**Mohit Sharma (2019CS10372) Vivek Choudhary (2019CS10413)**

Planning Taxi Domain

# Part A: Computing Policies

## Taxi Domains

* **State space:** We have defined it as a list of 5-element tuple (x, y, status, a, b), in which (x,y) represent coordinate of taxi, (a,b) represent coordinate of passenger and status is True or False which represent whether taxi is occupied by passenger or not. **Goal state** is reached when taxi has dropped the passenger at destination which means (goal[0],goal[1],False,goal[0],goal[1]), where goal is the drop location. Also, when status is True then the states contain only those tuples for which (x,y)=(a,b).

Hence size of states space = 5\*5\*5\*5 (for status=False) + 5\*5 (for status=True) = 650

* **Action space:** There are 6 actions possible [‘N’,’S’,’E’,’W’,’Pick’,’Drop’]. At every possible states(other than the goal state, in which episode ends and no further action takes place) all six actions can be performed, whether it will result in something or not doesn’t matter.
* **Transition model:** T(s,a,s’) <- transition[state][action][result], it stores probability of getting into state s’(result) from s(state) over taking action a(action). As this table is very large so in code we have stored only those T(s,a,s’) values which are non zero.
* **Reward model:** R(s,a,s’) <- reward[state][action][result], it stores reward for a(action) when applied from state s(initial state) and reaches on state s’(resultant state). Value of reward is as per stated in question. As this table is very large so only those R(s,a,s’) are stored in code for which T(s,a,s’) are non-zero.

Simulator: Implementation of different things like next state based on current state and current action, stochastic effect of actions, rewards for each action. A class MDP is made in code which have functions to support these stochastic effects of applications. Also it stored these states, actions, transition model and reward model.

Instance: An instance of the problem is defined uniquely by the grid(possible depots, its size and the walls it contains) and the destination depot on which the passenger has to be dropped finally to end the episode. On this instance, the initial location for taxi can be any grid while for the passenger it can be any depot.

## Value Iteration

Implemented value iteration by using the formula

V\_k+1(s) <- max over all a(actions available on s) (∑(over states s’) T(s,a,s’) \* [R(s,a,s’) + Y.V\_k(s’)] ), Y is discount factor. (in code V\_k+1=cvfn, V\_k=pvfn).

Termination of iteration when max-norm error becomes less than epsilon.

1. For discount factor = 0.9 and epsilon = 0.01, number of iterations required was 22.

The value of epsilon chosen is 0.01 because for discount factor of gamma, the value of reward +20 carried over to upto 20 states look ahead(going to passenger than going to drop location may take 10+10 actions in best case stochastic effects in a 5\*5 grid where these locations are farthest apart from each other). So the effect of +20 reward would reduce to (gamma^20)\*20. For gamma =0.9 this value would be approximately 2.43. But with stochastic effects this would reduce even further and thus it is better to have a even lower epsilon (< 2.43\*0.85) to account for these stochastic effects. Also, we have to run these value iterations for different discount factors and through above mentioned approximate calculations it can be observed that epsilon=0.01 holds good enough for gamma> 0.7.

1. Discount factor = 0.01 => 3 iteration

Discount factor = 0.10 => 4 iteration

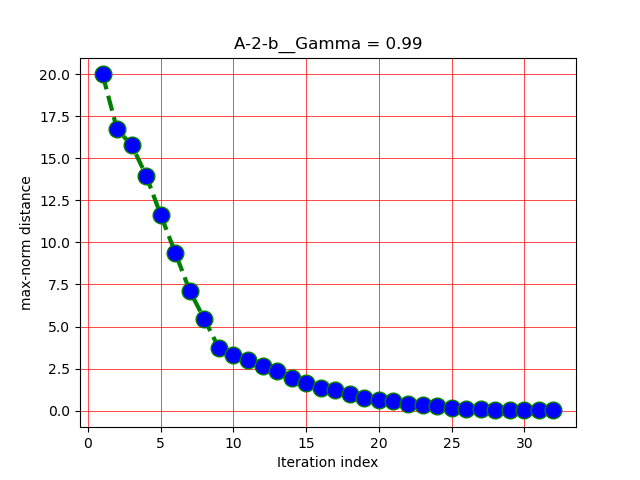
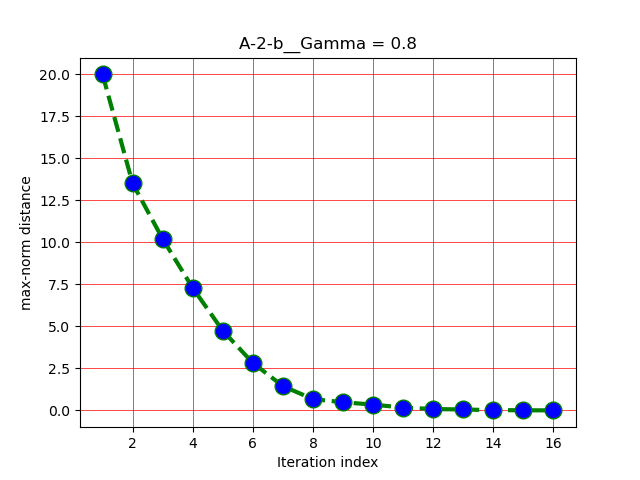
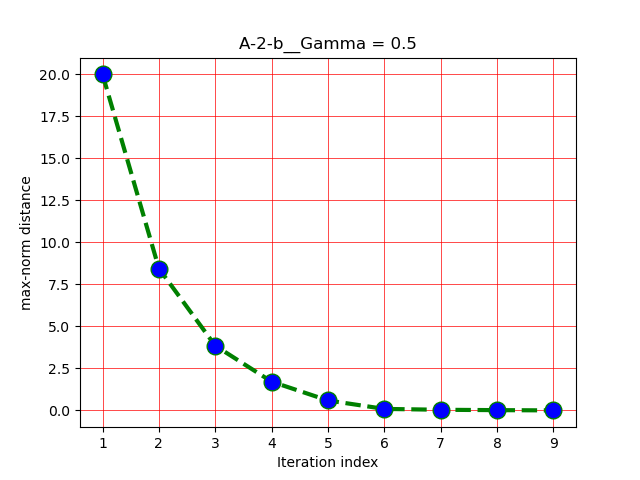
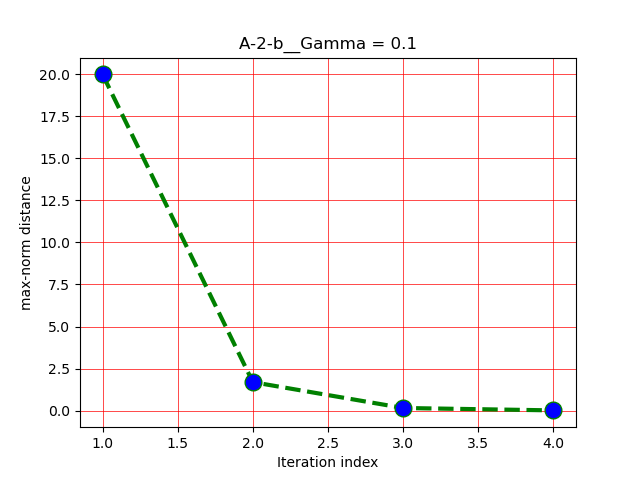
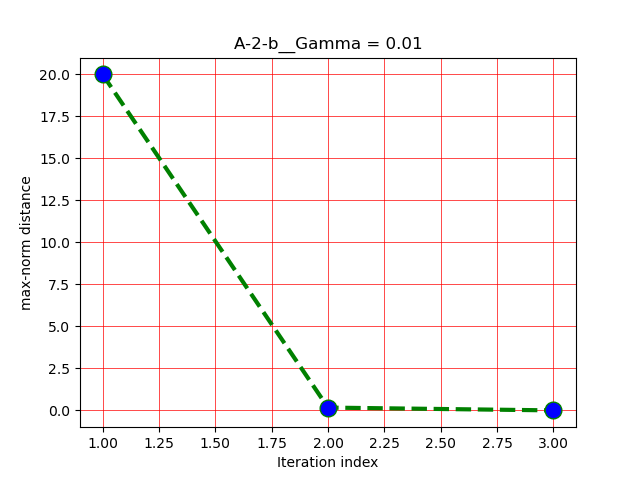
Discount factor = 0.50 => 9 iteration

Discount factor = 0.80 => 16 iteration

Discount factor = 0.99 => 32 iteration

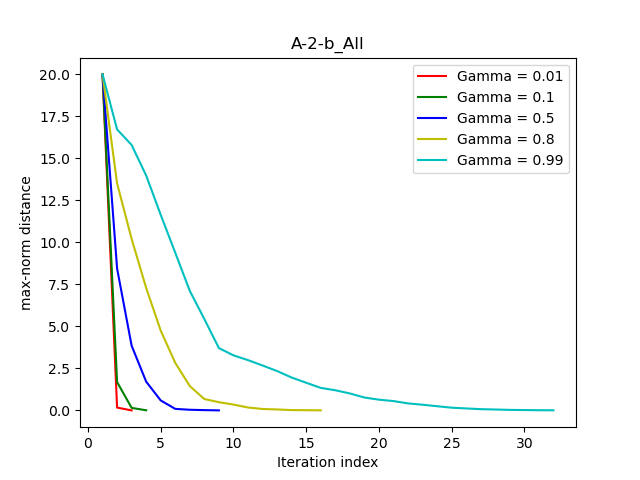
\*\* Epsilon was fixed to 0.01 for all above runs.

\*\* PLOT the graph b/w iteration index(x-axis) and max-norm(y-axis) for each and describe observation. \*\*



From above graphs we can see that as discount increases max-norm decrease slowly.

This happens because as we increase the value of discount factor then there is a larger effect of future rewards on the current value function and hence the change in value function is larger which implies that the max-norm error would be large and hence it converges slowly to the optimal value.



1. The common trend in all the runs is that the simulation for gamma = 0.1 never reaches the goal state and in-fact get stuck in grid with adjacent walls. This happens because the policy is made by giving more weightage to current goals rather than future rewards and thus the effect of +20 reward to end episode never gets accounted in the value iteration. Also our epsilon is 0.01 and for such a low value of gamma a epsilon in 2.E-19 range is needed even for a proper convergence of the value function to account the reward in all states properly.

For gamma=0.99, episodes end in most of the cases but still in some runs the policy only reaches till pickup state when only 20 steps of simulation are done. This is because of the stochastic effects of the simulation that the action given by policy does not follow the path of best possible state and rather follows the distribution given in question.

It is quite rare that policy gets stuck in gamma=0.99 as happening in gamma=0.1 case.

## Policy Iteration

For policy iteration we randomly choose some initial policy.

Then we repeat following steps till policy converges:

Evaluates the policy

Generate new improved policy based on evaluated value function

There are two ways to do policy iteration:

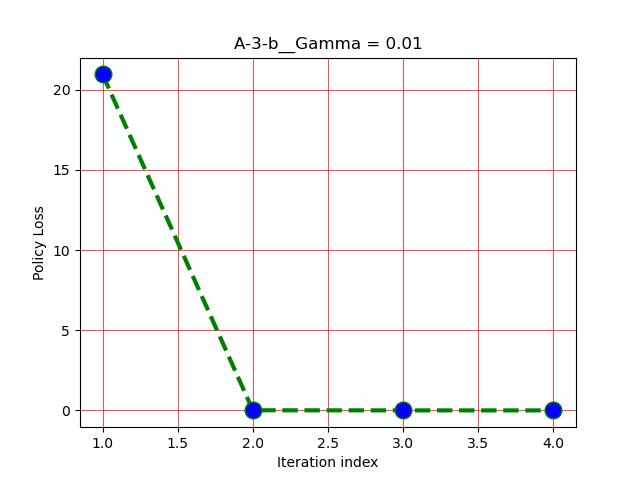
1. Value Iteration: We do similar steps as in value iteration method and just choose the action given by policy until the value function converges.
2. Linear Algebra: The value function evaluation is actually n equations with n variables.

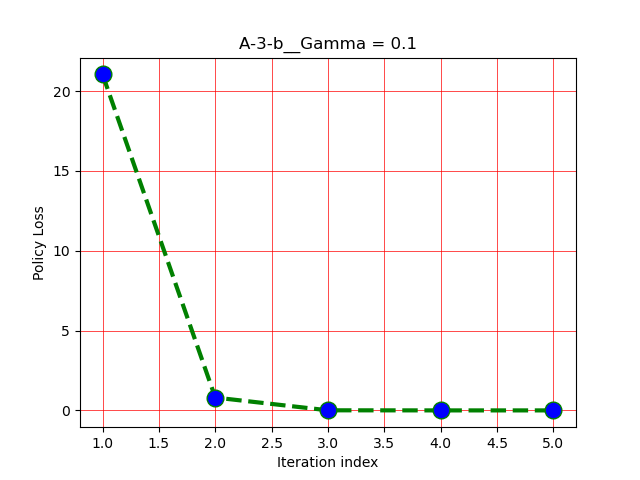
Here n is the total number of states (or n-1 variables and n-1 equations if not counting goal states value as an equation). We have solved these equations using numpy libraray from numply.linalg.solve() method. This takes O(n^3) time when doing exact calculations.

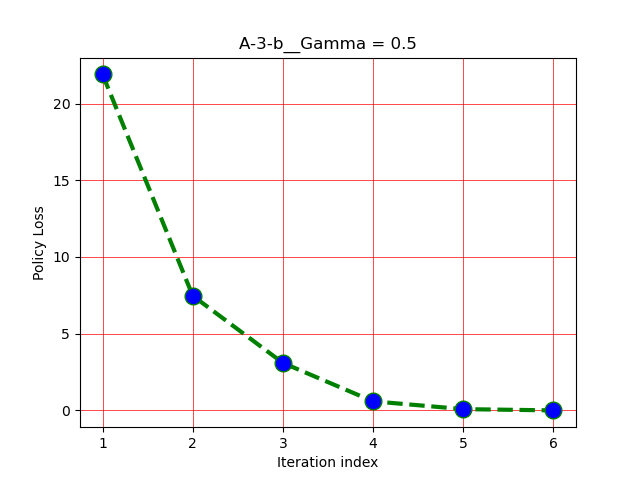
Method 1 of value iteration is better when the state space is large and actions have non zero probability for transition for most of the states as after some iterations the value function will start converging to approximate optimal values. Every iteration would take take O(n^2) even when actions can lead all states with some non-zero transition probability as we update n values and each value is based on n resultant states. In method2 of solving system of linear equations the time complexity is fixed to O(n^3) and hence a problem with large state space will take a lot of time.

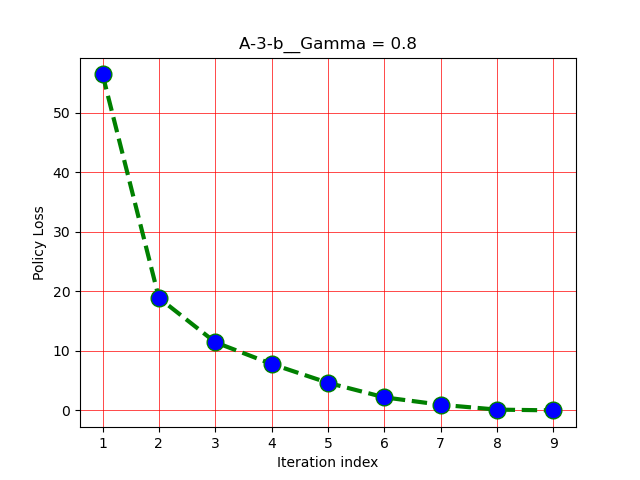
For given problem the state space is large and hence method 1 of value iteration is better.

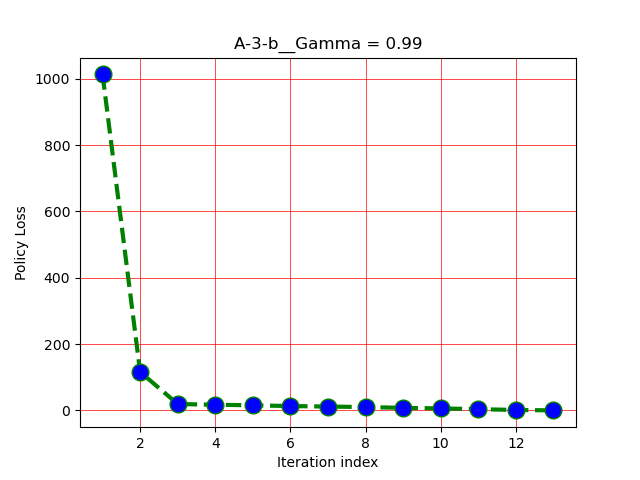
Plots of policy loss vs iteration index in policy iteration method.







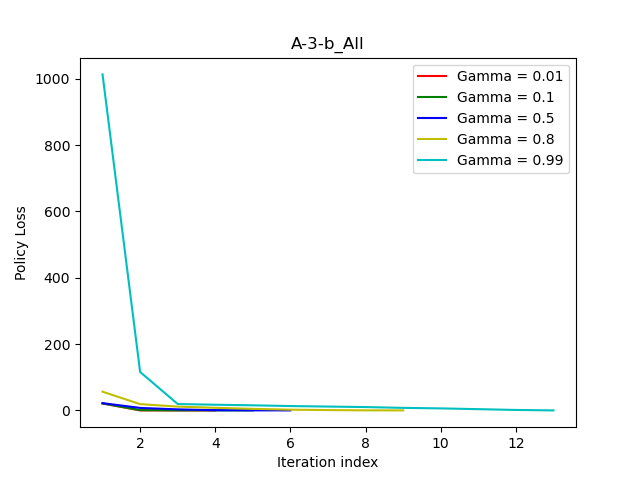




For high discount factors the initial policy loss is vey large as the random policy chosen does not take into account of the reward of +20 and similarly the policies made with low discount factor also does not take into account the future reward of +20 to much extent and hence the policy loss in case of low discount factor is small even at start.

As discount factor is increased it takes more iterations to reach convergence (similar reasons as in value iteration). Also, the policy loss is reduced majorly in the first few iterations itself as the notion of rewards is incorporated in it, the first iteration gives the states the policy which gives maximum reward going to states in 1 step look ahead.

Also, it can be observed that policy iteration converges earlier than value iteration for same value of discount factor.



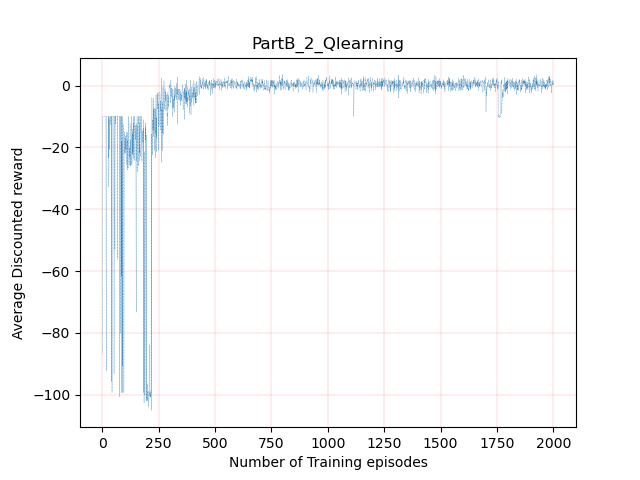
# Part B: Incorporating Learning

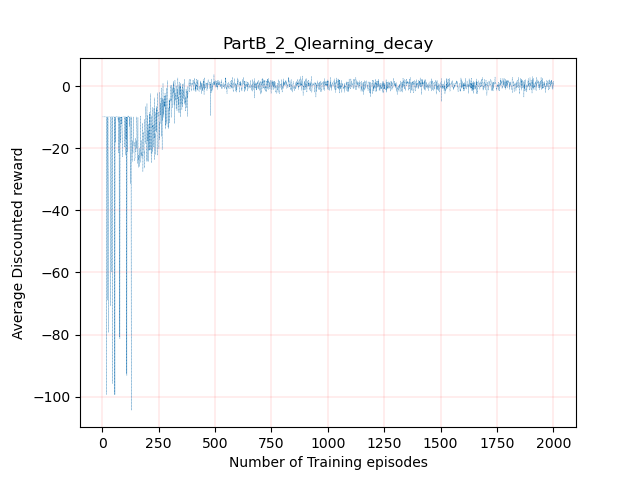
## 1) Learning algorithms

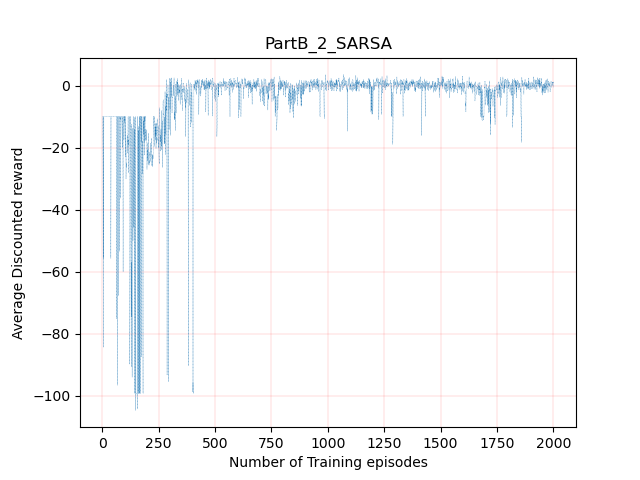
The four algorithms Q-learning, SARSA and their decaying exploration versions are implemented under the functions named as Qlearning, Qlearning\_decay, sarsa, sarsa\_decay.

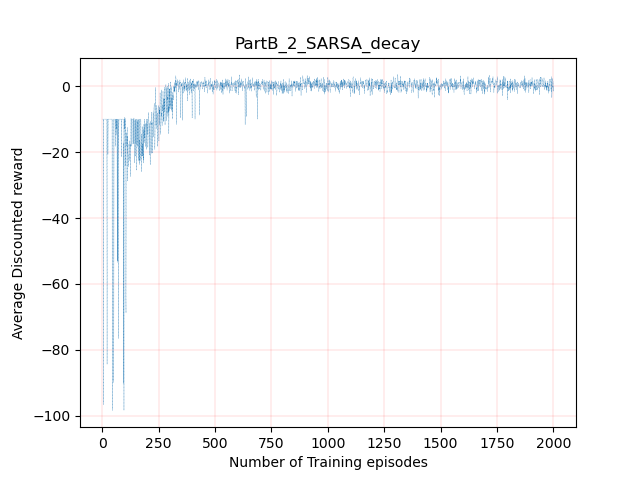
The algorithms with decaying exploration rate are decayed as epsilon/number of learning updates, where epsilon is starting value of exploration rate.

## 2) Convergence Analysis









In above plots the number of episodes used for training is 2000 and number of maximum steps taken per episode is 500. The average discounted reward is taken over 10 random initialisations of the taxi and passenger location in the depots.

Learning rate is taken 0.25 and discount factor is taken as 0.99.

Initial value of exploration rate is taken as 0.1

In case of algorithms with decaying exploration rate,

All the 4 algorithms converge to a discounted reward of 0. SARSA with a decaying exploration converges earliest followed by Q-learning with a decaying exploration rate. SARSA and Q-learning converges late relatively with almost similar number of episodes taken to converge. Still all algorithms converge within 500 episodes of training.

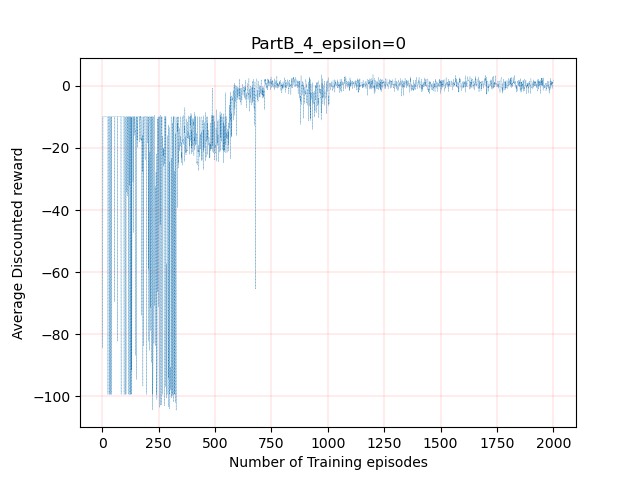
Algorithm with a decay in exploration have an early convergence in this problem because there is no unlikely action that can take place and provide a high reward, thus random explorations after certain training just account to some steps that eventually do not aid in the training by improving Q values for a better action in a given state. SARSA without any decay in exploration rate converges late because the selection of random action for exploration is carried forward in the next iteration. SARSA also shows a high variance in the discounted reward as compared to other algorithms due to carrying forward of this random exploration and thus providing bad actions to train.

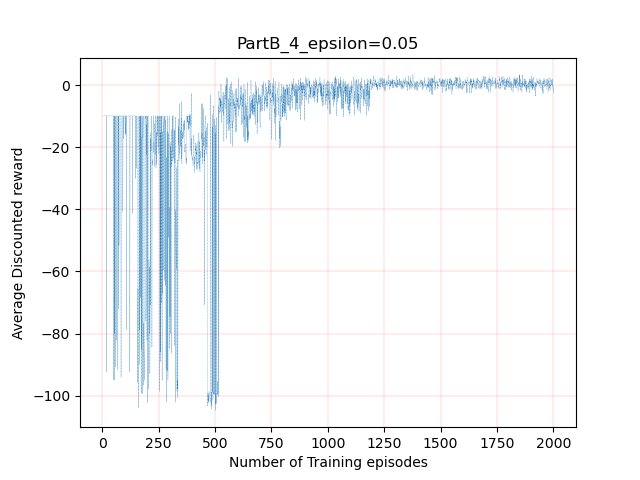
## 3)

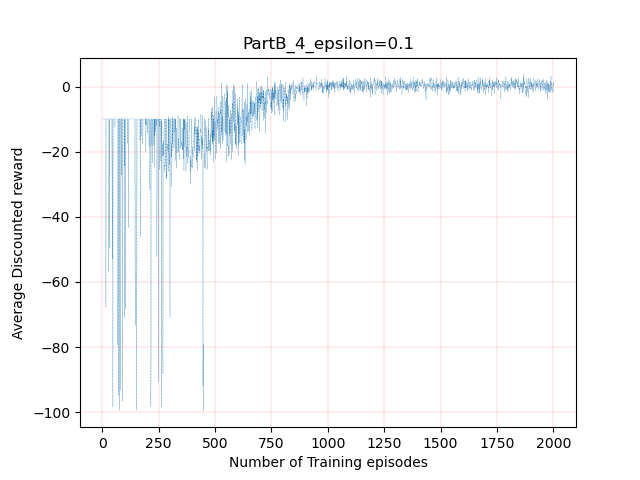
SARSA with decaying exploration is chosen for this part as it converges earliest

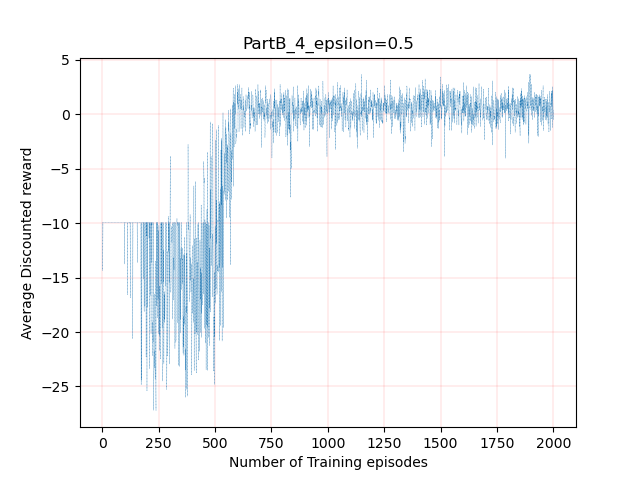
The algorithm is run by changing passenger and taxi’s initial location in different depots and every time the episode ends in less than 30 steps. In 1 out of 100 cases it takes more than 30 steps to end episode due to the negative effects of probabilistic distribution of actions.

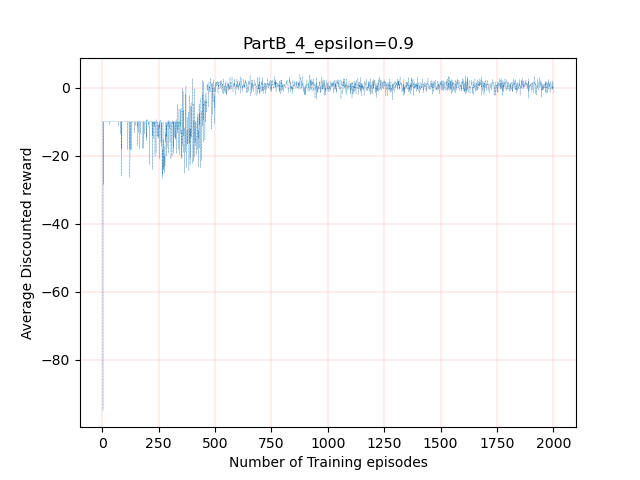
## 4)











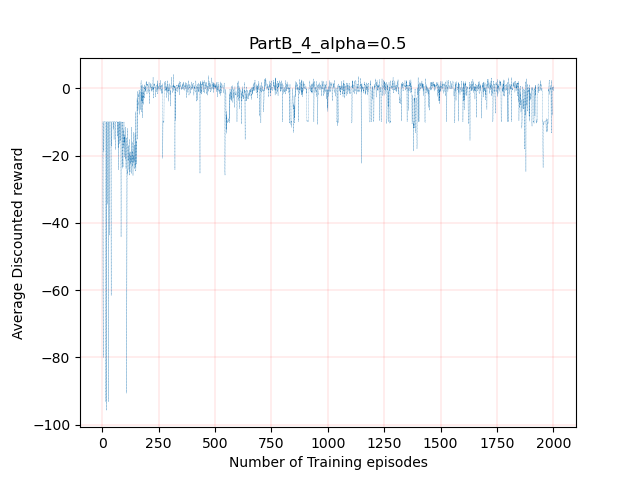
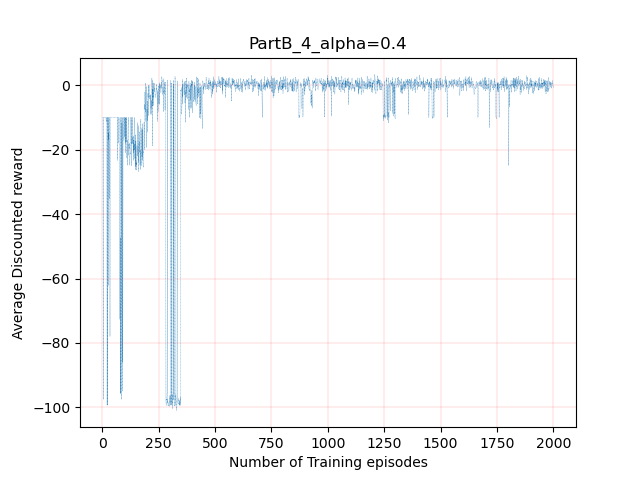
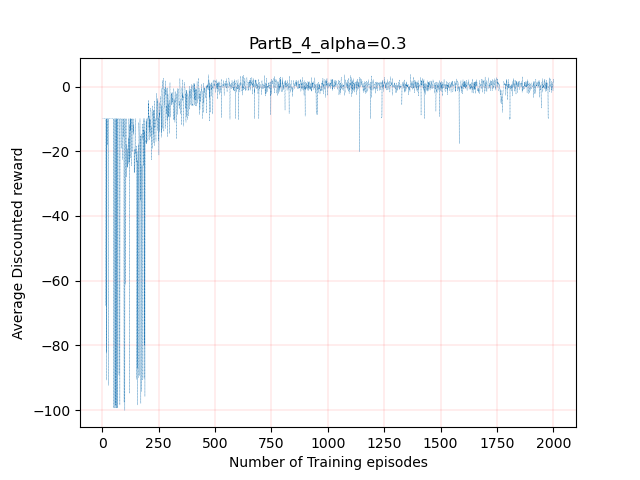
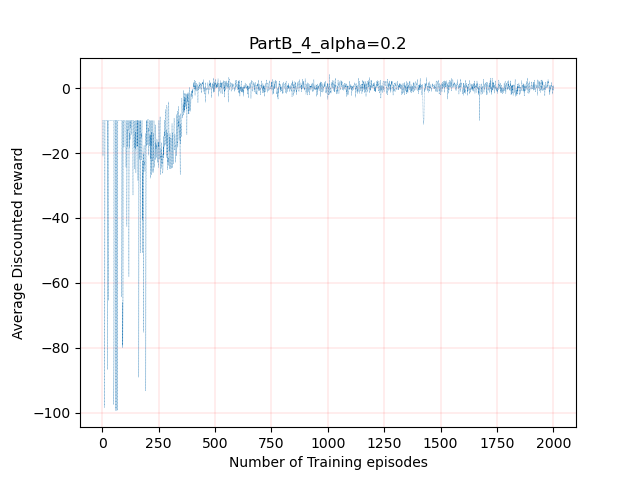
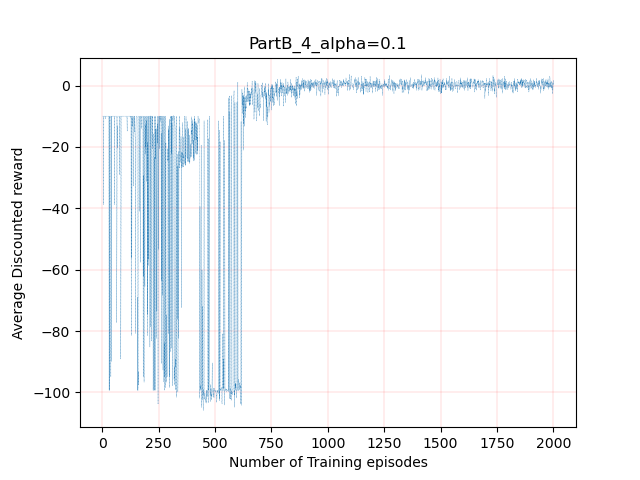
Learning rate was fixed to 0.1 in above plots, while discount factor was taken as 0.99.

Two things are observed in above plots:

1. As exploration rate is increased, the discounted rewards converge early.

When there is low exploration then initial policy which is actually a bad one makes the learning just stuck in that policy and so convergence is achieved lately when during start of new episode to train, we get chance of coming out of bad policies.

1. As exploration rate is increased, we get better reward using the policy learned during the learning phase, even before it converges. Reason is same as for above point.



Exploration rate was fixed to 0.1 in above plots, while discount factor was taken as 0.99.

As the learning rate is increased, the discounted reward converges early. But the variance of the discounted rewards obtained is also increased. High learning rate stresses to much on the new rewards we get in the learning step and that is the cause of high variance.

## 5)

The best algorithm is: Q learning decay

This algorithm shows the least variance and also converges early.

The given grid of 10\*10 is used to create an instance of the problem and then a policy is learned using Q-learning with decaying exploration rate where initial exploration rate is taken as 0.1, learning rate is taken as 0.25, discount factor is taken as 0.99. 10000 episodes of learning are used each with a max steps of 500.

The average discounted rewards it gives over 5 randomly created instance is: -13.085