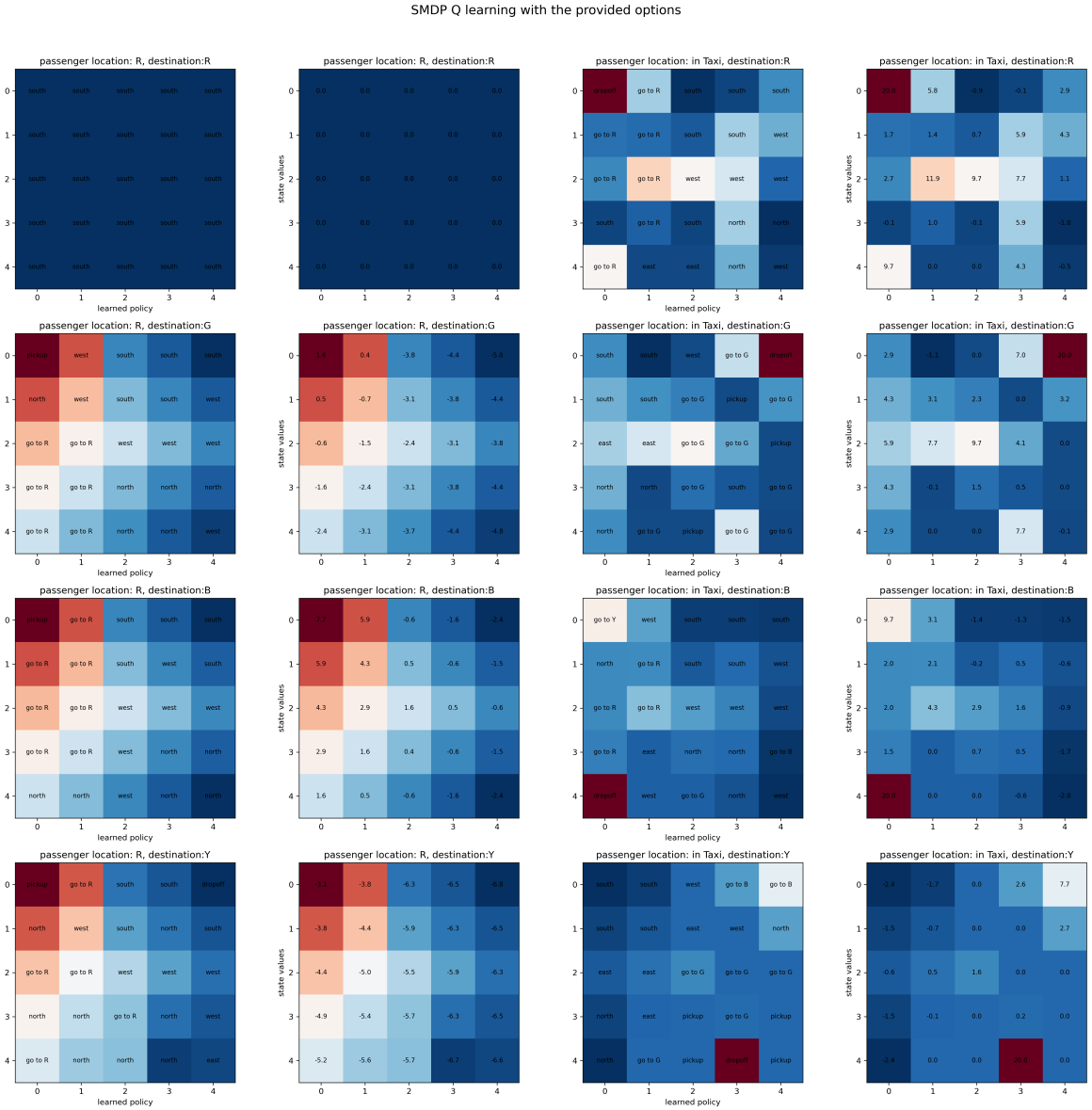

Figure 3: SMDP Q learning visualization using original option set

The visualization shows an analysis of learned policies and state values across different Taxi-v3 scenarios using SMDP Q-learning with landmark navigation options. Each row represents a different destination (R, G, Y, B), while columns alternate between displaying learned policies and corresponding state values for different passenger locations.

The learned policy reveals the hierarchical decision-making: the agent effectively utilizes navigation options ("go to R/G/Y/B") in states where reaching landmarks is beneficial, while appropriately choosing primitive actions (north/south/east/west) when finer control is needed. State values (right-side matrices) show logical progression with higher values (lighter colors) appearing in states closer to task completion, particularly when the passenger is already in the taxi and near the destination.

For example, in passenger location R/destination R scenarios (top row), the agent learns to first navigate to R for pickup, while for passenger in-taxi scenarios, it directly navigates toward the destination.

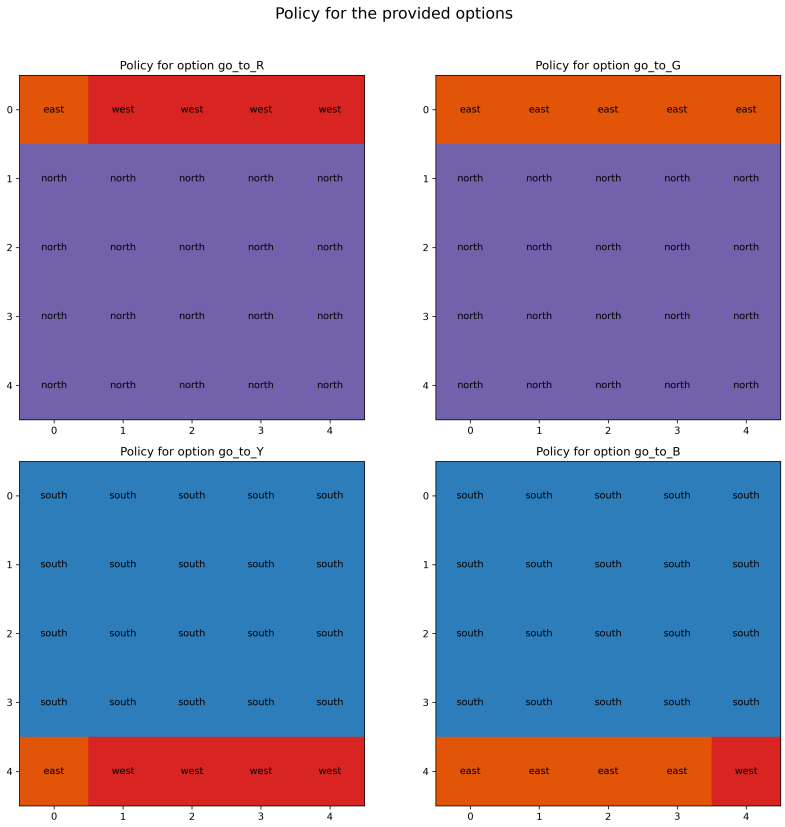

Figure 4: Visualization of the option policies

The visualization reveals deterministic, target-focused policies for each of the four navigation options in SMDP Q-learning, demonstrating the hierarchical nature of the algorithm. Each option follows a consistent vertical-then-horizontal movement strategy (north/south followed by east/west), creating simple yet effective subpolicies that enable the agent to efficiently navigate to landmark locations regardless of starting position.

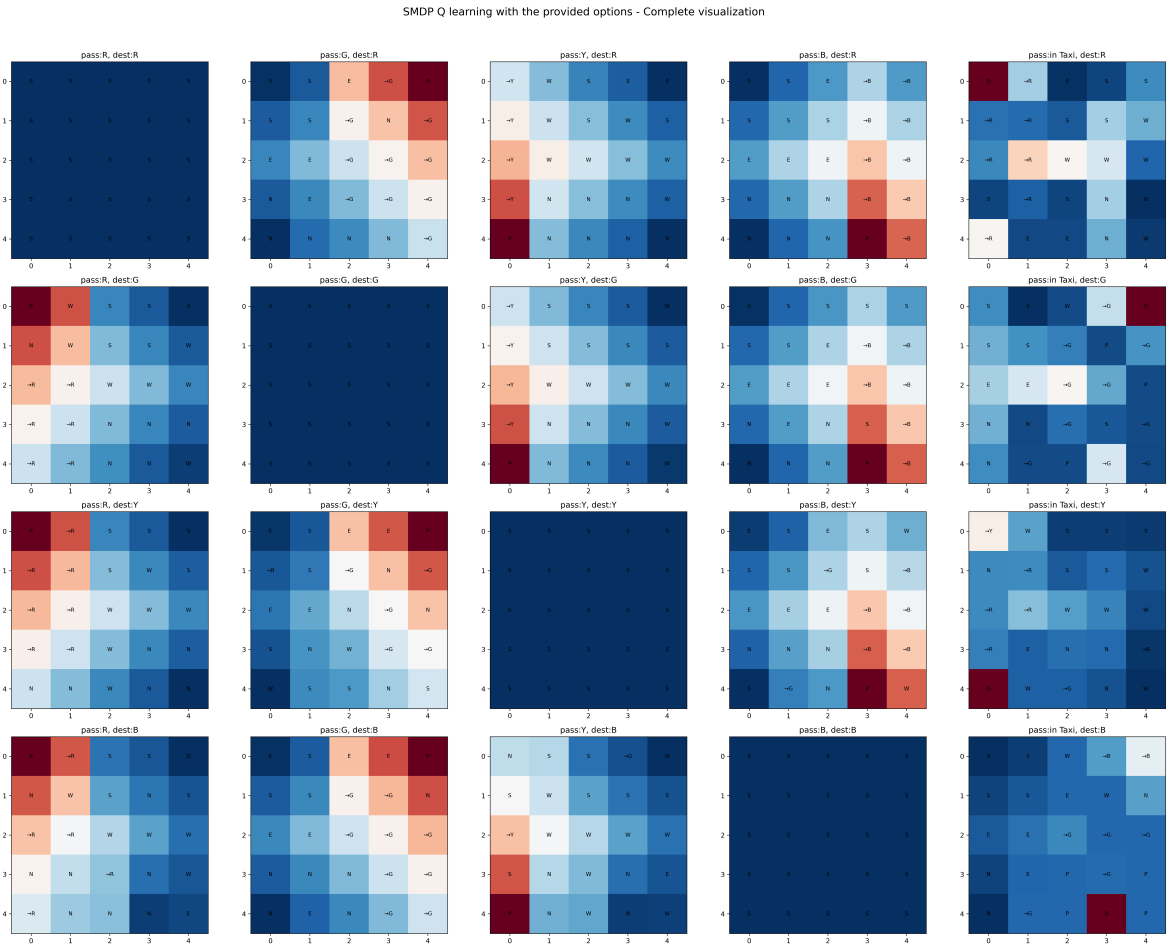

Figure 5: Full visualization of state and actions according to the options

The visualization reveals a comprehensive state-action policy map across all passenger-destination combinations in the Taxi-v3 environment, with each cell showing both learned Q-values (via color intensity) and optimal actions (via text labels). Clear patterns emerge where the agent intelligently selects navigation options (→R/G/Y/B) for efficient long-distance travel while using primitive actions (N/S/E/W) for fine-grained positioning near goals.

The darker blue regions represent states with lower expected returns, while lighter areas indicate higher-value states closer to task completion. Notably, states where passenger and destination match (diagonal subplots) show minimal policy differentiation, as the task is already complete, while states requiring complex navigation show sophisticated action selection combining both primitive actions and options.

4.2 SMDP with Passenger-Based Option Set

Results when using the passenger-based navigation options.

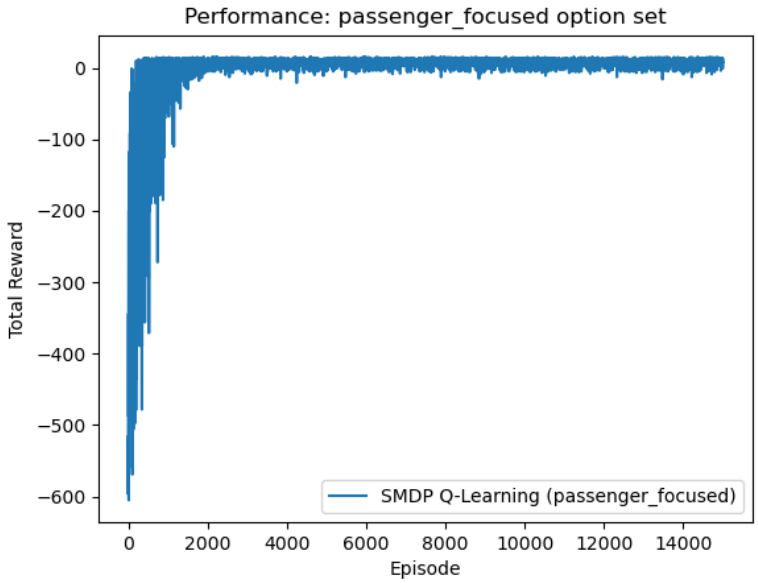

Figure 6: Reward plot for SMDP Q-learning with passenger-based option set

The passenger-focused option set demonstrates remarkably rapid convergence, reaching near-optimal performance around episode 2000 and maintaining exceptionally stable behavior with minimal fluctuations throughout extended training. This suggests the task-oriented approach effectively captures the underlying structure of the taxi problem, enabling highly efficient learning.

Compared to the landmark navigation options, the passenger-focused option converges faster and exhibits significantly more stable performance with fewer performance dips once trained. This demonstrates how a single task-oriented option can outperform multiple landmark-based options by directly encoding the complete task structure rather than requiring the agent to learn combinations of navigation primitives.

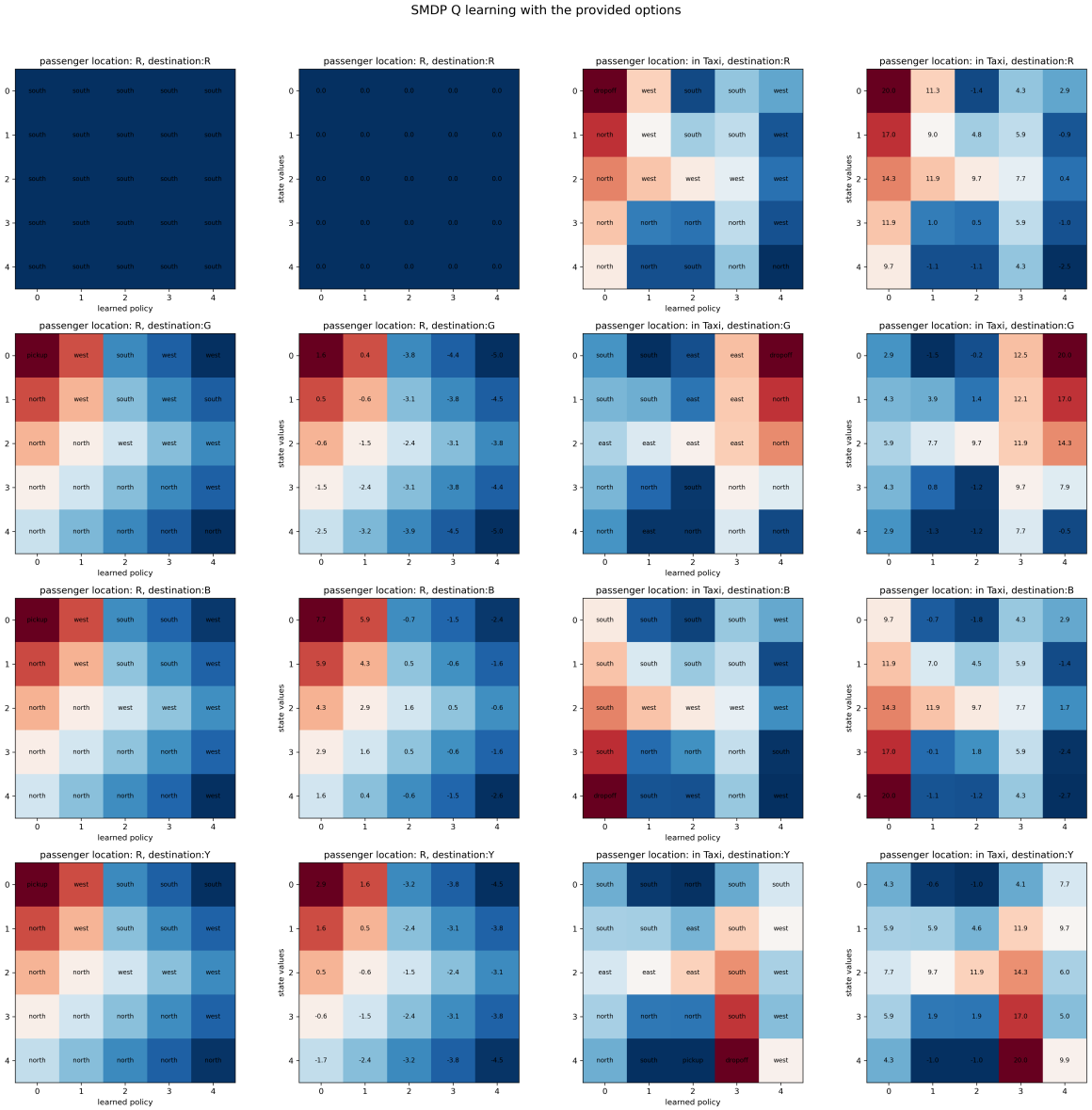

Figure 7: SMDP Q learning visualization using passenger-based option set

This visualization shows SMDP Q-learning results with the passenger-focused option set, displaying both learned policies and state values across different passenger-destination combinations. Unlike the landmark navigation option visualization described, this implementation relies predominantly on primitive directional actions (north/south/east/west) rather than hierarchical "go to" options, suggesting a different approach to temporal abstraction.

Notable differences include the consistent numerical state values displayed (ranging from negative to positive), which provide precise quantification of expected returns. The policy patterns show clear directional preferences based on taxi and passenger positions, with directional actions forming logical pathways toward goals. While the landmark option approach created distinct navigation zones using options, this visualization reveals more granular state-by-state decision making, potentially indicating a different balance between exploiting temporal abstraction and maintaining fine-grained control over movement

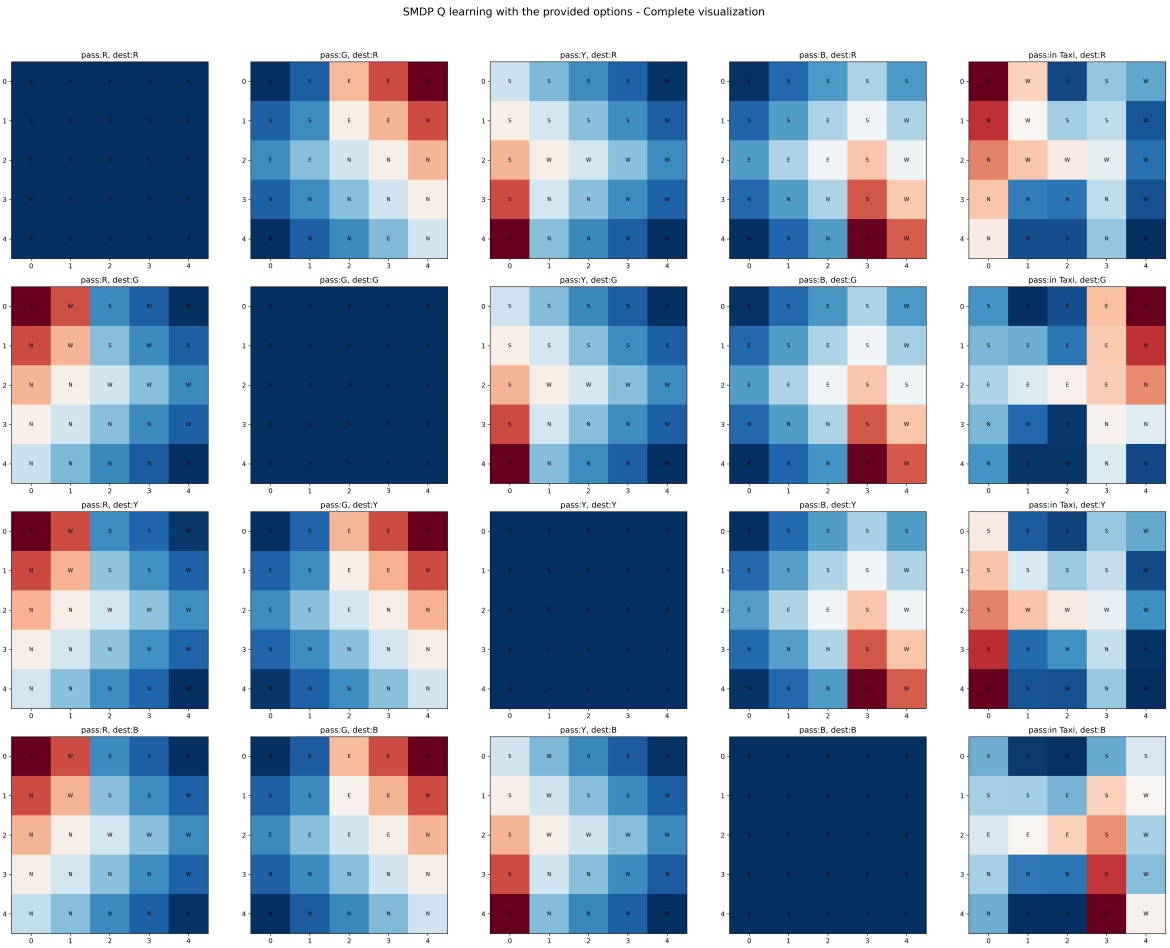

Figure 8: Full visualization of state and actions according to the passenger -based options

The visualization shows a comprehensive SMDP Q-learning policy map for the passenger-focused option set, with each subplot representing a different passenger-destination combination across a 5×5 grid. Unlike the description of the landmark navigation option approach, this implementation relies almost exclusively on primitive actions (N/S/E/W) rather than hierarchical options, suggesting a different approach to temporal abstraction where the agent has learned to solve the task primarily through sequences of primitive movements.

The color gradient indicates state values, with darker blue regions representing lower expected returns (particularly in diagonal subplots where passenger origin matches destination, indicating task completion). This visualization demonstrates how SMDP Q-learning can effectively learn primitive-action-based policies within a hierarchical framework, contrasting with the option-centric approach described in the query where navigation shortcuts (→R/G/Y/B) were prominently utilized for efficient travel.

4.3 Intra-Option Q-Learning (Bonus)

We now present the results for Intra-Option Q-Learning using both option sets and compare its performance with SMDP Q-Learning.

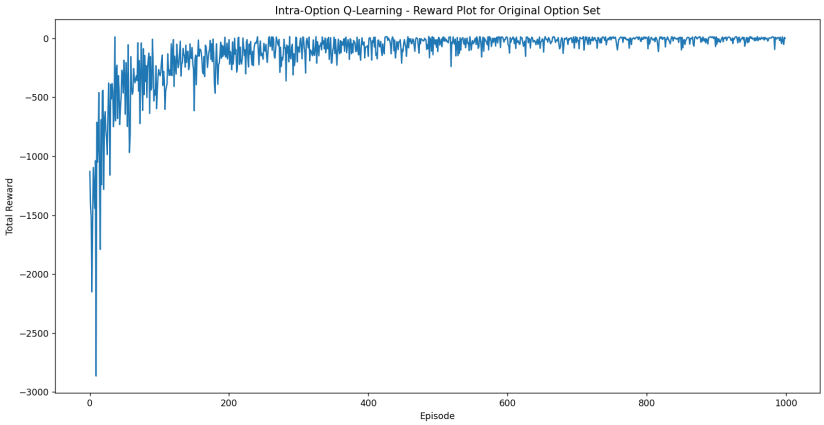

Figure 9: Reward plot for Intra-Option Q-learning with original option set

Intra-Option Q-Learning with the original option set shows a distinct learning progression. The agent initially experiences substantial negative rewards (below -2500) but rapidly improves within the first 200 episodes. By episode 400, the algorithm stabilizes with rewards consistently approaching zero, indicating successful task completion. The reduced variance in later episodes demonstrates the stability of the learned policy.

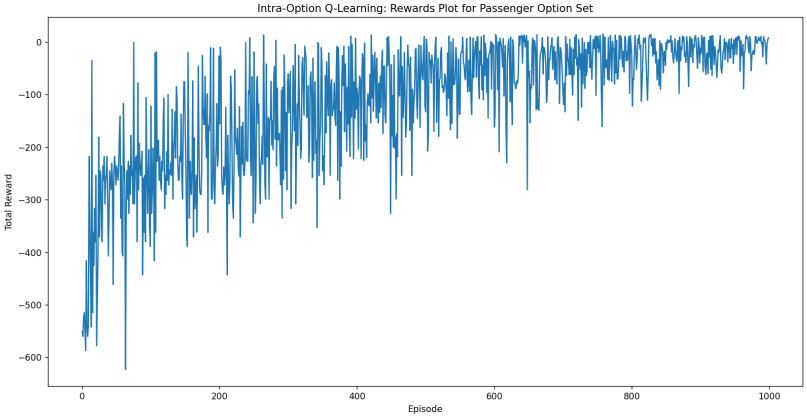

Figure 10: Reward plot for Intra-Option Q-learning with passenger-based option set

For the passenger-focused option set, we observe a less extreme initial performance (starting around -600) with a more gradual convergence pattern. While the agent achieves near-zero rewards by episode 500, there remains noticeable variability throughout training, indicating some policy instability even in later episodes.

Comparison with SMDP Q-Learning: Intra-Option Q-Learning demonstrates several key advantages over SMDP Q-Learning:

- **Accelerated early learning:** By updating option values at every primitive step rather than only at option termination, Intra-Option learning propagates value information more efficiently, particularly evident in the original option set's rapid convergence.
- **Sample efficiency:** Intra-Option learning makes more efficient use of experience, requiring fewer episodes to reach comparable performance levels.

- **Knowledge transfer across options:** The concurrent updates of multiple option values facilitate implicit knowledge sharing between related options.

However, the passenger-focused option set shows less dramatic improvement with Intra-Option learning compared to the original navigation options. This suggests that when options are already well-aligned with task structure (as in the passenger-focused design), the additional update mechanism provides less comparative advantage.

5 Conclusion

We implemented SMDP Q-learning on Taxi-v3 using two different sets of options. Both methods learn to transport the passenger in finite time. Intra-option updates generally accelerate early learning by propagating information at each primitive step. Future work could explore adding stochastic option policies or learning option policies end-to-end.

6 Code Repository

The complete source code is available on GitHub at: `https://github.com/DA6400-RL-JanMay2025/programming-assignment-03-me21b171_ch21b053`.

7 Contributions

1. M S Sai Teja: Worked on SMDP and Intra Options Q-Learning implementation, readme and corresponding parts of Intra Options in report. 2. Kush Shah: Worked on SMDP and Q-Learning Visualizations, and its corresponding parts of report and readme.