CS6700 Reinforcement Learning
Programming Assignment-3

K V Vikram (CS19B021), Kanishkan M S (ME19B192)

April 22 2023

Introduction

This assignment familiarized us with the popular Hierarchical RL techniques of SMDP Q-learning and Intra option Q-learning methods and were asked to implement these techniques on a simple taxi domain environment

There are 500 discrete states since there are 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 there are 400 states that can actually be reached during an episode. The missing states correspond to situations in which the passenger is at the same location as their destination, as this typically signals the end of an episode. Four additional states can be observed right after a successful episodes, when both the passenger and the taxi are at the destination. This gives a total of 404 reachable discrete states.

1. **Passenger locations:** 0: R(ed); 1: G(reen); 2: Y(ellow); 3: B(lue); 4: in taxi
2. **Destinations:** 0: R(ed); 1: G(reen); 2: Y(ellow); 3: B(lue)
3. **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 $\gamma = 0.9$

Wandb Sweep

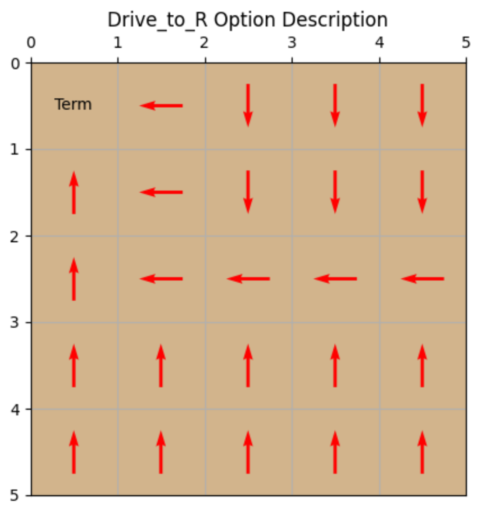
For hyper parameter tuning, a wandb sweep is made that runs 100 different hyper parameter combinations using bayesian hyperparameter optimization with each trained for 20000 episodes. The value range considered were: α from 0.2 to 0.7 and ϵ from 0.02 to 0.25. Results of all these tests can be accessed at these links: SMDP-Part1,Intra-Part1,SMDP-Part2,Intra-Part2.

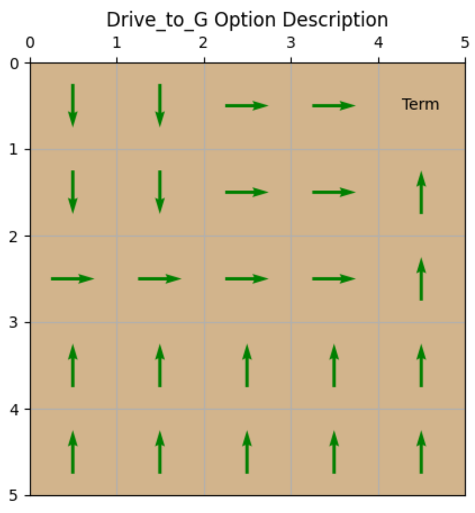
Performance metric: Average reward obtained in 1000 test episodes post training

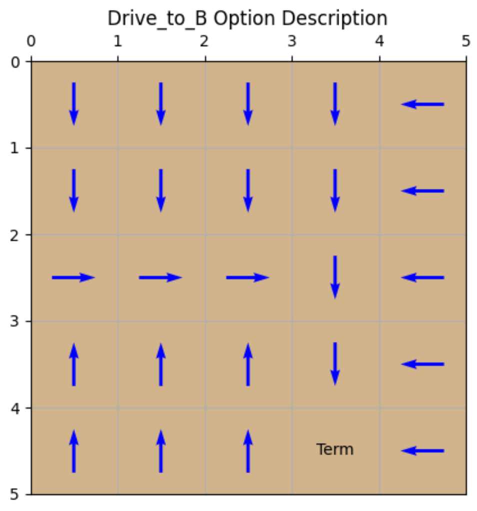
1 HRL with given set of options

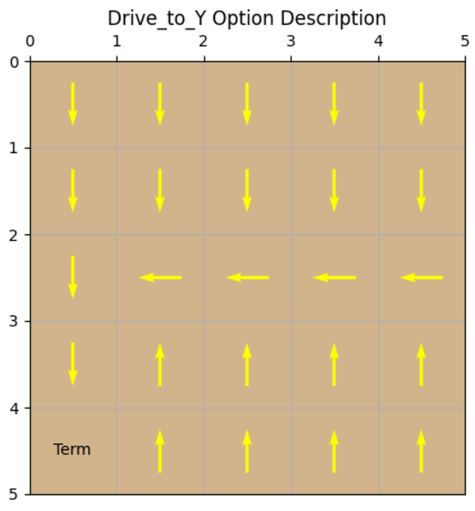
Option description

- These below diagrams show the given set of options which are to move the taxi to each of the four designated locations when the taxi is not already there.
- The options can be initiated in all states, except for the states where car's location is marked "Term".
- The options will be terminated in all states with the car's location marked "Term".









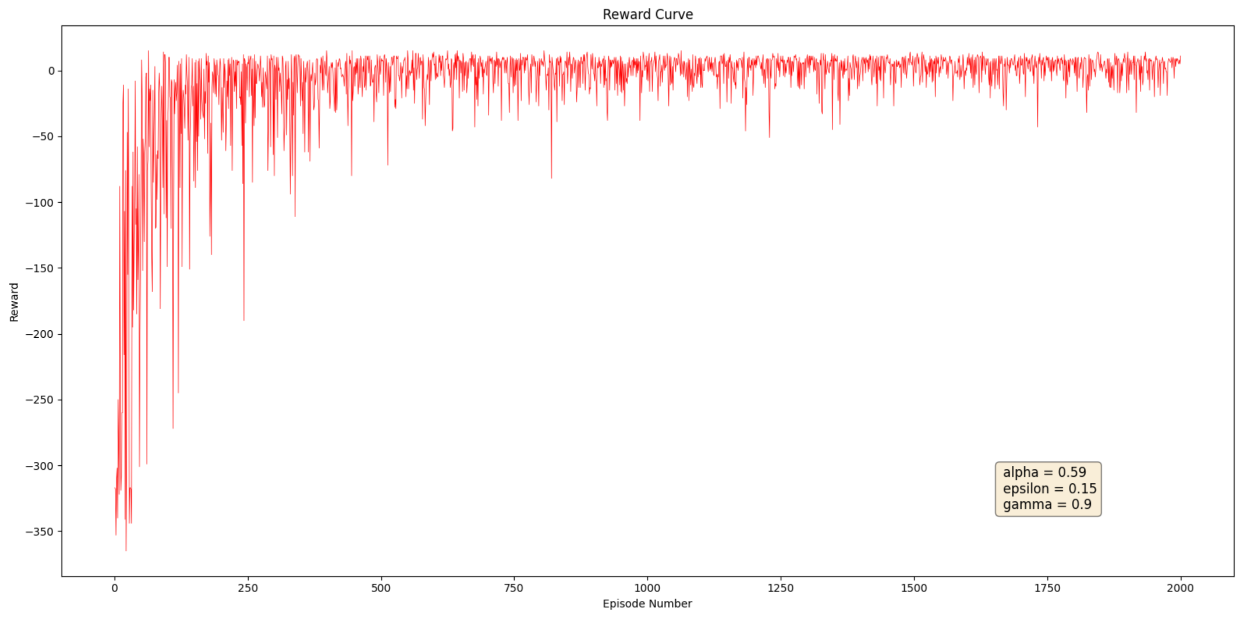
SMDP Q-Learning

The code used for implementing SMDP is given below. We can see that a running option o makes only 1 update. This update is for the q-value of $(initialstate, o)$ pair and this update is done only after the option terminates. Here, $initialstate$ is the state where o was started.

```
# Iterate over epcount episodes
for epidx in tqdm(range(epcount)):
  state = env.reset()
  done = False
  eps_reward = 0

  # While episode is not over
  while not done:
    # Choose action/option. Note : option should be allowed at the current co-
                                    ordinates
    st_coords = tuple(env.decode(state))[:2]
    dis_opt = (dis_opts[st_coords] if st_coords in dis_opts.keys() else None)
    action = egreedy_policy(q_values, state, epsilon, rg, dis_opt)
    # Checking if primitive action
    if action < 6:
      # Perform regular Q-Learning update for state-action pair
      next_state, reward, done, _ = env.step(action)
      q_values[state, action] += alpha * (reward + gamma*np.max(q_values[
                                    next_state]) - q_values[state, action])
      uf_values[state,action] += 1
      state = next_state
      eps_reward += reward
    else:
      # find the corresponding function for the option
      opt_func = opt_func_map[action]
      reward_bar = 0

      initial_state = state
      opt_steps = 0
      optdone = False
      while (optdone == False):
        optact, _ = opt_func(env, state)
        next_state, reward, done, _ = env.step(optact)
        _, optdone = opt_func(env, next_state)
        reward_bar = gamma * reward_bar + reward
        opt_steps += 1
        state = next_state
        eps_reward += reward

      # SMDP Q-Learning Update for options. We update the q-value of the (
                                    initial_state, option) pair at the end of
                                     option execution.
      q_values[initial_state, action] += alpha * (reward_bar + (gamma **
                                    opt_steps) * np.max(q_values[state]) -
                                    q_values[initial_state, action])
  uf_values[initial_state,action] += 1
  ep_rewards_list.append(eps_reward)
```

Tuned hyperparameters for this configuration: α=**0.59**, ϵ=**0.15**.

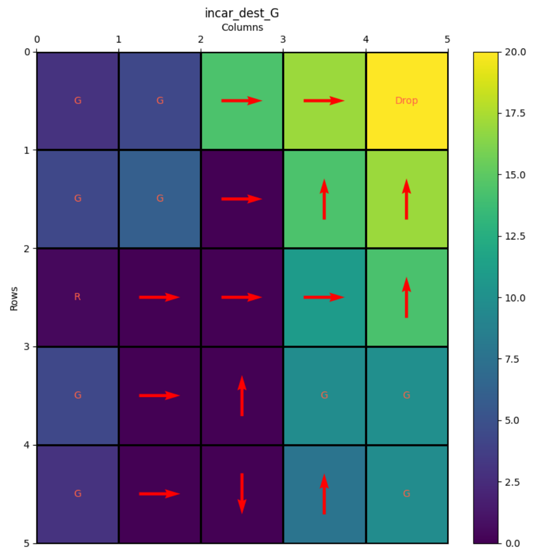
Average reward curve



Visualizing the learnt Q values

The below plotted heatmaps show actions taken at each locations clearly. Therefore, we will provide only non-trivialities in the descriptions. The Q value plots are very informative on their own. Arrows indicate primitive actions and letters indicate options. Here, the letter "X" stands to "Drive_to_X" option where $X \in \{R, G, Y, B\}$.

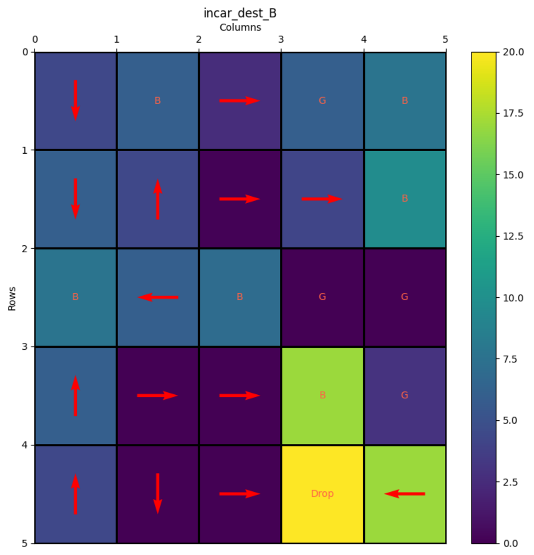
Passenger location: Inside car, Drop location: G



Inference

- The learnt policy chooses primitive actions and options almost in an equal manner
- There is a **"Drive_to_R"** option (not expected) at (2,0), this is due to infrequent updates at this location, as the car will directly go to the **Green** point, after picking up at **Red** and **Yellow** through the **Drive_to_G** option.
- If the pickup location is Blue, it shows an upward primitive action, and at the next step, option green is chosen, which is optimal. After picking up at **Red** and **Yellow** points, it directly takes the **Drive_to_G** option.

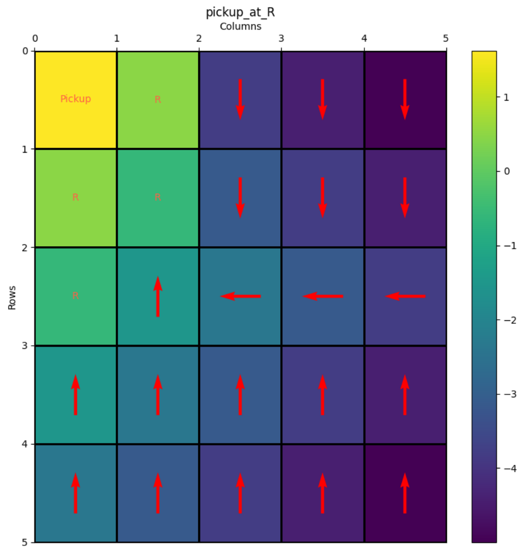
Passenger location: Inside car, Drop location: B



Inference

- Similar to previous case, the learnt policy chooses primitive actions and options almost in an equal manner.
- Post picking up at location Red, the policy chooses downward primitive actions twice and then takes the **Drive_to_B** option.
- Similarly post picking up at location Yellow, the policy chooses upward primitive actions twice and then takes the **Drive_to_B** option.

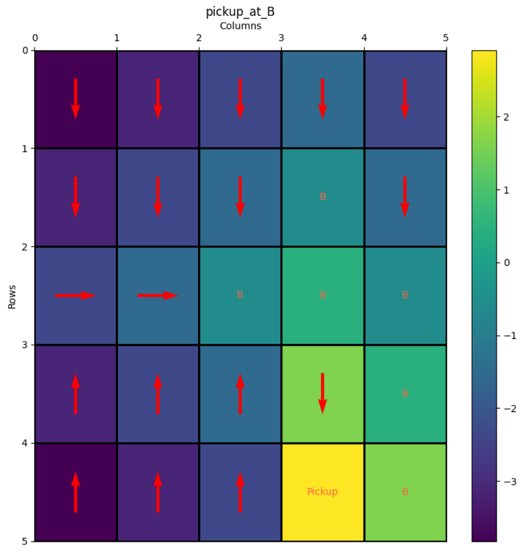
Passenger location: Pickup at R



Inference

- Incase of pickup situations, the learnt policy performs well in all three cases.

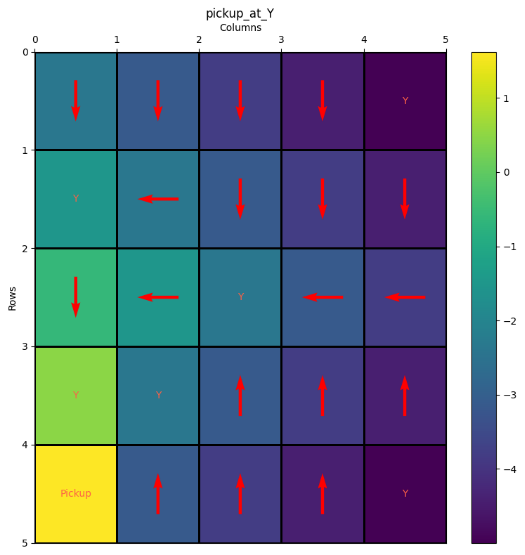
Passenger location: Pickup at B



Inference

- Choice of primitive actions are actually higher in these pickup cases.

Passenger location: Pickup at Y



Inference

- Similar output as the above two maps.

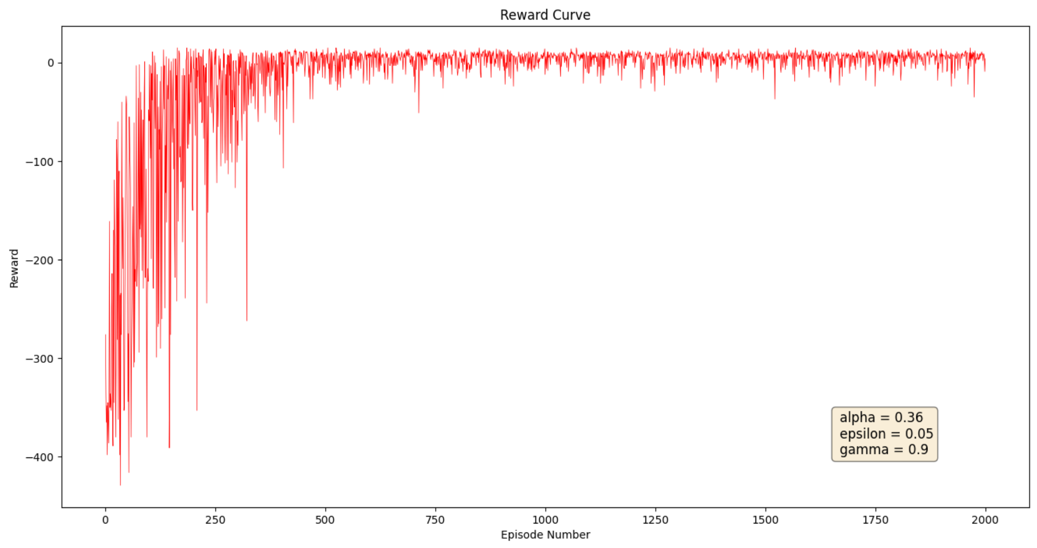
Intra-option Q-Learning

The code used for implementing Intra-option is given below. We can see that when an running option o chooses a at current state s, the q-value for (s, a) and the q-value for (o', a) for all options o' that choose action a at state s also get updated.

```
    # Iterate over epcount episodes
    for epidx in tqdm(range(epcount)):
      state = env.reset()
      done = False
      eps_reward = 0

      # While episode is not over
      while not done:
        # Choose action/option. Note : option should be allowed at the current co-
                                          ordinates
        st_coords = tuple(env.decode(state))[:2]
        dis_opt = (dis_opts[st_coords] if st_coords in dis_opts.keys() else None)
        action = egreedy_policy(q_values, state, epsilon, rg, dis_opt)

        # Checking if primitive action
        if action < 6:
          # Perform regular Q-Learning update for state-action pair
          next_state, reward, done, _ = env.step(action)
          q_values[state, action] += alpha * (reward + gamma*np.max(q_values[
                                          next_state]) - q_values[state, action])
          uf_values[state,action] += 1
          state = next_state
          eps_reward += reward
        else:
          # find the corresponding function for the option
          opt_func = opt_func_map[action]
          other_opts = [6,7,8,9]; other_opts.remove(action)
          optdone = False
          while (optdone == False):
            optact, _ = opt_func(env, state)
            next_state, reward, done, _ = env.step(optact)
            _, optdone = opt_func(env, next_state)
            eps_reward += reward
            # perform update for the optact - primitive action
            q_values[state, optact] += alpha * (reward + gamma*np.max(q_values[
                                     next_state]) - q_values[state, optact])
            uf_values[state, optact] += 1

            # find all options that choose the same primitive action at state
            opt_upd_list = [action]
            st_coords = list(env.decode(state))[:2]
            for oth in other_opts:
              if OPT_TO_POLICY_MAP[oth][st_coords[0], st_coords[1]] == optact:
                opt_upd_list.append(oth)
            # debug - print(st_coords, action, opt_upd_list)
            # perform updates for all options that choose the same primitive
                                     action at state
            nst_coords = list(env.decode(next_state))[:2]
            for opt in opt_upd_list:
              term_matrix = OPT_TO_TERM_MAP[opt]
              if term_matrix[nst_coords[0], nst_coords[1]] == 1:
                # if the option terminates, we do total max over all actions(and
                                     options) in next state
                q_values[state, opt] += alpha * (reward + gamma * np.max(q_values[
                                     next_state]) - q_values[state, opt])
                uf_values[state, opt] += 1
              else:
                # if it does not, we use the option q-value in next state for
                                     update
                q_values[state, opt] += alpha * (reward + gamma * (q_values[
                                     next_state, opt]) - q_values[state, opt])
                uf_values[state, opt] += 1

            state = next_state
    ep_rewards_list.append(eps_reward)
```

Tuned hyperparameters for this configuration: α=**0.36**, ϵ=**0.05**.

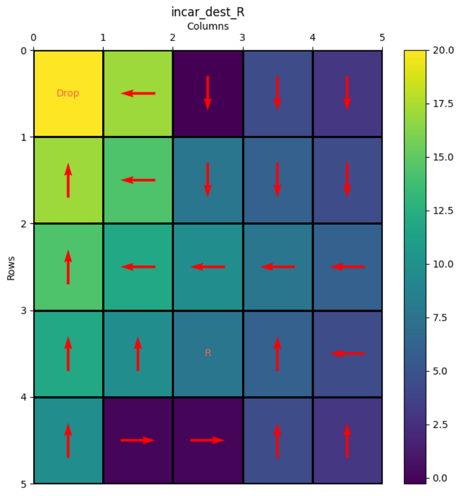
Average reward curve



Visualizing the learnt Q values

The below plotted heatmaps show actions taken at each locations clearly. Therefore, we will provide only non-trivialities in the descriptions. The Q value plots are very informative on their own. Arrows indicate primitive actions and letters indicate options. Here, the letter "X" stands to "Drive_to_X" option where $X \in \{R, G, Y, B\}$.

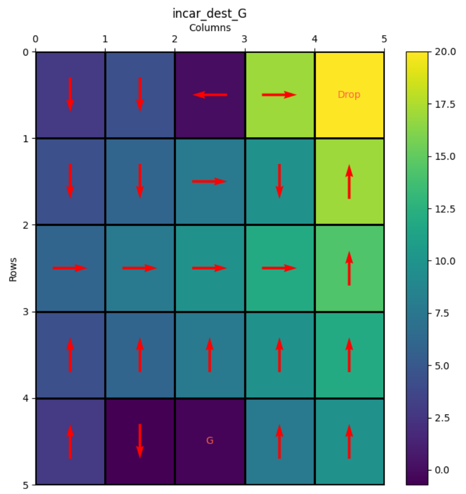
Passenger location: Inside car, Drop location: R



Inference

- The policy has learnt well enough with the choice of hyperparameters.

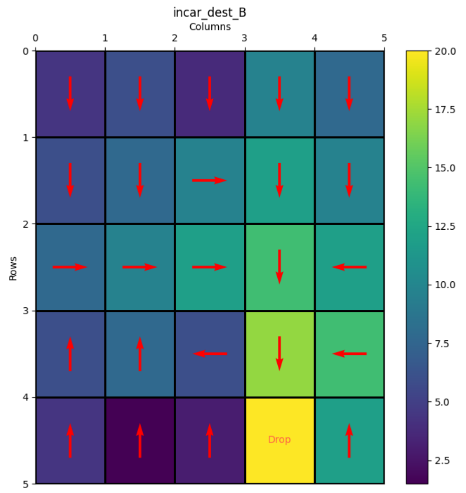
Passenger location: Inside car, Drop location: G



Inference

- General trend observed in all these maps for Intra option Q-learning is that primitive actions are highly preferred.

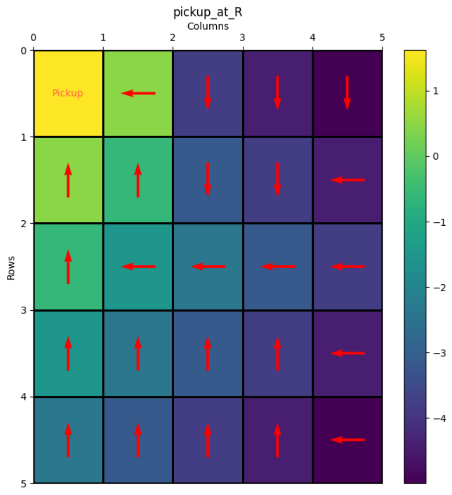
Passenger location: Inside car, Drop location: B



Inference

- The agent reaches drop location with no anomalies occuring in any position in the map.

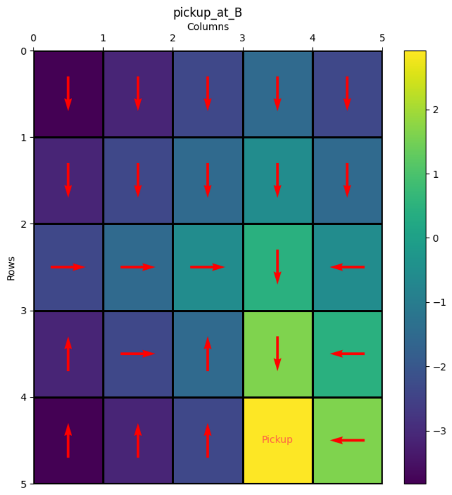
Passenger location: Pickup at R



Inference

- In case of pickup situations, as it can be observed that, none of actions are based on options.

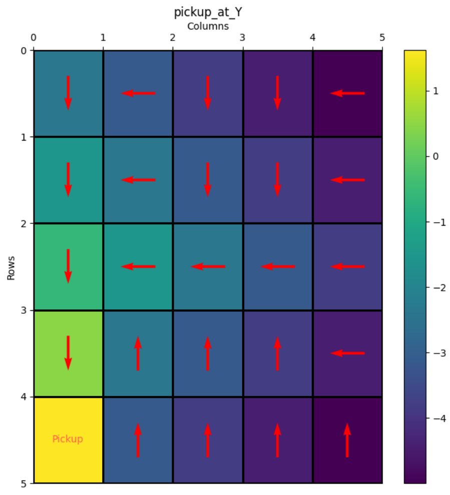
Passenger location: Pickup at B



Inference

- It reaches location red comfortably with primitive actions.

Passenger location: Pickup at Y



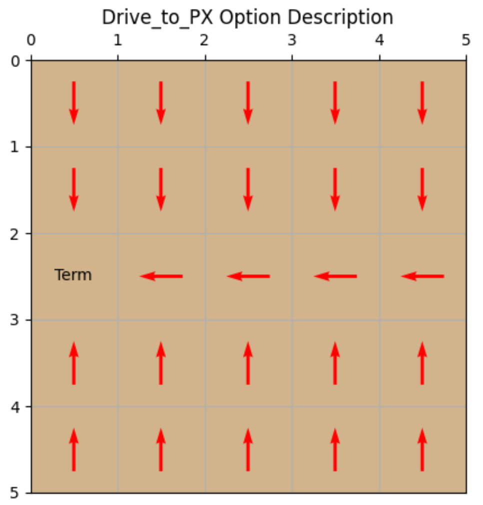
Inference

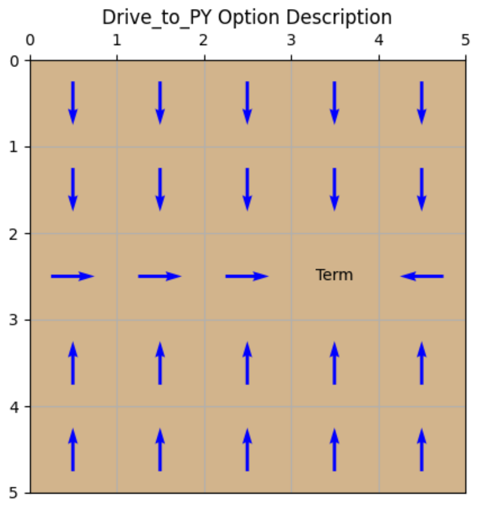
- Same as location red, here also the agent reaches comfortably location Blue with just primitive actions.

2 HRL with alternate set of options

Option description

- These below diagrams show an alternate set of options in which the agent will be taken to the 2 special locations namely PX = (2, 0) and PY = (2,3).
- The options can be initiated in all states, except for the states where car's location is marked "Term".
- The options will be terminated in all states with the car's location marked "Term"
- Reasoning for choosing this set of options are:
  - Most of the times, agent has to go through this locations PX and PY, it can be beneficial to provide options that will guide the agent towards these locations first. And then, the agent may go to the color locations for pickup/drop-off.
  - This way, number of options have been reduced, thus might lead to quicker convergence and lesser chance of sub-optimal policy.



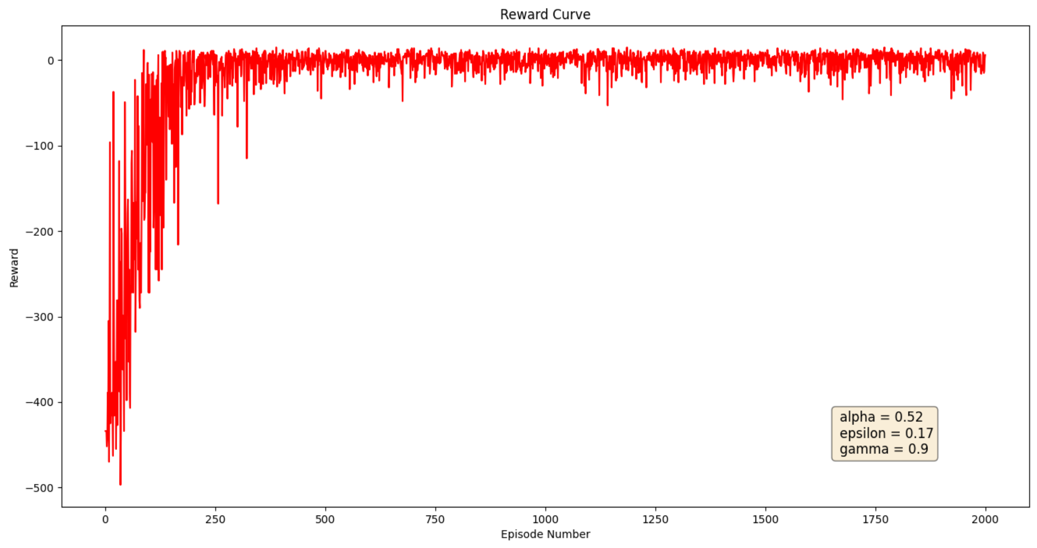


SMDP Q-Learning

The code used for implementing SMDP is same as that for Part 1 but the alternate options are used instead of the given options.

Tuned hyperparameters for this configuration: α**=0.52,** ϵ**=0.17**.

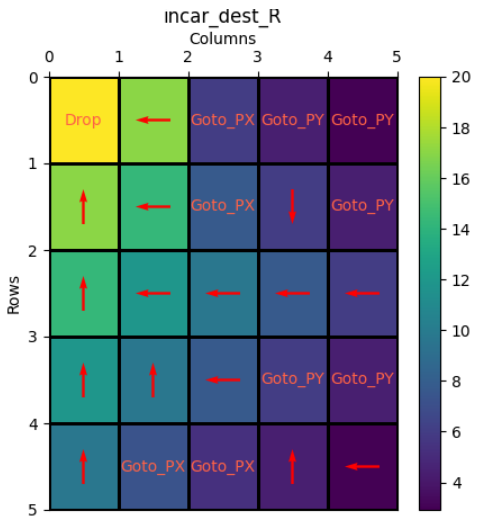
Average reward curve



Visualizing the learnt Q values

The below plotted heatmaps show actions taken at each locations clearly. Therefore, we will provide only non-trivialities in the descriptions. The Q value plots are very informative on their own. Arrows indicate primitive actions and strings indicate options. In the Q plots, ”Goto” is the same as ”Drive_to”.

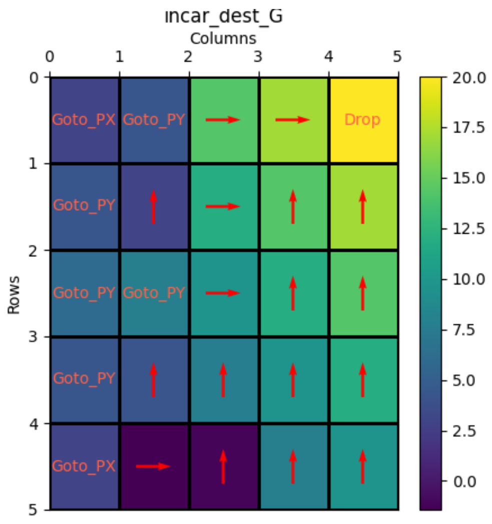
Passenger location: Inside car, Drop location: R



Inference

- For the alternate set of options that we considered, it can be seen that, the agent learns both primitive actions and options.

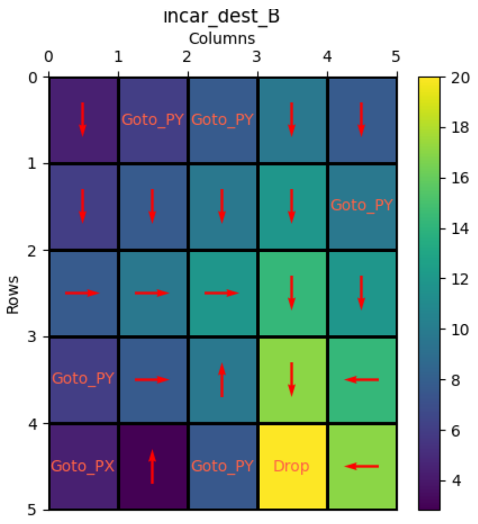
Passenger location: Inside car, Drop location: G



Inference

- Sometimes, the actions is ”Go to PY” even though,it will pass through PX along the way, but at the end our goal is to reach our drop location, and that is achieved with ease.

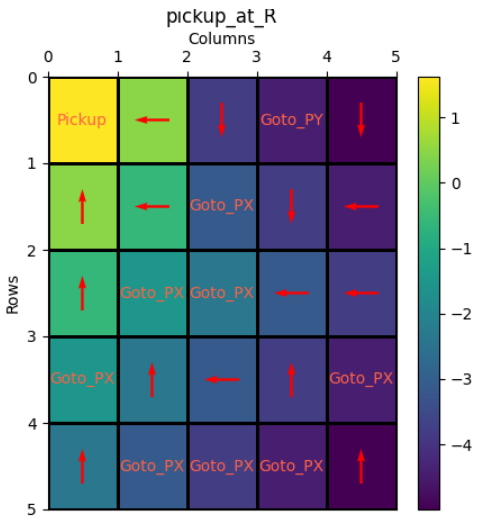
Passenger location: Inside car, Drop location: B



Inference

- Similar to the above two cases, the agent reaches the drop location B.

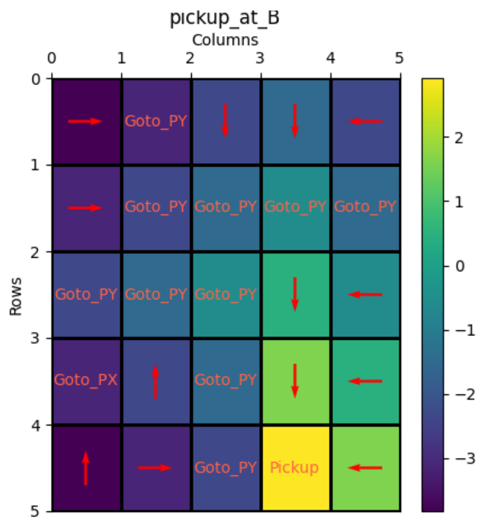
Passenger location: Pickup at R



Inference

- In case of pickup situations, options are preferred largely for our set of alternate options, than the given set of options.

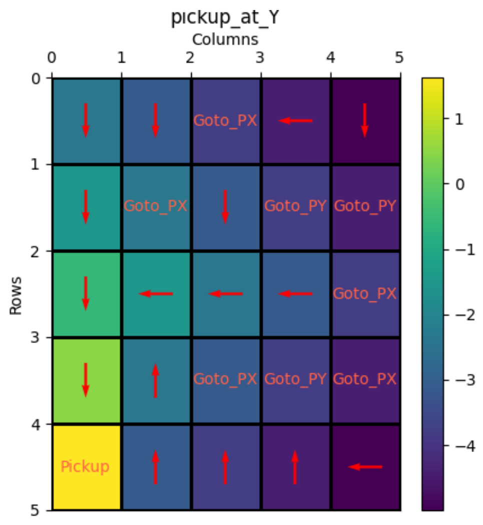
Passenger location: Pickup at B



Inference

- Agent reaches the pickup location at B with both set of actions(primitive and non-primitive).

Passenger location: Pickup at Y



Inference

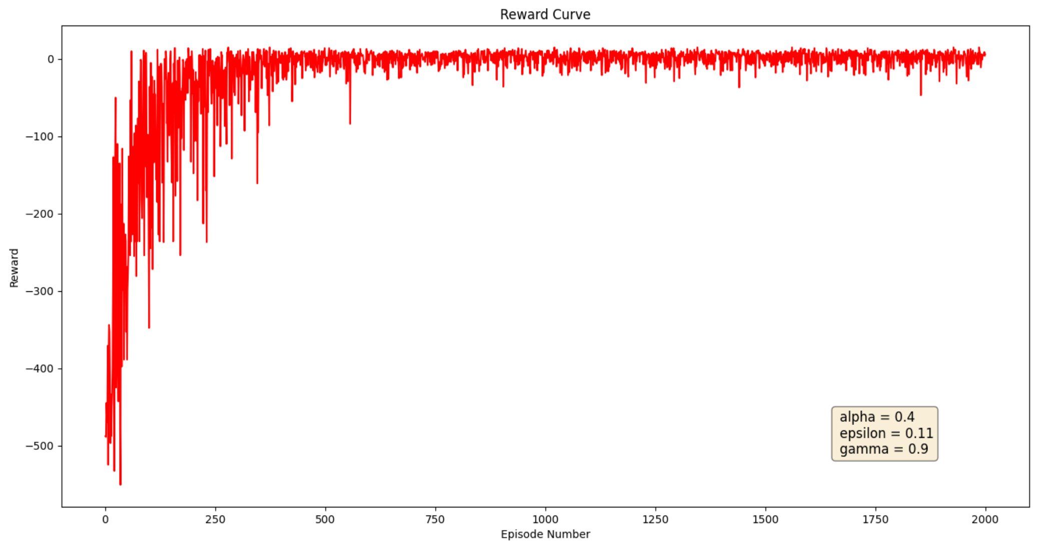
- Agent reaches the pickup location at Y with both set of actions(primitive and non-primitive).

Intra-option Q-Learning

The code used for implementing Intra-Option is same as that for Part 1 but the alternate options are used instead of the given options.

Tuned hyperparameters for this configuration: α=**0.4**, ϵ=**0.11**.

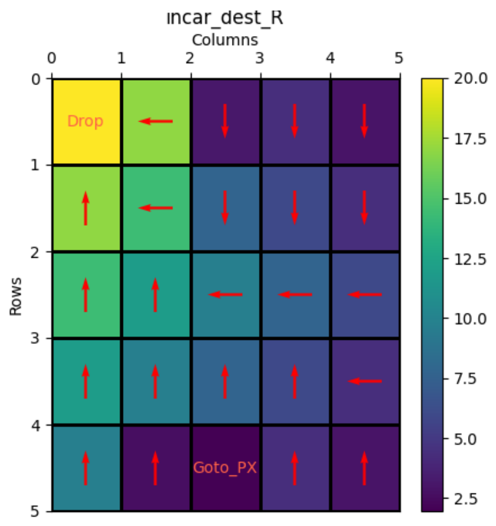
Average reward curve



Visualizing the learnt Q values

The below plotted heatmaps show actions taken at each locations clearly. Therefore, we will provide only non-trivialities in the descriptions. The Q value plots are very informative on their own. Arrows indicate primitive actions and strings indicate options. In the Q plots, "Goto" is the same as "Drive_to".

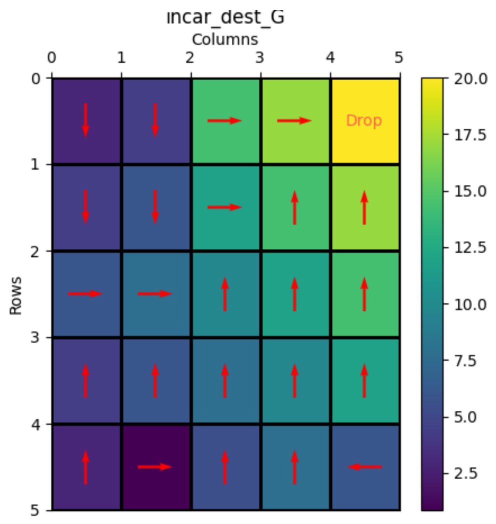
Passenger location: Inside car, Drop location: R



Inference

- Even with the alternate set of options, intra option Q-learning still has learnt primitive actions.

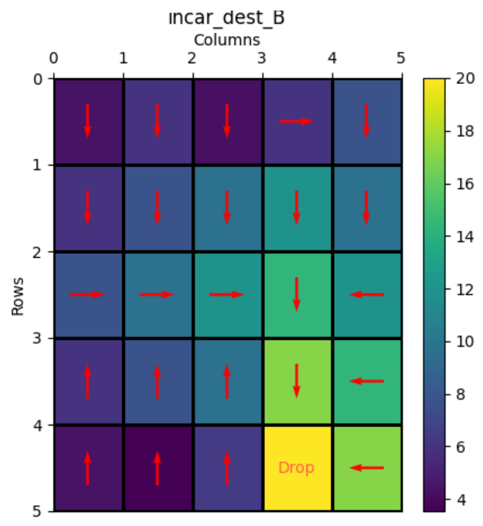
Passenger location: Inside car, Drop location: G



Inference

- Agent reaches drop location G with only primitive actions.

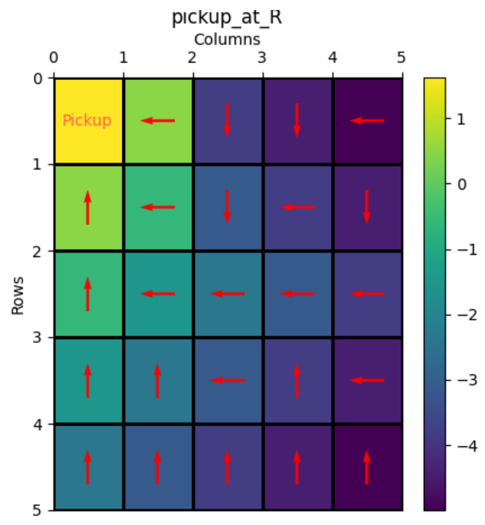
Passenger location: Inside car, Drop location: B



Inference

- Similar to the above case, agent reaches B with just primitive set of actions.

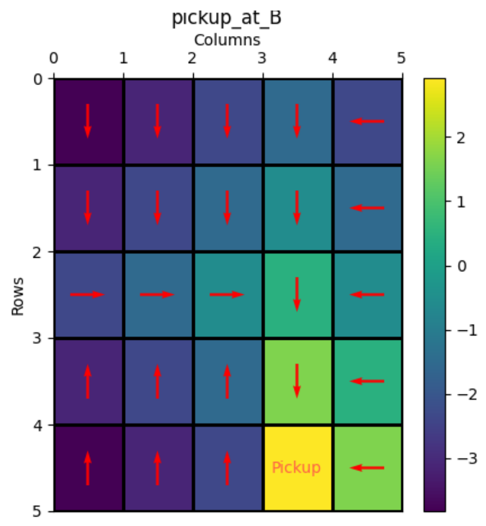
Passenger location: Pickup at R



Inference

- For all these pickup situations, there is not even a single non-primitive action that the policy has learnt.

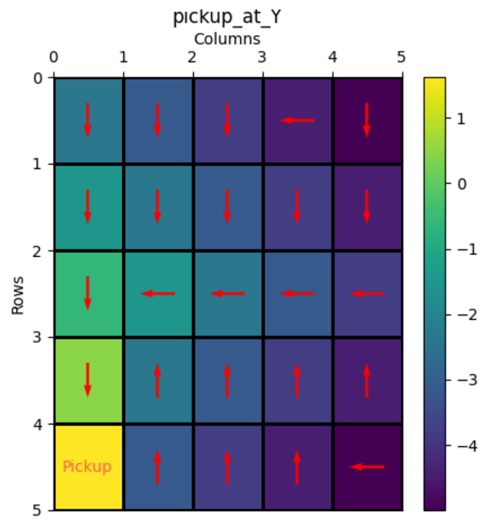
Passenger location: Pickup at B



Inference

- Agent reaches pickup location B, and the action map looks almost similar to PY option description map.

Passenger location: Pickup at Y



Inference

- Agent reaches pickup location Y, and the action map looks almost similar to PX option description map.

3 Comparing SMDP and Intra option:

3.1 General comments

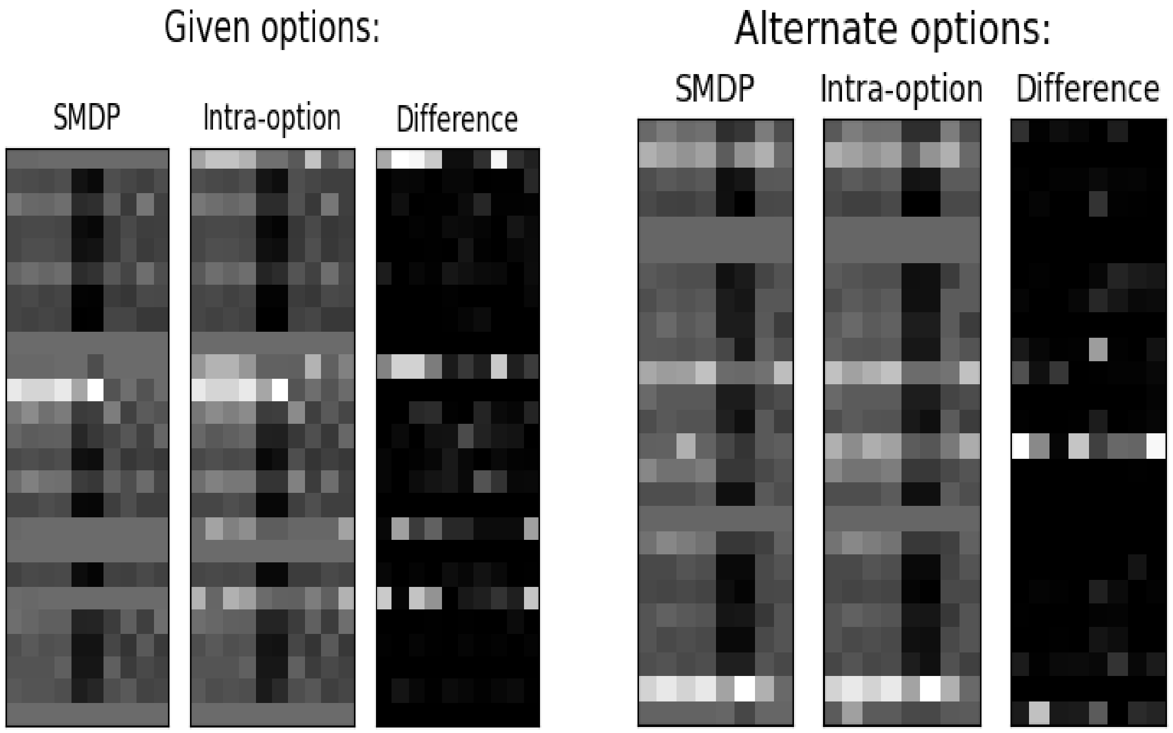
- In the Q plots so far, we note that Intra-option Q-Learning (IOQ) prefers primitive action over options, whereas this is not the case with SMDP Q-Learning.
- This is due to the way in which the update rule is defined. The Q-values for both primitive actions and options are **updated at each time step** in IOQ.
- So, in IOQ, the Q values for the actions are also frequently updated. We note that **numpy's argmax breaks ties in favour of lower index**. This results in **primitive actions [index** $\in \{0, 1 \cdots 5\}$**] being chosen when acting greedily** as seen in the Q plots. This is the reason why there are very few options in the Q plots for IOQ.
- As IOQ updates many more state-action pairs, it results in faster convergence.
- IOQ is also **versatile and robust** than SMDP. In SMDP Q-Learning for the given options **Drive to (R,G,Y,B) options**, when the passenger was **inside the car** and his destination was **Green**, the optimal action at (2,0) was to take **Drive_to_Red** option. We saw this earlier - this was the description for the first Q plot in this report. Say by some miracle, the car has a passenger with destination **Green** and ends up at $(2, 0)$ - this agent will now go to **Red** and then go to **Green**. This is clearly sub-optimal. IOQ does not have these issues since the q-values for the primitive actions chosen by a running option are also updated.

3.2 Comparative Study

To analyze the Q values learned and update counts of Intra-Option QLearning and SMDP QLearning, we conducted a comparative study. Here, we ran SMDP and Intra-option Q-Learning with the same set of hyperparameters : α=0.4 and ϵ=0.15.

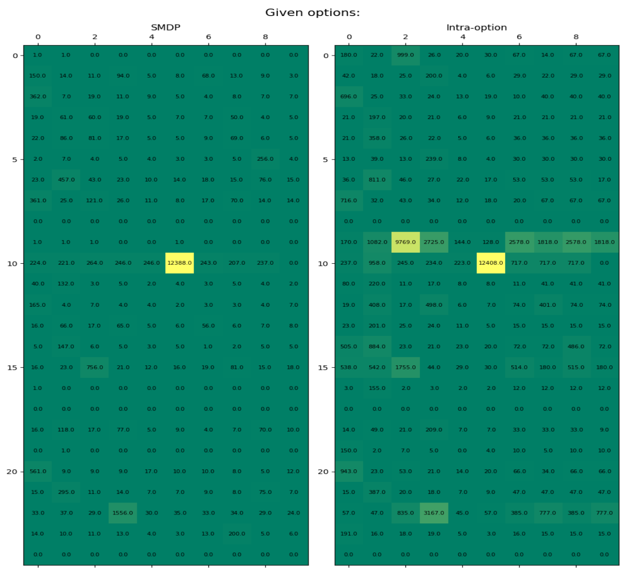
Both the below Q-value and Update frequency plot comparisons consider 25 random states out of the 500 total states. In the heatmap, rows represent the states, and column represents 10 and 8 different actions for the given options and alternate options respectively.

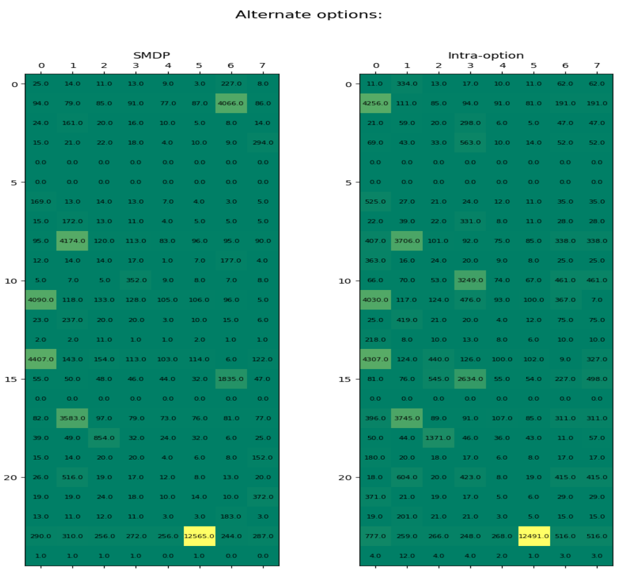
Q-value comparison:



- The Q values learnt by both SMDP and Intra-Option QLearning are almost the same.
- This is evident from the difference heatmap, in which almost all (state,action) pairs have values close to zero[and are black].

Update frequency comparison:





- By keeping track of the total update count, it is evident that the total number of updates in Intra-option Q-Learning is higher than SMDP Q-Learning.
- From the above heatmaps, we can see that for most (s, a) pairs, Intra-Option QLearning makes more updates than SMDP Q-Learning.

4 Links:

- Link to all the plots for different configurations: Reward curves and Q values visualized
- Link to the notebooks that we used to produce these results : Notebooks