**PROJECT 3 SOLUTIONS**

**Pseudo Code For The MDP software**

import random library //to be used in generating random numbers

declare:- reward, Y, max\_error, number\_of\_actions, Actions, Number\_of\_rows, Number\_of\_columns, Utility, Policy, Solution

Function PrintEnviron(array, policy):

declare result:- //to be used as the array to store the path

for any column in the list of columns do:

if row is equal to column equal to 1 then:-

value:- = “wall”

else if row less or equal 1 and column equal 5 then:- // setting the boundaries

value:- = +1 when row is 0 else -1

else:-

value:- = the action and the current gridcell

result :- = value + “|”

result :- = result + “\n” Adding new line to the steps

return result

Function policyEval(policy, Utility):-

while True then:-

Declare nextUtility and error:-

for row in NumRows do:-

For column in Num\_cols do:-

if row:- less than 1 and column equal 3 or row equal column equal 1 then:-

continue //Skip the current iteration

nextUtiliy[row][col] = clavulateUtility(Utility, row, columns, policy)

error:- = max(error, absolute value of next utility current index

Set Utility to nextUtility

If error is less than (1-Y)/Y do:-

break // breaking the infinite while loop

return utility

Function getUtility(Utility, row, column, action):-

Declare dr, dc, newR and newC:-

if newR is less than 0 or newC less than 0 or newR greater than num\_Row or newC greater than or equal new\_color newR equal newC equal 1 then:-

return utility[column][row]

else:-

return[newR][newC]

Function calculateUtility(Utility, row, column, action):-

u;- = reward,

u;- += 0.1 \* Y \* getUtility(UTILITY, row, column, (action-1)%4)

u;- += 0.8 \* Y \* getUtility(UTILITY, row, column, action)

u;- += u += 0.1 \* Y \* getUtility(UTILITY, row, column, (action+1)%4)

return u

Function policyIter( policy, Utility):-

Set steps to 0

while True do:- //set an infinite loop to get the optimal path

Utility equal policyEval(policy, Utility)

Unchanged equal true

for row in num\_row do:-

for column in num\_col:-

if row is less or equal 1 and column equal 3 or row equal column equal 1 then:-

continue

maxAction and maxU equal to None

for action in num\_acctions:-

u equal calculateUtility(Utility, row, column, action)

if u is less than maxU then:-

maxAction and maxU equal action and u respectively

if maxU is less than the calculatedUtility then:-

Policy[row][column] equal maxAction

Unhanged equal false

Steps equal to steps + 1

If unchanged then:-

break

printEnviron(policy)

Print steps

Get the path evaluated

Print the path

return policy

main method:-

Declare the tax position, passenger 1 location and passenger 2 position

Display the output for the first episode

Get the solution aka the best path

for i in 5:-

Display the output for the ith episode

solution equal printEnviron //Get the solution aka the best path

Print solution

//THE END

**Source Code For MDP program**

import random

# Arguments: Reward + Disount should be equal to 1

REWARD = -0.10 # constant reward for non-terminal states

Y = 0.90

MAX\_ERROR = 10\*\*(-3)

# Set up the initial environment

NUM\_ACTIONS = 4

ACTIONS = [(1, 0), (0, -1), (-1, 0), (0, 1)] # South, West, North, East Since pickup and drop off can be either if these dirtections

NUM\_ROW = 3

NUM\_COL = 4

UTILITY = [[0, 0, 0, 1], [0, 0, 0, -1], [0, 0, 0, 0], [0, 0, 0, 0]]

policy = [[random.randint(0, 3) for j in range(NUM\_COL)] for i in range(NUM\_ROW)] # construct a random policy

SOLUTION = ""

# Visualizing the enviroment.

def printEnviron(arr, policy=False):

result = ""

for row in range(NUM\_ROW):

result += "|"

for column in range(NUM\_COL):

# Position for T cannot be the initial position

if row == column == 1:

value = "WALL"

# Setting the boundaries for the rows and columns to be used as steps

elif row <= 1 and column == 5:

value = "+1" if row == 0 else "-1"

else:

value = ["South", "West", "North", "East"][arr[row][column]]

result += " " + value[:5].ljust(5) + " |" # format

result += "\n"

return result

# Performing some simplified value iteration steps to get an approximation of the utilities

def policyEval(policy, UTILITY):

while True:

nextUtility = [[0, 0, 0, 1], [0, 0, 0, -1], [0, 0, 0, 0], [0, 0, 0, 0]]

error = 0

# Looping through the rows and columns to help evaluate the policy

for row in range(NUM\_ROW):

for column in range(NUM\_COL):

if (row <= 1 and column == 3) or (row == column == 1):

continue

nextUtility[row][column] = calculateUtility(UTILITY, row, column, policy[row][column]) # simplified Bellman update

error = max(error, abs(nextUtility[row][column]-UTILITY[row][column]))

UTILITY = nextUtility

if error < MAX\_ERROR \* (1-Y) / Y:

break

return UTILITY

# Getting the utility of the state reached by performing the given action from the given state

def getUtility(UTILITY, row, column, action):

dr, dc = ACTIONS[action]

newR, newC = row+dr, column+dc

if newR < 0 or newC < 0 or newR >= NUM\_ROW or newC >= NUM\_COL or (newR == newC == 1): # collide with the boundary or the wall

return UTILITY[row][column]

else:

return UTILITY[newR][newC]

# Calculating the utility of a state given the action

def calculateUtility(UTILITY, row, column, action):

u = REWARD

u += 0.1 \* Y \* getUtility(UTILITY, row, column, (action-1)%4)

u += 0.8 \* Y \* getUtility(UTILITY, row, column, action)

u += 0.1 \* Y \* getUtility(UTILITY, row, column, (action+1)%4)

return u

def policyIter(policy, UTILITY, tax, p1p, p2p):

steps = 0;

while True:

UTILITY = policyEval(policy, UTILITY)

unchanged = True

changes = random.randint(0, 1)

path = "tax: " , tax ," => "

# Iterating through the policy

for row in range(NUM\_ROW):

for column in range(NUM\_COL):

if (row <= 1 and column == 3) or (row == column == 1):

continue

maxAction, maxU = None, -float("inf")

for action in range(NUM\_ACTIONS):

u = calculateUtility(UTILITY, row, column, action)

if u > maxU:

maxAction, maxU = action, u

if maxU > calculateUtility(UTILITY, row, column, policy[row][column]):

policy[row][column] = maxAction # the action that maximizes the utility

unchanged = False

steps = steps +1

if unchanged:

break

p1d = [random.randint(1, 5),random.randint(1, 5)]

p2d = [random.randint(1, 5),random.randint(1, 5)]

if changes == 0:

changes = random.randint(0, 1)

path = ''.join(map(str, path)) , "p2p: ",p2p," => "

if changes == 0:

path = ''.join(map(str, path)) + "p1p: ",p1p," => "

path = ''.join(map(str, path)) , "p2d: ",p2d," => "

path = ''.join(map(str, path)) , "p1d: ",p1d

elif changes == 1:

path = ''.join(map(str, path)) , "p2d: ",p2d," => "

path = ''.join(map(str, path)) , "p1p: ",p1p," => "

path = ''.join(map(str, path)) , "p1d: ",p1d

else:

path = ''.join(map(str, path)) , "p1p: ",p1p," => "

changes = random.randint(0, 1)

if changes == 0:

path = ''.join(map(str, path)) , "p2p: ",p2p," => "

path = ''.join(map(str, path)) , "p1d: ",p1d," => "

path = ''.join(map(str, path)) , "p2d: ",p2d

elif changes == 1:

path = ''.join(map(str, path)) , "p1d: ",p1d," => "

path = ''.join(map(str, path)) , "p2p: ",p2p," => "

path = ''.join(map(str, path)) , "p2d: ",p2d

printEnviron(policy)

print("Steps = ",steps)

listToStr = ' '.join([str(elem) for elem in path])

print("The path is: ", path)

print("The optimal path is:")

return policy

if \_\_name\_\_ == "\_\_main\_\_":

tax = [2,4]

p1p = [1,5]

p2p = [5,1]

print("Episode 1 from figure 1:")

print("The tax location is ", tax)

print("The first passenger location is ", p1p)

print("The second pasenger location is ", p2p)

# Getting the original policy from figure one.

policy = [[3, 1, 2, 0], [1, 1, 2, 3], [0, 3, 0, 3]]

# Printing the optimal path

policy = policyIter(policy, UTILITY, tax, p1p, p2p)

SOLUTION = printEnviron(policy)

print(SOLUTION)

# Iterating throuh the policy.

for i in range(4):

tax = [random.randint(1, 5),random.randint(1, 5)]

p1p = [random.randint(1, 5),random.randint(1, 5)]

p2p = [random.randint(1, 5),random.randint(1, 5)]

print("Episode ", i+2,":")

print("The tax location is ", tax)

print("The first passenger location is ", p1p)

print("The second pasenger location is ", p2p)

policy = [[random.randint(0, 3) for j in range(NUM\_COL)] for i in range(NUM\_ROW)]

# Printing the optimal policy

policy = policyIter(policy, UTILITY, tax, p1p, p2p)

SOLUTION = printEnviron(policy)

print(SOLUTION)

**Solutions For the Questions**

**Question 1:**

The best path calculation is done by checking the grid cells that will lead to less steps and reach the destinations required.

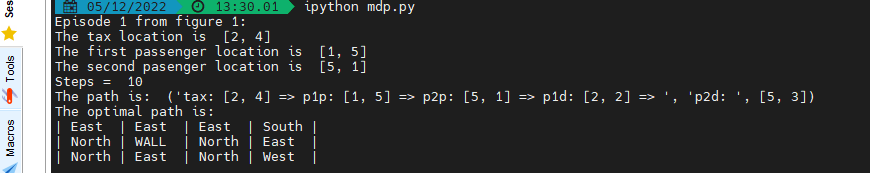
The taxi will go North to cell (1,4) pick passenger at L1 cell (1,5) by traversing west, go Back East to drop the passenger, Move East to cell (1,4) then south one cell to cell (2,4) then move east two steps to cell (2,2), Go South to cell (3,2), go east to cell (3,1) which is the drop off for the first passenger and pick up for the third passenger. The taxi picks the second passenger and moves to the South two cells and reaches the destination for the second passenger at cell (5,1).. And this leads to the end of the journey. This results in ten steps in total.

**Total steps moved: 10 steps**

**The best path is: T(2,4) => p1p(1,5) => p2p(3,1) => p1d(3,1) => p2d(5,1)**

**Question 2:**

The output for the first episode is as shown below in the screenshot:



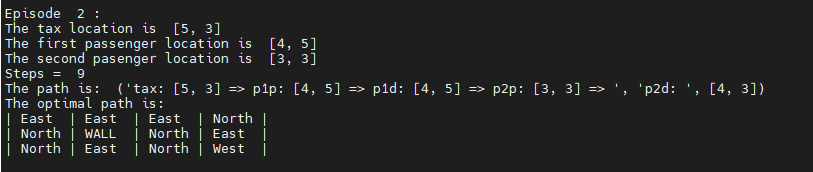
The path is totally different from the path produced by executing the MDP program. This is caused by the different paths that can be followed to get the exact path without passing through the illegal area. The grid has two options to follow in every state that the agent is in for all the states. This results in the agent having actions to follow that are not similar hence the different actions and still have the same number of steps.

**Question 3:**

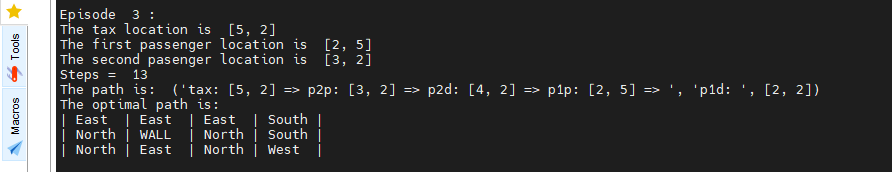
**Printing out the locations and the path**

The screenshots below show the locations and the paths for the four other random episodes:

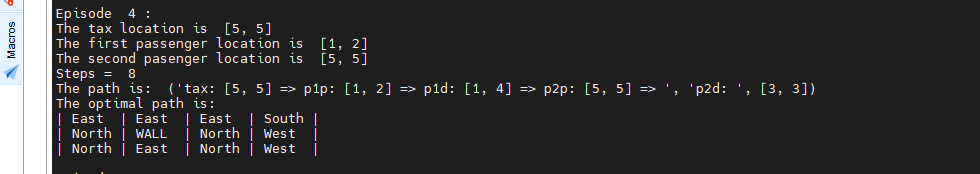
Second Episode:



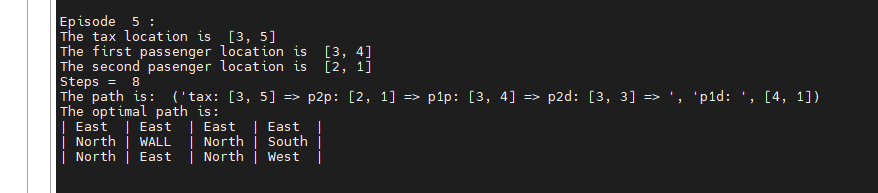
Third Episode:



Fourth Episode:



Fifth Episode:



**THE END**