CS6700 Reinforcement Learning PA-3

Pranav Debbad (ED21B046), Adithyaa RG (ED21B005)

April 20, 2024

Github Link to the code

1 Environment Description

The environment for this task is the taxi domain. It is a 5x5 matrix, where each cell is a position your taxi can stay at. There is a single passenger who can be either picked up or dropped or is being transported. There are four designated locations in the grid world indicated by R(ed), G(reen), Y(ellow), and B(lue). When the episode starts, the taxi starts at a random square and the passenger is at a random location. The taxi drives to the passenger's location, picks up the passenger, drives to the passenger's destination (another one of the four designated locations), and then drops off the passenger. Once the passenger is dropped off, the episode ends.

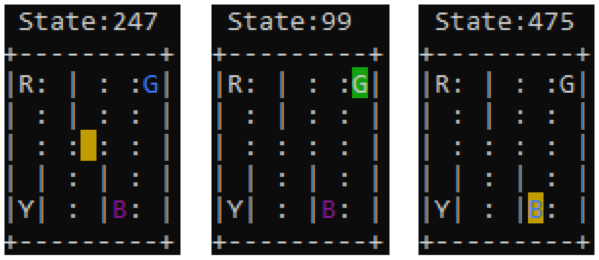
There are 500 discrete states in this environment, representing 25 taxi positions, 5 possible locations of the passenger (including the case when the passenger is in the taxi), and 4 destination locations. Note that 400 states can actually be reached during an episode. The missing states correspond to situations wherein the passenger is at the same location as their destination, which typically signals the end of an episode. Four additional states can be observed right after a successful episode when both the passenger and the taxi are at the destination. This gives a total of 404 reachable discrete 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)

Rewards:

- -1 per step unless other reward is triggered.
- +20 delivering passenger.
- -10 executing "pickup" and "drop-off" actions illegally.

The discount factor is taken to be $\gamma = 0.9$.



```
State:247      State:99       State:475
+---------+    +---------+    +---------+
|R: | : :G|    |R: | : :G|    |R: | : :G|
| : | : : |    | : | : : |    | : | : : |
| : : : : |    | : : : : |    | : : : : |
| | : | : |    | | : | : |    | | : | : |
|Y| : |B: |    |Y| : |B: |    |Y| : |B: |
+---------+    +---------+    +---------+
```

Figure 1: Taxi Domain

2 Algorithms

Each option consists of an input set (I), termination probability (β) and policy (π). It is essentially a high-level action that executes a series of primitive actions before terminating at a state according to β. Options help break down tasks into manageable sub-goals, enabling more efficient exploration and learning.

2.1 Semi-Markov Decision Process(SMDP) Q-Learning

An SMDP, unlike an MDP, accounts for the time between multiple options. This time can be an integral or a real value. Here we consider each primitive action to take a unit time. So time taken for an option to complete is the number of primitive actions taken until the completion of the option. The Q-Learning update equation for the same is:

$$Q_{k+1}(s,o) \leftarrow (1-\alpha_k)Q_k(s,o) + \alpha_k\Big[r + \gamma^{\tau} max_{o' \epsilon O_{s'}} Q_k(s',o')\Big]$$

where r is the accumulated return over the option for executing τ timesteps.

For each primitive action taken, a regular Q-Learning update is done, and when a non-primitive option is taken, its Q-value is updated using the SMDP Q-Learning update. For each step within the option, the option policy is also updated by updating its Q-map using Q-Learning for the primitive action taken at that step.

2.2 Intra-Option Q-Learning

SMDP Q-Learning waits till the end of an option to perform the update, which might be underusing the experience gained by the agent from executing the option. Intra-option Q-Learning updates the higher-level Q-value for the option at every step within it besides updating the current option's Q-values. It also updates the value of every option that takes the same action.

$$U_k(s_t,o) = (1-\beta(s))Q_k(s,o) + \beta(s) max_{o' \epsilon O} Q_k(s',o')$$
$$Q_{k+1}(s_t,o) \leftarrow (1-\alpha_k)Q_k(s_t,o) + \alpha_k\Big[r_{t+1} + \gamma U_k(s_t,o)\Big]$$

3 Actions and Options

3.1 Options Set 1

There are 6 discrete deterministic actions:

- 0: move south
- 1: move north
- 2: move east
- 3: move west
- 4: pick passenger up
- 5: drop passenger off

Additionally, options to move the taxi to each of the four designated locations, executable when the taxi is not already there, are also provided. This gives us 4 options:

- 6: toR (terminates when the taxi reaches R)
- 7: toG (terminates when the taxi reaches G)
- 8: toY (terminates when the taxi reaches Y)

- 9: toB (terminates when the taxi reaches B)

All of these options only navigate the taxi to the designated locations. Hence, they only use primitive actions to move north, south, east, and west. Each option acts greedily with its Q-map using ϵ-greedy or softmax, thus each option has a subpolicy defined by its own Q-map.

Thus we use a total of 10 options, which includes 6 primitive actions and 4 options defined on those primitive actions.

3.1.1 Code Snippets

A few important code snippets have been attached to explain our implementation clearly.

Action Selection Policy:

```
def softmax(Q, state, tau):
    q_values = Q[state]
    q_values = q_values / tau
    max_q = np.max(q_values)
    e = np.exp(q_values - max_q)
    dist = e / np.sum(e)
    action = np.random.choice(len(dist), p=dist)
    return action

def epsilon_greedy(Q, state, epsilon):
    if random.uniform(0, 1) < epsilon:
        action = np.random.randint(0, 4)
    else:
        action = np.argmax(Q[state])
    return action
```

These functions are used to select an action based on Q-values for the state, at both levels (choosing an option/primitive action or choosing primitive actions within each option). We use a decaying ϵ/τ to balance between exploration and exploitation.

Option Execution:

```
def execute_option(env, Q_values, Q_options, Q_freq, option_freq, \
                   option, state, policy = softmax):
    optdone = False
    optact = 0
    if option < 6:
        next_state, reward, done, _, _ = env.step(option)
        optdone = done
        optact = option
        Q_values[state][option] = Q_values[state][option]+ALPHA_Q*(reward+ \
                    GAMMA*np.max(Q_values[next_state])-Q_values[state][option])
        Q_freq[state][option] += 1
        reward_bar = reward

    if option >= 6:
        reward_bar = 0
        chosen_option = options.index(option)
        chosen_Q = Q_options[chosen_option]
        counter = 0
        current_state = state
        while optdone == False:
            state_value = 5 * decode_state(env, state)[0] +\
            decode_state(env, state)[1]
            optact = policy(chosen_Q, state_value, TAU)
            next_state, reward, done, _, _ = env.step(optact)
            next_state_value = 5 * decode_state(env, next_state)[0] + \
                decode_state(env, next_state)[1]
            reward_bar += reward * (GAMMA ** counter)
            GOAL_REWARD = 30
            if decode_state(env, next_state)[:2] == goal_states[chosen_option]:
                optdone = True
                Q_values[current_state, option] += ALPHA_Q * (reward_bar - \
                                Q_values[current_state, option] + \
                                GAMMA**counter * np.max(Q_values[next_state]))
                Q_freq[current_state, option] += 1
                reward += GOAL_REWARD
            chosen_Q[state_value][optact] = chosen_Q[state_value][optact] + \
                ALPHA_OPTIONS * (reward + GAMMA * np.max(\
                chosen_Q[next_state_value])-chosen_Q[state_value][optact])
            option_freq[chosen_option][state_value][optact] += 1
            counter += 1

            state = next_state
        Q_options[chosen_option] = chosen_Q

    return next_state,reward_bar,done,[Q_values,Q_options,Q_freq,option_freq]
```

The function defined here performs regular Q-Learning for primitive actions (option<6) and SMDP Q-Learning for non-primitive options (>= 6). This can be extended to intra-options by having higher-level Q updates at each step (similar to the sub-option Q updates) within the option instead of a single update after termination of the option.

3.1.2 Results and Discussion

- **SMDP Q-Learning**

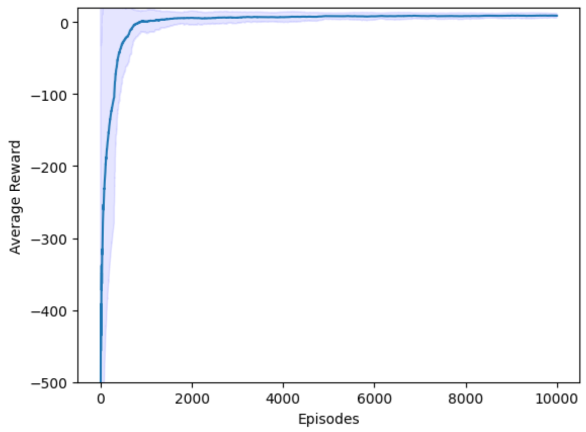


Figure 2: Mean Score window(size 100) plot over 10000 episodes

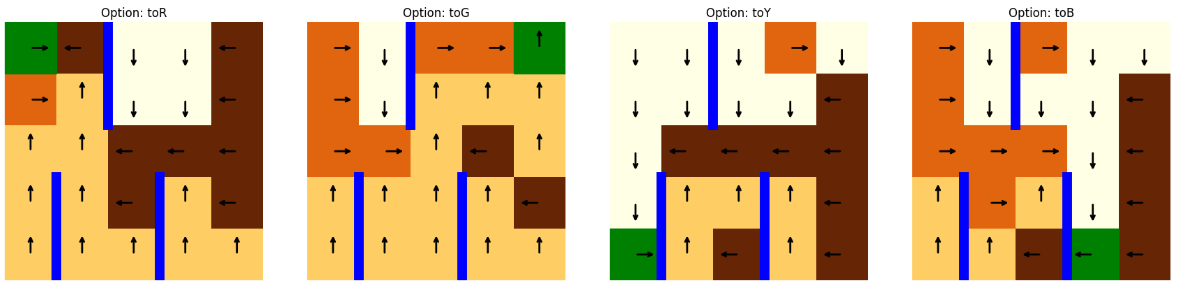


Figure 3: Learned Q-map for each option

In Fig 3, different directions are represented using different colours. The goal state is represented in green.
Since we are only navigating to these goal locations irrespective of the passenger position and destination, we used an array of $(25, 4)$ to represent the states. The plot shows the policies based on Q-values learned by each option, which the options will use to execute the primitive actions. As we can see, all of them have trained well and each state takes the agent closer to the goal state.

Large positive rewards are given in the environment only when the whole task is completed by doing a proper drop action. This meant, there could be cases where reaching the option termination would give the same reward as reaching any other state within the option (suppose when the passenger is not in the taxi). In such a case, the option policy would not learn to navigate well. To combat this, we have implemented a pseudo-reward for each option, so that whenever it reaches its termination state, we give it a reward of +30.

This leads to quicker training consistently and options learning very well. They also showed interesting results in the overall performance, improving the total reward and reducing computation time significantly.

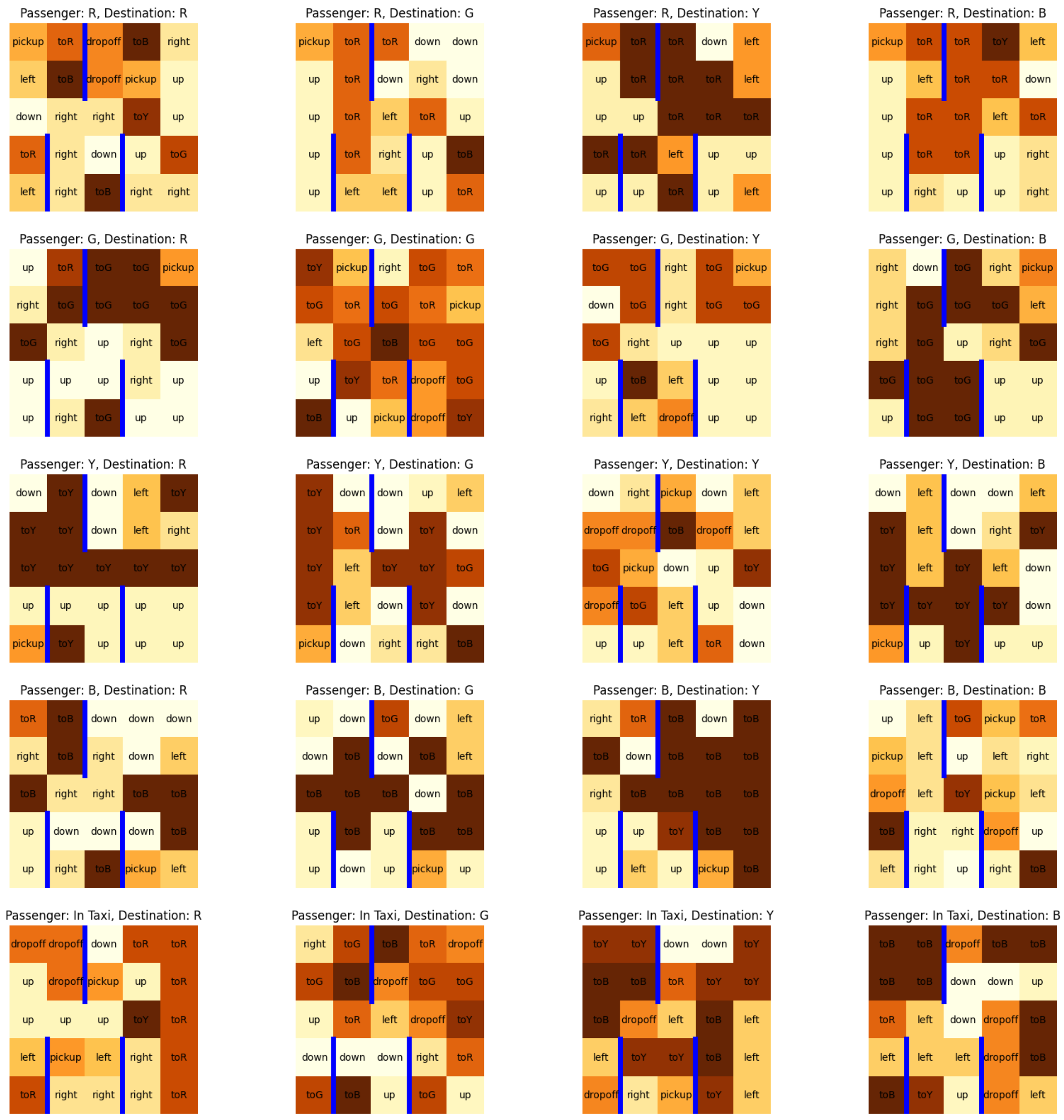


Figure 4: Learned option for each state

- As seen above, we can see that when the task is to go to the passenger at any location, not every state chooses to use an option, but a significant number of states prefer options over primitive actions as seen in most cases. There are still some states in the Q-map that haven't learned properly. This could be because, for the current task, that state might not have been visited often, due to a simpler path or not sampling start location over there.
- There are still a few states, such as when the passenger is in R and Destination state is G [Fig 4 row 1, column 2], where primitive actions still dominate and training has not given the best possible solutions. It could be because the option is not visited much in some states, which means that the Q-value for the option is not updated there in SMDP Q-Learning. This is expected to improve in Intra-Options Q-Learning since an update is done for each state that the agent passes through during the option execution as well.
- In the cases when the passenger is in taxi, it usually starts at one of the 4 goal locations. As we can see, in every case, it prefers to choose an option in those states, thus leading to it being updated regularly but irregular updates of the other states leading to some random

actions in a few states. This should also be improved in Intra-Options Q-Learning.

- The mean reward after learning for 10000 episodes was **9.44** averaged over 5 runs.

- **Intra-Option Q-Learning**

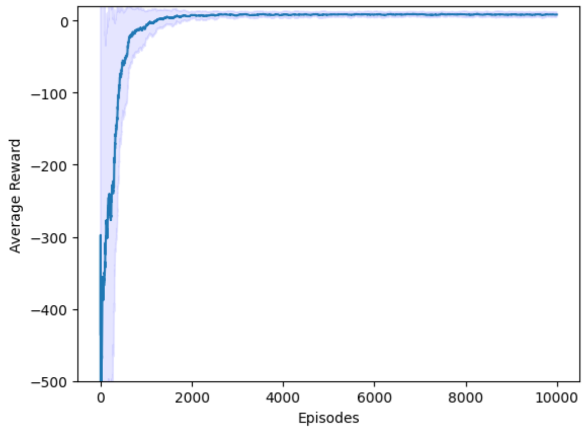


Figure 5: Mean Score window(size 100) plot over 10000 episodes

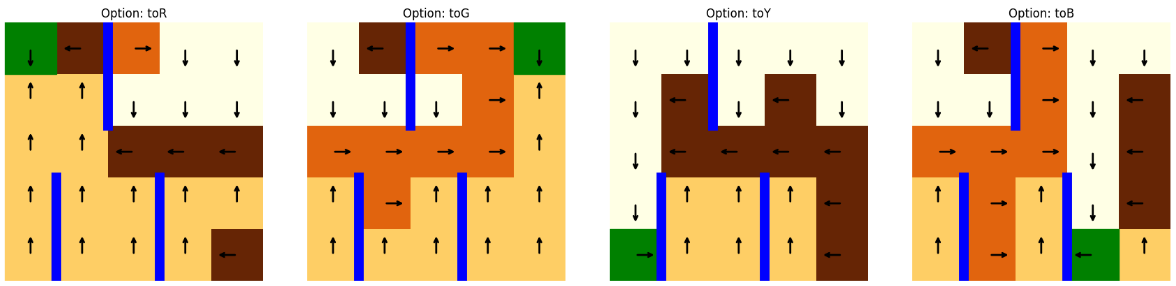


Figure 6: Learned Q-map for each option

Like the previous case in SMDP, each option has learned to navigate really well to reach the corresponding goal locations. The same pseudo-rewards were given to speed up the learning. Consistently, it was giving faster results compared to without pseudo-rewards.

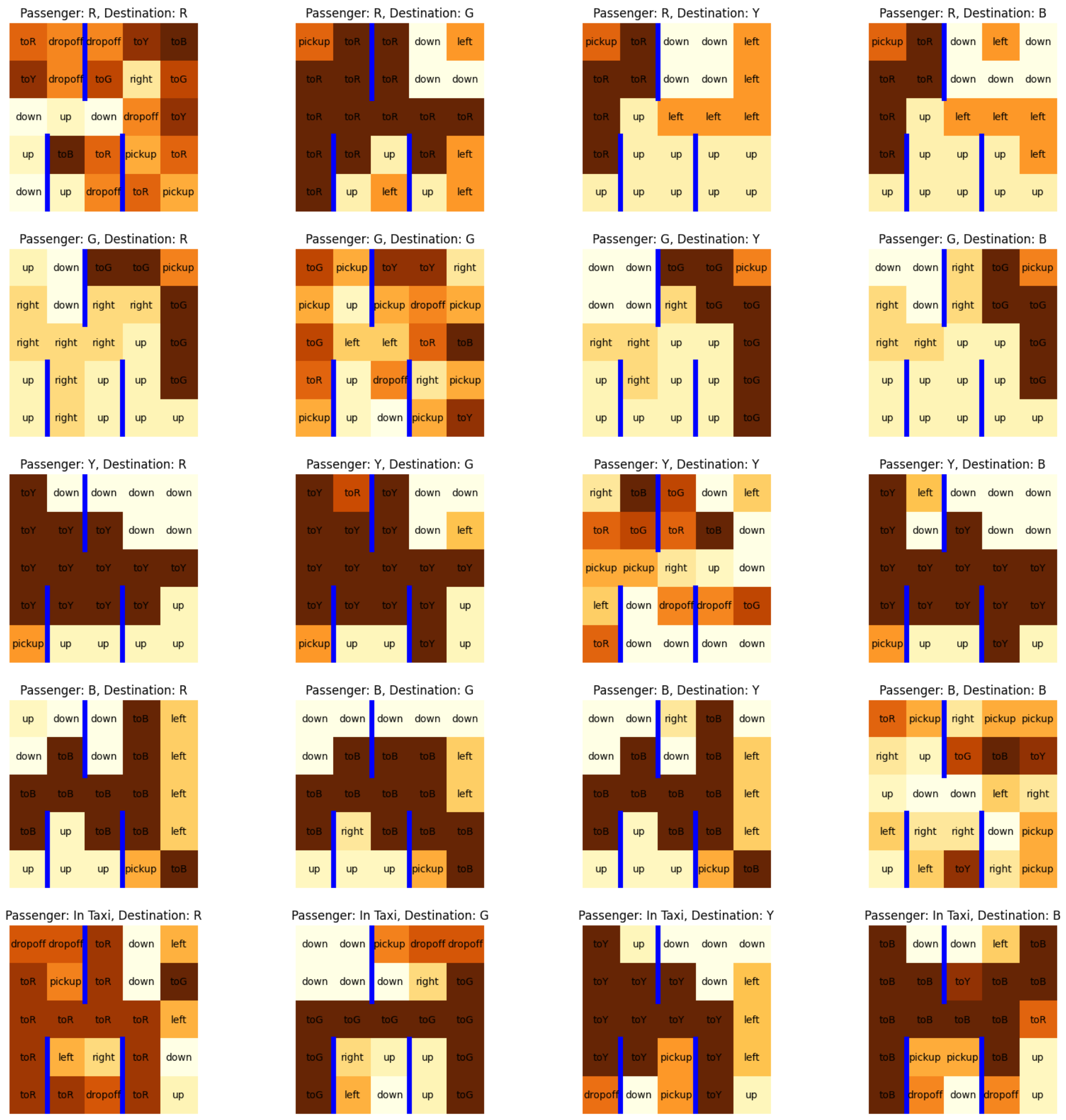


Figure 7: Learned option for each state

– As seen in the above figure, this prefers options compared to primitive actions in most cases, especially when picking up a passenger. There are also nearly no selection of the wrong primitive action between pickup and dropoff in any of the states, unlike in SMDP. This could be attributed to the update in every step of an option, which means that the agent uses experience more efficiently.

– One reason why options might be preferred more in certain states over primitive actions is so the agent can avoid the pickup/drop actions in neighbouring states which penalize the agent. In the states adjacent to these states, options generally dominate over primitive actions. This caused a few random pickup/drop actions in SMDP Q-Learning in states that were barely visited outside of option executions. However, Intra-Options solves this issue by having updates even when the state is visited during option execution. It may be noted that the incorrect drop/pickup actions are found here mainly in states that would not be passed through for the dominant option execution since these are the only less-updated states now.

  - Whenever the passenger is picked up, as we can see from the last row of the Q-map, those states always choose to use an option instead of an action. Since we update in every step of the option, the option for each state is learnt much better compared to SMDP. This clearly shows the advantage of Intra option Q-learning over SMDP Q-Learning.
  - The mean reward after learning for 10000 episodes was **8.98** averaged over 5 runs.

3.2 Options Set 2

Primitive Actions: There are 6 discrete deterministic actions:

- 0: move south
- 1: move north
- 2: move east
- 3: move west
- 4: pick passenger up
- 5: drop passenger off

Options:

- The options are defined at 3 different levels. At the top level, we learn to choose between `get` and `put`, which refer to getting and putting a passenger.
- At the second level, we choose either to navigate to one of the 4 goals (or) to choose a primitive action between pickup/drop based on the option selected at the top level.
- If a navigate action is chosen at the second level, then we choose between 4 primitive actions which specify the direction of motion for the taxi.

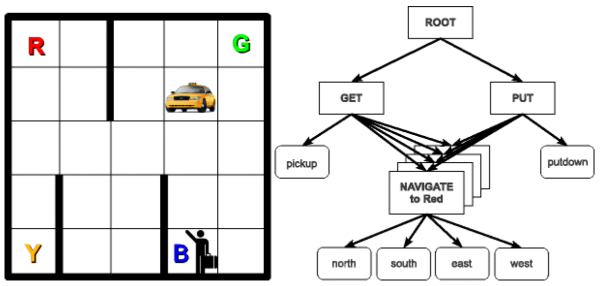


Figure 8: Taxi problem and an action hierarchy

The hierarchy of options is shown in this image. These options were chosen based on the class slides and existing literature. This image is from the paper Bayes-adaptive hierarchical MDPs

3.2.1 Code Snippets

Since only the options vary from the previous implementation, only that snippet has been provided here. Note that Intra-Options Q-Learning is shown here.

Level 1 Option Execution:

```
def execute_level_1_selection(env, state, QL1, QL2, QL3, option, policy, TAU):

    if option < 2:
        optdone = False
        optact = 0
        counter = 0
        reward_bar = 0
        chosen_q = QL2[option]
        steps = 0
        while not optdone:
            optact = policy(chosen_q, state, TAU)
            next_state, reward, done, [QL2, QL3], pick_drop_flag, steps_level_2 = \
execute_level_2_options(env, QL2, QL3, state, option, optact, policy, TAU)
            reward_bar += reward * GAMMA**counter
            steps += steps_level_2
            chosen_q[state][optact] += ALPHA_OPTIONS * (reward + GAMMA * \
np.max(chosen_q[next_state]) - chosen_q[state][optact])
            if pick_drop_flag:
                optdone = True
                QL1[state][option] += ALPHA_Q * (reward - QL1[state][option] + \
GAMMA * np.max(QL1[next_state]))
            else:
                optdone = False
                QL1[state][option] += ALPHA_Q * (reward - QL1[state][option] +\
(QL1[next_state][option]))
            counter += 1
            state = next_state
        QL2[option] = chosen_q
        return next_state, reward_bar, done, [QL1, QL2, QL3], steps
```

Level 2 Option Execution

```
def execute_level_2_options(env, QL2, QL3, state, ol1, ol2, policy, TAU):
    optdone = False
    optact = 0
    if ol2 == 0: # pickup/dropoff
        if ol1 == 0:
            chosen_action = 4
        elif ol1 == 1:
            chosen_action = 5
        steps = 1


        next_state, reward, done, _, _ = env.step(chosen_action)
        state_decoded = decode_state(env, state)
        next_state_decoded = decode_state(env, next_state)
        PICKUP_REWARD = 30 if chosen_action==4 and state_decoded[2]!=4 \
    and next_state_decoded[2]==4 else 0
        QL2[ol1][state][ol2] += ALPHA_OPTIONS * \
    (reward + PICKUP_REWARD + GAMMA * np.max(QL2[ol1][next_state]) -\
    QL2[ol1][state][ol2])
        return next_state, reward, done, [QL2, QL3], True, steps

    else: # getR, getG, getY, getB
        reward_bar = 0
        chosen_action_index = get_put_options.index(ol2) - 1
        counter = 0
        current_state = state
        chosen_q = QL3[chosen_action_index]
        while not optdone:
            state_value = 5 * decode_state(env, state)[0] + \
        decode_state(env, state)[1]
            optact = policy(chosen_q, state_value, TAU)
        next_state, reward, done, _, _ = env.step(optact)
        next_state_value = 5 * decode_state(env, next_state)[0] + \
    decode_state(env, next_state)[1]
        reward_bar += reward*GAMMA**counter

        action_array = []
        for i in range(len(QL2)):
            for j in range(len(QL2[i][state])):
                if j!=0:
                    choose_option_array = policy(QL3[j-1], state_value, TAU)
                    action_array.append([i, choose_option_array])
        if decode_state(env, next_state)[:2] == goal_states[chosen_action_index]:
            optdone = True
            QL2[ol1][state][ol2] += ALPHA_OPTIONS * \
                (reward - QL2[ol1][state][ol2] + \
                 GAMMA * np.max(QL2[ol1][next_state]))
            for state_set in action_array:
                QL2[state_set[0]][state][state_set[1]] += ALPHA_OPTIONS * \
                    (reward - QL2[state_set[0]][state][state_set[1]] + \
                   GAMMA * np.max(QL2[state_set[0]][next_state]))
        else:
            optdone = False
            QL2[ol1][state][ol2] += ALPHA_OPTIONS * \
                (reward - QL2[ol1][state][ol2] + \
                 GAMMA * (QL2[ol1][next_state][ol2]))
            for state_set in action_array:
                QL2[state_set[0]][state][state_set[1]] += ALPHA_OPTIONS * \
                (reward - QL2[state_set[0]][state][state_set[1]] + \
                 GAMMA * (QL2[state_set[0]][next_state][state_set[1]]))
        GOAL_REWARD = 30 if optdone else 0
        IN_TAXI_REWARD = -2 if ol1 == 1 and decode_state(env, state)[2]==4 else 0
        chosen_q[state_value][optact] += ALPHA_OPTIONS_1 * (reward + \
    GOAL_REWARD + IN_TAXI_REWARD + GAMMA * np.max(chosen_q[next_state_value]) - \
    chosen_q[state_value][optact])

        counter += 1
        state = next_state
    QL3[chosen_action_index] = chosen_q
    steps = counter
    return next_state, reward_bar, done, [QL2, QL3], False, steps
```

3.2.2 Results & Discussion

- **SMDP Q-Learning**

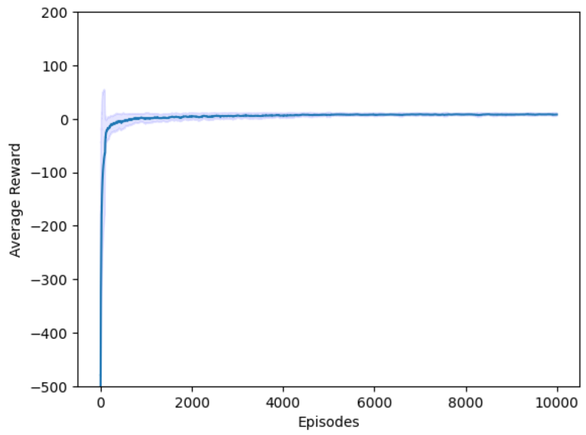


Figure 9: Reward Plot with new options

The mean reward for this set of options using SMDP Q-Learning is **6.452194518585356**
From Figure 10, we can observe that

- At the top level, our options choose between `get` and `put`. This could have been a step that can just be done without any learning using the state details. We wanted to explore how well these options could be learnt. As seen in the images, it learnt the options perfectly.
- When a passenger is to be picked up, all the states are choosing the option get, which has sub-options to pickup and navigate to the passenger locations.
- Similarly, when the user is in the taxi, the start states would be either R,G,B or Y. So it directly chooses the put option. Put chooses between navigating to the goal and dropoff of a passenger. This would be the ideal actions once a passenger is picked up.
- This results in other states not learning to put, when the passenger is in the car, because once the option is chosen in one of the designated locations, it navigates straight to the destination, drops the passenger and terminates, thus not updating the other states in SMDP Q-Learning.

From Figure 11, we can observe that

- As seen from the previous case, get is chosen only when the passenger is outside the taxi in either of R,G,B or Y.
- When get is chosen, we see that pick option is only chosen at the passenger's location and the other locations only use getting to the passenger's location.
- Even in states where, it seems like the wrong option is chosen, it is generally 1-3 extra steps to reach the goal as they are directed to a state that is moving towards the goal.

From 12, we can observe that

- As seen from the level 1 options, put is chosen only when the passenger is inside the taxi. It is chosen right at the state where the passenger is picked up.
- The plots are very clear and directly choose the option to go to their corresponding goal. Due to the same reason as in level 1 options, the other states don't learn here.

These options clearly explains how the whole hierarchy works and explains the decision making of the whole problem at all levels

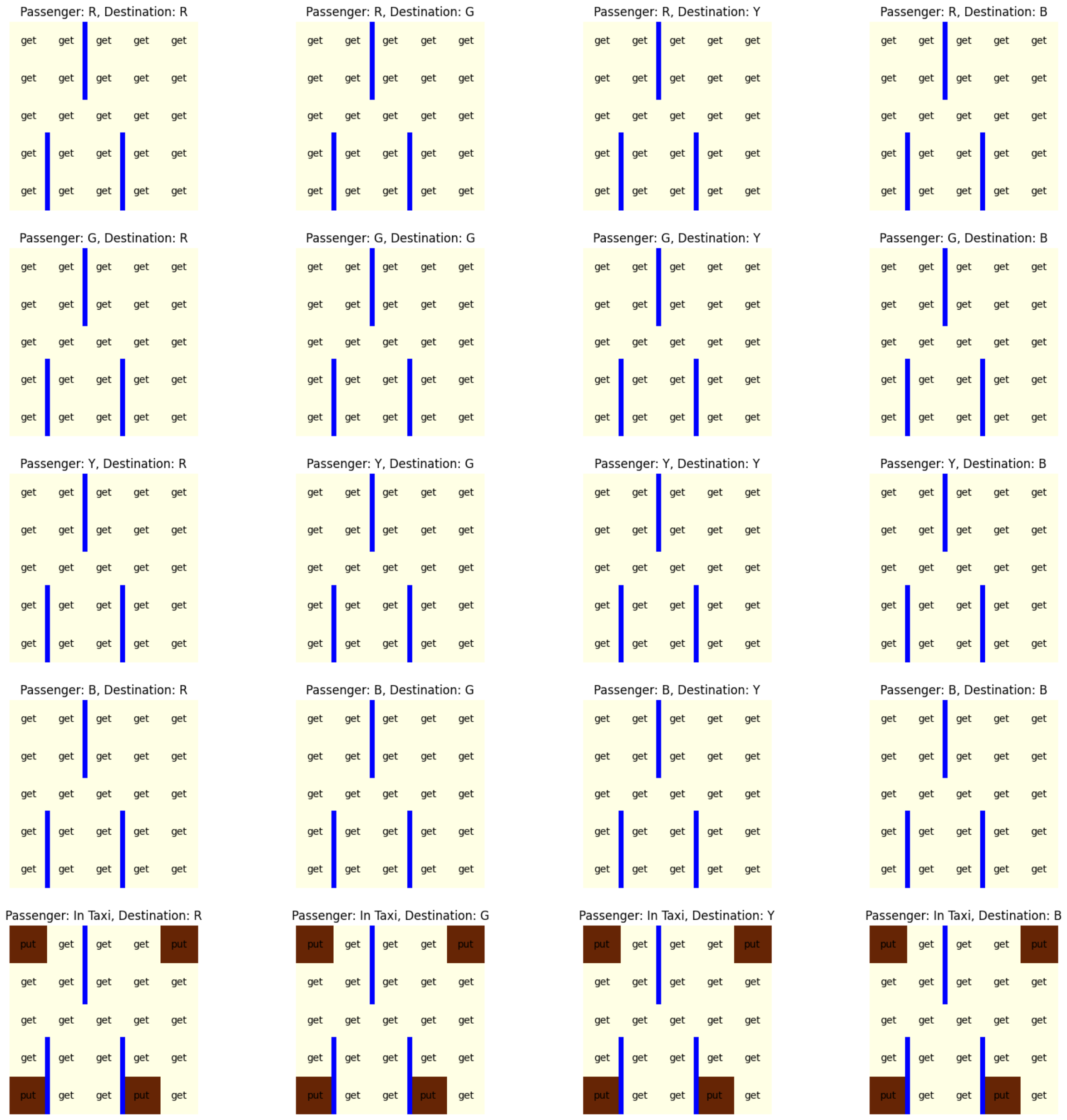


Figure 10: Level 1 options

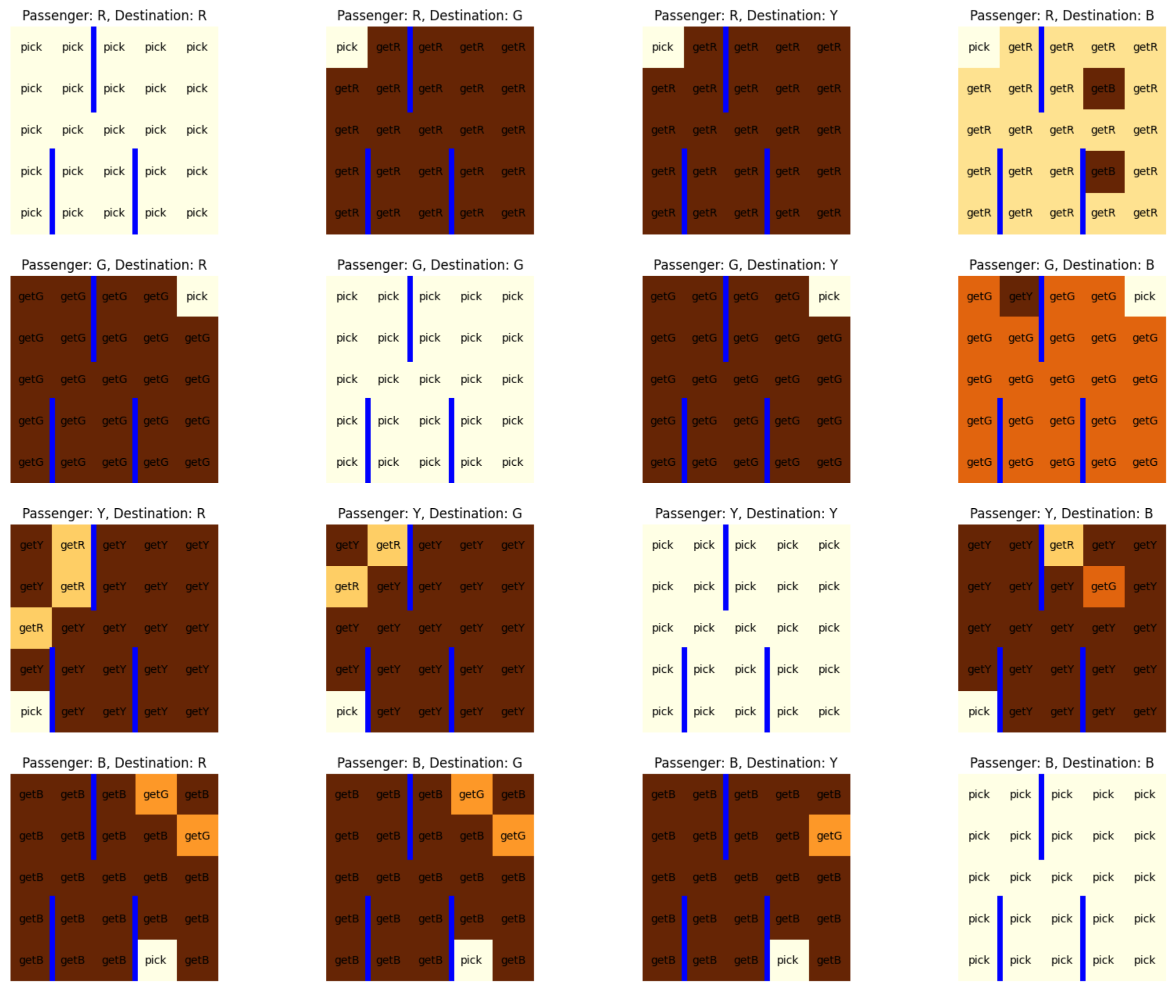


Figure 11: Level 2 options: Get

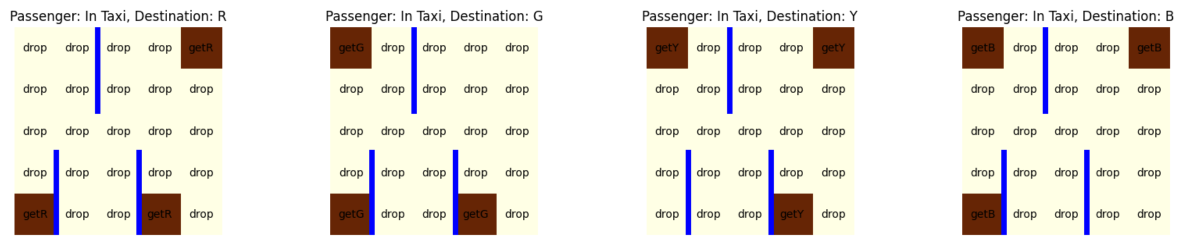


Figure 12: Level 2 options: Put

- **Intra Option Q-Learning**

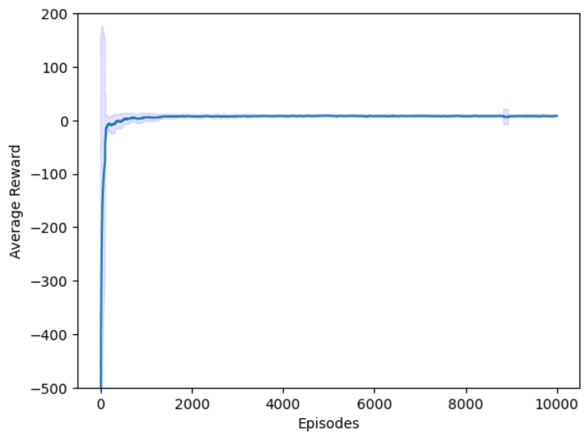


Figure 13: Reward Plot with new options

The mean reward for these options using Intra Options Q-Learning is **7.677131388235193**

- As seen in Fig 14, the top level has learned the same options. Both SMDP and IOQL have learned them in the same way.

- In Fig 15, the options have learned well. These options are not as good compared to their SMDP counterparts. There are many states which take a longer route to the goal. This might not be desirable all the time.
- This could be again because of updates in IOQL updating every state visited during the execution of an option.

- In Fig 16, we can see a significant difference between Intra Option Q-Learning and SMDP Q-Learning.
- When a taxi picks up a passenger, he still starts only at 3 different states out of R,G,B,Y. At all these states the options are learnt properly in all 4 plots.
- Due to IOQL learning, even when an option is executing, the other states in between also learn some policies unlike in the case of SMDP.
- They are not always the best policy as seen in many states above, but if we are to follow the learned Q-Map greedily, we would never have to choose an option in these states until termination of the previous option which already takes you to the goal.

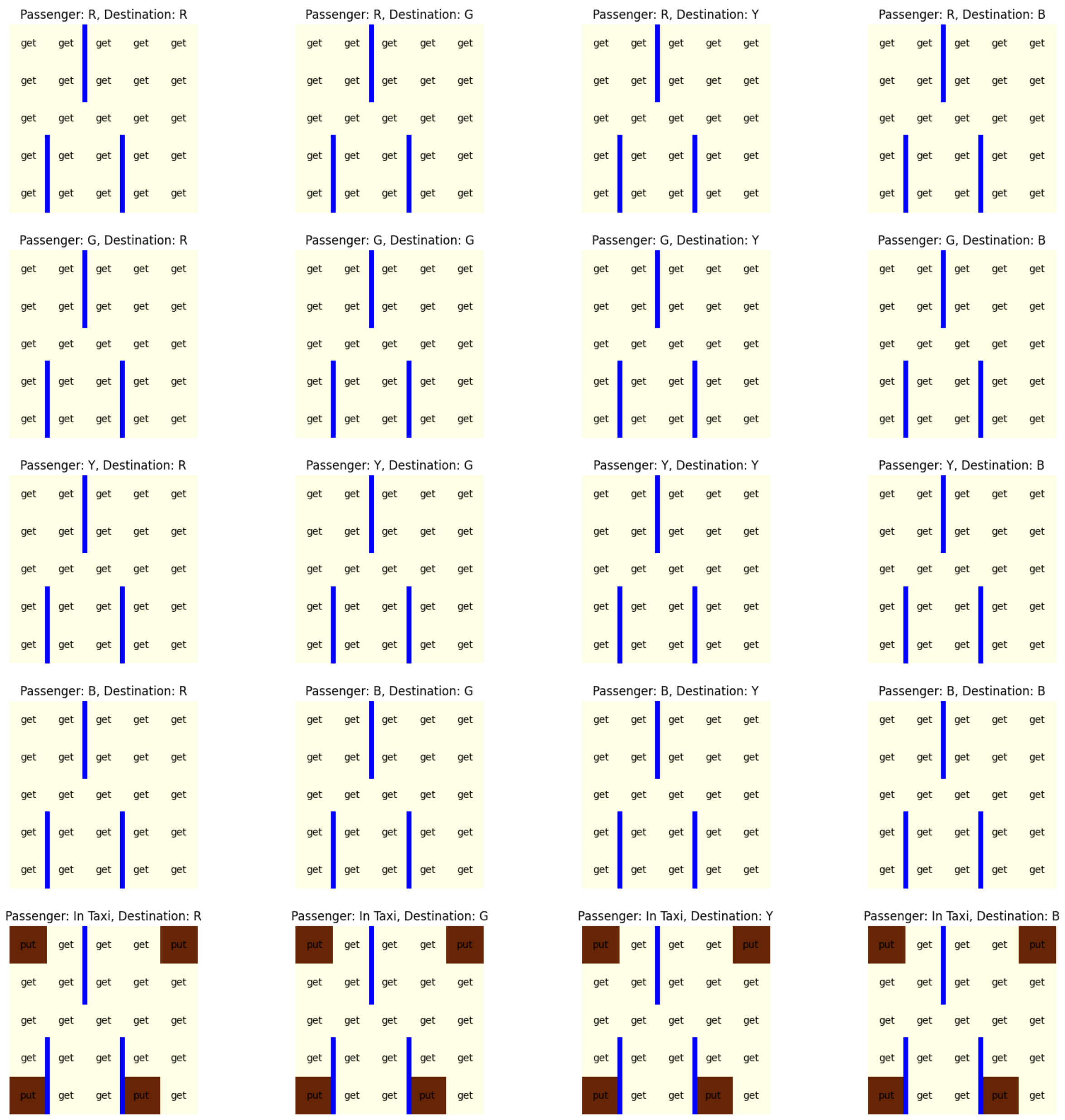

Passenger: R, Destination: R
Passenger: R, Destination: G
Passenger: R, Destination: Y
Passenger: R, Destination: B
Passenger: G, Destination: R
Passenger: G, Destination: G
Passenger: G, Destination: Y
Passenger: G, Destination: B
Passenger: Y, Destination: R
Passenger: Y, Destination: G
Passenger: Y, Destination: Y
Passenger: Y, Destination: B
Passenger: B, Destination: R
Passenger: B, Destination: G
Passenger: B, Destination: Y
Passenger: B, Destination: B
Passenger: In Taxi, Destination: R
Passenger: In Taxi, Destination: G
Passenger: In Taxi, Destination: Y
Passenger: In Taxi, Destination: B
get
put

Figure 14: Level 1 options

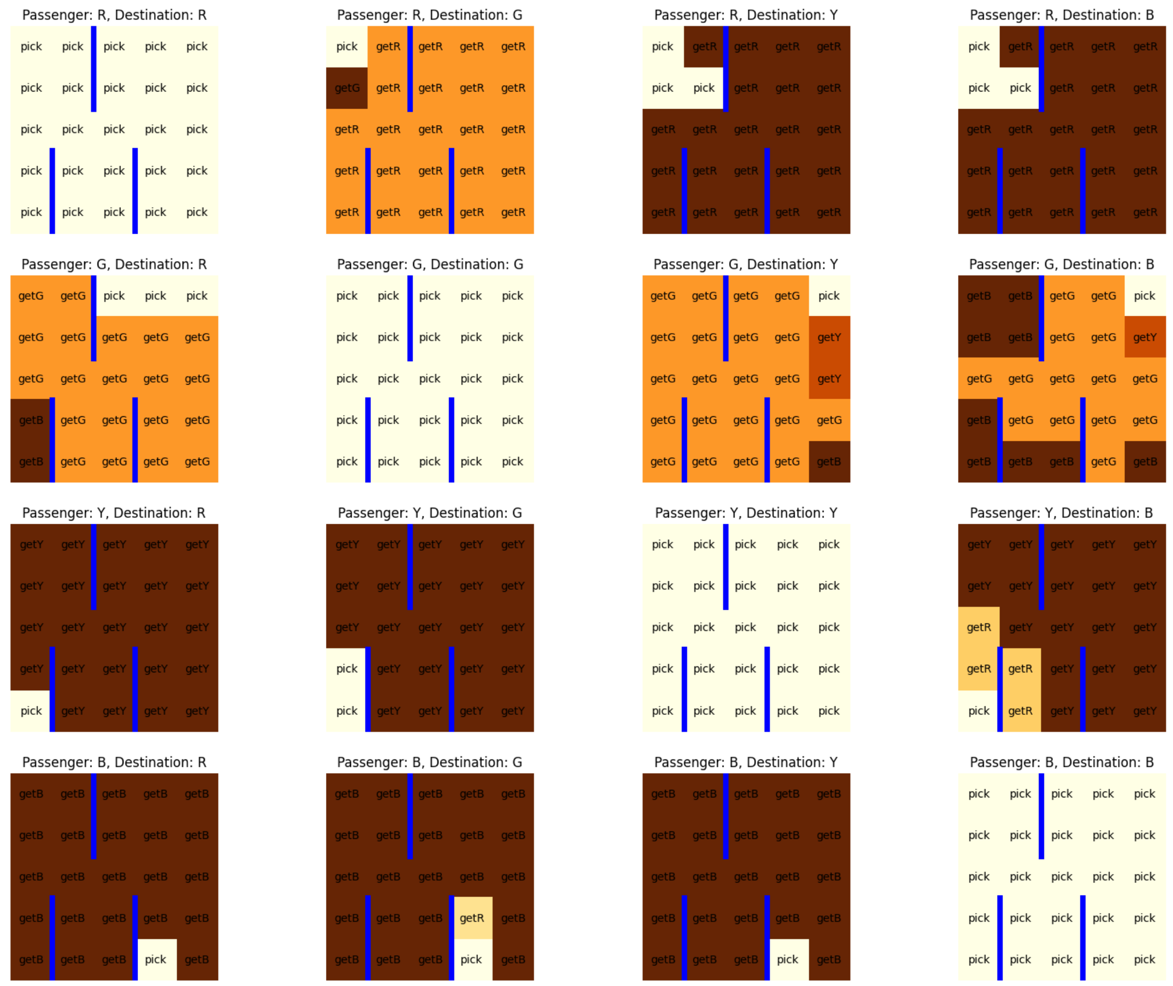


Figure 15: Level 2 options: get

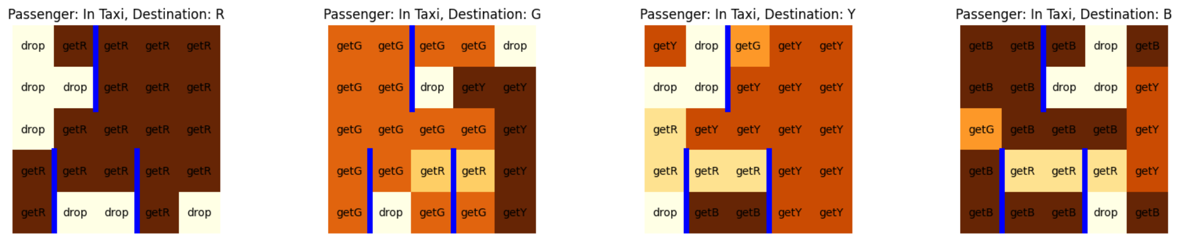


Figure 16: Level 2 options: put

3.3 Options Set 3

Primitive Actions: There are 6 discrete deterministic actions:

- 0: move south
- 1: move north
- 2: move east
- 3: move west
- 4: pick passenger up
- 5: drop passenger off

Options: Additionally, we have provided four options that take the agent "near" each of the goal states. This is done by defining regions containing all states from which the goal state are directly accessible, i.e. without any walls in between. Thus, the options are:

- 6: goR: takes the agent to region R (red)
- 7: goG: takes the agent to region G (green)
- 8: goY: takes the agent to region Y (yellow)
- 9: goB: takes the agent to region B (blue)

3.3.1 Code Snippets

Option Execution:

```
def execute_option(env, state, option, Q_select_option, \
Q_within_option, policy=softmax):

    optdone = False

    if option < len(prim_actions):
        next_state, reward, done, _, _ = env.step(option)
        optdone = done
        optact = option
        Q_select_option[state][option] += ALPHA_Q * (reward + GAMMA *\
    np.max(Q_select_option[next_state]) - Q_select_option[state][option])
        return next_state, reward, done, [Q_select_option, Q_within_option]

    if option >= len(prim_actions):
        reward_bar = 0
        optnum = options.index(option)
        counter = 0
        current_state = state
        while optdone == False:
            optact = policy(Q_within_option[optnum], state, TAU)
            next_state, reward, done, _, _ = env.step(optact)
            reward_bar += reward * (GAMMA ** counter)
            if decode_state(env, next_state)[:2] in regions[optnum]:
                optdone = True
                Q_select_option[current_state, option] += ALPHA_Q * (reward_bar -\
        Q_select_option[current_state, option] + GAMMA**counter * \
        np.max(Q_select_option[next_state]))
            REGION_REWARD = 30 if optdone else 0
            Q_within_option[optnum, state, optact] += ALPHA_OPTIONS * (reward + \
        REGION_REWARD + GAMMA * np.max(Q_within_option[optnum, next_state])\
          - Q_within_option[optnum, state, optact])
            counter += 1
            state = next_state
        return next_state, reward_bar, done, [Q_select_option, Q_within_option]
```

3.3.2 Results & Discussion

- **SMDP Q-Learning**

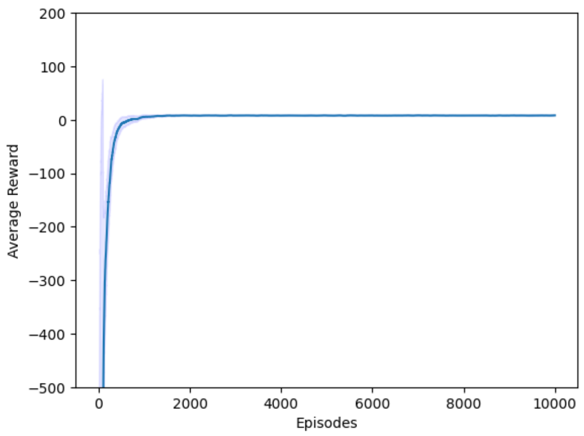


Figure 17: Mean Score window(size 100) plot over 10000 episodes

  – In Fig 18, we can see the regions R, G, Y, B outlined in lines of the colour of the goal state they are defined for. The regions have been defined as overlapping, so sometimes the option toR and toY may be interchanged, and also toG and toB may be exchanged. This is due to those common points being an entry gate-way to either of the regions
  – When the passenger is at one of the designated locations (pickup required), the first four rows show that options are preferred quite often in states outside of that location's region.
  – Within the region, the agent has correctly learned the primitive actions required to reach the goal state, and is properly picking/dropping the passenger as well.
  – When the passenger is in the taxi, a lot of states in between have random actions, because of the use of options in neighbouring states which cause these states to not be updated. This is a downside of SMDP Q-Learning: some states are very rarely updated causing them to prefer wrong actions.
  – Selection of these options also show how these algorithms learn a sequence of options to complete the execution of a task similar to learning a sequence of primitive actions.
  – Fig 19 shows the policy within option toR, and it is clear that the agent has learned to navigate itself to the states outlined in red. Similarly, the policies within the options toY, toG and toB have learned well to go to the corresponding regions. This was achieved by giving the agent a pseudo-reward of +30 for reaching the desired region.
  – The obtained mean reward after training for 10000 episodes averaged over 5 runs was **7.67**.

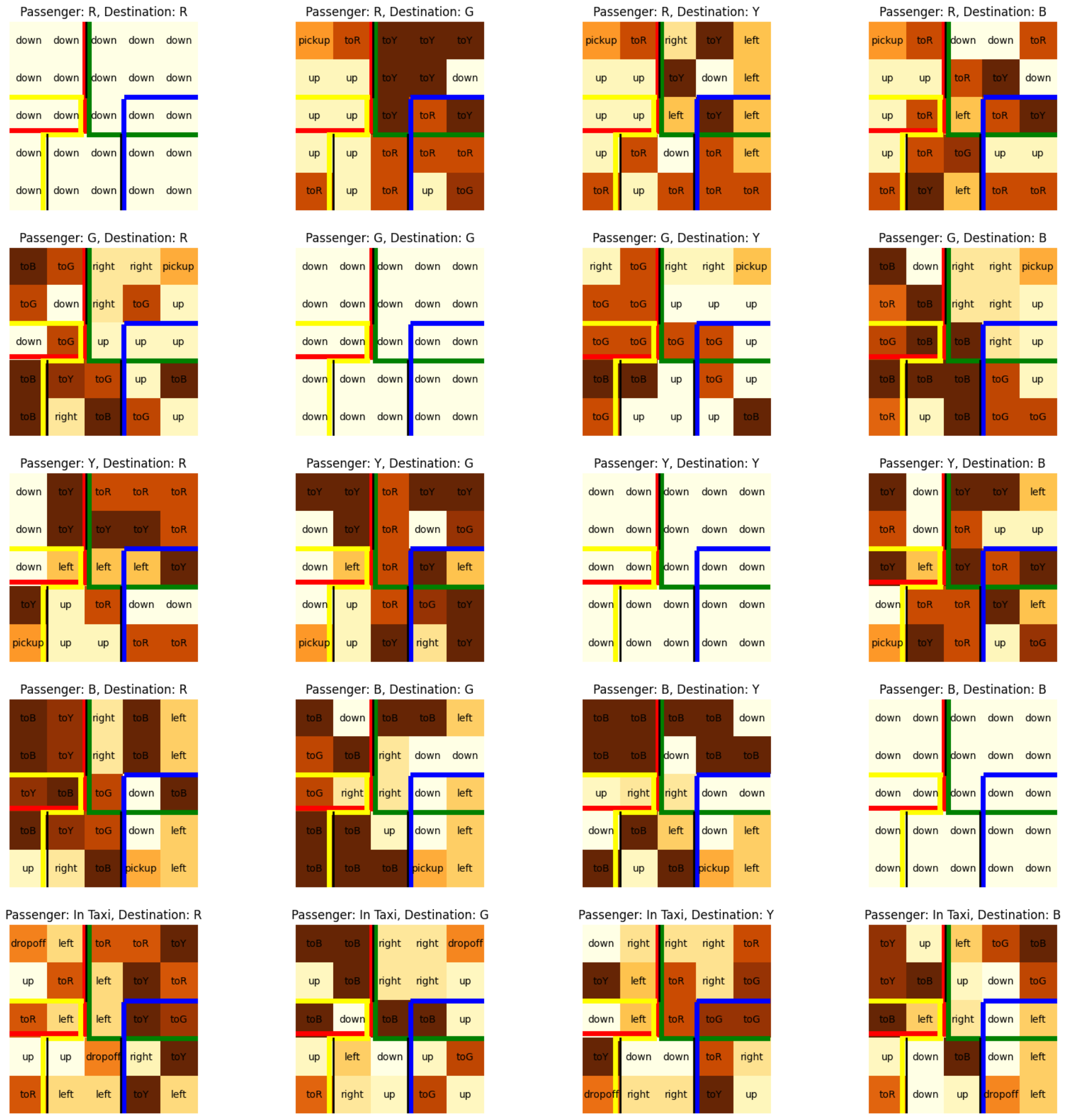


Figure 18: Main policy: SMDP QL

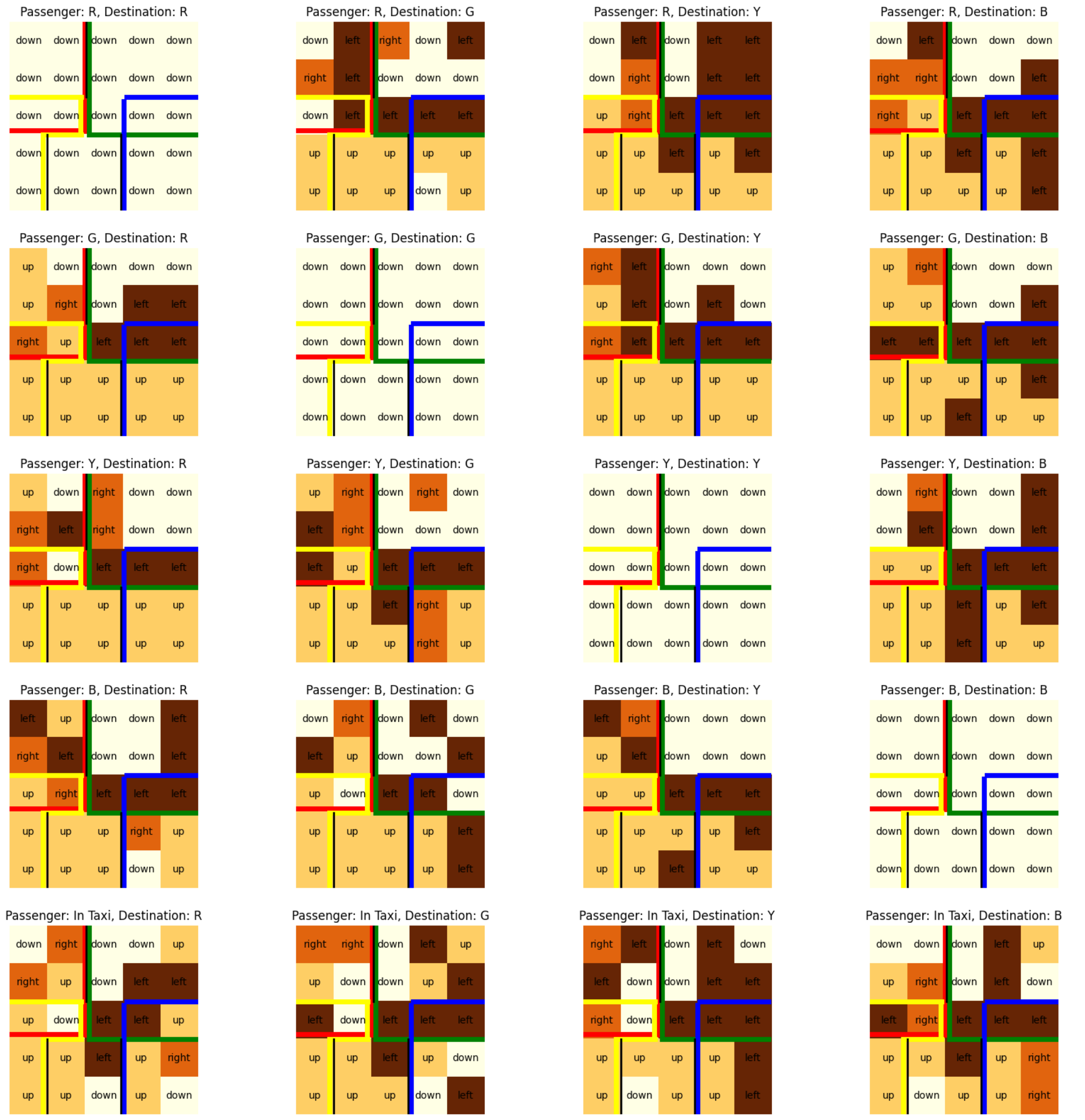


Figure 19: Sub-policy to navigate to region R

- **Intra-Option Q-Learning**

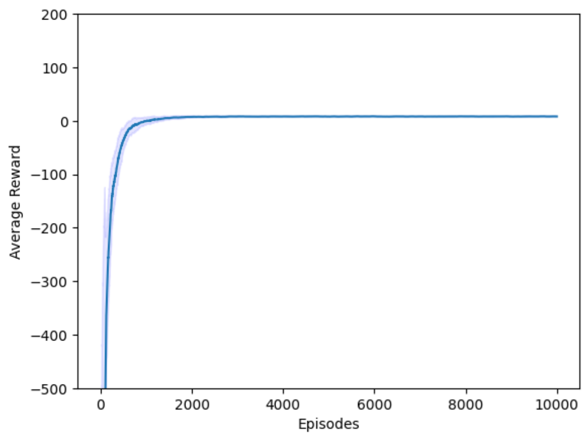


Figure 20: Mean Score window(size 100) plot over 10000 episodes

  – The sub-option policies were learned as well as in the SMDP Q-Learning case, which was expected as both use the same Q-Learning updates for the option policy. This again used the same pseudo-reward of +30 for reaching the region.
  – In Fig 22, we see that the agent rarely prefers using options and when it does it usually does it irrespective of the passenger location and destination, it just uses it to go to the centre row.
  – This could be because of the updates done to every option that would have taken the same action in Intra-Options, which cause the options to compete with each other and confuse the agent.
  – Here, since overlapping regions were used, the agent might have not learned as well because the same states would have high Q-values, say for reaching red and yellow regions. This also causes competing options and reduces the efficiency of learning.
  – The obtained mean reward after training for 10000 episodes averaged over 5 runs was **7.87**.
  – As a possible improvement to this, we tried defining non-overlapping regions. While this did improve the mean reward to **8.12**, the main policy learned still did not significantly change. On removing the similar options update step, however, options were seen to be preferred more. Thus, the multiple updates in Intra-Options hinders learning of options in case of these ”region”-based options.

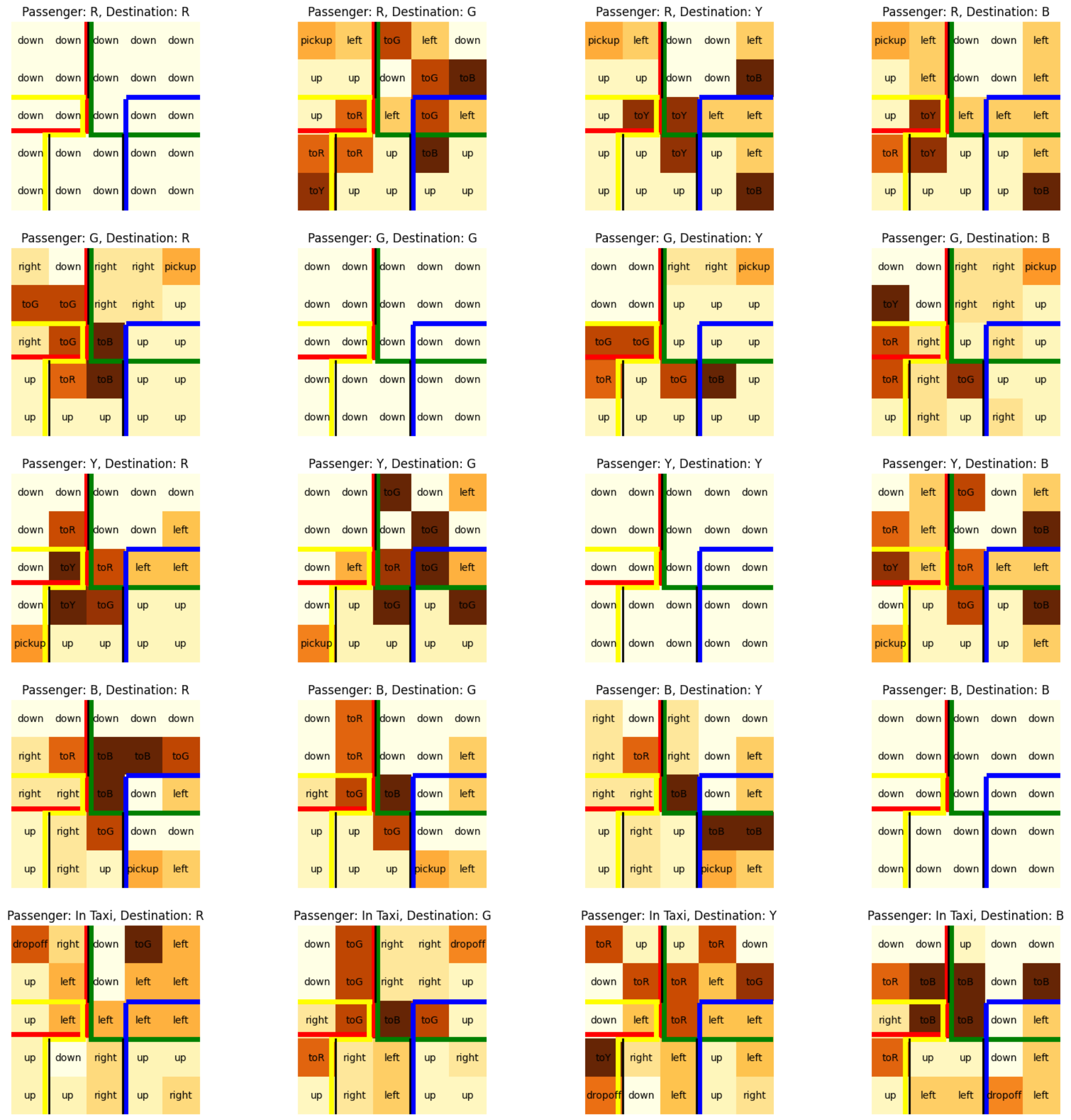


Figure 21: Main policy: IOQL

3.4 SMDP vs Intra-Options Q-Learning

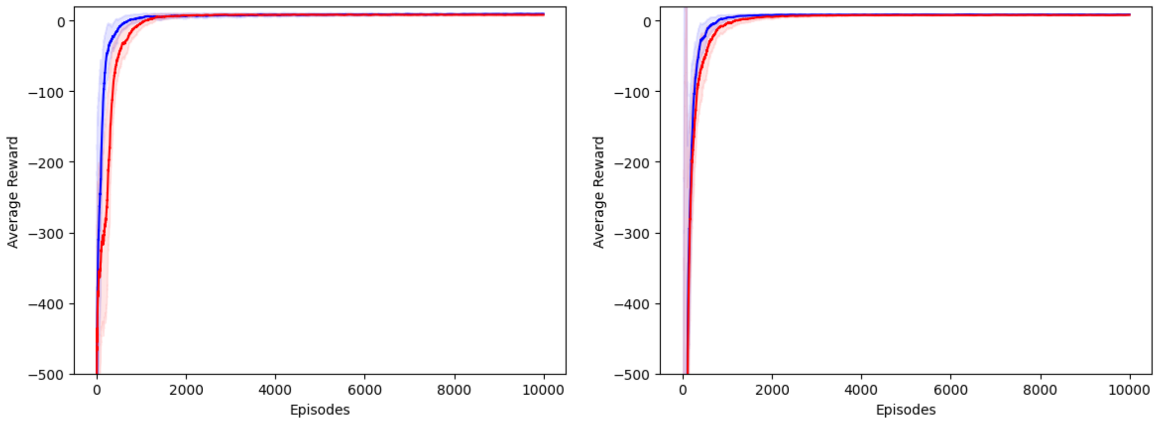


Figure 22: Mean Reward for SMDP (blue) and Intra-Options (red) Q-Learning for Option 1 (left) and Option 3 (right)

- For most sets of options, Intra-Options Q-Learning would learn to prefer non-primitive options over primitive actions for more states. This is because Intra-Options Q-Learning performs updates on the option Q values at every step and for each state along the path, as opposed to SMDP Q-Learning which updates the option Q value only after termination and only for the state in which the option is initiated.
- Intra-Options Q-Learning uses experience from the option execution more efficiently than SMDP Q-Learning. It requires fewer samples to learn the policy, leading to lesser need for exploration and faster convergence.
- SMDP Q-Learning gives better mean reward in the first set of options, but Intra-Options performs better in the other two cases. However, as shown in Fig 22, SMDP Q-Learning learns slightly faster than Intra-Options Q-Learning.
- For the last set of options, because of the definition of overlapping regions and the multiple options updates which cause the options to compete with each other, Intra-Option does not learn to use options much. However, it does give a better performance than SMDP in terms of mean reward.

Thus, based on the defined options structure, Intra-Options and SMDP Q-Learning may behave differently, and the choice between both must be done on an application-basis.