University of Tehran

College of Engineering

School of Electrical and Computer Engineering

Intelligent Systems

Dr.Hosseini

Homework 6

Soroush Mesforush Mashhad

SN:810198472

Bahman 01

Contents

Abstract

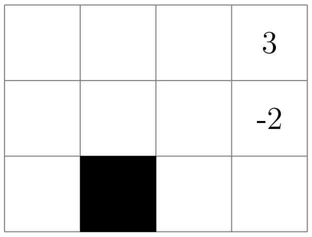
In this project we shall begin by performing the **Policy Iteration Algorithm** on the a given space.

In the second section we shall implement a problem via value iteration.

In the final section we shall implement a game via reinforcement learning.

1 Question 1 : Model based reinforcement learning (Analytic)

In this part we shall perform the policy iteration algorithm three times on the following space.



The probability to move in the desired direction is 0.6 and the other directions is 0.2 accordingly, it is important to note that the agent doesn't get penalized for moving.

1.1 Policy Iteration

We know that the policy iteration method is used to obtain the optimal policy to maximize long term rewards in reinforcement learning.

Policy iteration consists of three steps.

- **Random Policy Initialization**
- **Policy Evaluation**
- **Policy Improvement**

We shall follow the following steps.

- **Step 1**

  We randomly initialize the policy and set each state to zero.

- **Step 2**

  We use the **Bellman Equation**

$$
\begin{aligned}
V(s) &= r(s) + \gamma \max\left(\sum_{s',r} p(s',r|s,\pi(s))V(s')\right)\\
V_\pi(s) &= R_s^{\pi(s)} + \gamma \sum_{s'} P_{ss'}^{\pi(s)} V_\pi(s')\\
V(s) &= \sum_a \pi(s,a) \sum_{s'} P_{ss'}^a \left[R_{ss'}^a + \gamma V(s')\right]
\end{aligned}
$$

- **Step 3** Here we update the policy.

$$
\begin{aligned}
\pi(s) &= arg\max_a \sum_{s',r} p(s',r|s,a)V(s')\\
\pi'(s) &:= arg\max\left(R_s^a + \gamma \sum_{s'} P_{ss'}^a V_\pi(s')\right)
\end{aligned}
$$

We must repeat steps 2 and 3 until convergence occurs. What we explained was very theoretical and not very heuristic, we shall use the third form of the **Bellman** equation here.

1.1.1 Initialization

First of all we shall depict the block numbering for brevity.

0	3	5	8
1	4	6	9
2	■	7	10

We set each of the blocks to 0(other than 8 and 9 obviously), and we also set an arbitrary initial policy as below.

0	0	0	3
0	0	0	-2
0	■	0	0

→	↑	→	3
→	↑	↓	-2
←	■	→	→

1.1.2 First Iteration

Now we shall utilize the given equations.

$s = 0 \rightarrow V(s) = 0.6(0 + 0.2 \times 0) + 0.2(0 + 0.2 \times 0) + 0.2(0 + 0.2 \times 0) = 0$

$s = 1 \rightarrow V(s) = 0.6(0 + 0.2 \times 0) + 0.2(0 + 0.2 \times 0) + 0.2(0 + 0.2 \times 0) = 0$

$s = 2 \rightarrow V(s) = 0.6(0 + 0.2 \times 0) + 0.2(0 + 0.2 \times 0) + 0.2(0 + 0.2 \times 0) = 0$

$s = 3 \rightarrow V(s) = 0.6(0 + 0.2 \times 0) + 0.2(0 + 0.2 \times 0) + 0.2(0 + 0.2 \times 0) = 0$

$s = 4 \rightarrow V(s) = 0.6(0 + 0.2 \times 0) + 0.2(0 + 0.2 \times 0) + 0.2(0 + 0.2 \times 0) = 0$

$s = 5 \rightarrow V(s) = 0.6(0 + 0.2 \times 3) + 0.2(0 + 0.2 \times 0) + 0.2(0 + 0.2 \times 0) = 0.36$

$s = 6 \rightarrow V(s) = 0.6(0 + 0.2 \times 0) + 0.2(0 + 0.2 \times -2) + 0.2(0 + 0.2 \times 0) = -0.08$

$s = 7 \rightarrow V(s) = 0.6(0 + 0.2 \times 0) + 0.2(0 + 0.2 \times 0) + 0.2(0 + 0.2 \times 0) = 0$

$s = 8 \rightarrow V(s) = 3$

$s = 9 \rightarrow V(s) = -2$

$s = 10 \rightarrow V(s) = 0.6(0 + 0.2 \times 0) + 0.2(0 + 0.2 \times -2) + 0.2(0 + 0.2 \times 0) = -0.08$

So the updated table shall be as below.

0	0	0.36	3
0	0	-0.08	-2
0	■	0	-0.08

Now we must improve the policy.

$$\pi(s) = arg\max_a \sum_{s',r} p(s',r|s,a)V(s')$$

We shall write the needed equations.

$$\begin{aligned}
&\text{Policy doesn't change for s = 0,1,2,5}\\
&s = 3, \quad \rightarrow \text{ is chosen}\\
&s = 6, \quad \uparrow \text{ is chosen}\\
&s = 7, \quad \rightarrow \text{ is chosen}\\
&s = 10, \quad \uparrow \text{ is chosen}
\end{aligned}$$

The complete calculations have been included for one state as an example.

$$\begin{aligned}
&\leftarrow: \pi(5) = 0.6 \times 0 + 0.2 \times 0 + 0.2 \times -0.08 = -0.016\\
&\rightarrow: \pi(5) = 0.6 \times 3 + 0.2 \times -0.08 + 0.2 \times 0 = 1.64\\
&\downarrow: \pi(5) = 0.6 \times -0.08 + 0.2 \times 3 + 0.2 \times 0 = 0.12\\
&\uparrow: \pi(5) = 0.6 \times 0 + 0.2 \times 3 + 0.2 \times 0 = 0.6\\
&\pi(s) = arg\max_a \sum_{s',r} p(s',r|s,a)V(s'), \quad \rightarrow \text{ is chosen}
\end{aligned}$$

The improved policy grid is as follows.

→	→	→	3
→	←	↑	-2
←	■	↑	←

1.1.3 Second Iteration

→	→	→	3
→	←	↑	-2
←	■	↑	←

0	0	0.36	3
0	0	-0.08	-2
0	■	0	-0.08

Similar to the first iteration we have:

$s = 0 \rightarrow V(s) = 0.6(0 + 0.2 \times 0) + 0.2(0 + 0.2 \times 0) + 0.2(0 + 0.2 \times 0) = 0$

$s = 1 \rightarrow V(s) = 0.6(0 + 0.2 \times 0) + 0.2(0 + 0.2 \times 0) + 0.2(0 + 0.2 \times 0) = 0$

$s = 2 \rightarrow V(s) = 0.6(0 + 0.2 \times 0) + 0.2(0 + 0.2 \times 0) + 0.2(0 + 0.2 \times 0) = 0$

$s = 3 \rightarrow V(s) = 0.6(0 + 0.2 \times 0.36) + 0.2(0 + 0.2 \times 0) + 0.2(0 + 0.2 \times 0) = 0.0432$

$s = 4 \rightarrow V(s) = 0.6(0 + 0.2 \times 0) + 0.2(0 + 0.2 \times 0) + 0.2(0 + 0.2 \times 0) = 0$

$s = 5 \rightarrow V(s) = 0.6(0 + 0.2 \times 3) + 0.2(0 + 0.2 \times -0.08) + 0.2(0 + 0.2 \times 0.36) = 0.3712$

$s = 6 \rightarrow V(s) = 0.6(0 + 0.2 \times 0.36) + 0.2(0 + 0.2 \times -2) + 0.2(0 + 0.2 \times 0.0) = -0.0368$

$s = 7 \rightarrow V(s) = 0.6(0 + 0.2 \times -0.08) + 0.2(0 + 0.2 \times -0.08) + 0.2(0 + 0.2 \times 0) = -0.0128$

$s = 8 \rightarrow V(s) = 3$

$s = 9 \rightarrow V(s) = -2$

$s = 10 \rightarrow V(s) = 0.6(0 + 0.2 \times 0) + 0.2(0 + 0.2 \times -2) + 0.2(0 + 0.2 \times -0.08) = -0.0832$

So the updated table shall be as below.

0	0.0432	0.3712	3
0	0	-0.0368	-2
0	■	-0.0128	-0.0832

Now we shall update the policy in a manner similar as before. The updated policy table is as follows.

→	→	→	3
→	↑	↑	-2
←	■	↑	←

1.1.4 Third Iteration

In the third iteration we shall perform everything the same as before.

→	→	→	3
→	↑	↑	-2
←	■	↑	←

0	0.0432	0.3712	3
0	0	-0.0368	-2
0	■	-0.0128	-0.0832

$s = 0 \rightarrow V(s) = 0.6(0 + 0.2 \times 0.0432) + 0.2(0 + 0.2 \times 0) + 0.2(0 + 0.2 \times 0) \approx 0.005$

$s = 1 \rightarrow V(s) = 0.6(0 + 0.2 \times 0) + 0.2(0 + 0.2 \times 0) + 0.2(0 + 0.2 \times 0) = 0$

$s = 2 \rightarrow V(s) = 0.6(0 + 0.2 \times 0) + 0.2(0 + 0.2 \times 0) + 0.2(0 + 0.2 \times 0) = 0$

$s = 3 \rightarrow V(s) = 0.6(0 + 0.2 \times 0.3712) + 0.2(0 + 0.2 \times 0) + 0.2(0 + 0.2 \times 0.0432) = 0.046$

$s = 4 \rightarrow V(s) = 0.6(0 + 0.2 \times 0.0432) + 0.2(0 + 0.2 \times -0.0368) + 0.2(0 + 0.2 \times 0) = 0.004$

$s = 5 \rightarrow V(s) = 0.6(0 + 0.2 \times 3) + 0.2(0 + 0.2 \times -0.0368) + 0.2(0 + 0.2 \times 0.3712) = 0.3734$

$s = 6 \rightarrow V(s) = 0.6(0 + 0.2 \times 0.3712) + 0.2(0 + 0.2 \times -2) + 0.2(0 + 0.2 \times 0.0) = -0.0354$

$s = 7 \rightarrow V(s) = 0.6(0 + 0.2 \times -0.0368) + 0.2(0 + 0.2 \times -0.0832) +$
$0.2(0 + 0.2 \times -0.0128) = -0.008$

$s = 8 \rightarrow V(s) = 3$

$s = 9 \rightarrow V(s) = -2$

$s = 10 \rightarrow V(s) = 0.6(0 + 0.2 \times -0.0128) + 0.2(0 + 0.2 \times -2) +$
$0.2(0 + 0.2 \times -0.0832) = -0.085$

So the table shall be as below.

0.005	0.046	0.3734	3
0	0.004	-0.0354	-2
0	■	-0.008	-0.085

We can update the policy table as such.

→	→	→	3
↑	↑	↑	-2
←	■	↑	←

Just as a sidenote it is worth noting that in the $s = 2$ block there is no difference how we choose the moving style, so to make the policy look better we do the following.

→	→	→	3
↑	↑	↑	-2
↑	■	↑	←

2 Question 2 : Model Based Reinforcement Learning (Implementation)

First of all, we shall illustrate our problem just to make everything clear.

We face a classic reinforcement learning problem known as the **Gambler's problem**, someone wants to take part in a gambling festival, he places his bids on the outcome of a coin toss, if heads appears he wins the stakes and if tails appears he loses his stakes, the goal for this gambler is to reach 100 dollars.

In this problem, we show the probability of heads coming out of a coin toss with p_h, the bid the gambler places is always an integer number, whilst we our modeling the problem, each transition that results in a victory has a +1 reward and 0 in other cases. Due to the fact that we have an undiscounted Markov decision process the value of γ shall be set to 1 accordingly.

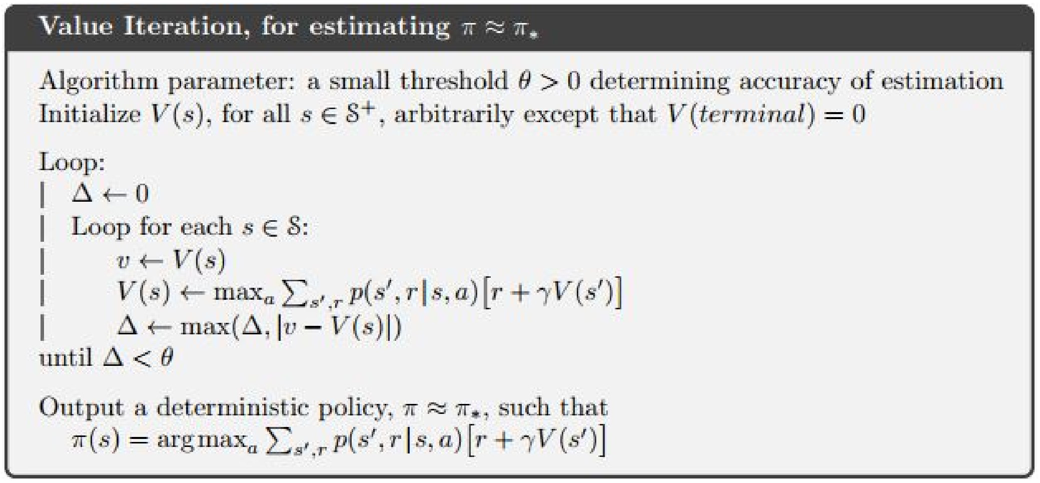
2.1 Value iteration algorithm

First of all, we shall explain the value iteration algorithm to freshen our minds. As we had in our lecture notes we have.

- Initialize V(s) arbitrarily

- Loop until policy is good enough
  - Loop for $s \in S$
    - Loop for $a \in A$
      - $Q(s,a) = \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V(s')\right]$
      - V(s) = max Q(s,a)

Another notation is as described in the Sutton and Barto reinforcement learning textbook.



```
Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
|  $\Delta \leftarrow 0$
|  Loop for each $s \in \mathcal{S}$:
|       $v \leftarrow V(s)$
|       $V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V(s')\right]$
|       $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
   $\pi(s) = \arg\max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V(s')\right]$
```

Figure 1: Value Itration

I have used the logic behind this notation in my implementation. Now I shall give some clear and explanation on how we approach to solve this problem.

As explained in the assignment description we face an undiscounted Markov decision process, first of all we must understand the difference between the state and acion in this problem, we define the states (s) as the fortune of the gambler, obviously this is something between 1 and 99 because if his fortune equals 0 then he has become bankrupt and if he reaches 100 he has won, as for the actions, we logically define them as the bids the gambler places, we must pay attention that we define the upper bound of this bids corresponding to the p_h values, this shall be discussed with more detail, also we have initialized $Fortune = 100$ as 1 and $Fortune = 0$ as 0 which are the states where the game is finished, this is done merely to have a better plot.

What we shall do is exactly implement the algorithm in **Figure 1** as follows.

```
import numpy as np
import pandas as pd
import seaborn as sn
from matplotlib import pyplot as plt
import random

```

Figure 2: Included libraries

```
#head probs
p_h1 = 0.25
p_h2 = 0.55
p_t1 = 1 - p_h1
p_t2 = 1 - p_h2
#discount factor
gamma = 1
#State count
Nstates = 100 #from 0 to 99

```

Figure 3: Preliminary definitions

Now we go on to define a functions which solves the **Bellman Equations** accordingly, this is done exactly as the algorithm depicted in previous pages.

```
 def BellmanEquationForValue(Vs,Rs,state,p_h,p_t):
 #Here we solve the Bellman equation and determine the Delta
and best actions
 bids = range(1,min(state+1,101-state))
 BellmanSummation = np.zeros(Nstates+1)
 for i in bids:
 #Here we implement the summation needed in the bellman
equation and store it
 # in the array (sum(p(s,s',a)[r + gamma*V(s'))
 # we initialize the indice for heads and tails easily
 heads = i+state
 tails = -i+state
 BellmanSummation[i] = p_h*(Rs[heads]+gamma*Vs[heads]) + p_t
*(Rs[tails]+gamma*Vs[tails])
 OptAction = max(BellmanSummation)
 VsBellman = OptAction
 return VsBellman , OptAction

```

Figure 4: Bellman Equation solver

Here we have defined the bids in the range of 1 to $\min(s, 100 - s)$, now we shall explain why we do this, first of all 100 is the maximum amount that we can bet, in a betting game we aim to not lose all our money, this move doesn't allow us to bet anything more than 50 dollars, hence the risk of bankruptcy is very low, in the value iteration method we don't really mind to increase the iterations, though it is good not to do so, but using this method we have a good probability of not losing, there probably are other versions to set the bid range, one might be to go all in and test our luck, which is risky and I have not attempted.

Now we shall continue with our code.

```python
def OptPolicyFinder(Vs,Rs,state,p_h,p_t):
#Here find the optimal policy as the arguement of the maximum in the bellman summation
bids = range(1,min(state+1,101-state))
BellmanSummation = np.zeros(Nstates+1)
for i in bids:
heads = i+state
tails = -i+state
BellmanSummation[i] = p_h*(Rs[heads]+gamma*Vs[heads]) + p_t*(Rs[tails]+gamma*Vs[tails])
OptPol = np.argmax(BellmanSummation)
return OptPol

```

Figure 5: Policy finder

```python
def GamblerProb(p_h,p_t):
threshold = 1e-10 #Theta in textbooks such as Sutton Barto
#Rewards Array
Rs = np.zeros(Nstates + 1)
Vs = np.zeros(Nstates + 1)
OptPolicies = np.zeros(Nstates)
#We set Rs[0]= 0 and Rs[100] = 1 as discussed before.
Rs[100] = 1
Rs[0] = 0 # Just written to express its importance
Delta = 0.01
while Delta>threshold:
Delta = 0
for state in range(1,Nstates):
VsBellman,OptAction = BellmanEquationForValue(Vs,Rs,state,p_h,p_t)
Delta = max(Delta,np.abs(OptAction-Vs[state]))
Vs[state] = OptAction
for state in range(1,Nstates):
OptPolicies[state] = OptPolicyFinder(Vs,Rs,state,p_h,p_t)
return Vs,OptPolicies

```

Figure 6: Main Gambler function

Now we go on to obtain the needed plots.

2.2 $P_h = 0.25$

The plots are as follows.

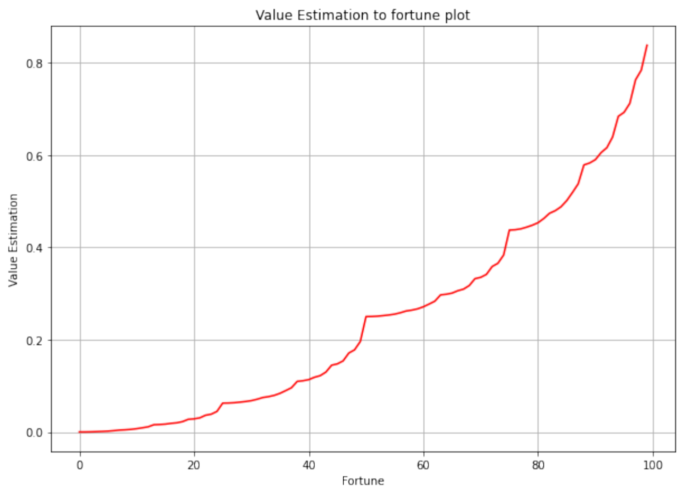


Figure 7: Value Estimation to fortune

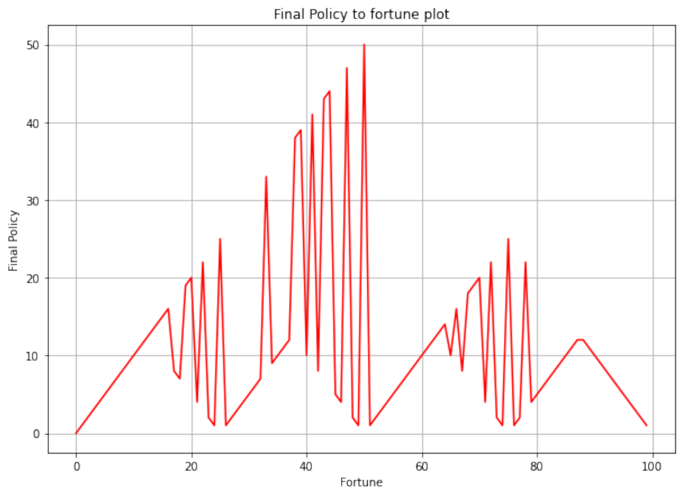


Figure 8: Final Policy to fortune (Normal plot)

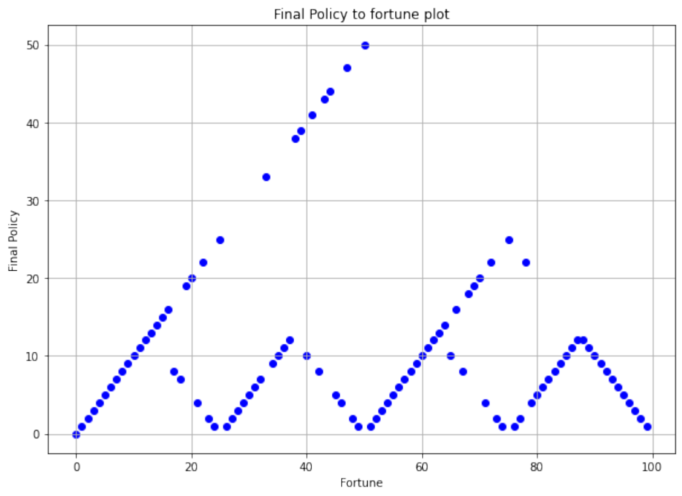


Figure 9: Final Policy to fortune (Scatter plot)

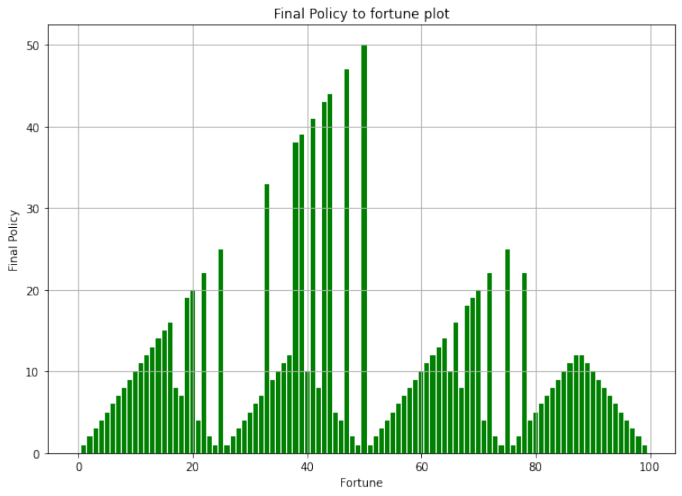


Figure 10: Final Policy to fortune (Bar plot)

As we can see the policy keeps changing based on the fortune the agent has, there are some ups and downs in the spikes in the bar plot, they can be controlled or render different results by changing the θ(threshold) value, we expected the changes in policy to be considerable and have a somewhat iterative manner and a peak somewhere near the middle when the probability of heads is less than tails which has occurred respectively.

2.3 $P_h = 0.55$

The plots are as follows.

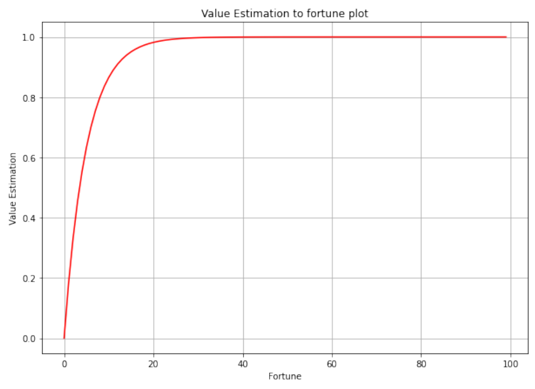


Figure 11: Value Estimation to fortune

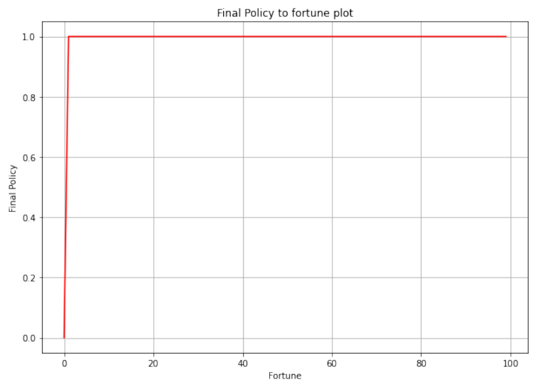


Figure 12: Final Policy to fortune (Normal plot)

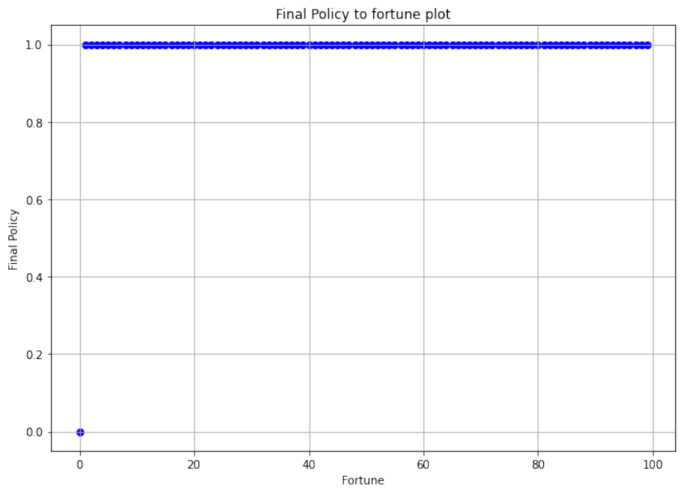


Figure 13: Final Policy to fortune (Scatter plot)

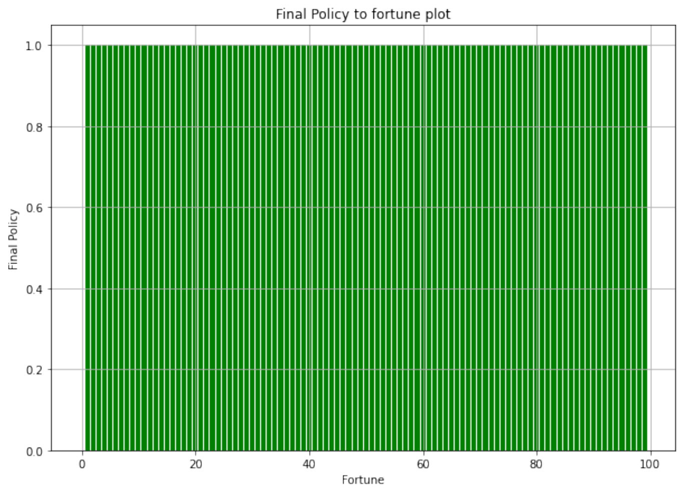


Figure 14: Final Policy to fortune (Bar plot)

Here we expected the policy to converge faster due to the probability of heads being more than tails, this has occurred fashionably, so if the probability of heads is higher we have verified that convergence shall occur much sooner and more effectively.

3 Question 3 : Model Free reinforcement learning(Implementation)

First of all we shall explain the problem for better understanding. We want to solve the problem of a taxi who picks up and drops off passengers at specific locations. The environment is as follows.

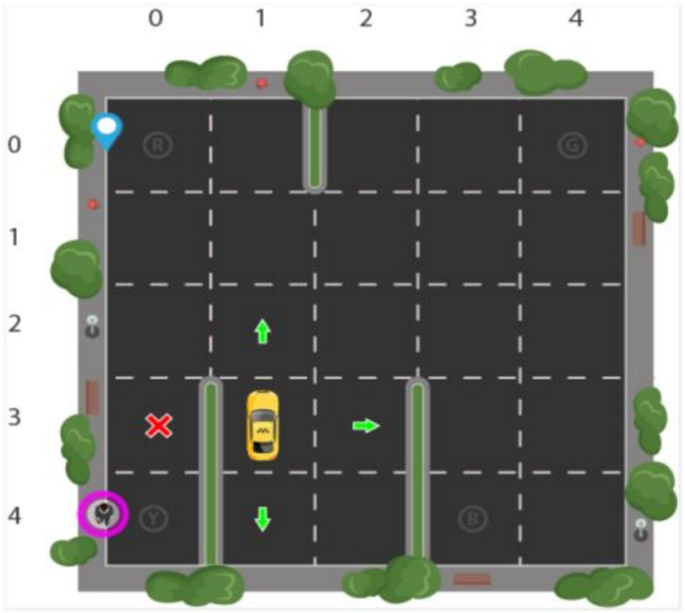


Figure 15: Taxi Environment

The Taxi gets as negative punishment for bumping against the wall, a negative walking punishment and -10 punishment for dropping off the passenger in the wrong position and 20 reward.

Defining Helper functions

Some of our helper functions are as follows.

```python
def plotbar(x,y,xlabel,ylabel,title):
fig = plt.figure(figsize=(10,7))
plt.bar(x,y,color = 'g')
plt.title(title)
plt.ylabel(ylabel)
plt.xlabel(xlabel)
plt.grid()

```

Figure 16: Plotter

```python
def ComparatorA(StepsA,RewardTotA,TotalPenaltyA,DropOffPenaltyA):
AverageRewardsPerMoveA = [i / j for i, j in zip(RewardTotA,StepsA)]
AverageStepsinEpisodesA = np.mean(StepsA)
AveragePenaltyInEpisodesA = np.mean(TotalPenaltyA)
AverageDropPenaltyInEpisodesA = np.mean(DropOffPenaltyA)
return AverageRewardsPerMoveA,AverageStepsinEpisodesA,AveragePenaltyInEpisodesA
    ,AverageDropPenaltyInEpisodesA

```

Figure 17: Compare

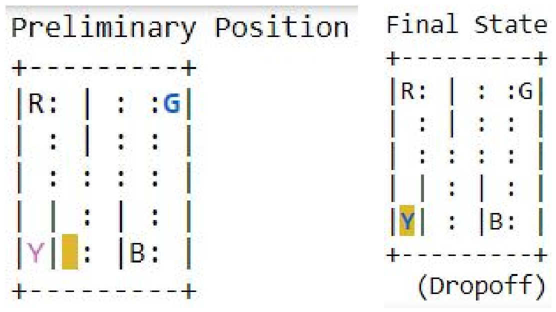
3.1 Part A : Solving via random navigation

Here we have created a loop to solve the problem based on random movements, the taxi driver is quite crazy in this case. The implementation we have used is as follows.

```
def TaxiForoneScenario():
Steps = 0
RewardTot = 0
NegativeRewards = 0
MoveandCollisionPenalty = 0
DropOffPenalty = 0
TotalPenalty = 0
env = gym.make("Taxi-v3").env
env.reset()
print('Preliminary Position')
env.render()
while 1==1:
action = env.action_space.sample()
state, reward, done, info = env.step(action)
print('------------------------------------')
print(f'Move number {Steps+1}')
env.render()
if reward<0:
TotalPenalty +=1
if reward == -1:
MoveandCollisionPenalty += 1
NegativeRewards +=reward
if reward == -10:
DropOffPenalty += 1
NegativeRewards +=reward
RewardTot +=reward
Steps += 1
if done==True:
break
print(f'Total Moves made by cab is :  {Steps}')
print(f'Moving and Collision Penalty obtained is : {MoveandCollisionPenalty}')
print(f'DropOff Penalty obtained is : {DropOffPenalty}')
print(f'Total Penalty obtained is : {TotalPenalty}')
print(f'Total Reward obtained is : {RewardTot}')
print('------------------------------------------')
print('Final State')
env.render()
```

Figure 18: Taxi for one scenario

Firstly we solve the taxi problem for one scenario, we have calculated the rewards and penalties and different kinds of rewards and printed them accordingly.



```
Preliminary Position
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
```

```
Final State
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (Dropoff)
```

Figure 19: Preliminary and Final position

```
Total Moves made by cab is :  2225
Moving and Collision Penalty obtained is : 1487
DropOff Penalty obtained is : 737
Total Penalty obtained is : 2224
Total Reward obtained is : -8837

```

Figure 20: Output

We see that the cab has reached the output but in a very horrible manner, this driver is incompetent and isn't recommended at all, the total steps is 2225 which is terrible, now we shall go on to perform some experiments we name episodes.

$$TotalSteps = 2225, \quad TotalPenalty = 2224, \quad DropOffPenalty = 737$$

```
def TaxiWithMultipleEpisodes(EpisodeNum):
StepsA =[]
RewardTotA = []
NegativeRewardsA = []
MoveandCollisionPenaltyA = []
DropOffPenaltyA = []
TotalPenaltyA = []
env = gym.make("Taxi-v3").env
for i in range(EpisodeNum):
Steps = 0
RewardTot = 0
NegativeRewards = 0
MoveandCollisionPenalty = 0
DropOffPenalty = 0
TotalPenalty = 0
env.reset()
print(f'Preliminary Position of Episode {i+1}')
env.render()
while 1==1:
action = env.action_space.sample()
state, reward, done, info = env.step(action)
if reward<0:
TotalPenalty +=1
if reward == -1:
MoveandCollisionPenalty += 1
NegativeRewards +=reward
if reward == -10:
DropOffPenalty += 1
NegativeRewards +=reward
RewardTot +=reward
Steps += 1
if done==True:
StepsA.append(Steps)
RewardTotA.append(RewardTot)
NegativeRewardsA.append(NegativeRewards)
DropOffPenaltyA.append(DropOffPenalty)
TotalPenaltyA.append(TotalPenalty)
MoveandCollisionPenaltyA.append(MoveandCollisionPenalty)
break
print('--------------------------------------------------')
print(f'Final State of Episode {i+1}')
env.render()
return StepsA,RewardTotA,TotalPenaltyA,DropOffPenaltyA

```

Figure 21: Output

After running this function we go on to plot some useful graphs as follows.

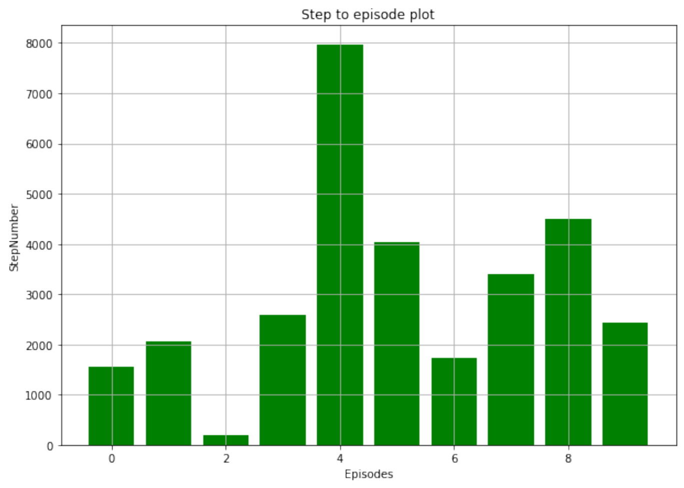


Figure 22: Step to Episode Plot

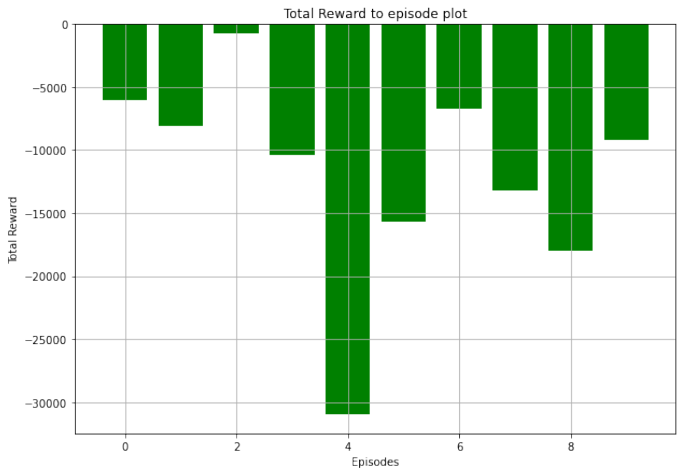


Figure 23: Total Reward to Episode Plot

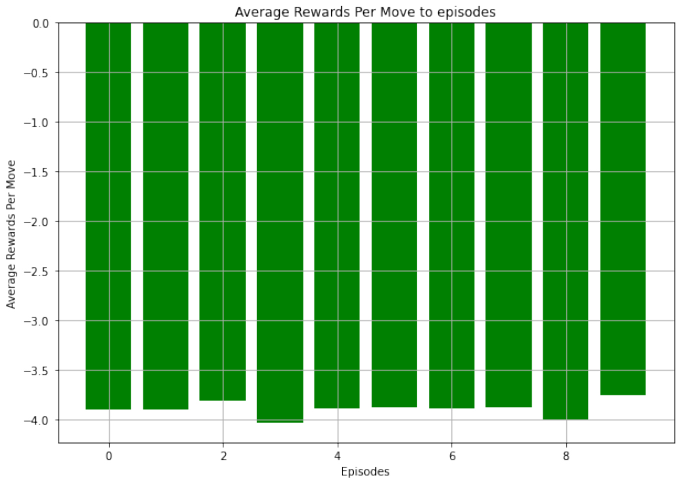


Figure 24: Average Rewards per move to Episode Plot

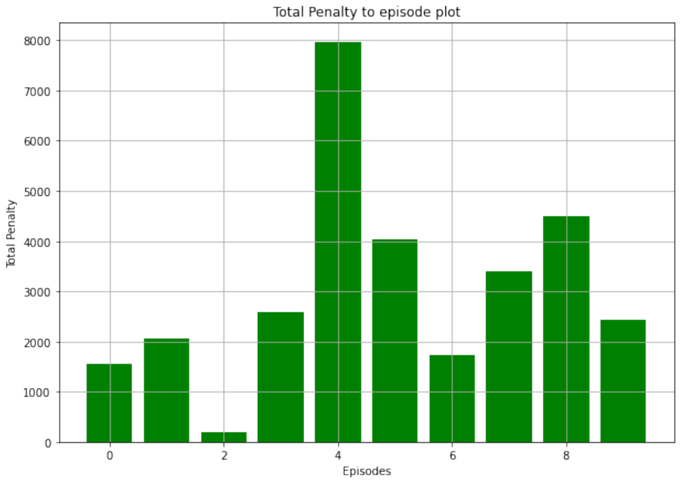


Figure 25: Total Penalty to Episode Plot

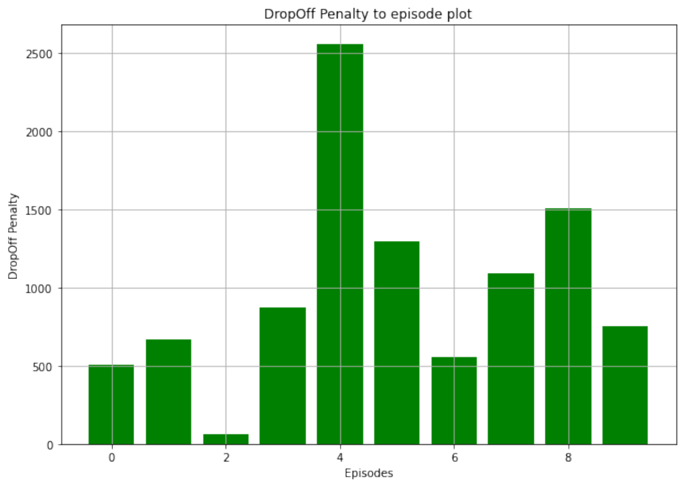


Figure 26: Dropoff Penalty to Episode Plot

The depicted output values are as follows.

```
print(f'AverageRewardsPerMoveA : {AverageRewardsPerMoveA}')
print(f'AverageStepsinEpisodesA : {AverageStepsinEpisodesA}')
print(f'AveragePenaltyInEpisodesA : {AveragePenaltyInEpisodesA}')
print(f'AverageDropPenaltyInEpisodesA : {AverageDropPenaltyInEpisodesA}')

AverageRewardsPerMoveA : [-3.894230769230769, -3.899806389157793,
    -3.8085106382978724, -4.0278529980657645, -3.888400150810607,
    -3.8776569451309935, -3.8853503184713376, -3.87488986784141,
    -4.00133155792277, -3.7577868852459018]
AverageStepsinEpisodesA : 3048.0
AveragePenaltyInEpisodesA : 3047.0
AverageDropPenaltyInEpisodesA : 985.8

```

Figure 27: Output

$$AverageStepsinEpisodes = 3048.0, \quad AveragePenaltyInEpisodes = 3047.0,$$
$$AverageDropPenaltyInEpisodes = 985.8$$

3.2 Part B : Solving via Q-Learning

Here we use the **Q-Learning** method, to do so we firstly give a quick explanation about **Q-Learning**.

3.2.1 Q-Learning

Q-Learning is a model free reinforcement learning method, in this method we aim to directly learn the optimal Q function, to do so we must perform updates after each action as such:

$$Q(s,a) = Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right)$$

If we depict the algorithm in a sequential manner we have:

1. Start with initial Q function
2. Take action using ϵ - greedy explore and exploit policy
3. Perform update
   $$Q(s,a) = Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right)$$
4. Go to step 2

Now we can go back to our original problem. We have implemented the said algorithm exactly as explained above, the codes are as follows.

```
#HyperParameters
alpha = 0.1 #Learning rate
gamma = 0.75 # discount rate
epsilon = 0.1 # for the epsilon - greedy method

```

Figure 28: Hyperparameters

```
def epsilongreedy(QTable,s,epsilon,env):
#exploration
if (random.uniform(0,1)<epsilon):
return env.action_space.sample()
#Otherwise we exploit
else:
return np.argmax(QTable[s])

```

Figure 29: ϵ greedy function

```
def QlearningTrainer(TrainEpisodes):
env = gym.make("Taxi-v3").env
QTable = np.zeros([500,6])
StepsTrain =[]
for i in range(TrainEpisodes):
#To create a new environement to learn
s = env.reset()
Steps = 0
TotalPenalty=0
while 1==1:
#Here we get the best action
a = epsilongreedy(QTable,s,epsilon,env)
#We obtain the next state
ns, reward, done, info = env.step(a)
OldQ = QTable[s,a]
NextQ = np.max(QTable[ns])
QTable[s,a] = OldQ + alpha*(reward+gamma*NextQ-OldQ)
Steps+=1
if done==True:
break
s = ns
StepsTrain.append(Steps)
return QTable,StepsTrain

```

Figure 30: Q-Learning trainer

The algorithm has been implemented in the above function as explained before. We shall test the new method via the following function then go on to plot the desired graphs.

```
def TaxiWithMultipleEpisodesQlearned(EpisodeNum,QTable):
StepsB =[]
RewardTotB = []
NegativeRewardsB = []
MoveandCollisionPenaltyB = []
DropOffPenaltyB = []
TotalPenaltyB = []\begin{figure}[H]
env = gym.make("Taxi-v3").env \begin{center}
for i in range(EpisodeNum):   \includegraphics[width=9cm]{13.png}
Steps = 0   \caption{Step to Episode Plot}
RewardTot = 0 \end{center}
NegativeRewards = 0\end{figure}
MoveandCollisionPenalty = 0\begin{figure}[H]
DropOffPenalty = 0  \begin{center}
TotalPenalty = 0    \includegraphics[width=9cm]{14.png}
state=env.reset()   \caption{Total Reward to Episode Plot}
print(f'Preliminary Position of Episode {i+1}') \end{center}
env.render()\end{figure}
while 1==1:\begin{figure}[H]
action = np.argmax(QTable[state]) \begin{center}
state, reward, done, info = env.step(action)    \includegraphics[width=9cm]{15.
  png}
if reward<0:    \caption{Average Rewards per move to Episode Plot}
TotalPenalty +=1  \end{center}
if reward == -1:\end{figure}
MoveandCollisionPenalty += 1
NegativeRewards +=reward\begin{figure}[H]
if reward == -10: \begin{center}
DropOffPenalty += 1   \includegraphics[width=9cm]{16.png}
NegativeRewards +=reward    \caption{Total Penalty to Episode Plot}
RewardTot +=reward  \end{center}
Steps += 1\end{figure}
if done==True:\begin{figure}[H]
StepsB.append(Steps)  \begin{center}
RewardTotB.append(RewardTot)    \includegraphics[width=9cm]{17.png}
NegativeRewardsB.append(NegativeRewards)    \caption{ Dropoff Penalty to
  Episode Plot}
DropOffPenaltyB.append(DropOffPenalty)  \end{center}
TotalPenaltyB.append(TotalPenalty)\end{figure}
MoveandCollisionPenaltyB.append(MoveandCollisionPenalty)
break
print('————————————————————————————————————')
print(f'Final State of Episode {i+1}')
env.render()
env.close()
return StepsB,RewardTotB,TotalPenaltyB,DropOffPenaltyB
```

Figure 31: Q-Learning tester

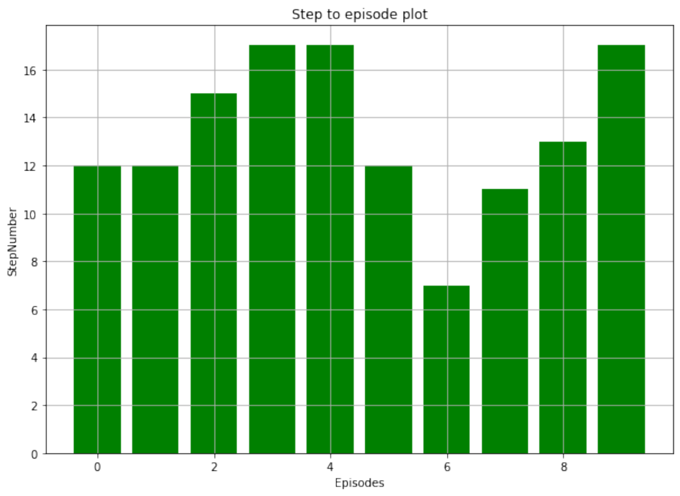


Figure 32: Step to Episode Plot

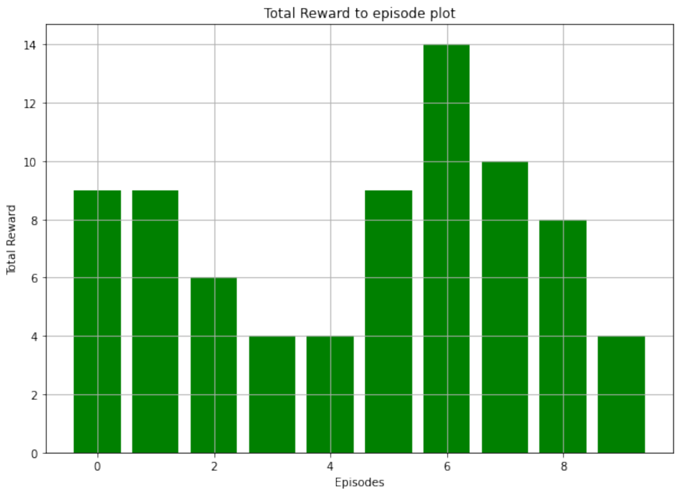


Figure 33: Total Reward to Episode Plot

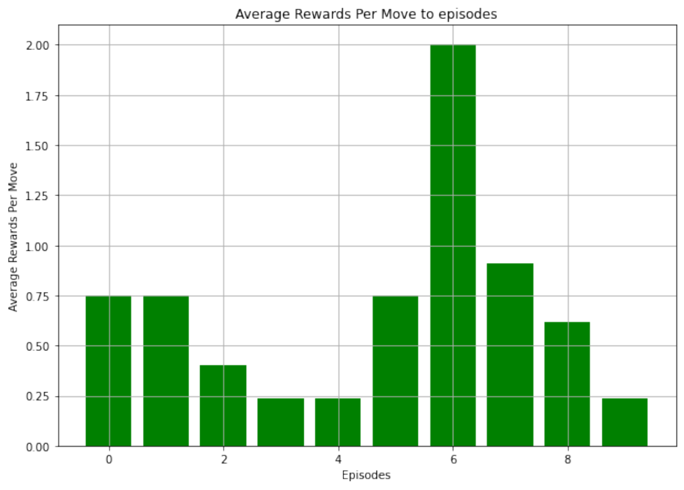


Figure 34: Average Rewards per move to Episode Plot

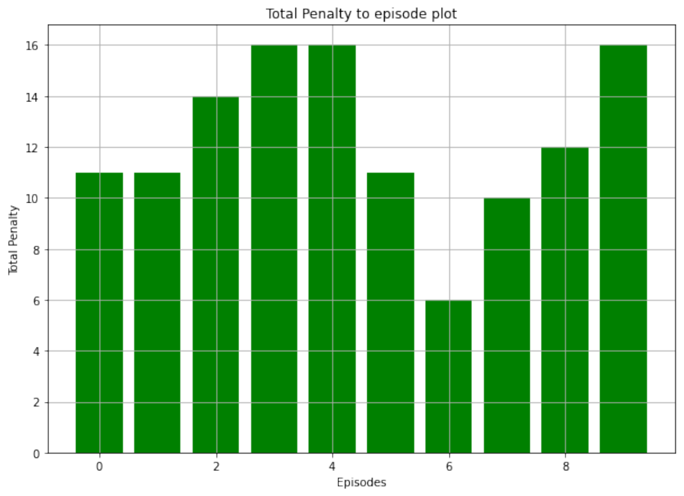


Figure 35: Total Penalty to Episode Plot

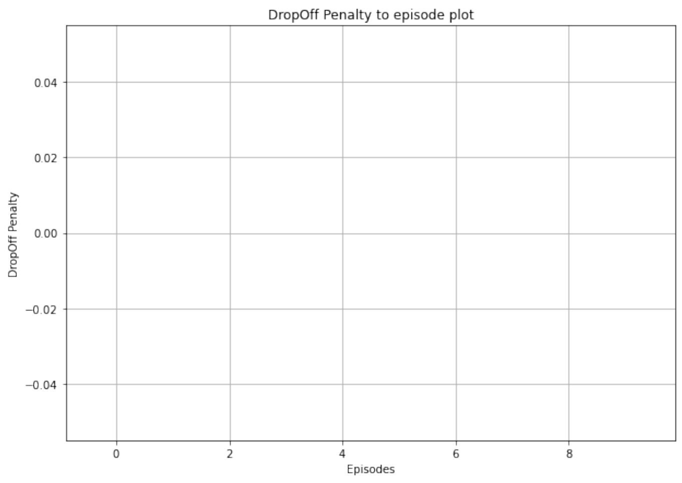


Figure 36: Dropoff Penalty to Episode Plot

As we can see the dropoff penalty shall become zero, which means the driver is in a more mentally stable condition, all in all the only kind of penalty we have is the walking penalty(and attempt to move through walls) which is very slight.

The depicted output values are as follows.

```
print(f'AverageRewardsPerMoveB : {AverageRewardsPerMoveB}')
print(f'AverageStepsinEpisodesB : {AverageStepsinEpisodesB}')
print(f'AveragePenaltyInEpisodesB : {AveragePenaltyInEpisodesB}')
print(f'AverageDropPenaltyInEpisodesB : {AverageDropPenaltyInEpisodesB}')

AverageRewardsPerMoveB : [0.75, 0.75, 0.4, 0.23529411764705882,
   0.23529411764705882, 0.75, 2.0, 0.9090909090909091, 0.6153846153846154,
   0.23529411764705882]
AverageStepsinEpisodesB : 13.3
AveragePenaltyInEpisodesB : 12.3
AverageDropPenaltyInEpisodesB : 0.0

```

Figure 37: Output

$$AverageStepsinEpisodes = 13.3, \quad AveragePenaltyInEpisodes = 12.3,$$
$$AverageDropPenaltyInEpisodes = 0.0$$

As expected the agent renders in a much better and intelligent manner compared to the random approach.

3.2.2 Algorithm to improve convergence speed

Now we shall attempt to change the rewards such that the we converge faster. My proposition is that we check that in each state where our passenger start position and the destination is, then we shall reduce the negative walking punishment for the cells that are good to reach these positions.

```
def QlearningTrainerBetter(TrainEpisodes):
env = gym.make("Taxi-v3").env
QTableB = np.zeros([500,6])
StepsTrainB =[]
for i in range(TrainEpisodes):
#To create a new environement to learn
s = env.reset()
Steps = 0
TotalPenalty=0
print(i)
while 1==1:
#Here we get the best action
a = epsilongreedy(QTableB,s,epsilon,env)
#We obtain the next state
row,col, pind, dind = env.decode(s)
CabL = int(str(row) + str(col))
ns, reward, done, info = env.step(a)
if ((pind==0 and dind==1)or(pind==1 and dind==0))and((CabL==0 or CabL==1 or
  CabL==2 or CabL==3  or CabL==10 or CabL==11 or CabL==12 or CabL==13) and a
  !=5):
reward=-0.75
elif ((pind==0 and dind==2)or(pind==2 and dind==0))and((CabL==0 or CabL==10 or
  CabL==20 or CabL==30 or CabL==40)and a!=5):
reward=-0.75
elif ((pind==0 and dind==3)or(pind==3 and dind==0))and((CabL==0 or CabL==10 or
  CabL==1 or CabL==11 or CabL==12 or CabL==21 or CabL==22 or CabL==23 or CabL
  ==33) and a!=5):
reward=-0.75
elif ((pind==1 and dind==2)or(pind==2 and dind==1))and((CabL==4 or CabL==14 or
  CabL==3 or CabL==2 or CabL==12 or CabL==13 or CabL==22 or CabL==11 or CabL
  ==10 or CabL==21 or CabL==20 or CabL==30)and a!=5):
reward=-0.75
elif ((pind==1 and dind==3)or(pind==3 and dind==1))and((CabL==3 or CabL==4 or
  CabL==13 or CabL==14 or CabL==23 or CabL==24 or CabL==33 or CabL==34 or CabL
  ==44) and a!=5):
reward=-0.75
elif ((pind==2 and dind==3)or(pind==3 and dind==2))and((CabL==30 or CabL==20 or
   CabL==21 or CabL==22 or CabL==23 or CabL==33) and a!=5):
reward=-0.75
OldQ = QTableB[s,a]
NextQ = np.max(QTableB[ns])
QTableB[s,a] = OldQ + alpha*(reward+gamma*NextQ-OldQ)
Steps+=1
if done==True:
break
s = ns
StepsTrainB.append(Steps)
return QTableB,StepsTrainB
```

Figure 38: Smart Q-Learning

```
def TaxiWithMultipleEpisodesQlearnedBetter(EpisodeNum,QTable):
StepsBe =[]
RewardTotBe = []
NegativeRewardsBe = []
MoveandCollisionPenaltyBe = []
DropOffPenaltyBe = []
TotalPenaltyBe = []
env = gym.make("Taxi-v3").env
for i in range(EpisodeNum):
Steps = 0
RewardTot = 0
NegativeRewards = 0
MoveandCollisionPenalty = 0
DropOffPenalty = 0
TotalPenalty = 0
state=env.reset()
print(f'Preliminary Position of Episode {i+1}')
env.render()
while 1==1:
action = np.argmax(QTable[state])
state, reward, done, info = env.step(action)
if reward<0:
TotalPenalty +=1
if reward == -1:
MoveandCollisionPenalty += 1
NegativeRewards +=reward
if reward == -10:
DropOffPenalty += 1
NegativeRewards +=reward
RewardTot +=reward
Steps += 1
if done==True:
StepsBe.append(Steps)
RewardTotBe.append(RewardTot)
NegativeRewardsBe.append(NegativeRewards)
DropOffPenaltyBe.append(DropOffPenalty)
TotalPenaltyBe.append(TotalPenalty)
MoveandCollisionPenaltyBe.append(MoveandCollisionPenalty)
break
print('--------------------------------------')
print(f'Final State of Episode {i+1}')
env.render()
env.close()
return StepsBe,RewardTotBe,TotalPenaltyBe,DropOffPenaltyBe

```

Figure 39: Smart Q-Learning Tester

The plots are as follows.

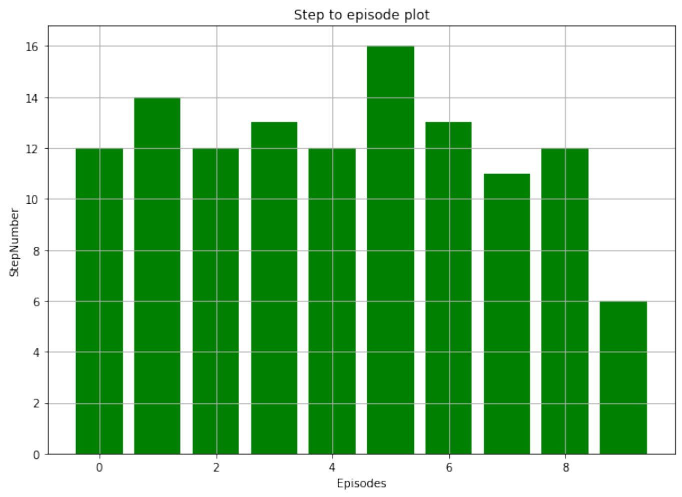


Figure 40: Step to Episode Plot

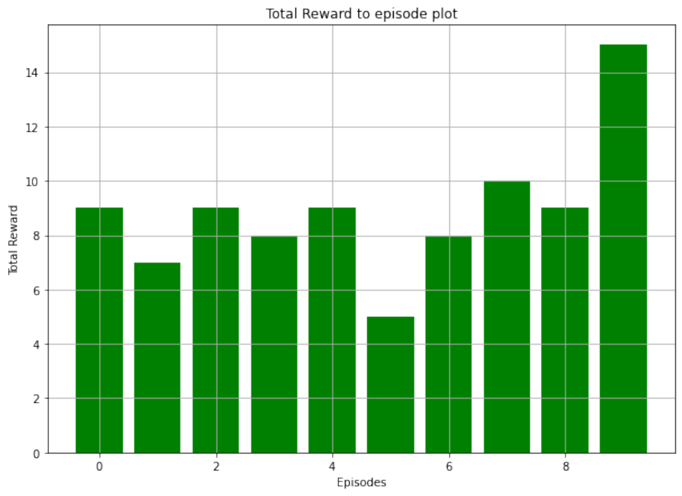


Figure 41: Total Reward to Episode Plot

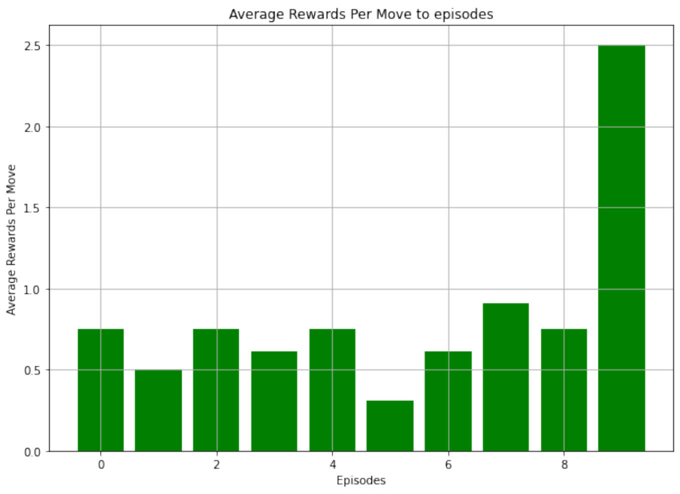


Figure 42: Average Rewards per move to Episode Plot

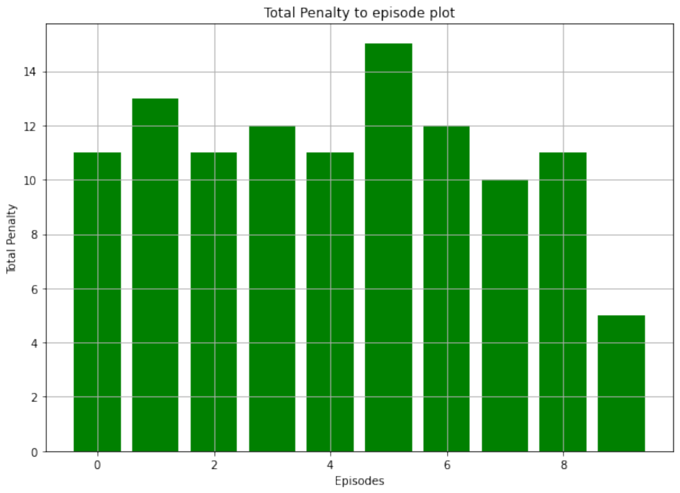


Figure 43: Total Penalty to Episode Plot

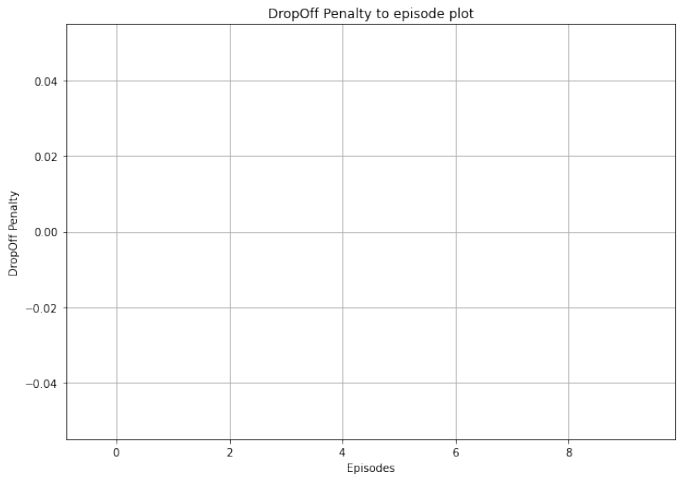


Figure 44: Dropoff Penalty to Episode Plot

The depicted output values are as follows.

```
print(f'AverageRewardsPerMoveBe : {AverageRewardsPerMoveBe}')
print(f'AverageStepsinEpisodesBe : {AverageStepsinEpisodesBe}')
print(f'AveragePenaltyInEpisodesBe : {AveragePenaltyInEpisodesBe}')
print(f'AverageDropPenaltyInEpisodesBe : {AverageDropPenaltyInEpisodesBe}')

AverageRewardsPerMoveBe : [0.75, 0.5, 0.75, 0.6153846153846154, 0.75, 0.3125,
  0.6153846153846154, 0.9090909090909091, 0.75, 2.5]
AverageStepsinEpisodesBe : 12.1
AveragePenaltyInEpisodesBe : 11.1
AverageDropPenaltyInEpisodesBe : 0.0

```

Figure 45: Output

$$AverageStepsinEpisodes = 12.1, \quad AveragePenaltyInEpisodes = 12.1,$$
$$AverageDropPenaltyInEpisodes = 0.0$$

As we can see the convergence rate of the agent has improved.

References

[1] Reshad Hosseini, *Intelligent Systems Lecture Notes, Fall 01*

[2] Richard S. Sutton, Andrew G Barto, *Reinforcement learning: An Introduction, 2nd edition, MIT Press, 2018*

[3] Policy Iteration in RL: A step by step Illustration

[4] The Gambler's Problem and Beyond

[5] What is reinforcement learning?