CS6700
REINFORCEMENT LEARNING
PROGRAMMING ASSIGNMENT - 3

TEAM: Anirud N (CE21B014) , Leon Davis M S (ED21B038)

Link for Github repository (click here)(https://github.com/AnirudN/CS6700_CE21B014_ED21B038_PA3/tree/main)

Introduction :

This assignment aims at familirising and implementing the 1-Step SMDP and Intra option Q Learning Algorithm to solve the Open AI Taxi V-3 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 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 another reward is triggered.
• +20 delivering passengers.
•-10 executing ”pickup” and ”drop-off” actions illegally.
The discount factor is taken to be $\gamma = 0.9$.

Flow of Report :

In this Assignment we have implemented SMDP and Intra Option Q learning for the following settings:

1) **Learning the options for reaching R,G,B,Y simultaneously and performing the SMDP update**

2) **Deterministic options to reach the states R,G,B,Y**
3) **Custom options are used - Option 1- to reach the state (2,0) and option 2- to reach (2,3) in the grid. Performing SMDP and INTRA Option Q Learning for both the cases**
    a) **Deterministic options ie the option policy is predefined**
    b) **The options have been learnt simultaneously**

Options that the assignment asked us to look for:

1) Reach state R
2) Reach State G
3) Reach State Y
4) Reach State B

These 4 options are the alternative options considered below.

1) Learning the options R,G,Y,B Simultaneously

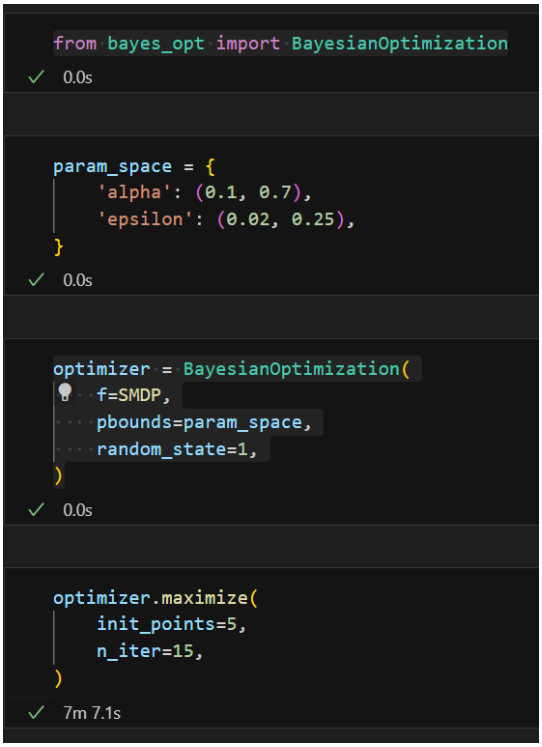
SMDP Q LEARNING

a) Tuning of Hyperparameters:

Gamma was set to 0.9 (discount factor)

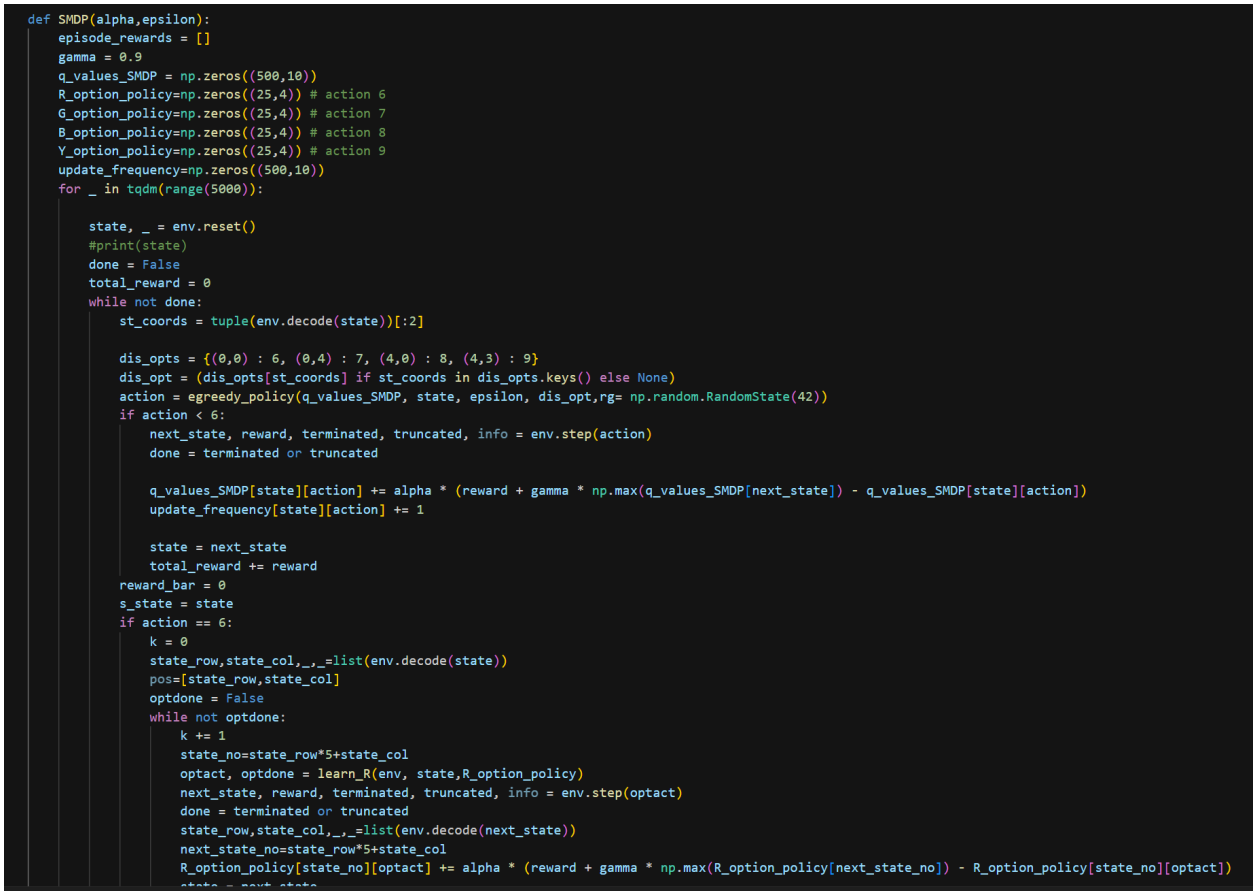
We need to Tune the hyperparameters alpha and epsilon.

Inorder to tune the hyperparameters, we used Bayesian Optimization.



The search space : alpha (0.1 to 0.7) and epsilon (0.02,0.25)

The function SMDP, which runs the smdp algorithm for 5000 iterations and it returns the total rewards of the last 1000 episodes run. This was used as the metric to tune the hyperarameter to maximise the total reward of last 1000 episodes.



```python
def SMDP(alpha,epsilon):
    episode_rewards = []
    gamma = 0.9
    q_values_SMDP = np.zeros((500,10))
    R_option_policy=np.zeros((25,4)) # action 6
    G_option_policy=np.zeros((25,4)) # action 7
    B_option_policy=np.zeros((25,4)) # action 8
    Y_option_policy=np.zeros((25,4)) # action 9
    update_frequency=np.zeros((500,10))
    for _ in tqdm(range(5000)):

        state, _ = env.reset()
        #print(state)
        done = False
        total_reward = 0
        while not done:
            st_coords = tuple(env.decode(state))[:2]

            dis_opts = {(0,0) : 6, (0,4) : 7, (4,0) : 8, (4,3) : 9}
            dis_opt = (dis_opts[st_coords] if st_coords in dis_opts.keys() else None)
            action = egreedy_policy(q_values_SMDP, state, epsilon, dis_opt,rg= np.random.RandomState(42))
            if action < 6:
                next_state, reward, terminated, truncated, info = env.step(action)
                done = terminated or truncated

                q_values_SMDP[state][action] += alpha * (reward + gamma * np.max(q_values_SMDP[next_state]) - q_values_SMDP[state][action])
                update_frequency[state][action] += 1

                state = next_state
                total_reward += reward
            reward_bar = 0
            s_state = state
            if action == 6:
                k = 0
                state_row,state_col,_,_=list(env.decode(state))
                pos=[state_row,state_col]
                optdone = False
                while not optdone:
                    k += 1
                    state_no=state_row*5+state_col
                    optact, optdone = learn_R(env, state,R_option_policy)
                    next_state, reward, terminated, truncated, info = env.step(optact)
                    done = terminated or truncated
                    state_row,state_col,_,_=list(env.decode(next_state))
                    next_state_no=state_row*5+state_col
                    R_option_policy[state_no][optact] += alpha * (reward + gamma * np.max(R_option_policy[next_state_no]) - R_option_policy[state_no][optact])
                    state = next_state
```

```
            while not optdone:
                k += 1
                state_no=state_row*5+state_col
                optact, optdone = learn_R(env, state,R_option_policy)
                next_state, reward, terminated, truncated, info = env.step(optact)
                done = terminated or truncated
                state_row,state_col,_,_=list(env.decode(next_state))
                next_state_no=state_row*5+state_col
                R_option_policy[state_no][optact] += alpha * (reward + gamma * np.max(R_option_policy[next_state_no]) - R_option_policy[state_no][optact])
                state = next_state
                reward_bar = gamma * reward_bar + reward
                state_row,state_col,_,_=list(env.decode(state))
                pos=[state_row,state_col]

            q_values_SMDP[s_state][6] += alpha * (reward_bar + (gamma ** k) * np.max(q_values_SMDP[next_state]) - q_values_SMDP[s_state][6])
            update_frequency[state][6] += 1
            total_reward += reward_bar
        if action == 7:
            k=0
            state_row,state_col,_,_=list(env.decode(state))
            pos=[state_row,state_col]
            optdone = False
            while (optdone == False):
                k+=1
                state_no=state_row*5+state_col
                optact,optdone = learn_G(env,state,G_option_policy)
                next_state, reward, terminated, truncated, info = env.step(optact)
                done = terminated or truncated
                state_row,state_col,_,_=list(env.decode(next_state))
                next_state_no=state_row*5+state_col
                G_option_policy[state_no][optact] += alpha * (reward + gamma * np.max(R_option_policy[next_state_no]) - R_option_policy[state_no][optact])
                state = next_state
                reward_bar = gamma*reward_bar + reward
                state_row,state_col,_,_=list(env.decode(state))
                pos=[state_row,state_col]

            q_values_SMDP[s_state][7] += alpha * (reward_bar + (gamma**k) * np.max(q_values_SMDP[next_state]) - q_values_SMDP[s_state][7])
            update_frequency[state][7]+=1
            total_reward += reward_bar

        if action == 8:
                k=0
                optdone = False
                state_row,state_col,_,_=list(env.decode(state))
                pos=[state_row,state_col]
                while (optdone == False):
```

```
if action == 8:
        k=0
        optdone = False
        state_row,state_col,_,_=list(env.decode(state))
        pos=[state_row,state_col]
        while (optdone == False):
            k+=1
            state_no=state_row*5+state_col
            optact,optdone = learn_Y(env,state,Y_option_policy)
            next_state, reward, terminated, truncated, info = env.step(optact)
            done = terminated or truncated
            state_row,state_col,_,_=list(env.decode(next_state))
            next_state_no=state_row*5+state_col
            Y_option_policy[state_no][optact] += alpha * (reward + gamma * np.max(R_option_policy[next_state_no]) - R_option_policy[state_no][optact
            state = next_state
            reward_bar = gamma*reward_bar + reward
            state_row,state_col,_,_=list(env.decode(state))
            pos=[state_row,state_col]
        q_values_SMDP[s_state][8] += alpha * (reward_bar + (gamma**k) * np.max(q_values_SMDP[next_state]) - q_values_SMDP[s_state][8])

        update_frequency[state][8]+=1
        total_reward += reward_bar

if action == 9:
        k=0
        optdone = False
        state_row,state_col,_,_=list(env.decode(state))
        pos=[state_row,state_col]
        while (optdone == False):
            k+=1
            state_no=state_row*5+state_col
            optact,optdone = learn_B(env,state,B_option_policy)
            next_state, reward, terminated, truncated, info = env.step(optact)
            done = terminated or truncated
            state_row,state_col,_,_=list(env.decode(next_state))
            next_state_no=state_row*5+state_col
            B_option_policy[state_no][optact] += alpha * (reward + gamma * np.max(R_option_policy[next_state_no]) - R_option_policy[state_no][optact
            state = next_state
            reward_bar = gamma*reward_bar + reward
            state_row,state_col,_,_=list(env.decode(state))
            pos=[state_row,state_col]
        q_values_SMDP[s_state][9] += alpha * (reward_bar + (gamma**k) * np.max(q_values_SMDP[next_state]) - q_values_SMDP[s_state][9])
        update_frequency[state][9]+=1
        total_reward += reward_bar
```

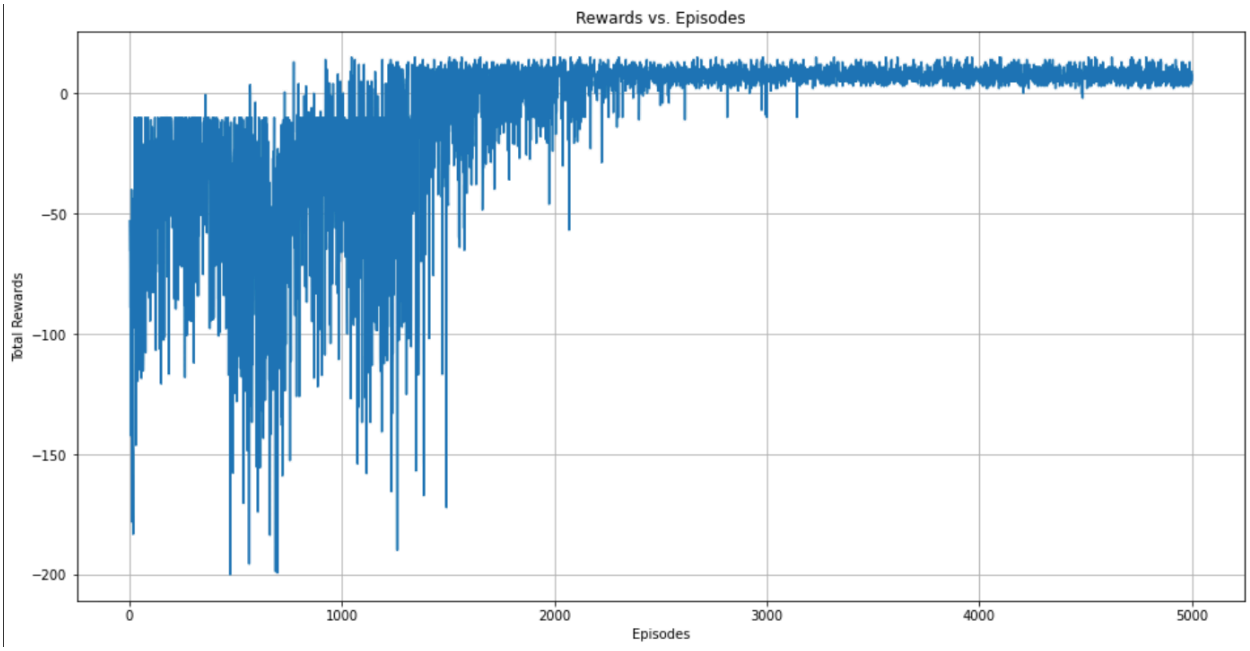
```
            if action == 9:
                    k=0
                    optdone = False
                    state_row,state_col,_,_=list(env.decode(state))
                    pos=[state_row,state_col]
                    while (optdone == False):
                        k+=1
                        state_no=state_row*5+state_col
                        optact,optdone = learn_B(env,state,B_option_policy)
                        next_state, reward, terminated, truncated, info = env.step(optact)
                        done = terminated or truncated
                        state_row,state_col,_,_=list(env.decode(next_state))
                        next_state_no=state_row*5+state_col
                        B_option_policy[state_no][optact] += alpha * (reward + gamma * np.max(R_option_policy[next_state_no]) - R_option_policy[state_no][optact
                        state = next_state
                        reward_bar = gamma*reward_bar + reward
                        state_row,state_col,_,_=list(env.decode(state))
                        pos=[state_row,state_col]
                    q_values_SMDP[s_state][9] += alpha * (reward_bar + (gamma**k) * np.max(q_values_SMDP[next_state]) - q_values_SMDP[s_state][9])
                    update_frequency[state][9]+=1
                    total_reward += reward_bar


            if done:
                episode_rewards.append(total_reward)
    return sum(episode_rewards[4000:])
```

The above snippets show the code for the function “SMDP” that implements the SMDP algorithm, for 5000 iterations and returns the total reward of last 1000 episodes.

```
print(optimizer.max)
✓ 0.0s
{'target': 8039.0, 'params': {'alpha': 0.338060484538402, 'epsilon': 0.1439278488207721}}
```

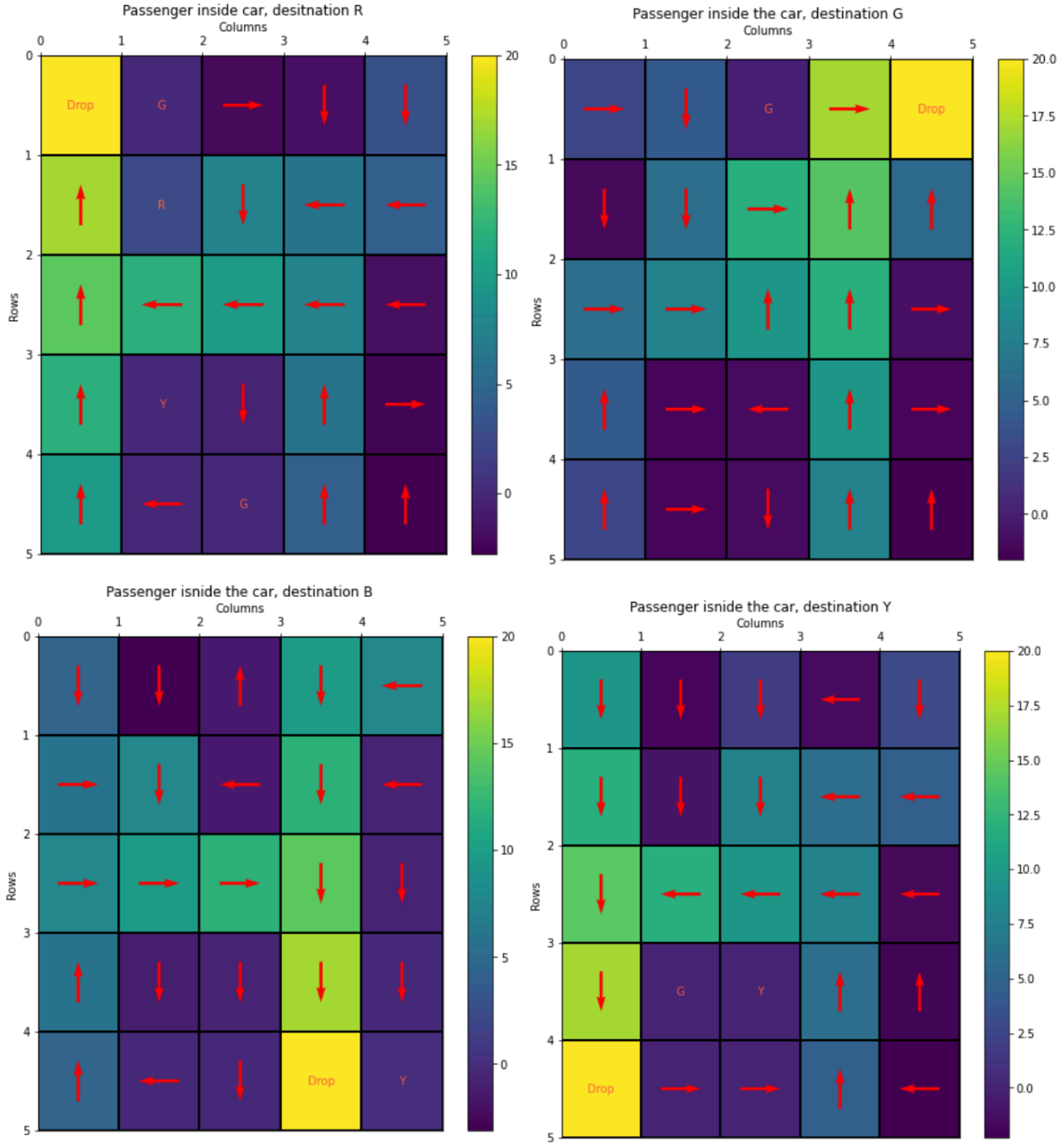
The Optimal Parameters were found to be
Alpha = 0.33 and epsilon = 0.144

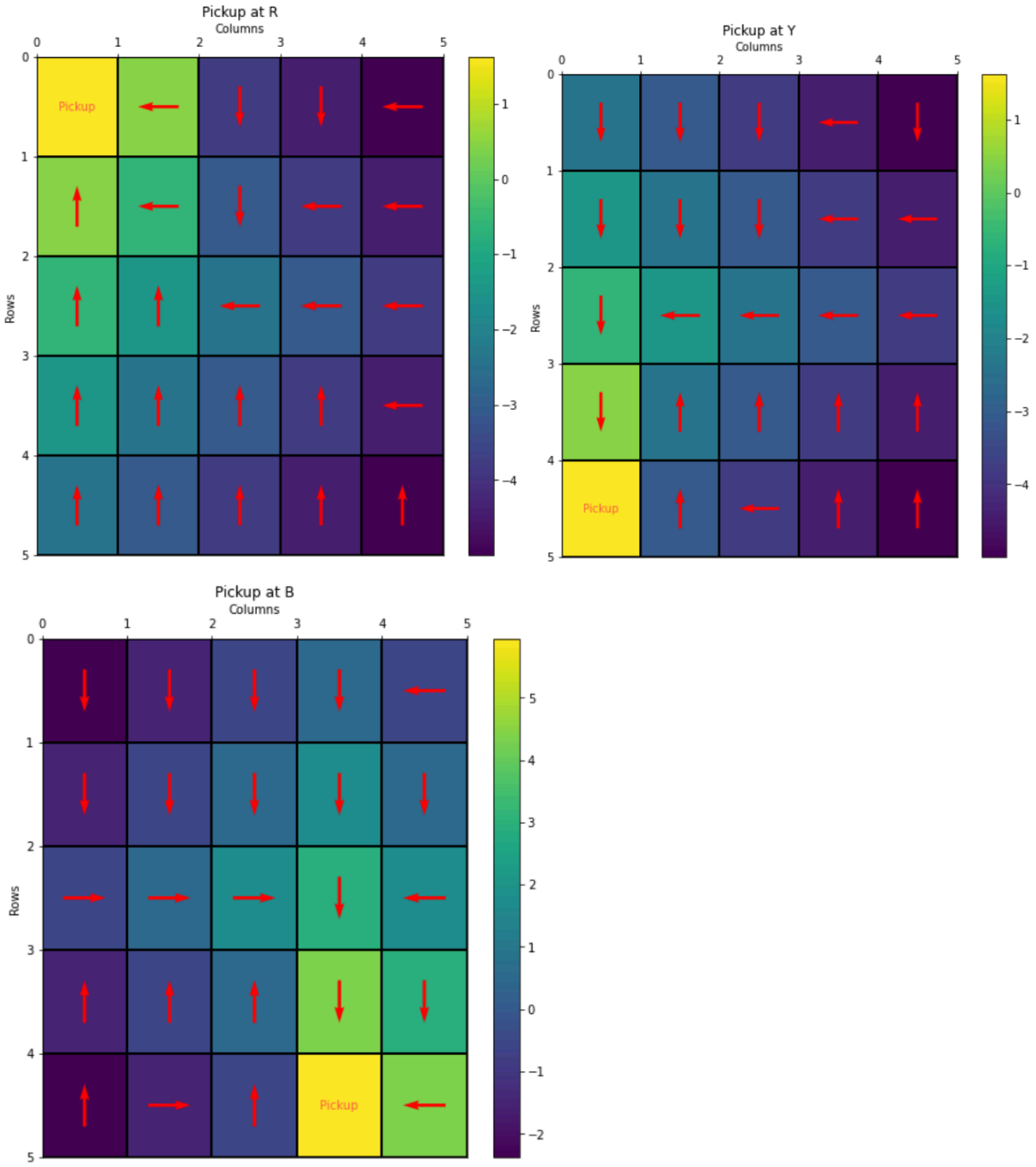
Plots and Inferences:



The above shows the plot for rewards vs episodes for each episode.

Plots for Heatmaps of Q Values for 50,00,000 Iterations of SMDP Learning:





From the above, it is evident that, when the training is done for a very long number of episodes, i.e. 50,00,000 episodes, the primitive options are chosen more than the defined options that we have defined. While The options R,G,B and Y are also chosen at times, But in usual, when training these options along

with the primitive options for a longer duration or number of episodes, it is observed that at the optimal policy, primitive options are chosen more.
In case of pickup situations, as it can be observed, none of the actions are based on options.
General trend observed in all these maps for SMDP Q-learning is that primitive actions are highly preferred.

Intra option Q LEARNING

Tuning of Hyperparameters:

Gamma was set to 0.9 (discount factor)
We need to Tune the hyperparameters alpha and epsilon.
Inorder to tune the hyperparameters, we used Bayesian Optimization.

The search space : alpha (0.1 to 0.7) and epsilon (0.02,0.25)
The function SMDP, which runs the smdp algorithm for 5000 iterations and it returns the total rewards of the last 1000 episodes run. This was used as the metric to tune the hyperarameter to maximise the total reward of last 1000 episodes.

```
print(optimizer.max)
✓ 0.0s
{'target': 8073.0, 'params': {'alpha': 0.1400844438047716, 'epsilon': 0.14288597991937635}}
```

The optimal hyperparameters were found to be
Alpha = 0.1401 and epsilon = 0.143

```python
def ioql(alpha,epsilon):
    episode_rewards = []
    gamma = 0.9
    q_values_IOQL = np.zeros((500,10))
    R_option_policy=np.zeros((25,4))
    G_option_policy=np.zeros((25,4))
    B_option_policy=np.zeros((25,4))
    Y_option_policy=np.zeros((25,4))
    for _ in tqdm(range(5000)):

        state, _ = env.reset()

        done = False
        total_reward = 0
        while not done:
            st_coords = tuple(env.decode(state))[:2]
            dis_opts = {(0,0) : 6, (0,4) : 7, (4,0) : 8, (4,3) : 9}
            dis_opt = (dis_opts[st_coords] if st_coords in dis_opts.keys() else None)
            action = egreedy_policy(q_values_IOQL, state, epsilon, dis_opt,rg= np.random.RandomState(42))
            if action < 6:
                next_state, reward, terminated, truncated, info = env.step(action)
                done = terminated or truncated
                q_values_IOQL[state][action] += alpha * (reward + gamma * np.max(q_values_IOQL[next_state]) - q_values_IOQL[state][action])
                update_frequency[state][action] += 1
                state = next_state
                total_reward += reward
            reward_bar = 0
            s_state = state
            if action == 6:
                k = 0
                state_row,state_col,_,_=list(env.decode(state))
                pos=[state_row,state_col]
                optdone = False
                while not optdone:
                    k += 1
                    state_no=state_row*5+state_col
                    optact, optdone = learn_R(env, state,R_option_policy)
                    next_state, reward, terminated, truncated, info = env.step(optact)
                    done = terminated or truncated
                    state_row,state_col,_,_=list(env.decode(next_state))
                    next_state_no=state_row*5+state_col
                    q_values_IOQL[state, optact] += alpha * (reward + gamma*np.max(q_values_IOQL[next_state]) - q_values_IOQL[state, optact])
                    update_frequency[state, optact] += 1
                    R_option_policy[state_no][optact] += alpha * (reward + gamma * np.max(R_option_policy[next_state_no]) - R_option_policy[state_no][optact])
                    options_to_update = [action]
```

```python
                    options_to_update = [action]
                    st_coords = list(env.decode(state))[:2]
                    other_options = [7,8,9]
                    for oth in other_options:
                        if oth == 7:
                            pa,pd = learn_G(env,state,G_option_policy)
                            if pa == optact:
                                options_to_update.append(oth)
                        if oth == 8:
                            pa,pd = learn_Y(env,state,G_option_policy)
                            if pa == optact:
                                options_to_update.append(oth)
                        if oth == 9:
                            pa,pd = learn_B(env,state,G_option_policy)
                            if pa == optact:
                                options_to_update.append(oth)
                    nst_coords = list(env.decode(next_state))[:2]
                    for opt in options_to_update:
                            term_matrix = OPT_TO_TERM_MAP[opt]
                            if term_matrix[nst_coords[0], nst_coords[1]] == 1:
                                q_values_IOQL[state, opt] += alpha * (reward + gamma * np.max(q_values_IOQL[next_state]) - q_values_IOQL[state, opt])
                                update_frequency[state, opt] += 1
                            else:
                                # if it does not, we use the option q-value in next state for update
                                q_values_IOQL[state, opt] += alpha * (reward + gamma * (q_values_IOQL[next_state, opt]) - q_values_IOQL[state, opt])
                                update_frequency[state, opt] += 1
                    state = next_state

                    reward_bar = gamma * reward_bar + reward
                    state_row,state_col,_,_=list(env.decode(state))
                    pos=[state_row,state_col]
                total_reward += reward_bar
            if action == 7:
                k = 0
                state_row,state_col,_,_=list(env.decode(state))
                pos=[state_row,state_col]
                optdone = False
                while not optdone:
                    k += 1
                    state_no=state_row*5+state_col
                    optact, optdone = learn_G(env, state,R_option_policy)
                    next_state, reward, terminated, truncated, info = env.step(optact)
                    done = terminated or truncated
                    state_row,state_col,_,_=list(env.decode(next_state))
                    next_state_no=state_row*5+state_col
                    q_values_IOQL[state, optact] += alpha * (reward + gamma*np.max(q_values_IOQL[next_state]) - q_values_IOQL[state, optact])
                    update_frequency[state, optact] += 1
```

```
                optact, optdone = learn_G(env, state,R_option_policy)
                next_state, reward, terminated, truncated, info = env.step(optact)
                done = terminated or truncated
                state_row,state_col,_,_=list(env.decode(next_state))
                next_state_no=state_row*5+state_col
                q_values_IOQL[state, optact] += alpha * (reward + gamma*np.max(q_values_IOQL[next_state]) - q_values_IOQL[state, optact])
                update_frequency[state, optact] += 1
                R_option_policy[state_no][optact] += alpha * (reward + gamma * np.max(R_option_policy[next_state_no]) - R_option_policy[state_no][optact])
                options_to_update = [action]
                st_coords = list(env.decode(state))[:2]
                other_options = [6,8,9]
                for oth in other_options:
                    if oth == 6:
                        pa,pd = learn_R(env,state,G_option_policy)
                        if pa == optact:
                            options_to_update.append(oth)
                    if oth == 8:
                        pa,pd = learn_Y(env,state,G_option_policy)
                        if pa == optact:
                            options_to_update.append(oth)
                    if oth == 9:
                        pa,pd = learn_B(env,state,G_option_policy)
                        if pa == optact:
                            options_to_update.append(oth)
                nst_coords = list(env.decode(next_state))[:2]
                for opt in options_to_update:
                        term_matrix = OPT_TO_TERM_MAP[opt]
                        if term_matrix[nst_coords[0], nst_coords[1]] == 1:
                            q_values_IOQL[state, opt] += alpha * (reward + gamma * np.max(q_values_IOQL[next_state]) - q_values_IOQL[state, opt])
                            update_frequency[state, opt] += 1
                        else:
                            q_values_IOQL[state, opt] += alpha * (reward + gamma * (q_values_IOQL[next_state, opt]) - q_values_IOQL[state, opt])
                            update_frequency[state, opt] += 1
                state = next_state

                reward_bar = gamma * reward_bar + reward
                state_row,state_col,_,_=list(env.decode(state))
                pos=[state_row,state_col]
            total_reward += reward_bar
        if action == 8:
            k = 0
            state_row,state_col,_,_=list(env.decode(state))
            pos=[state_row,state_col]
            optdone = False
            while not optdone:
                k += 1
                state_no=state_row*5+state_col
```

```
        if action == 8:
            k = 0
            state_row,state_col,_,_=list(env.decode(state))
            pos=[state_row,state_col]
            optdone = False
            while not optdone:
                k += 1
                state_no=state_row*5+state_col
                optact, optdone = learn_Y(env, state,R_option_policy)
                next_state, reward, terminated, truncated, info = env.step(optact)
                done = terminated or truncated
                state_row,state_col,_,_=list(env.decode(next_state))
                next_state_no=state_row*5+state_col
                q_values_IOQL[state, optact] += alpha * (reward + gamma*np.max(q_values_IOQL[next_state]) - q_values_IOQL[state, optact])
                update_frequency[state, optact] += 1
                R_option_policy[state_no][optact] += alpha * (reward + gamma * np.max(R_option_policy[next_state_no]) - R_option_policy[state_no][optact])
                options_to_update = [action]
                st_coords = list(env.decode(state))[:2]
                other_options = [6,7,9]
                for oth in other_options:
                    if oth == 6:
                        pa,pd = learn_R(env,state,G_option_policy)
                        if pa == optact:
                            options_to_update.append(oth)
                    if oth == 7:
                        pa,pd = learn_G(env,state,G_option_policy)
                        if pa == optact:
                            options_to_update.append(oth)
                    if oth == 9:
                        pa,pd = learn_B(env,state,G_option_policy)
                        if pa == optact:
                            options_to_update.append(oth)
                nst_coords = list(env.decode(next_state))[:2]
                for opt in options_to_update:
                        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_IOQL[state, opt] += alpha * (reward + gamma * np.max(q_values_IOQL[next_state]) - q_values_IOQL[state, opt])
                            update_frequency[state, opt] += 1
                        else:
                            # if it does not, we use the option q-value in next state for update
                            q_values_IOQL[state, opt] += alpha * (reward + gamma * (q_values_IOQL[next_state, opt]) - q_values_IOQL[state, opt])
                            update_frequency[state, opt] += 1
                state = next_state

                reward_bar = gamma * reward_bar + reward
                state_row,state_col,_,_=list(env.decode(state))
```

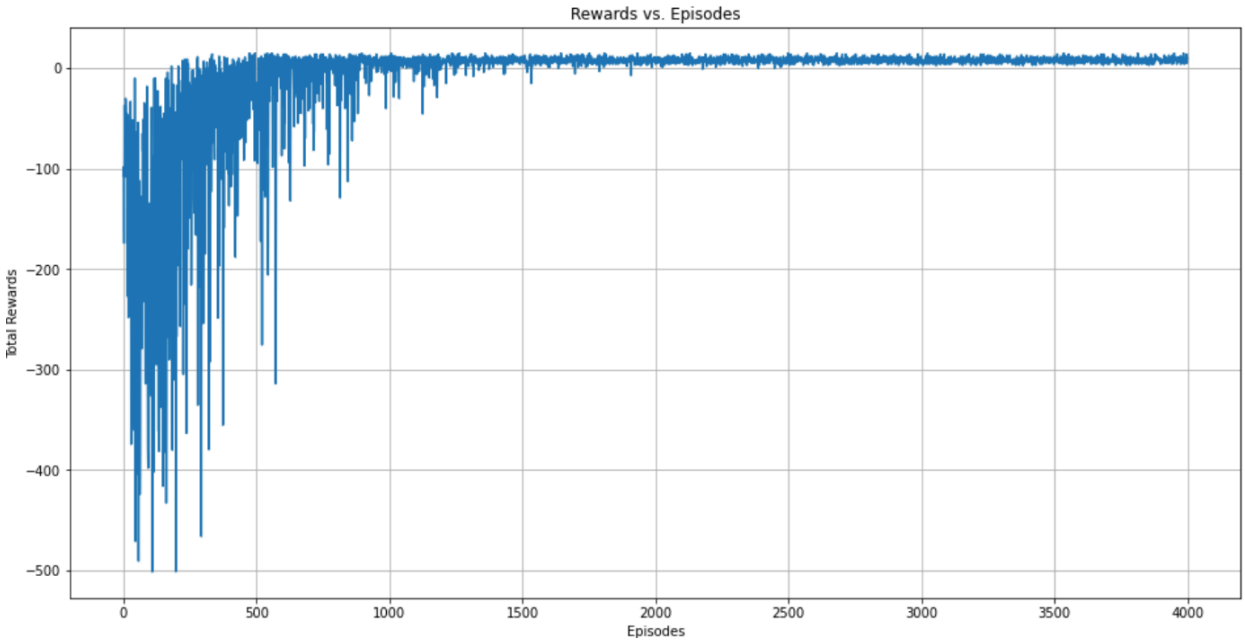
```
                reward_bar = gamma * reward_bar + reward
                state_row,state_col,_,_=list(env.decode(state))
                pos=[state_row,state_col]
            total_reward += reward_bar

        if action == 9:
            k = 0
            state_row,state_col,_,_=list(env.decode(state))
            pos=[state_row,state_col]
            optdone = False
            while not optdone:
                k += 1
                state_no=state_row*5+state_col
                optact, optdone = learn_B(env, state,R_option_policy)
                next_state, reward, terminated, truncated, info = env.step(optact)
                done = terminated or truncated
                state_row,state_col,_,_=list(env.decode(next_state))
                next_state_no=state_row*5+state_col
                q_values_IOQL[state, optact] += alpha * (reward + gamma*np.max(q_values_IOQL[next_state]) - q_values_IOQL[state, optact])
                update_frequency[state, optact] += 1
                R_option_policy[state_no][optact] += alpha * (reward + gamma * np.max(R_option_policy[next_state_no]) - R_option_policy[state_no][optact])
                options_to_update = [action]
                st_coords = list(env.decode(state))[:2]
                other_options = [6,7,8]
                for oth in other_options:
                    if oth == 6:
                        pa,pd = learn_R(env,state,G_option_policy)
                        if pa == optact:
                            options_to_update.append(oth)
                    if oth == 7:
                        pa,pd = learn_G(env,state,G_option_policy)
                        if pa == optact:
                            options_to_update.append(oth)
                    if oth == 8:
                        pa,pd = learn_Y(env,state,G_option_policy)
                        if pa == optact:
                            options_to_update.append(oth)
                nst_coords = list(env.decode(next_state))[:2]
                for opt in options_to_update:
                        term_matrix = OPT_TO_TERM_MAP[opt]
                        if term_matrix[nst_coords[0], nst_coords[1]] == 1:
                            q_values_IOQL[state, opt] += alpha * (reward + gamma * np.max(q_values_IOQL[next_state]) - q_values_IOQL[state, opt])
                            update_frequency[state, opt] += 1
                        else:
                            q_values_IOQL[state, opt] += alpha * (reward + gamma * (q_values_IOQL[next_state, opt]) - q_values_IOQL[state, opt])
                            update_frequency[state, opt] += 1
```

```
                R_option_policy[state_no][optact] += alpha * (reward + gamma * np.max(R_option_policy[next_state_no]) - R_option_policy[state_no][optact])
                options_to_update = [action]
                st_coords = list(env.decode(state))[:2]
                other_options = [6,7,8]
                for oth in other_options:
                    if oth == 6:
                        pa,pd = learn_R(env,state,G_option_policy)
                        if pa == optact:
                            options_to_update.append(oth)
                    if oth == 7:
                        pa,pd = learn_G(env,state,G_option_policy)
                        if pa == optact:
                            options_to_update.append(oth)
                    if oth == 8:
                        pa,pd = learn_Y(env,state,G_option_policy)
                        if pa == optact:
                            options_to_update.append(oth)
                nst_coords = list(env.decode(next_state))[:2]
                for opt in options_to_update:
                        term_matrix = OPT_TO_TERM_MAP[opt]
                        if term_matrix[nst_coords[0], nst_coords[1]] == 1:
                            q_values_IOQL[state, opt] += alpha * (reward + gamma * np.max(q_values_IOQL[next_state]) - q_values_IOQL[state, opt])
                            update_frequency[state, opt] += 1
                        else:
                            q_values_IOQL[state, opt] += alpha * (reward + gamma * (q_values_IOQL[next_state, opt]) - q_values_IOQL[state, opt])
                            update_frequency[state, opt] += 1
                state = next_state

                reward_bar = gamma * reward_bar + reward
                state_row,state_col,_,_=list(env.decode(state))
                pos=[state_row,state_col]
            total_reward += reward_bar
        if done:
            episode_rewards.append(total_reward)
    return sum(episode_rewards[4000:])
```

The above are the code snipped for the ioql function which returns the total rewards of last 1000 episodes(out of 5000 episodes). Maximizing this gave us the hyperparameter

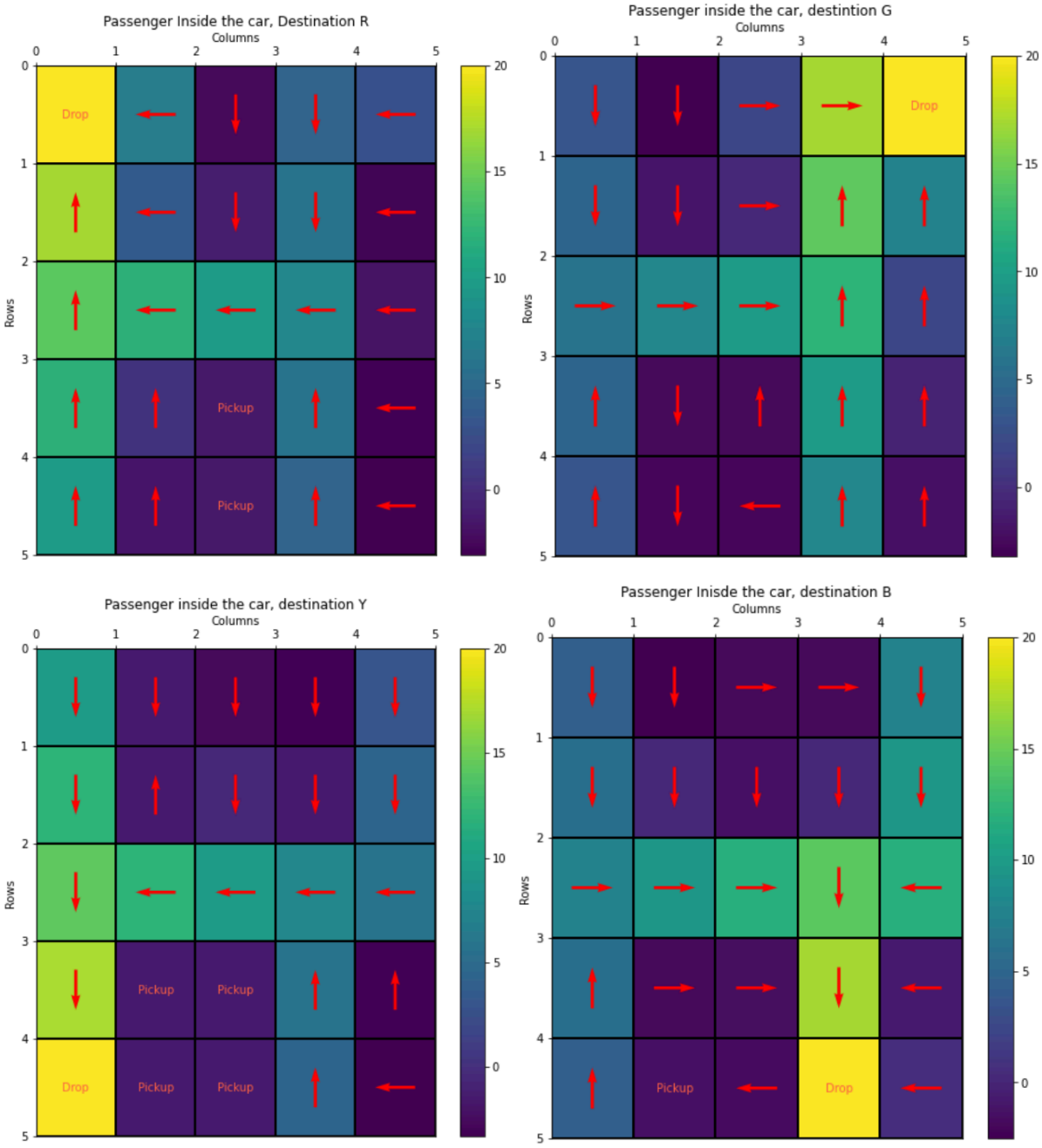
Now using these hyperparameters, we performed Intra option Q Learning for 5000000 episodes . The code for this is very similar to the code shown above, with the number of episodes alone changed.

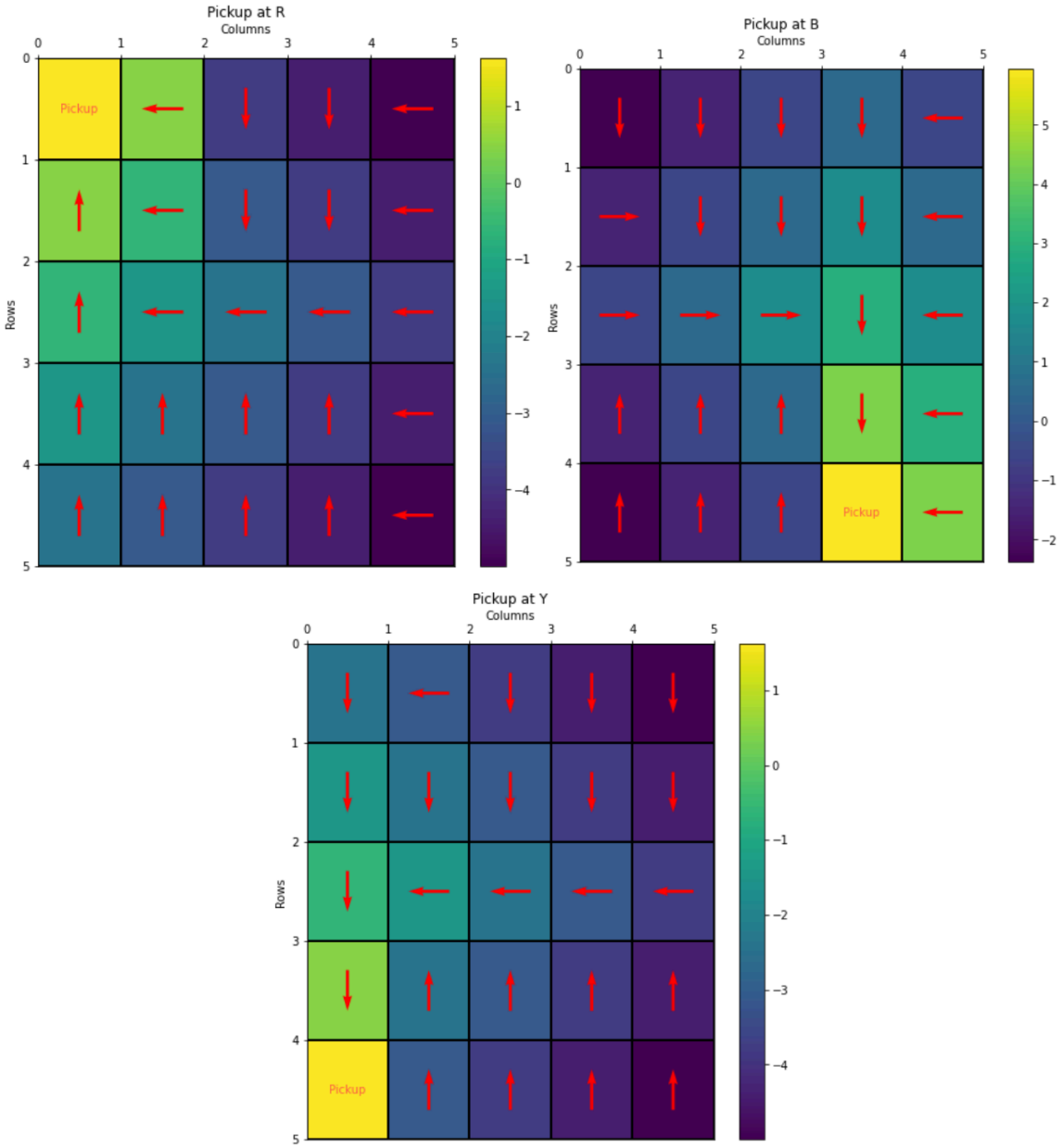
Plots and Inferences:



The above shows the plot for rewards vs episodes for each episode.

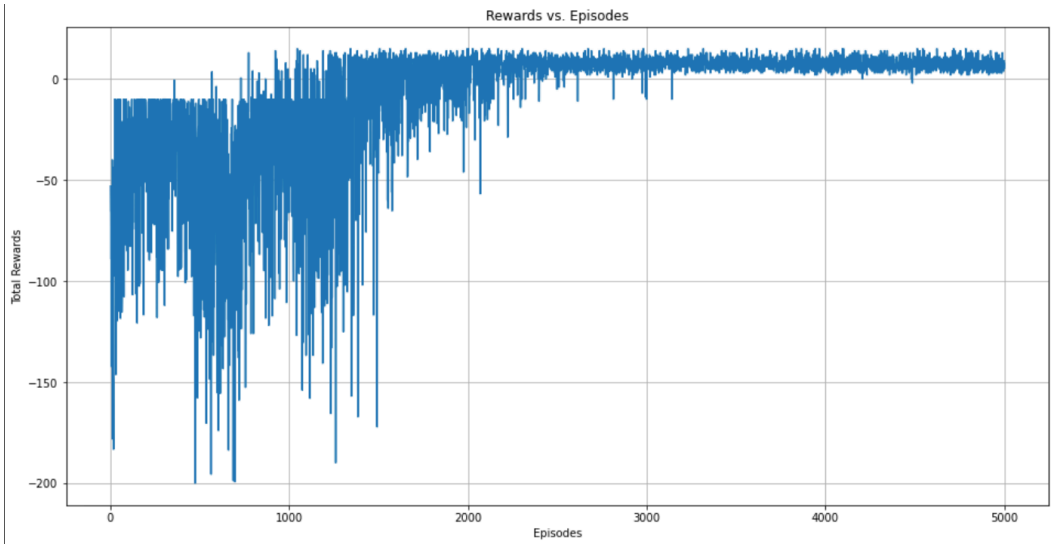
Plots for Heatmaps of Q Values for 50,00,000 Iterations of Intra Q Learning Learning:


Passenger Inside the car, Destination R
Columns
Rows
Drop
Pickup
Pickup
Passenger inside the car, destintion G
Columns
Rows
Drop
Passenger inside the car, destination Y
Columns
Rows
Pickup
Pickup
Drop
Pickup
Pickup
Passenger Inisde the car, destination B
Columns
Rows
Pickup
Drop

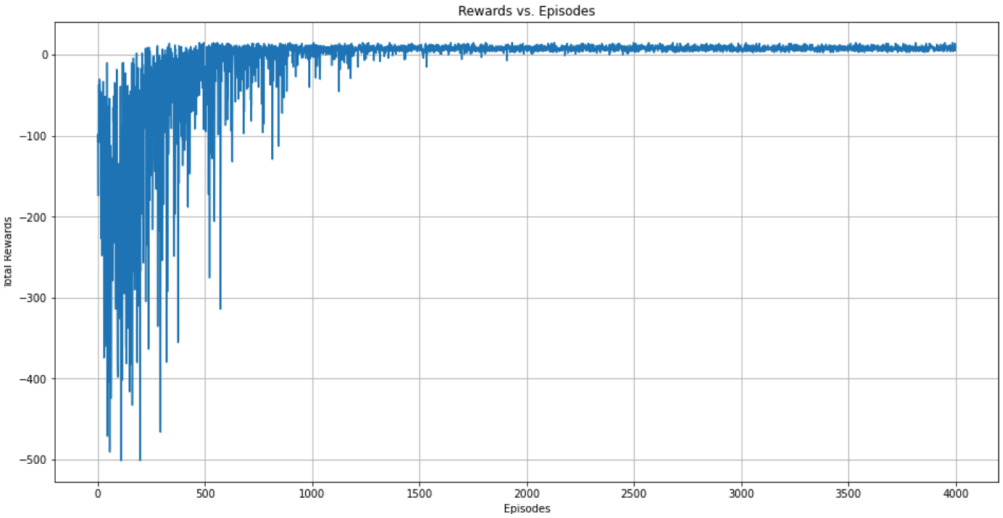


From the above, it is evident that, when the training is done for a very long number of episodes, i.e. 50,00,000 episodes, the primitive options are chosen more than the defined options that we have defined.

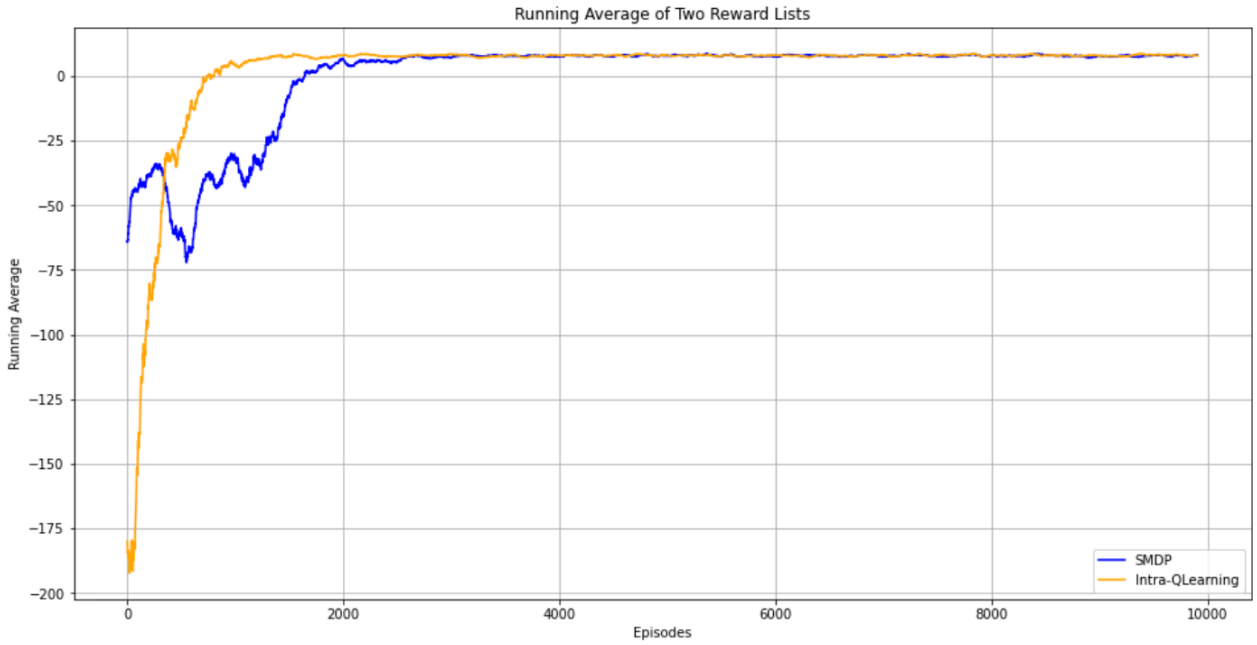
Comparison between SMDP and Intra Option Q Learning - Learning the options simultaneously



Plot for Total Rewards vs episodes for SMDP



Plot for total rewards vs episodes for Intra Option Q Learning



From the above plots, it is evident that Intra option Q learning learns Faster than SMDP, while the SMDP takes around 3000 episodes to start convergence to the Optimal policy, Intra Option Q Learning converges in around 1000 episodes for the same.

In the case of Intra, at the optimal policy, always the primitive options were chosen and they shadowed the use of the options R,G,B,Y.

A Possible Reason for faster convergence:

Intra option Q learning involves Off policy updates to learn the options and the q values of primitive options as well while performing the alternative options. Due to such an update, for the same number of episodes, more updates on q values happen in the Intra option over SMDP. So the Intra option converges faster to the optimal solution.

A Possible Reason for choosing primitive options over alternative options R,G,B,Y:

The mutually exclusiveness property of each alternative option with the primitive option is very less. In other words, Choosing an option - “go to R”, would basically also be represented as a combination of a set of primitive actions. So the option “go to R” usually works only if the goal state is R. This happens with a probability of 0.25, since there are 4 possible goal states. So, on a large run, learning under many episodes, leads to the agent using primitive actions over these alternative actions. It might be possible that the agent would choose alternative actions, if they are more mutually exclusive and the probability of

those alternative actions being chosen is higher. We would explore this in the later part of the report.

2) The Alternative options are deterministic and predefined SMDP Q LEARNING:

```
RED_MATRIX = np.array([[1,3,0,0,0],[1,3,0,0,0],[1,3,3,3,3],[1,1,1,1,1],[1,1,1,1,1]])
GRE_MATRIX = np.array([[0,0,2,2,1],[0,0,2,2,1],[2,2,2,2,1],[1,1,1,1,1],[1,1,1,1,1]])
YEL_MATRIX = np.array([[0,0,0,0,0],[0,0,0,0,0],[0,3,3,3,3],[0,1,1,1,1],[0,1,1,1,1]])
BLU_MATRIX = np.array([[0,0,0,0,3],[0,0,0,0,3],[2,2,2,0,3],[1,1,1,0,3],[1,1,1,0,3]])
```

As shown above, we defined a predefined matrix that contains the actions to be taken at each state when the particular alternative option is chosen.

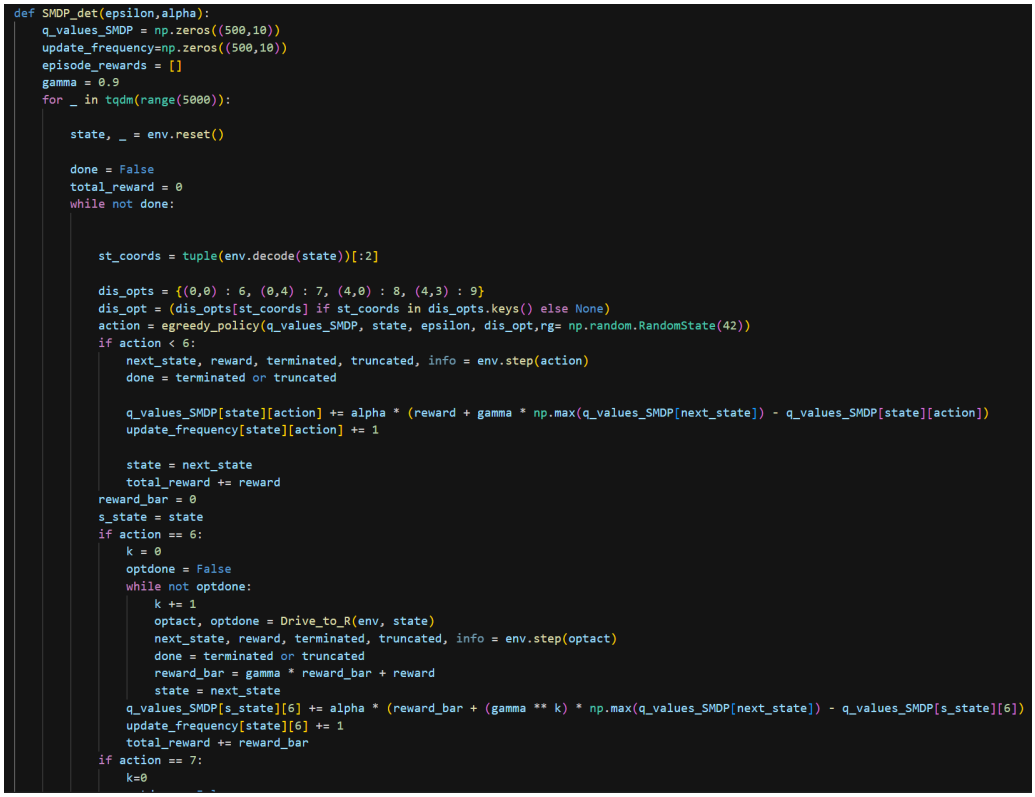
```python
def Drive_to_R(env, state):
    coords = list(env.decode(state))[:2]
    optdone = False
    optact = RED_MATRIX[coords[0], coords[1]]
    if (coords == [0, 0]):
        optdone = True
    return [optact, optdone]

def Drive_to_G(env, state):
    coords = list(env.decode(state))[:2]
    optdone = False
    optact = GRE_MATRIX[coords[0], coords[1]]
    if (coords == [0, 4]):
        optdone = True
    return [optact, optdone]

def Drive_to_Y(env, state):
    coords = list(env.decode(state))[:2]
    optdone = False
    optact = YEL_MATRIX[coords[0], coords[1]]
    if (coords == [4, 0]):
        optdone = True
    return [optact, optdone]

def Drive_to_B(env, state):
    coords = list(env.decode(state))[:2]
    optdone = False
    optact = BLU_MATRIX[coords[0], coords[1]]
    if (coords == [4, 3]):
        optdone = True
    return [optact, optdone]
```

The above functions help to choose the actions from the defined matrix at each state.

b) Tuning of Hyperparameters:
Gamma was set to 0.9 (discount factor)
We need to Tune the hyperparameters alpha and epsilon.
Inorder to tune the hyperparameters, we used Bayesian Optimization.

```python
from bayes_opt import BayesianOptimization
param_space = {


    'epsilon': (0.02, 0.25),
    'alpha': (0.1, 0.7),
}
optimizer = BayesianOptimization(
    f=SMDP_det,
    pbounds=param_space,
    random_state=1,
)
optimizer.maximize(
    init_points=5,
    n_iter=15,
)
```

The search space : alpha (0.1 to 0.7) and epsilon (0.02,0.25)
The function SMDP_det, which runs the smdp algorithm for 5000 iterations and it returns the total rewards of the last 1000 episodes run. This was used as the metric to tune the hyperarameter to maximise the total reward of last 1000 episodes.



```python
def SMDP_det(epsilon,alpha):
    q_values_SMDP = np.zeros((500,10))
    update_frequency=np.zeros((500,10))
    episode_rewards = []
    gamma = 0.9
    for _ in tqdm(range(5000)):

        state, _ = env.reset()

        done = False
        total_reward = 0
        while not done:


            st_coords = tuple(env.decode(state))[:2]

            dis_opts = {(0,0) : 6, (0,4) : 7, (4,0) : 8, (4,3) : 9}
            dis_opt = (dis_opts[st_coords] if st_coords in dis_opts.keys() else None)
            action = egreedy_policy(q_values_SMDP, state, epsilon, dis_opt,rg= np.random.RandomState(42))
            if action < 6:
                next_state, reward, terminated, truncated, info = env.step(action)
                done = terminated or truncated

                q_values_SMDP[state][action] += alpha * (reward + gamma * np.max(q_values_SMDP[next_state]) - q_values_SMDP[state][action])
                update_frequency[state][action] += 1

                state = next_state
                total_reward += reward
            reward_bar = 0
            s_state = state
            if action == 6:
                k = 0
                optdone = False
                while not optdone:
                    k += 1
                    optact, optdone = Drive_to_R(env, state)
                    next_state, reward, terminated, truncated, info = env.step(optact)
                    done = terminated or truncated
                    reward_bar = gamma * reward_bar + reward
                    state = next_state
                q_values_SMDP[s_state][6] += alpha * (reward_bar + (gamma ** k) * np.max(q_values_SMDP[next_state]) - q_values_SMDP[s_state][6])
                update_frequency[state][6] += 1
                total_reward += reward_bar
            if action == 7:
                k=0
```

```
            if action == 7:
                k=0
                optdone = False
                while (optdone == False):
                    k+=1
                    optact,optdone = Drive_to_G(env,state)
                    next_state, reward, terminated, truncated, info = env.step(optact)
                    done = terminated or truncated
                    reward_bar = gamma*reward_bar + reward
                    state = next_state
                q_values_SMDP[s_state][7] += alpha * (reward_bar + (gamma**k) * np.max(q_values_SMDP[next_state]) - q_values_SMDP[s_state][7])
                update_frequency[state][7]+=1
                total_reward += reward_bar

            if action == 8:
                    k=0
                    optdone = False
                    while (optdone == False):
                        k+=1
                        optact,optdone = Drive_to_Y(env,state)
                        next_state, reward, terminated, truncated, info = env.step(optact)
                        done = terminated or truncated
                        reward_bar = gamma*reward_bar + reward
                        state = next_state
                    q_values_SMDP[s_state][8] += alpha * (reward_bar + (gamma**k) * np.max(q_values_SMDP[next_state]) - q_values_SMDP[s_state][8])

                    update_frequency[state][8]+=1
                    total_reward += reward_bar

            if action == 9:
                    k=0
                    optdone = False
                    while (optdone == False):
                        k+=1
                        optact,optdone = Drive_to_B(env,state)
                        next_state, reward, terminated, truncated, info = env.step(optact)
                        done = terminated or truncated
                        reward_bar = gamma*reward_bar + reward
                        state = next_state
                    q_values_SMDP[s_state][9] += alpha * (reward_bar + (gamma**k) * np.max(q_values_SMDP[next_state]) - q_values_SMDP[s_state][9])
                    update_frequency[state][9]+=1
                    total_reward += reward_bar


            if done:
                episode_rewards.append(total_reward)
    return sum(episode_rewards[4000:])
```

The above snippets show the code for the function “SMDP” that implements the SMDP algorithm, for 5000 iterations and returns the total reward of last 1000 episodes.

```
print(optimizer.max)
```

✓ 0.0s

```
{'target': 8789.955083160046, 'params': {'alpha': 0.6498824007649348, 'epsilon': 0.04534743559454184}}
```

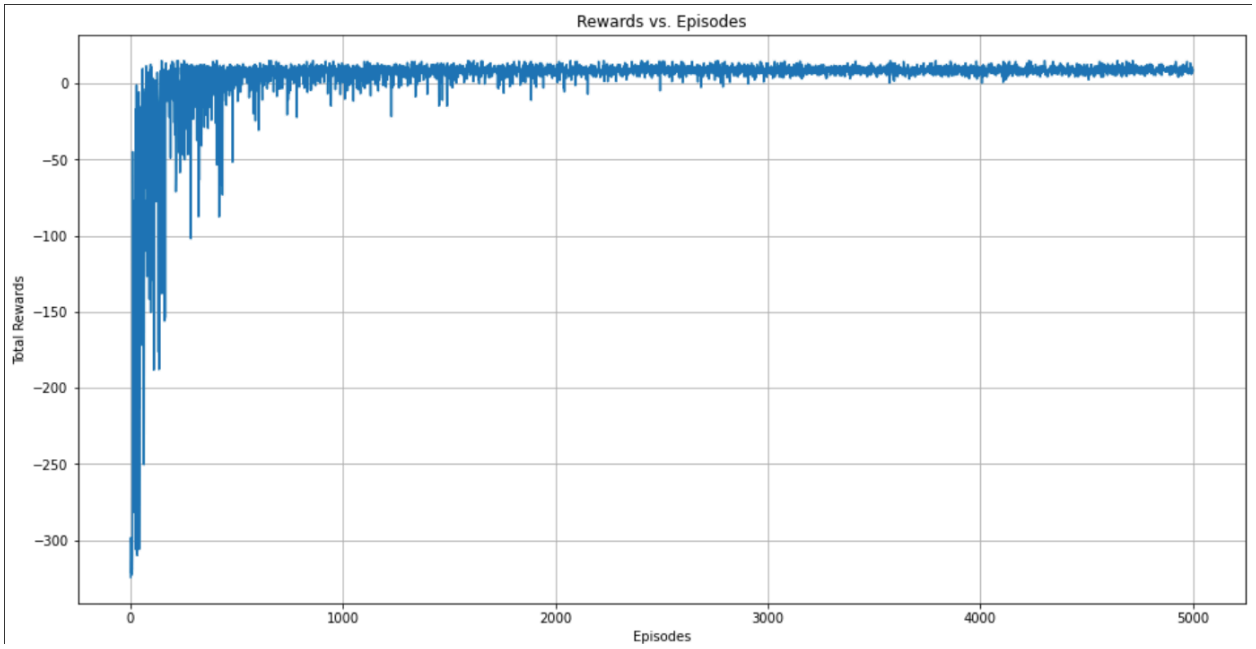
After tuning the hyperparameters, we received

Alpha = 0.64988

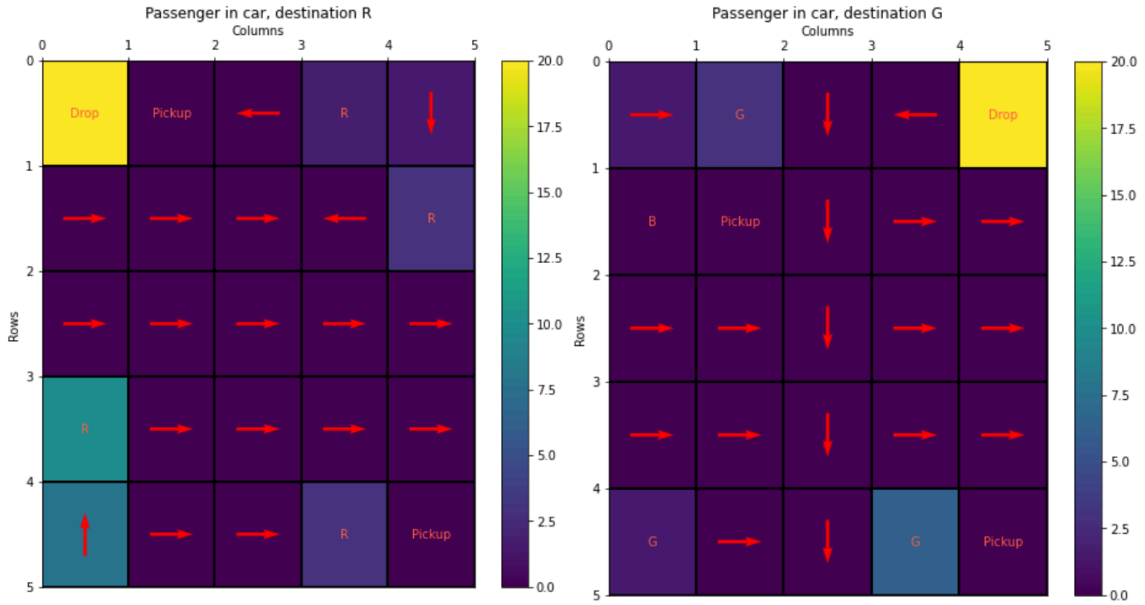
Epsilon = 0.04535

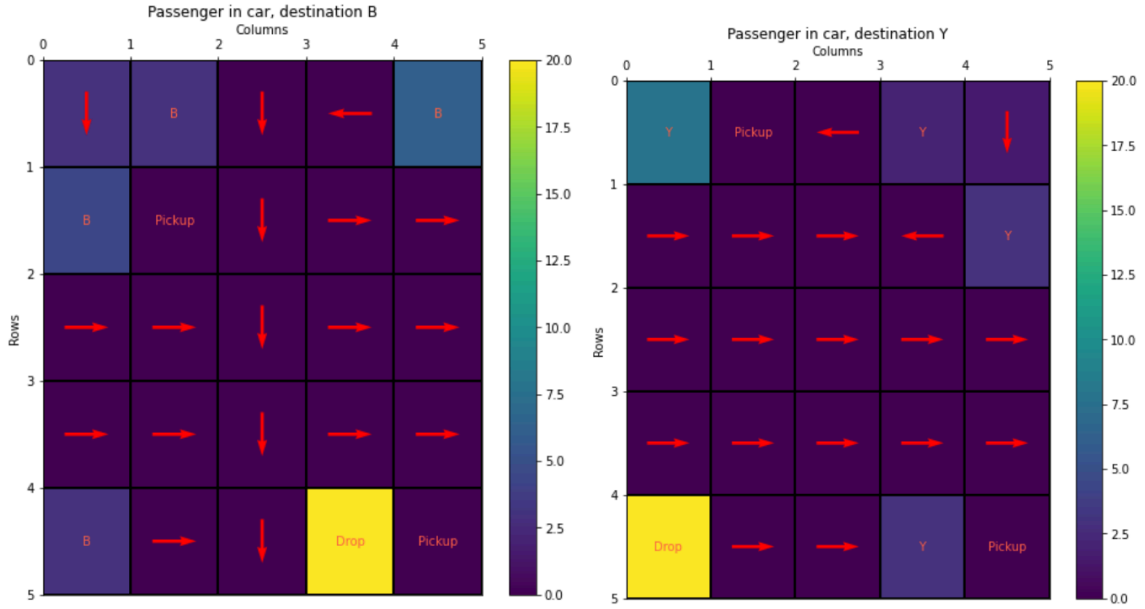
Now using these hyperparameters, we performed SMDP Q Learning for 5000000 episodes . The code for this is very similar to the code shown above, with the number of episodes alone changed.

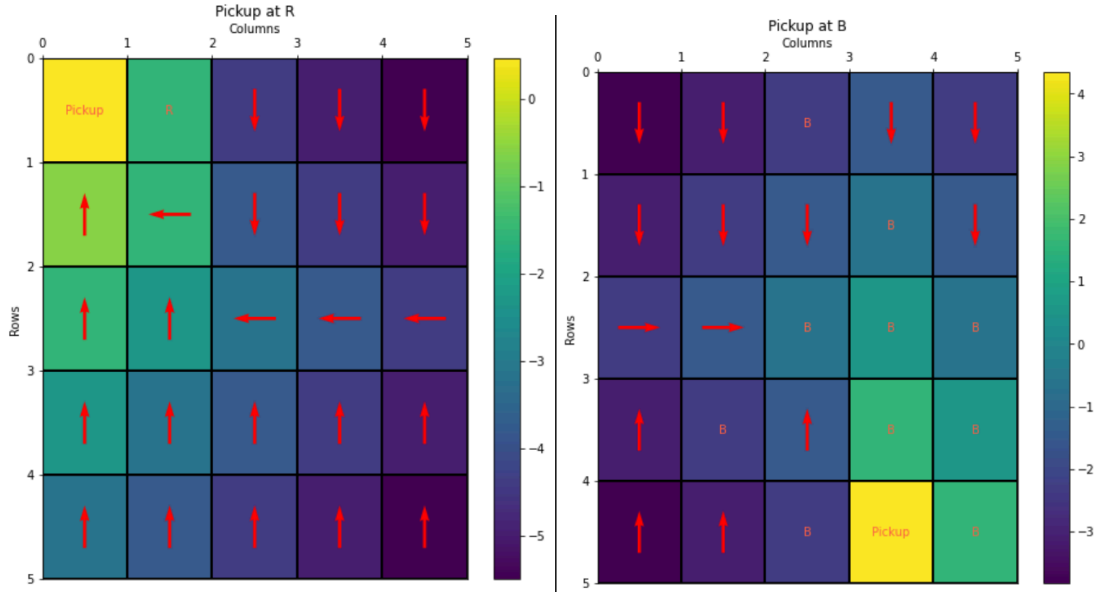
Plots and Inferences:

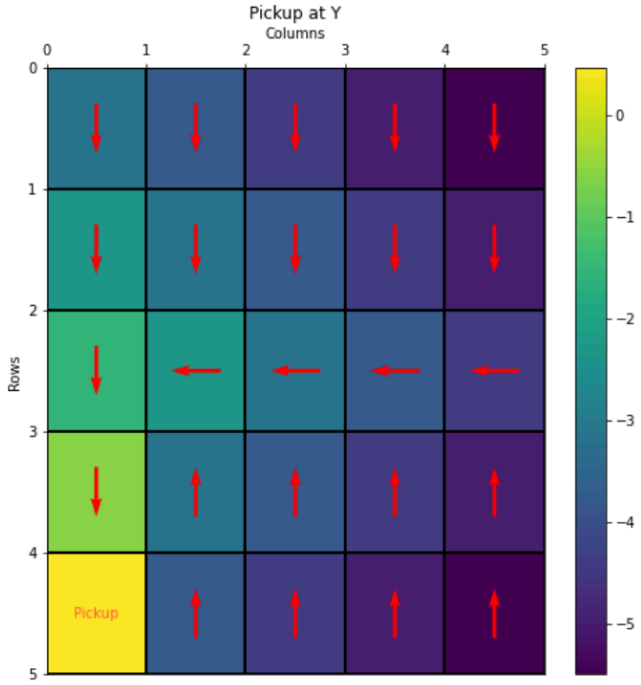


Plots for Heatmaps of Q Values for 50,00,000 Iterations of SMDP Q Learning :




Passenger in car, destination B
Columns
Rows
B
Pickup
Drop
Passenger in car, destination Y
Y
Pickup
Drop


Pickup at R
Columns
Rows
Pickup
R
Pickup at B
B
Pickup

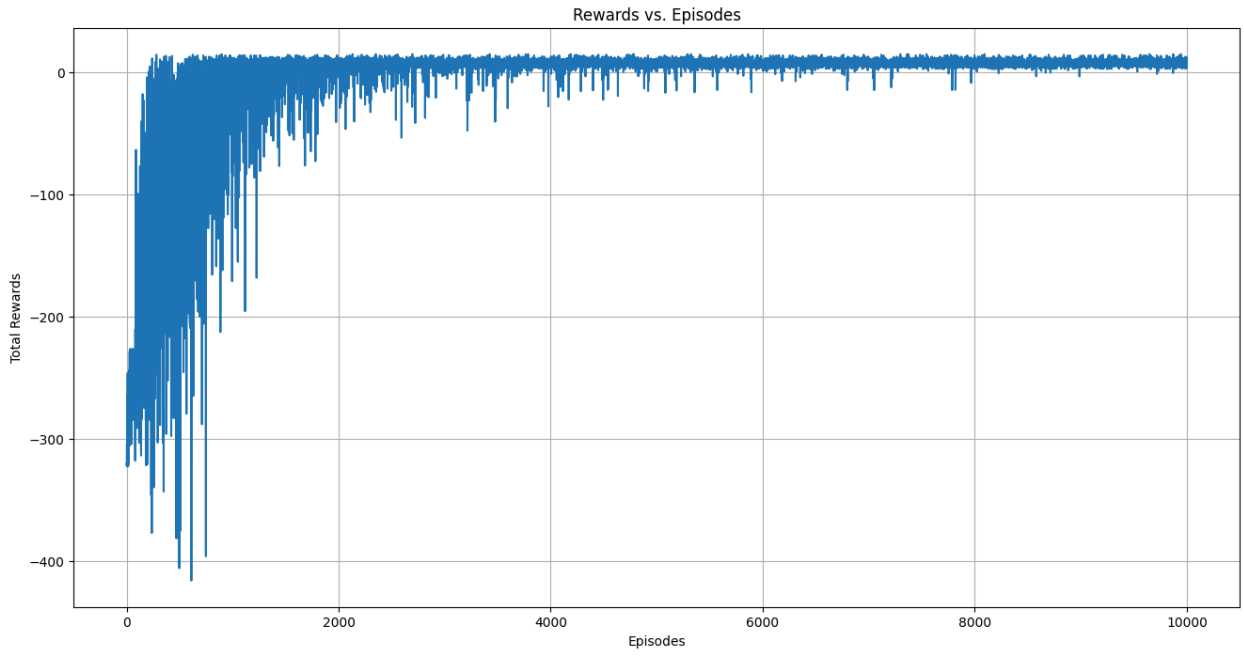


As seen from above, it's clear that the agent learns well for the chosen set of hyperparameters.
Both the primitive and the alternative options are chosen, but in most of the cases, the alternative options are chosen less times than the primitive options.

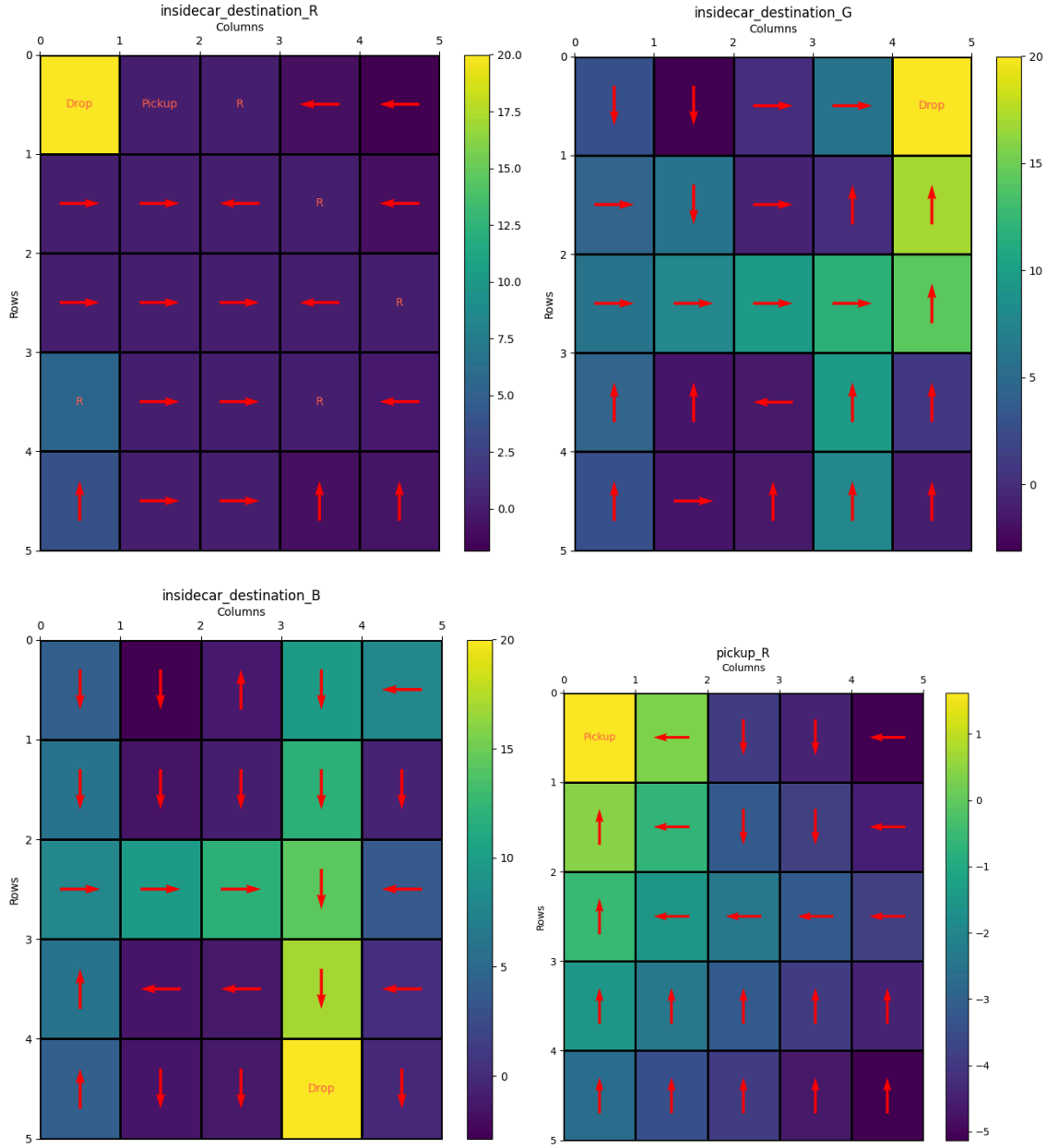
Intra Q LEARNING:

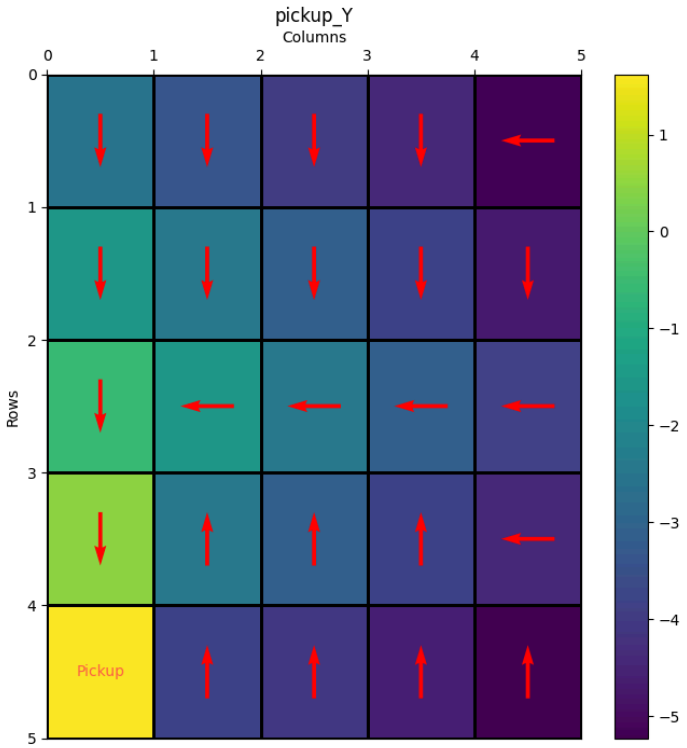
```
while not optdone:
    k += 1

    optact, optdone = Drive_to_R(env, state)
    next_state, reward, terminated, truncated, info = env.step(optact)
    done = terminated or truncated

    reward_bar = gamma * reward_bar + reward
    state = next_state
    opt_upd_list = [action]
    st_coords = list(env.decode(state))[:2]
    other_opts = [6,7,8,9]
    other_opts.remove(action)

    for oth in other_opts:
        if OPT_TO_POLICY_MAP[oth][st_coords[0], st_coords[1]]== optact:
            opt_upd_list.append(oth)

    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:

            q_values_SMDP[s_state][opt] += alpha * (reward_bar + (gamma ** k) * np.max(q_values_SMDP[next_state]) - q_values_SMDP[s_state][opt])
            update_frequency[state][opt] += 1
        else:

            q_values_SMDP[s_state][opt] += alpha * (reward_bar + (gamma ** k) *(q_values_SMDP[next_state][opt]) - q_values_SMDP[s_state][opt])
            update_frequency[state][opt] += 1

total_reward += reward_bar  # Accumulate reward for the episode
```

The update step for Intra-QLearning updates the Q-values of all the options in which the state-action is present.

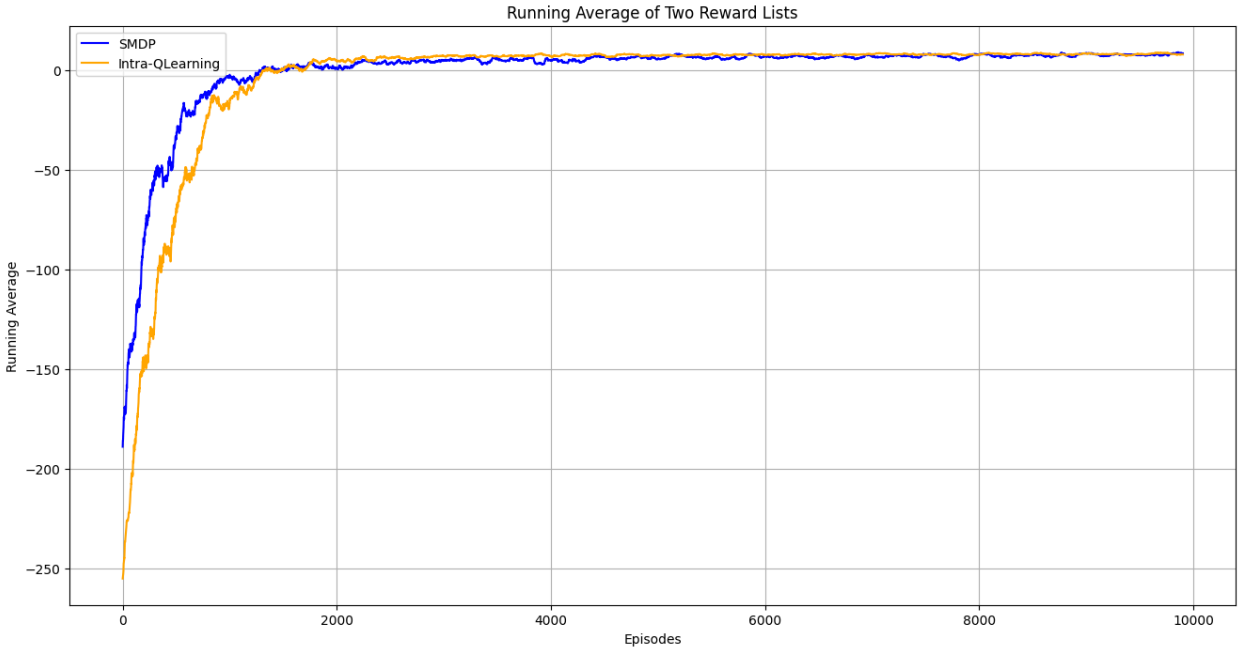


Plot for total rewards vs episodes for Intra Option Q Learning:





Comparison between SMDP and ioql - deterministic options:



Inference:
The above plot represents the running average reward values of SMDP and Intra-QLearning. Clearly, SMDP performs better as it has less regret compared to Intra-QLearning.However, they both converge to the same reward value.

Alternative Options:

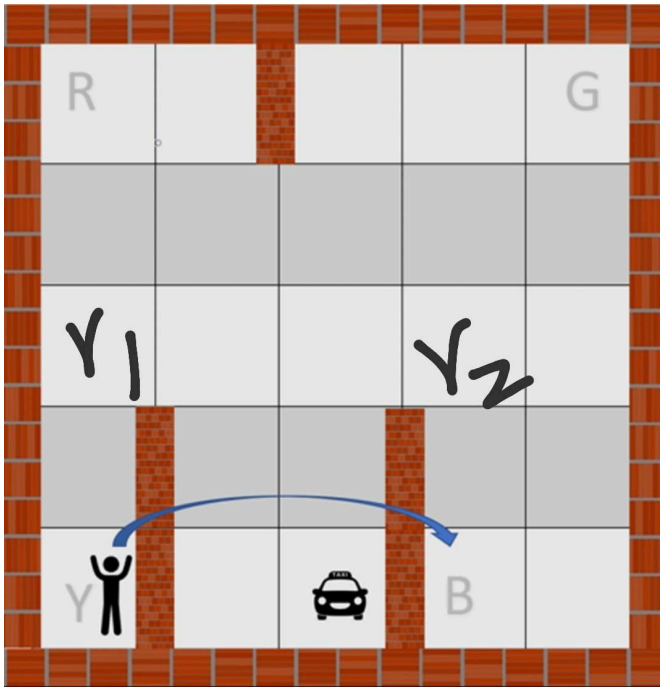
Part 3 of the assignment asks us to define alternate set of options that you can use to solve this problem, such that this set and the given options to move the taxi. And compute its performance and compare the SMDP with IOQL

These are the alternate options that we have chosen:

1) Go to state 1 - cell - (2,0) - Action r1
2) Go to state 2 - cell - (2,3) - Action r2

So in total there are 8 actions , 6 primitive and r1 and r2.

Why were these two options chosen?



As shown in the figure above of the environment, r1 denotes the option that reaches the state cell (2,0) from any cell of the environment and r2 denotes the option that reaches the state cell (2,3) from any cell of the environment.
From the Heatmaps in the previous experiments, it is clear that the cell (2,0) and (2,3) are more frequently visited than the other states in general. The state (2,0)

is in the middle of state cell R and state Y and the state (2,3) is in the middle of the state cell B and G.
The probability that agent goes from r1 to reach goal state (R or Y is nearby) in its optimal policy is 0.5. And similarly, the probability that the agent goes from r2 to reach goal state (G or B is nearby) in its optimal policy is 0.5. In the original case, this probability was lesser(0.25) as discussed earlier under the inference of SMDP vs Intra, due to less mutual exclusiveness of the options with the primitive actions.
So, we hope that choosing such alternative options would increase the chance of these options being a part of optimal policy that the agent learns over time.

1) The Alternative Options are deterministic and known policy: SMDP Q Learning

```
r1_MATRIX = np.array([[0,0,0,0,0],[0,0,0,0,0],[0,3,3,3,3],[1,1,1,1,1], [1,1,1,1,1]])


r2_MATRIX = np.array([[0,0,0,0,0],[0,0,0,0,0],[2,2,2,2,3],[1,1,1,1,1],[1,1,1,1,1]])
```

The above is the matric that contains the optimal policy to perform a particular alternate option.

```
def Drive_to_r1(env, state):
    coords = list(env.decode(state))[:2]
    optdone = False
    optact = r1_MATRIX[coords[0], coords[1]]

    if (coords == [2, 0]):
        optdone = True

    return [optact, optdone]

def Drive_to_r2(env, state):
    coords = list(env.decode(state))[:2]
    optdone = False
    optact = r2_MATRIX[coords[0], coords[1]]

    if (coords == [2, 3]):
        optdone = True

    return [optact, optdone]
```

✓ 0.0s

The code for SMDP is similar to the code done before.
Tuning of Hyperparameters:
Gamma was set to 0.9 (discount factor)

We need to Tune the hyperparameters alpha and epsilon.
Inorder to tune the hyperparameters, we used Bayesian Optimization.

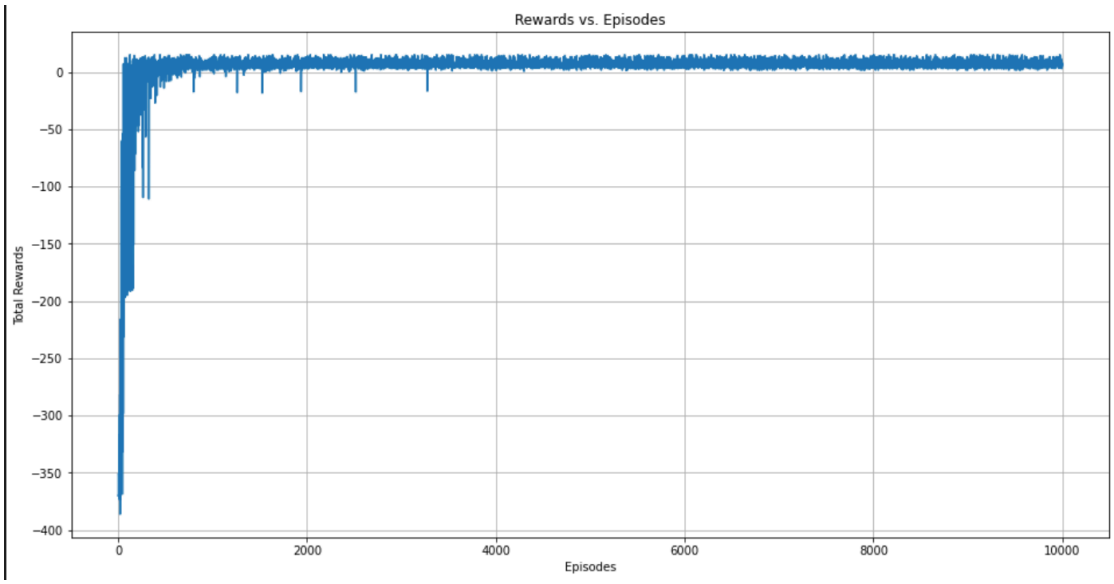
```
print(optimizer.max)
```
✓ 0.0s

```
{'target': 13037.186890000576, 'params': {'alpha': 0.5792554539550795, 'epsilon': 0.25}}
```

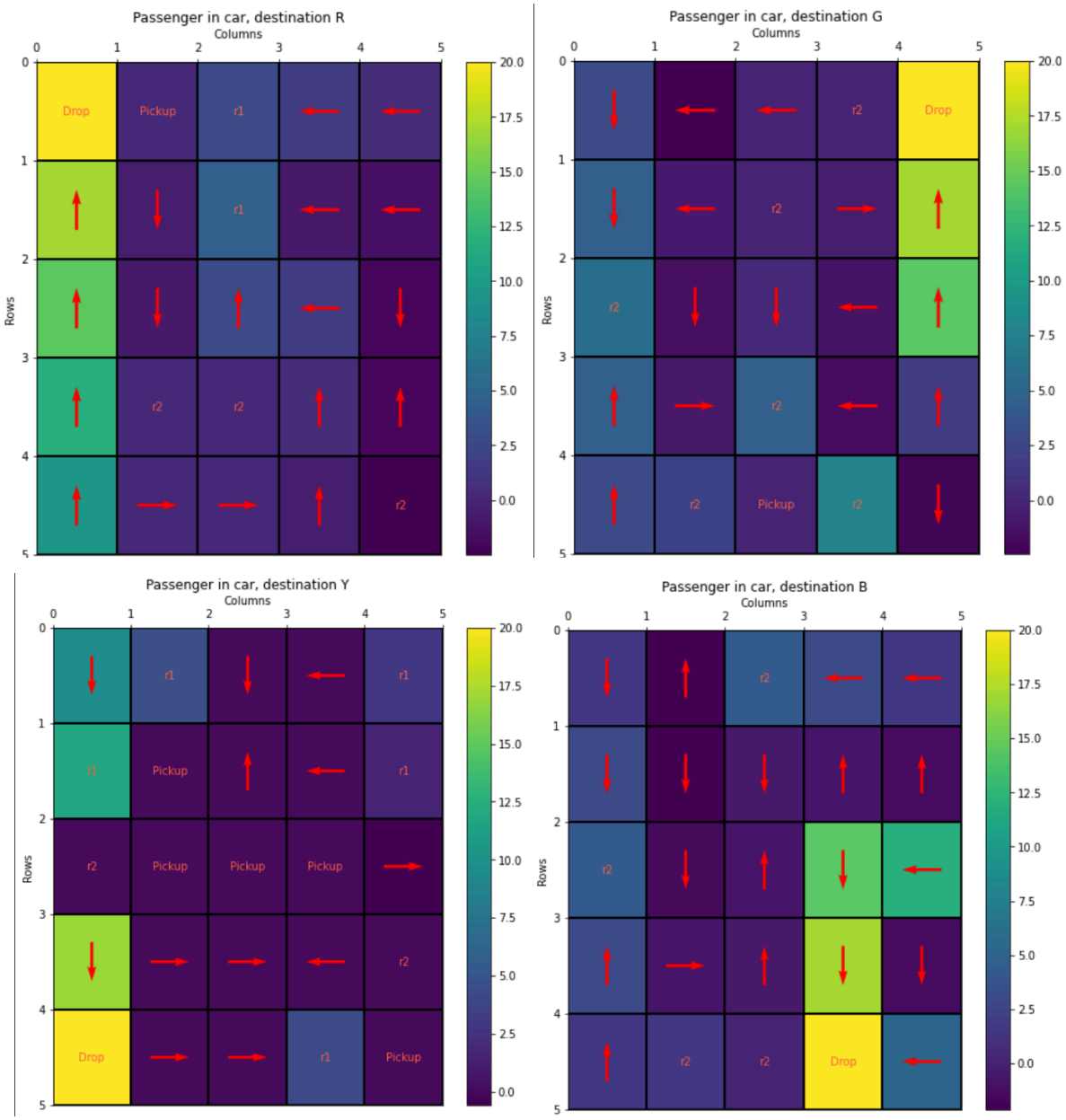
The search space : alpha (0.1 to 0.7) and epsilon (0.02,0.25)
The function SMDP, which runs the smdp algorithm for 5000 iterations and it returns the total rewards of the last 1000 episodes run. This was used as the metric to tune the hyperparameter to maximize the total reward of all the episodes.

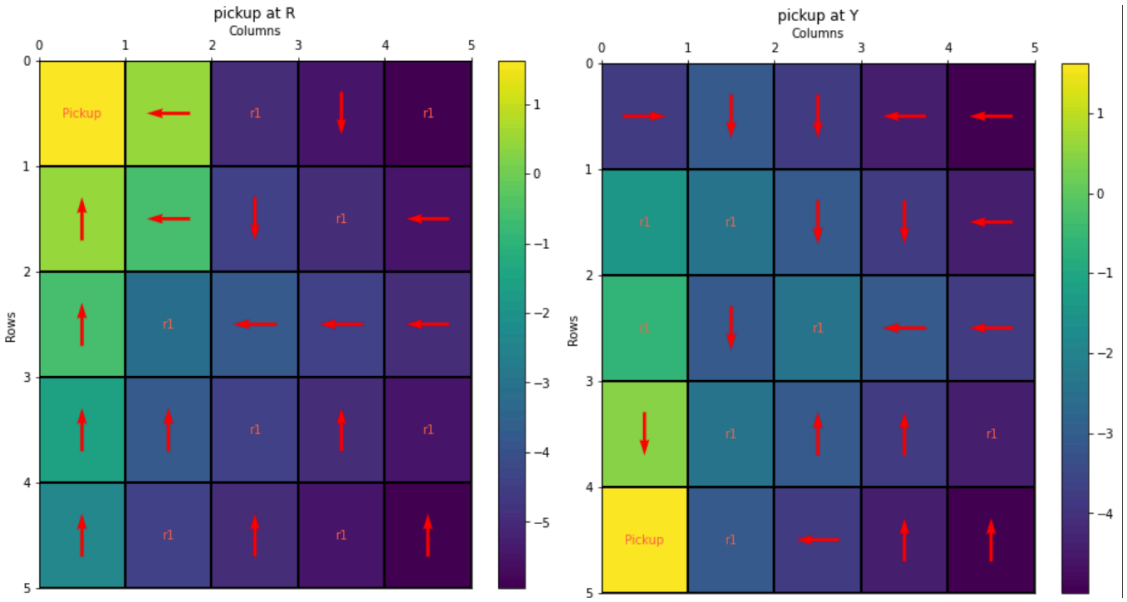
The optimal hyperparameters were found to be
Alpha = 0.579
Epsilon = 0.25

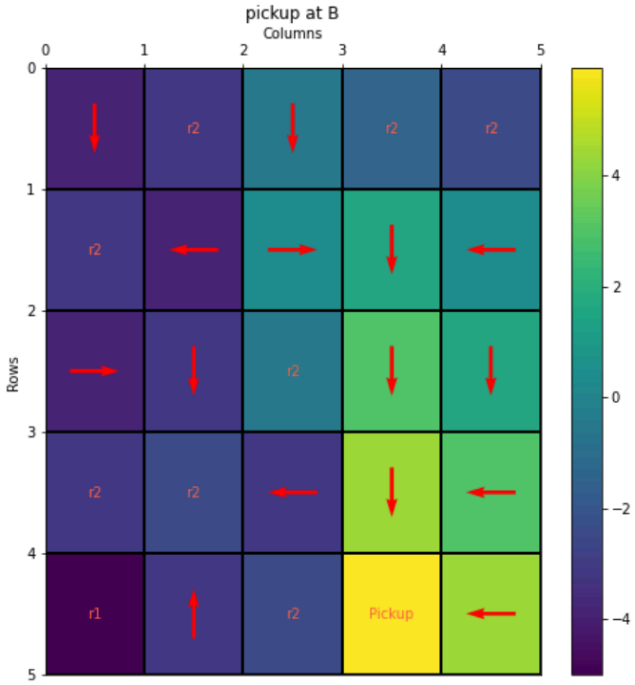
Plots for Custom Mutually Exclusive options : SMDP:



Heatmaps for visualizing the learnt policy and Q values:


Passenger in car, destination R
Columns
Rows
Drop
Pickup
r1
r2
Passenger in car, destination G
Drop
r2
Pickup
Passenger in car, destination Y
Drop
Pickup
Passenger in car, destination B
Drop





As expected, The choice of options r1 and r2, has shown that these options have a higher probability of being chosen in the optimal policy that the agent learnt. In case of pickup situations, options are preferred largely for our set of alternate options, than the given set of options.

Learning the options r1 and r2 Simultaneously: SMDP

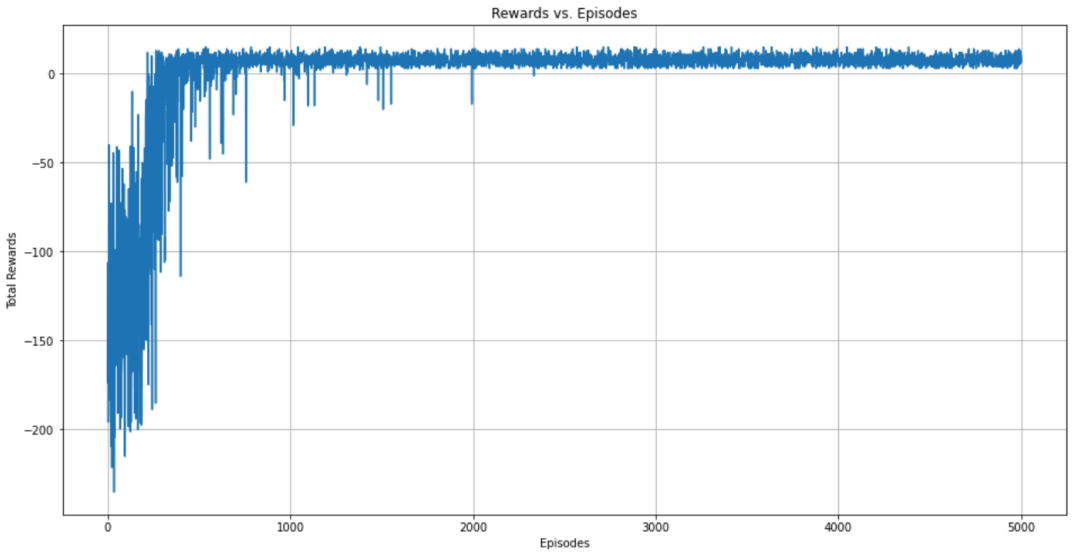
Tuning of Hyperparameters:
Gamma was set to 0.9 (discount factor)
We need to Tune the hyperparameters alpha and epsilon.
Inorder to tune the hyperparameters, we used Bayesian Optimization

```
print(optimizer.max)
✓ 0.0s
{'target': 5388.741594332561, 'params': {'alpha': 0.7, 'epsilon': 0.13865448466107505}}
```

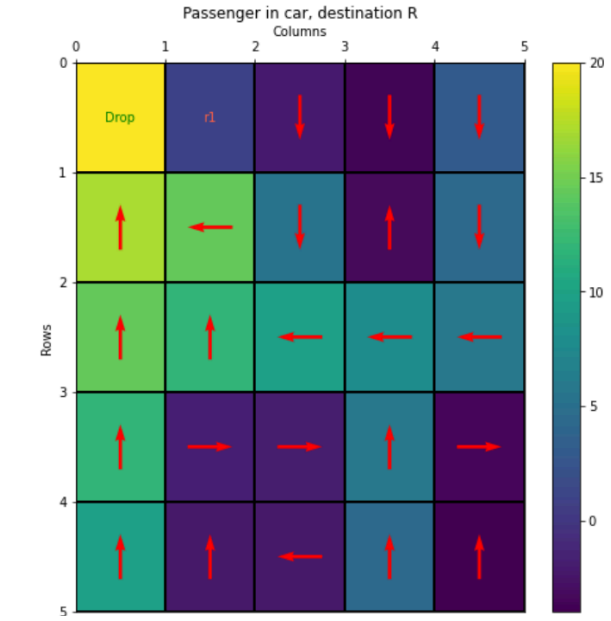
The search space : alpha (0.1 to 0.7) and epsilon (0.02,0.25)
The function SMDP, which runs the smdp algorithm for 5000 iterations and it returns the total rewards of the last 1000 episodes run. This was used as the metric to tune the hyperarameter to maximise the total reward of all the episodes.

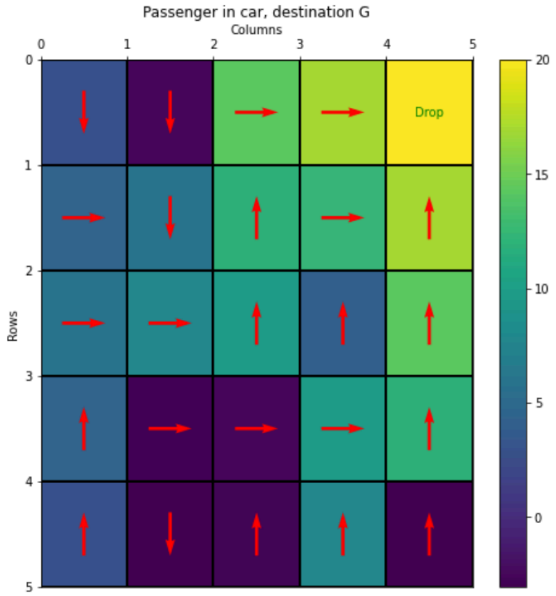
Alpha = 0.7
Epsilon = 0.138

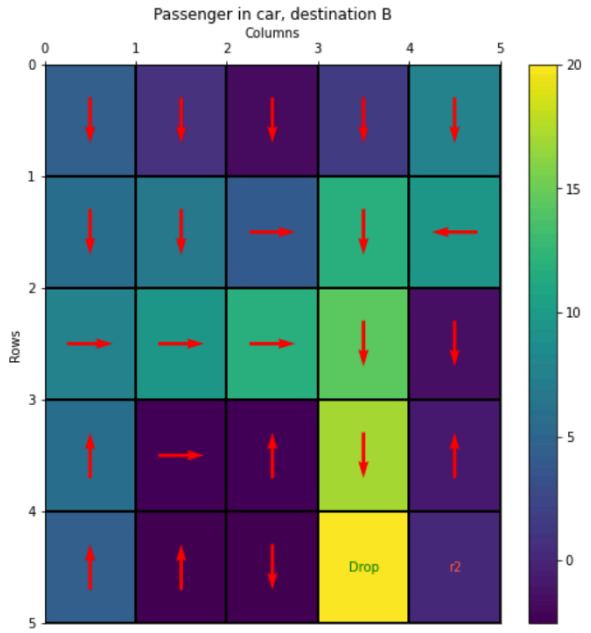
Plots for Custom Mutually Exclusive options : SMDP:

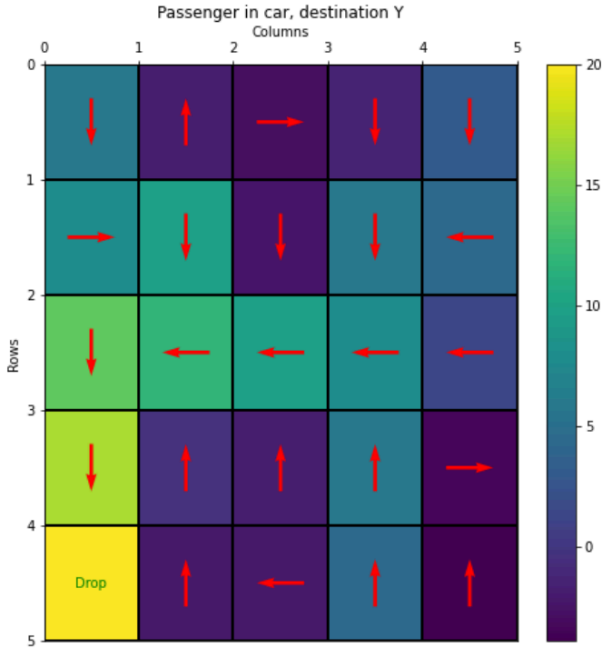


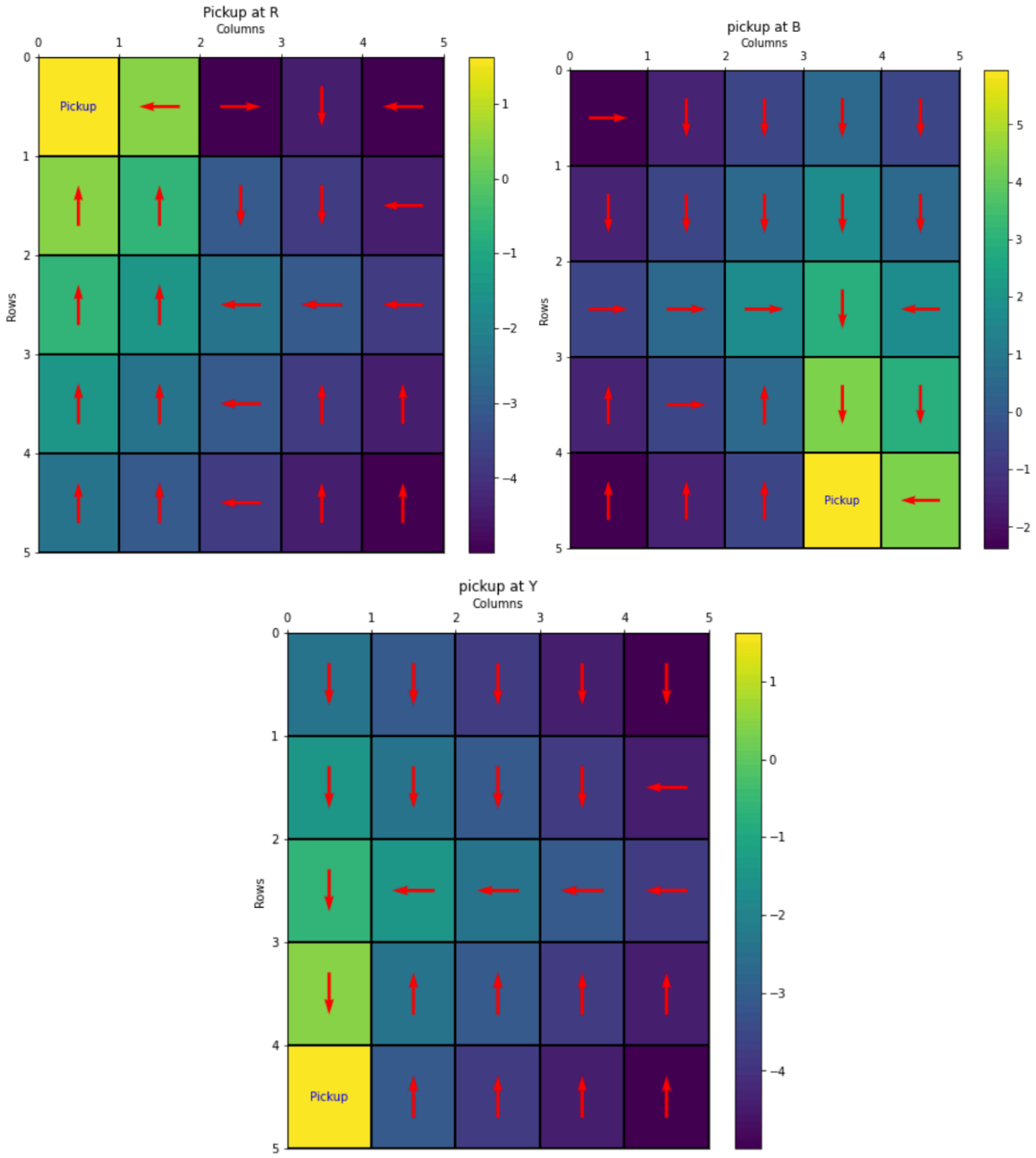
Heatmaps for visualizing the learnt policy and Q values:


Passenger in car, destination R
Columns
Rows
Drop
r1


Passenger in car, destination G
Columns
Rows
Drop


Passenger in car, destination B
Columns
Rows
Drop
r2


Passenger in car, destination Y
Columns
Rows
Drop



From the above heatmaps, the primitive actions are chosen more, than the alternative options,

INTRA OPTION:

A) Learning the options r1 and r2

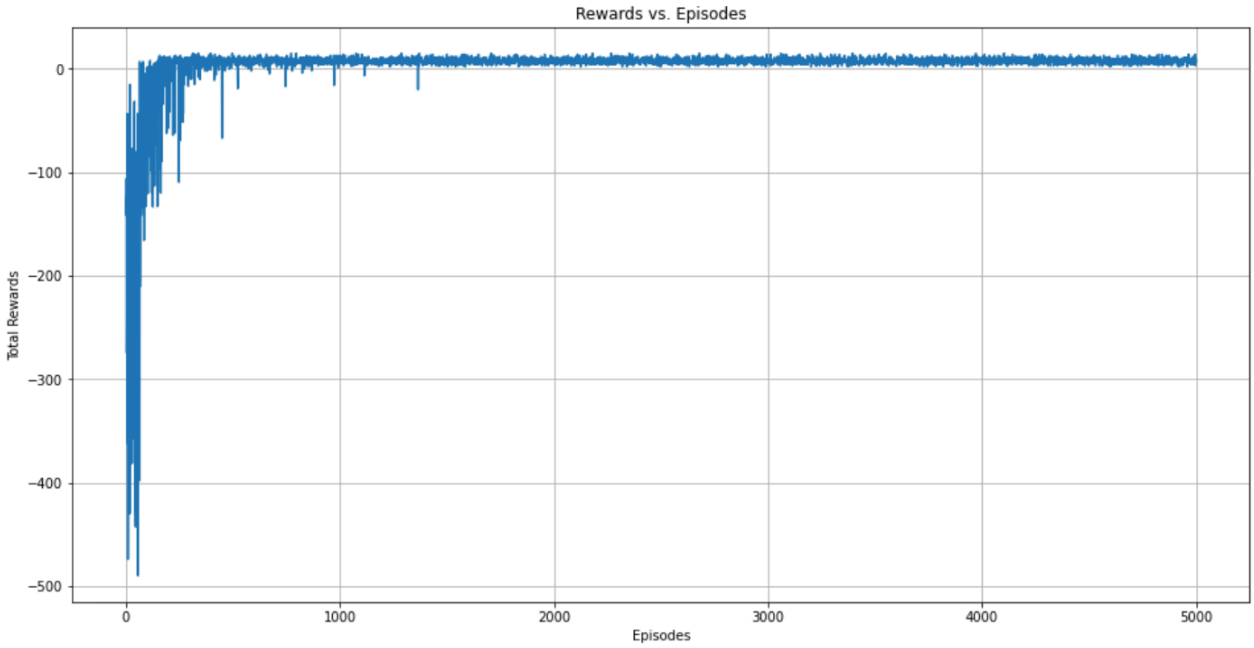
Tuning of Hyperparameters:
Gamma was set to 0.9 (discount factor)
We need to Tune the hyperparameters alpha and epsilon.
Inorder to tune the hyperparameters, we used Bayesian Optimization

```
print(optimizer.max)
✓ 0.0s
{'target': 18038.57484058183, 'params': {'alpha': 0.7, 'epsilon': 0.1576793782759692}}
```

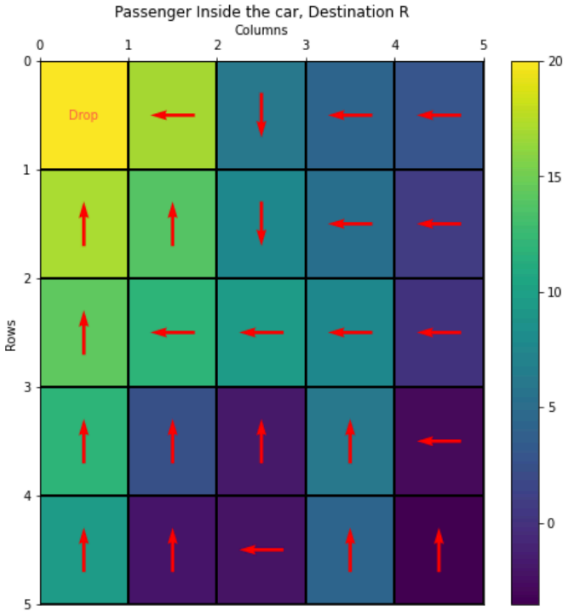
The search space : alpha (0.1 to 0.7) and epsilon (0.02,0.25)
The function SMDP, which runs the smdp algorithm for 5000 iterations and it returns the total rewards of the last 1000 episodes run. This was used as the metric to tune the hyperarameter to maximise the total reward of all the episodes.

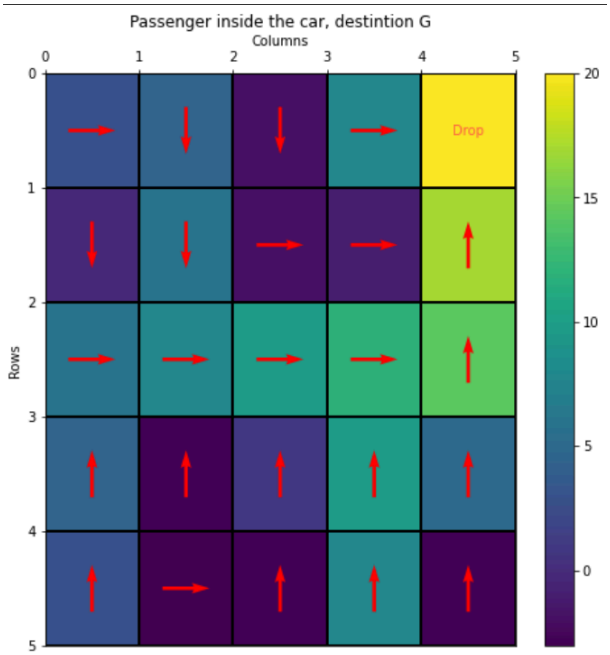
Alpha = 0.7
Epsilon = 0.157679

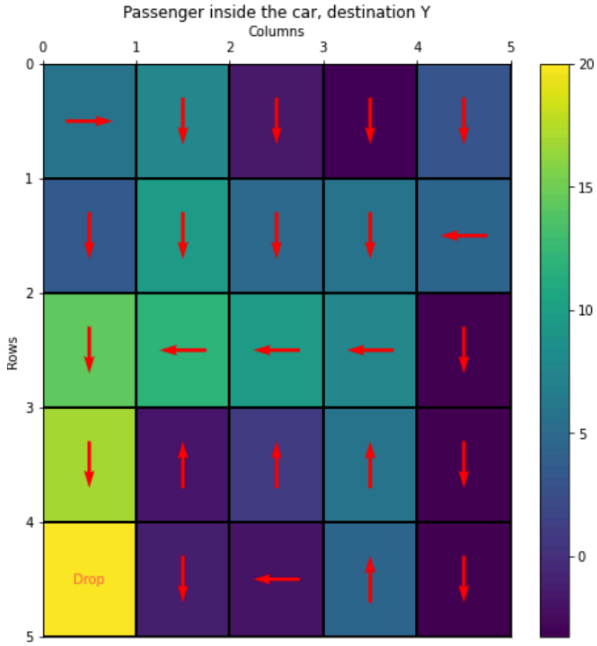
Plots for Custom Mutually Exclusive options : INTRA:

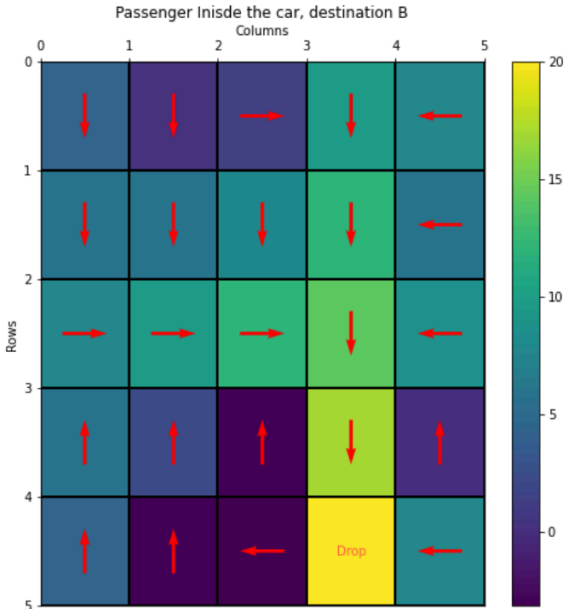


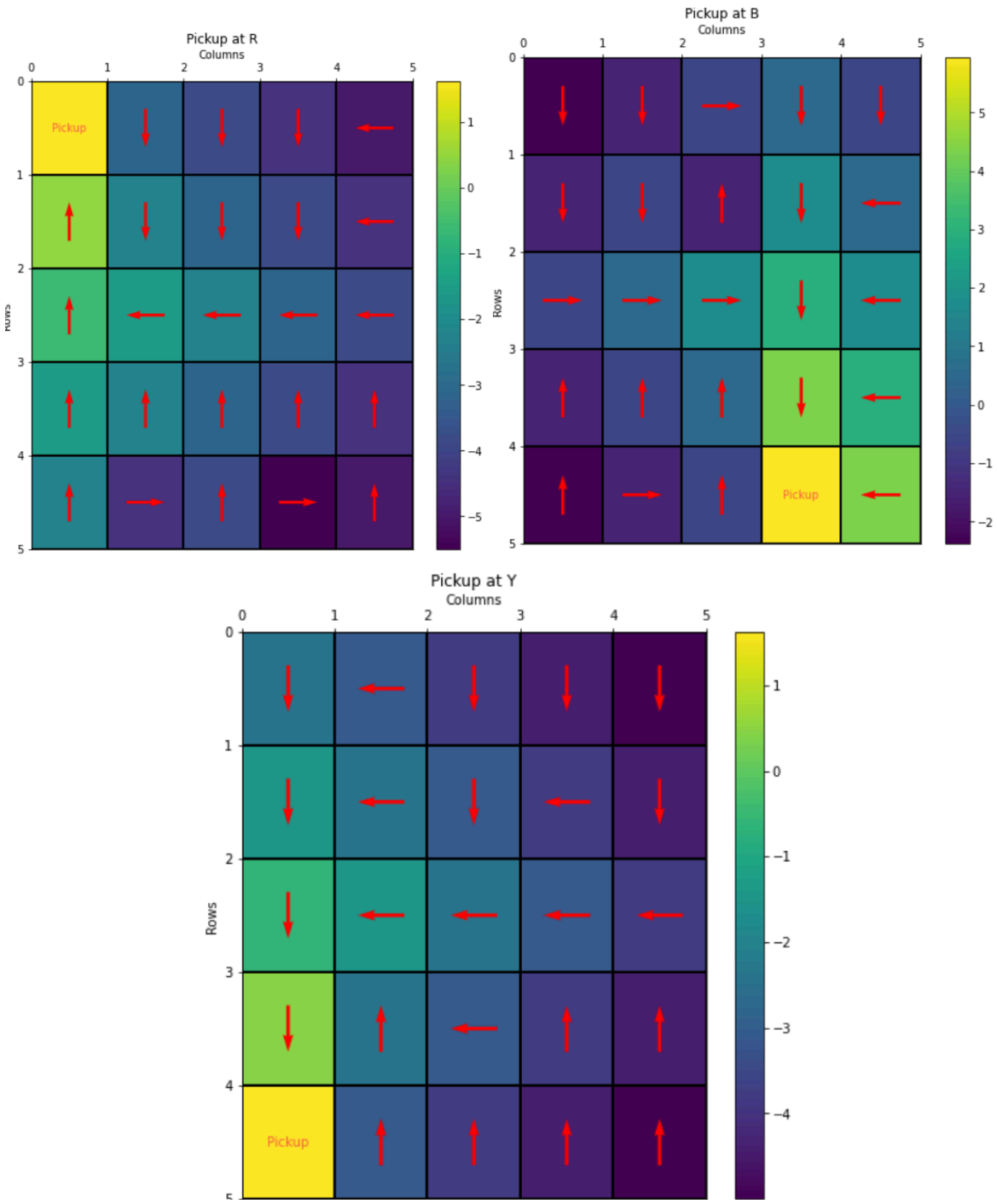
Heatmaps for visualizing the learnt policy and Q values:


Passenger Inside the car, Destination R
Columns
Rows
Drop


Passenger inside the car, destintion G
Columns
Rows
Drop

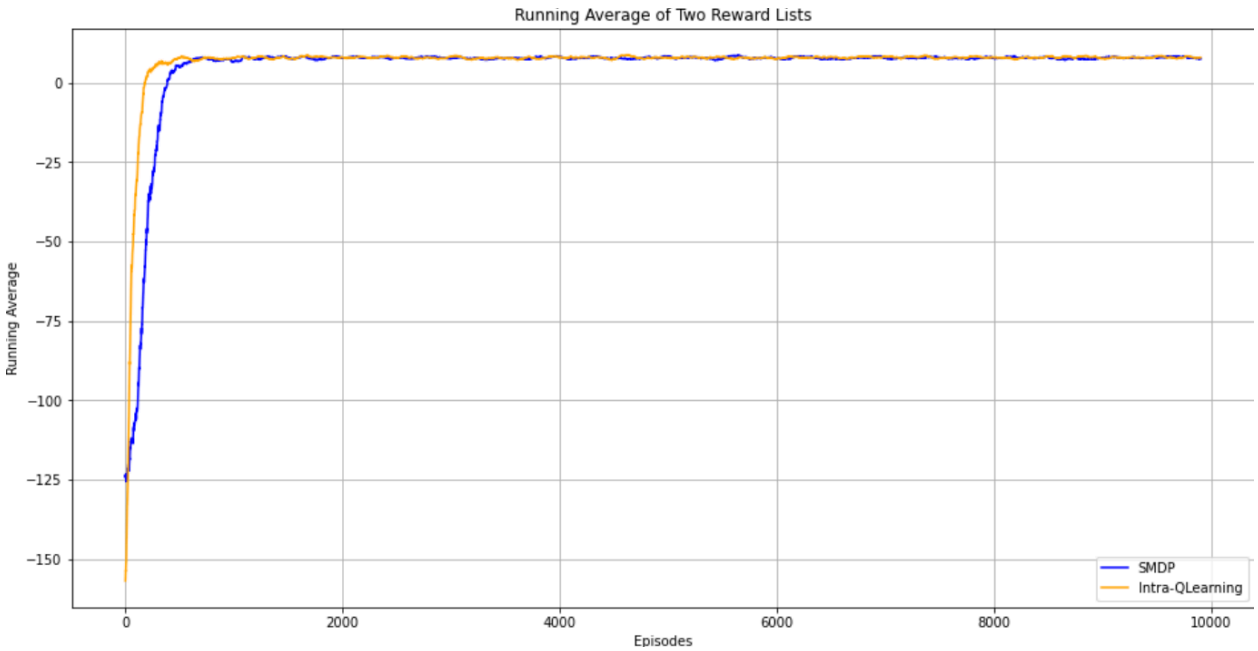

Passenger inside the car, destination Y
Columns
Rows
Drop


Passenger Inisde the car, destination B
Columns
Rows
Drop



From the above heatmaps, the primitive actions are chosen more, than the alternative options,

Comparison between SMDP and Intra Option Q Learning



As expected, From the above plots, it is evident that Intra option Q learning learns Faster than SMDP. They finally converge to similar values.
Also, it could be realized that the number of episodes required for convergence in this case is lesser than the number of episodes required for convergence in the case of 4 alternative options to reach R,Y, G and B. This could be because of “lesser number of options” to evaluate in this case. The results of this case of Learning the alternative options simultaneously are different from the case of Deterministic option policies in terms of the number of times the alternative options have been chosen in the final optimal policy learnt by the agent.

Link for Github repository (click here)(https://github.com/AnirudN/CS6700_CE21B014_ED21B038_PA3/tree/main)