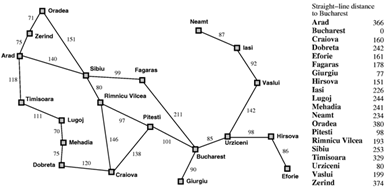
**AI Python Lab Record**

**NAME:** Ritika Subudhi

**USN:** 1NT18CS133

**SUBJECT CODE:** 18CSL58

1. We have the Map of Romania. In this map, the distance between various places in Romania is given. If we have to reach from one place to another place there exist several paths. Write a Python Program to find the shortest distance between any two places using a A\* search algorithm.



**SOLUTION:**

**Algorithm**:

// A\* Search Algorithm

1. Initialize the open list

2. Initialize the closed list

put the starting node on the open

list (you can leave its f at zero)

3. while the open list is not empty

a) find the node with the least f on

the open list, call it "q"

b) pop q off the open list

c) generate q's 8 successors and set their

parents to q

d) for each successor

i) if successor is the goal, stop search

successor.g = q.g + distance between

successor and q

successor.h = distance from goal to

successor (This can be done using many

ways, we will discuss three heuristics-

Manhattan, Diagonal and Euclidean

Heuristics)

successor.f = successor.g + successor.h

ii) if a node with the same position as

successor is in the OPEN list which has a

lower f than successor, skip this successor

iii) if a node with the same position as

successor is in the CLOSED list which has

a lower f than successor, skip this successor

otherwise, add the node to the open list

end (for loop)

e) push q on the closed list

end (while loop)

----------------------------------------------------------------------------------------------------------

**Files:**

**heuristics.txt** contains –

Arad, 366

Bucharest, 0

Craiova, 160

Dobreta, 242

Eforie, 161

Fagaras, 176

Giurgiu, 77

Hirsowa, 151

Lasi, 226

Lugoj, 244

Mehadia, 241

Neamt, 234

Oradea, 380

Pitesti, 100

Rimnicu Vilcea, 193

Sibiu, 253

Timisoara, 329

Urziceni, 80

Vaslui, 199

Zerind, 374

------------------------------------------------------------------------------------------------

**paths.txt** contains-

Arad, Zerind, 75

Arad, Sibiu, 140

Arad, Timisoara, 118

Zerind, Oradea, 71

Oradea, Sibiu, 151

Timisoara, Lugoj, 111

Sibiu, Fagaras, 99

Sibiu, Rimnicu Vilcea, 80

Lugoj, Mehadia, 70

Fagaras, Bucharest, 211

Rimnicu Vilcea, Pitesti, 97

Rimnicu Vilcea, Craiova, 146

Mehadia, Dobreta, 75

Bucharest, Pitesti, 101

Bucharest, Urziceni, 85

Bucharest, Giurgiu, 90

Pitesti, Craiova, 138

Craiova, Dobreta, 120

Urziceni, Hirsova, 98

Urziceni, Vaslui, 142

Hirsova, Eforie, 86

Vaslui, Lasi, 92

Lasi, Neamt, 87

----------------------------------------------------------------------------------------------------------

**Program:**

class PQueue():

def \_\_init\_\_(self):

self.dict = {}

self.keys = []

self.sorted = False

def \_sort(self):

self.keys = sorted(self.dict, key=self.dict.get, reverse=True)

self.sorted = True

def push(self, k, v):

self.dict[k] = v

self.sorted = False

def pop(self):

try:

if not self.sorted:

self.\_sort()

key = self.keys.pop()

value = self.dict[key]

self.dict.pop(key)

return key, value

except:

return None

def heuristics(path):

h = {}

with open(path, 'r') as file:

for line in file:

k, v = line.split(", ")

h[k] = int(v)

#print(h)

return h

def path\_costs(path):

c = {}

with open(path, 'r') as file:

for line in file:

line = line.split(", ")

v = int(line.pop())

e1 = line.pop()

e2 = line.pop()

if e1 not in c:

c[e1] = {}

if e2 not in c:

c[e2] = {}

c[e1][e2] = c[e2][e1] = v

#print(c)

return c

def a\_star(start, goal, h, g):

frontier = PQueue()

# pushing path and cost to pqueue

frontier.push(start, h[start])

while True:

# poping path with least cost

path, cost = frontier.pop()

print(path+ " " +str(cost))

# splitting out end node in path

end = path.split("->")[-1]

# removing heuristic value of end node from cost

cost -= h[end]

if goal == end:

break

for node, weight in g[end].items():

# adding edge weight(cost) and node heuristic to total cost

new\_cost = cost + weight + h[node]

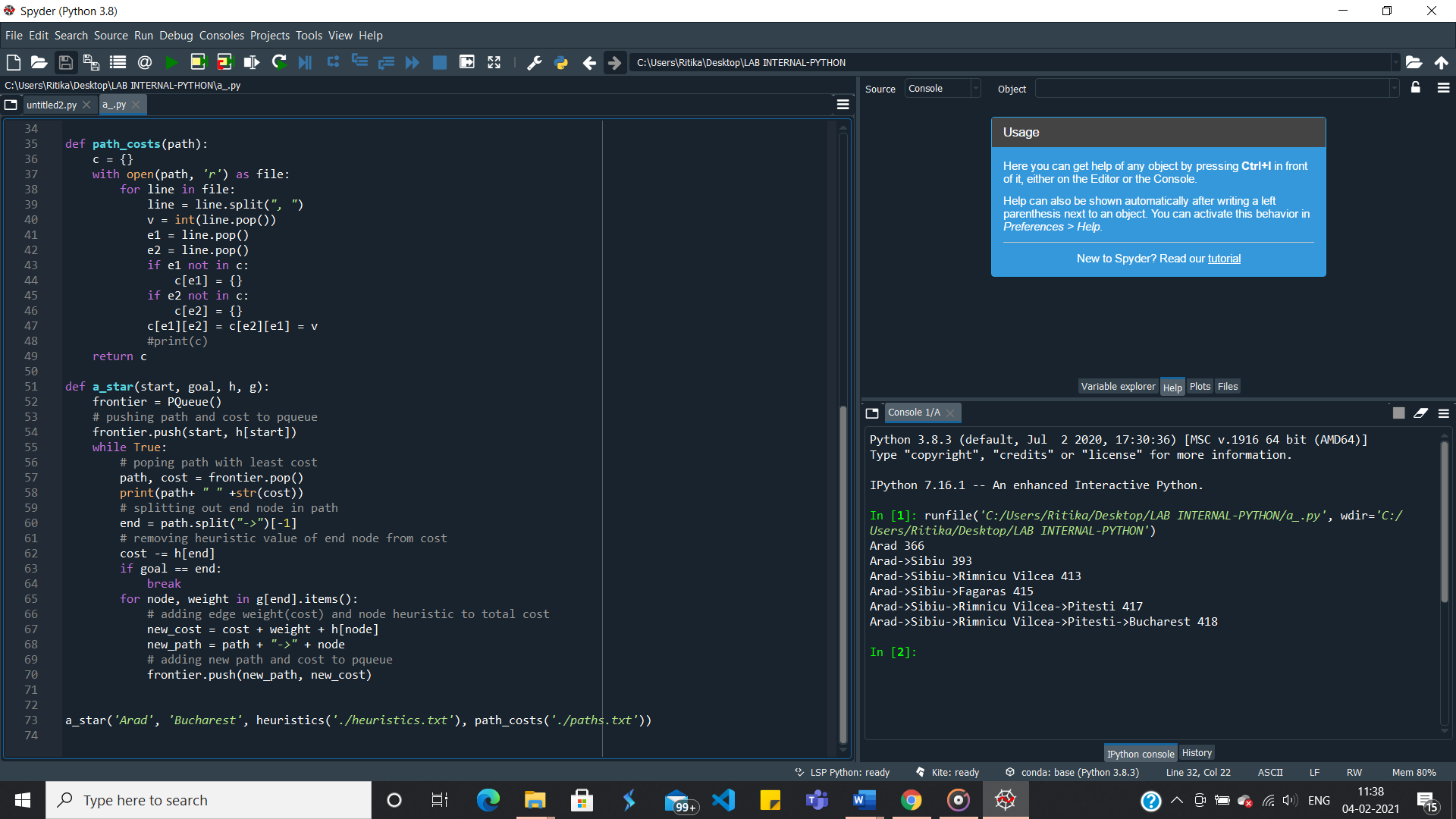
new\_path = path + "->" + node

# adding new path and cost to pqueue

frontier.push(new\_path, new\_cost)

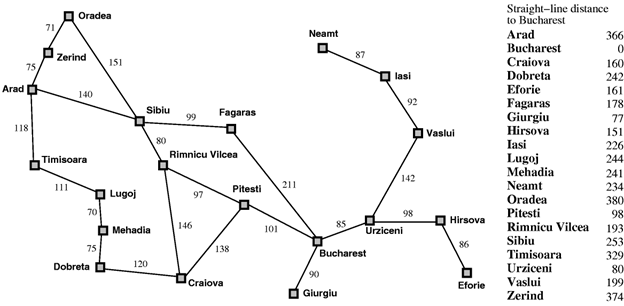
a\_star('Arad', 'Bucharest', heuristics('./heuristics.txt'), path\_costs('./paths.txt'))

---------------------------------------------------------------------------------------------------------

 **Output:**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

1. Problem Statement for uniform cost search: For the Romania map, the distance between various places are given. If we have to reach from one place to another place there exist several paths. Write a Python Program to find the shortest distance between any two places using a uniform cost search.



**SOLUTION:**

Algorithm:

Uniform-Cost Search is similar to Dijikstra’s algorithm .

In this algorithm from the starting state we will visit the adjacent states and will choose the least costly state then we will choose the next least costly state from the all un-visited and adjacent states of the visited states, in this way we will try to reach the goal state (note we wont continue the path through a goal state ), even if we reach the goal state we will continue searching for other possible paths( if there are multiple goals) . We will keep a priority queue which will give the least costliest next state from all the adjacent states of visited states.

**function** UNIFORM-COST-SEARCH (*problem*) **returns** a solution, or failure

*node* **<-** a node with STATE = *problem*.INITIAL-STATE, PATH-COST=0

*frontier* <- a priority queue ordered by PATH-COST, with node as the only element

*explored* <- an empty set

**loop do**

**if** EMPTY?(*frontier*) **then return** failure

*node* <- POP (*frontier*) /\*chooses the lowest -cost node in frontier\*/

if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)

add *node*.STATE to *explored*

**for each** *action* in *problem*.ACTIONS(*node*.STATE)**do**

*child* <- CHILD-NODE(*problem,node,action*)

**if** *child*.STATE is not in explored or frontier **then**

*frontier* <- INSERT(*child,frontier*)

**else if** *child*.STATE is in *frontier* with higher PATH-COST **then**

replace that *frontier* node with *child*

--------------------------------------------------------------------------------------------------------

**Files:**

**paths.txt** contains-

Arad, Zerind, 75

Arad, Sibiu, 140

Arad, Timisoara, 118

Zerind, Oradea, 71

Oradea, Sibiu, 151

Timisoara, Lugoj, 111

Sibiu, Fagaras, 99

Sibiu, Rimnicu Vilcea, 80

Lugoj, Mehadia, 70

Fagaras, Bucharest, 211

Rimnicu Vilcea, Pitesti, 97

Rimnicu Vilcea, Craiova, 146

Mehadia, Dobreta, 75

Bucharest, Pitesti, 101

Bucharest, Urziceni, 85

Bucharest, Giurgiu, 90

Pitesti, Craiova, 138

Craiova, Dobreta, 120

Urziceni, Hirsova, 98

Urziceni, Vaslui, 142

Hirsova, Eforie, 86

Vaslui, Lasi, 92

Lasi, Neamt, 87

**----------------------------------------------------------------------------------------------------------------**

**Program:**

class PQueue():

def \_\_init\_\_(self):

self.dict = {}

self.keys = []

self.sorted = False

def push(self, k, v):

self.dict[k] = v

self.sorted = False

def \_sort(self):

self.keys = sorted(self.dict, key=self.dict.get, reverse=True)

self.sorted = True

def pop(self):

try:

if not self.sorted:

self.\_sort()

key = self.keys.pop()

value = self.dict[key]

self.dict.pop(key)

return key, value

except:

return None

def path\_costs(path):

c = {}

with open(path, 'r') as file:

for line in file:

line = line.split(", ")

v = int(line.pop())

e1 = line.pop()

e2 = line.pop()

if e1 not in c:

c[e1] = {}

if e2 not in c:

c[e2] = {}

c[e1][e2] = c[e2][e1] = v

return c

def ucs(start, goal, g):

frontier = PQueue()

# pushing path and cost to pqueue

frontier.push(start, 0)

while True:

# poping path with least cost

path, cost = frontier.pop()

print(path+ " " +str(cost))

# splitting out end node in path

end = path.split("->")[-1]

if goal == end:

break

for node, weight in g[end].items():

# adding edge weight(cost) to total cost

new\_cost = cost + weight

new\_path = path + "->" + node

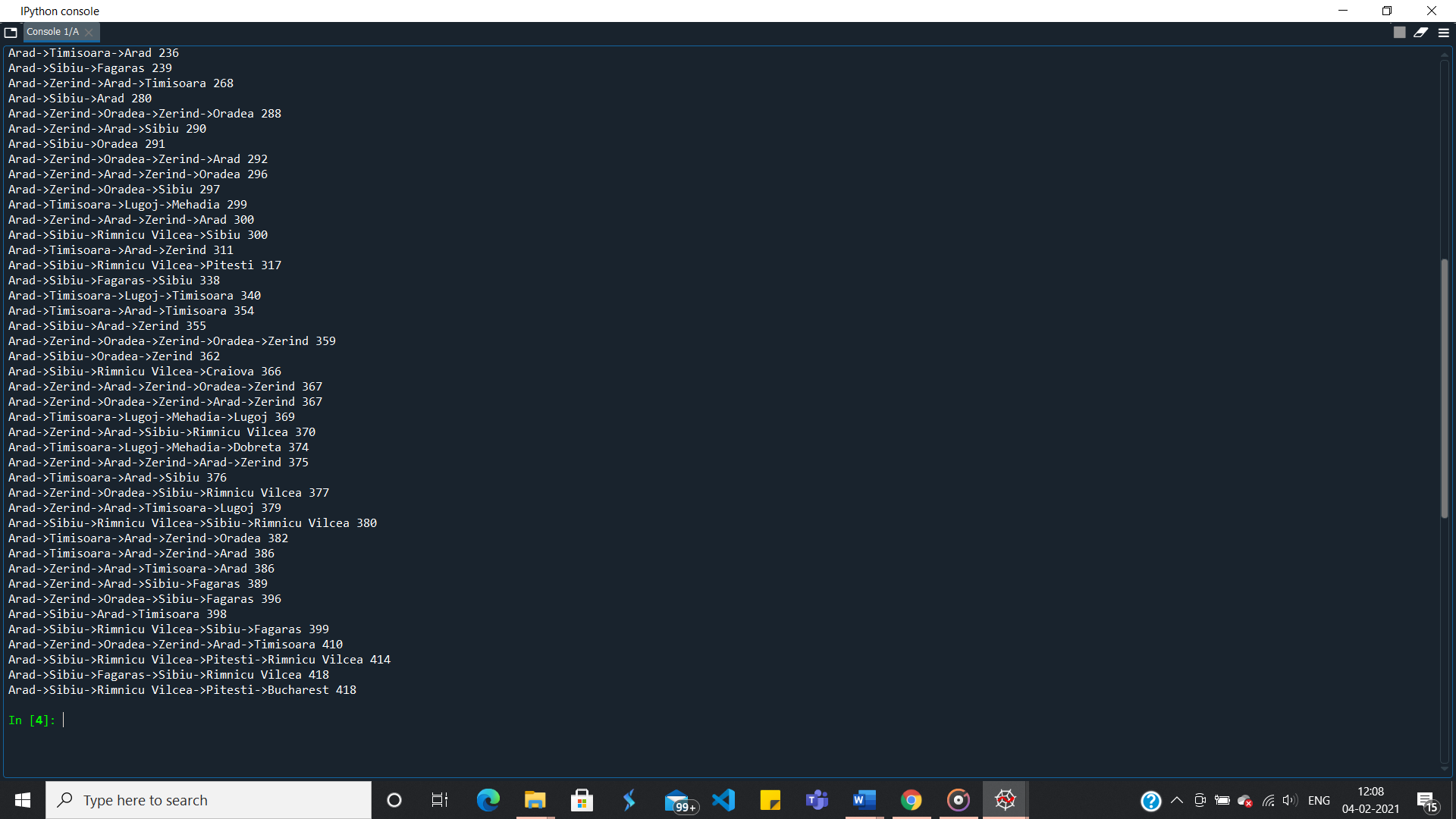
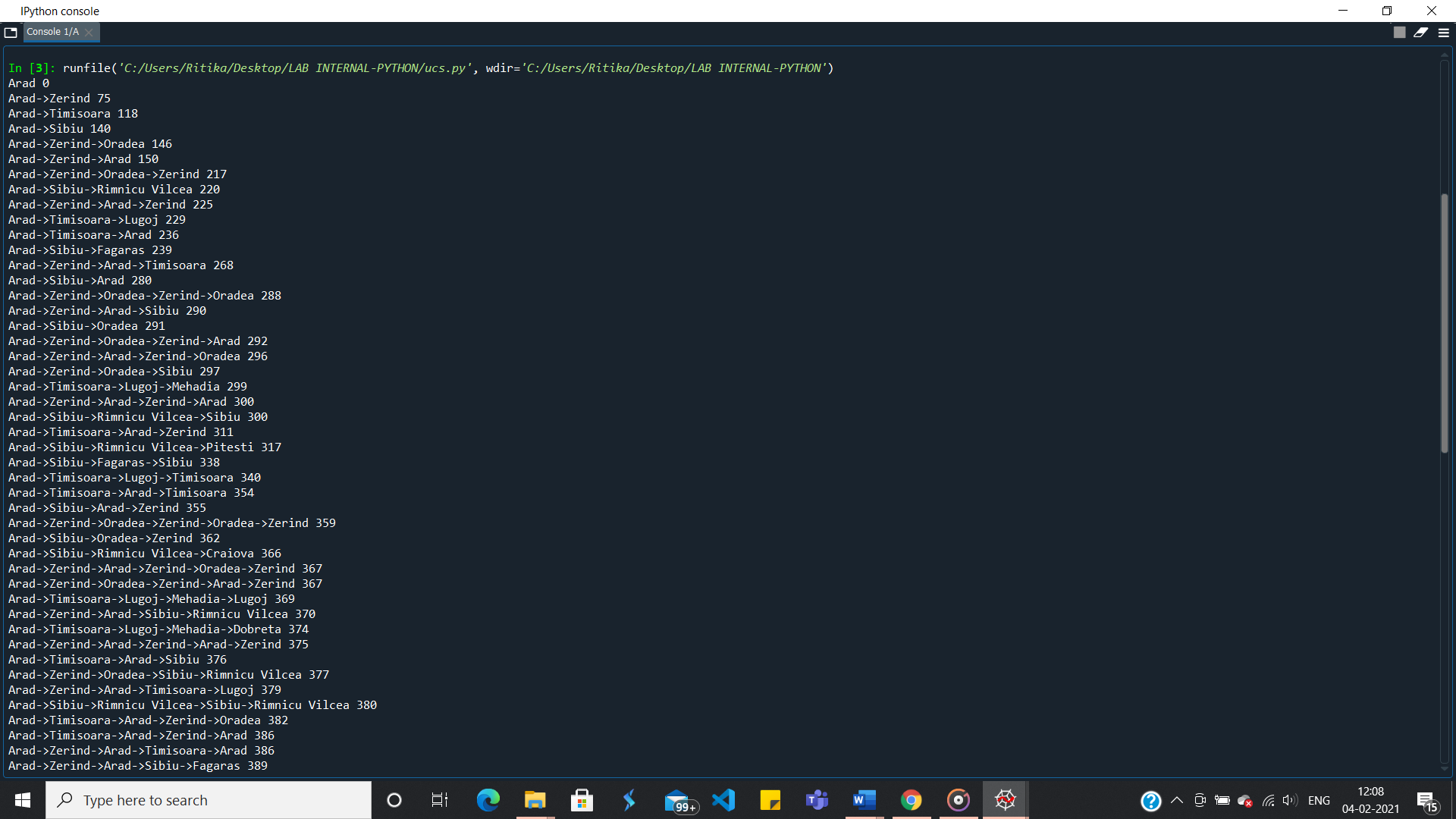
# adding new path and cost to pqueue

frontier.push(new\_path, new\_cost)

ucs('Arad', 'Bucharest', path\_costs('./paths.txt'))

--------------------------------------------------------------------------------------------------------------------------

**Output:**



**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

1. Problem Statement for Depth Limited Search: Design and develop a program in Python to print all the nodes reachable from a given starting node in a graph by using the Depth Limited Search method. Repeat the experiment for different Graphs.

**SOLUTION:**

**Algorithm:**

* The start node or node 1 is added to the beginning of the stack.
* Then it is marked as visited, and if node 1 is not the goal node in the search, then we push second node 2 on top of the stack.
* Next, we mark it as visited and check if node 2 is the goal node or not.
* If node 2 is not found to be the goal node, then we push node 4 on top of the stack.
* Now we search in the same depth limit and move along depth-wise to check for the goal nodes.
* If Node 4 is also not found to be the goal node and depth limit is found to be reached, then we retrace back to nearest nodes that remain unvisited or unexplored.
* Then we push them into the stack and mark them visited.
* We continue to perform these steps in iterative ways unless the goal node is reached or until all nodes within depth limit have been explored for the goal.

**Depth-limited search is found to terminate under these two clauses:**

* When the goal node is found to exist.
* When there is no solution within the given depth limit domain.

--------------------------------------------------------------------------------------------------------------------------

**Program:**

from collections import defaultdict

class Graph:

def \_\_init\_\_(self,vertices):

self.V = vertices

self.graph = defaultdict(list)

def addEdge(self,u,v):

self.graph[u].append(v)

def DLS(self,source,target,maxDepth):

if source == target : return True

if maxDepth <= 0 : return False

# recursively traversing the graph while searching

for i in self.graph[source]:

if(self.DLS(i, target, maxDepth-1)):

return True

return False

g = Graph(9)# creating the graph

g.addEdge(0, 1)

g.addEdge(0, 2)

g.addEdge(1, 3)

g.addEdge(1, 4)

g.addEdge(2, 5)

g.addEdge(2, 6)

g.addEdge(3,7)

g.addEdge(3,8)

target = 3

maxDepth = 3

source = 0

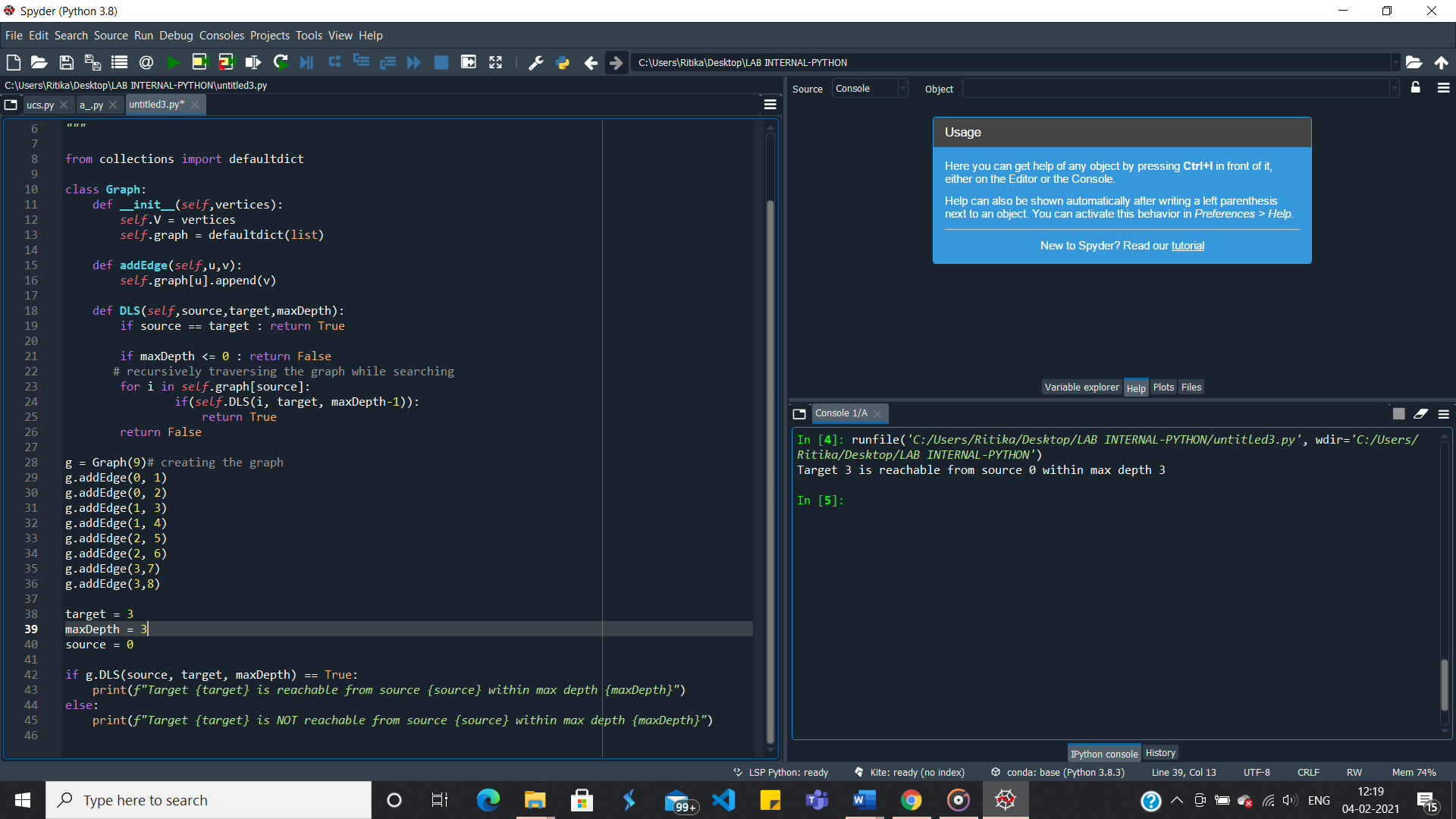
if g.DLS(source, target, maxDepth) == True:

print(f"Target {target} is reachable from source {source} within max depth {maxDepth}")

else:

print(f"Target {target} is NOT reachable from source {source} within max depth {maxDepth}")

---------------------------------------------------------------------------------------------------------------

**Output:**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Write a program to implement a Minimax decision-making algorithm, typically used in a turn-based, two player games. The goal of the algorithm is to find the optimal next move.

**SOUTION:**

**Algorithm:**

* Construct the complete game tree
* Evaluate scores for leaves using the evaluation function
* Back-up scores from leaves to root, considering the player type:
  + For max player, select the child with the maximum score
  + For min player, select the child with the minimum score
* At the root node, choose the node with max value and perform the corresponding move

**---------------------------------------------------------------------------------------------------------**

**Program:**

import math

import random

#minimax class

def minimax (currentDepth, nodeIndex, maxTurn, score, treeDepth):

# base case : treeDepth reached

if (currentDepth == treeDepth):

return score[nodeIndex]

if (maxTurn):

return max(minimax(currentDepth + 1, nodeIndex \* 2, False, score, treeDepth),

minimax(currentDepth + 1, nodeIndex \* 2 + 1,False, score, treeDepth))

else:

return min(minimax(currentDepth + 1, nodeIndex \* 2, True, score, treeDepth),

minimax(currentDepth + 1, nodeIndex \* 2 + 1,True, score, treeDepth))

# Driver code

score = random.sample(range(1, 50), 4)

print(str(score))

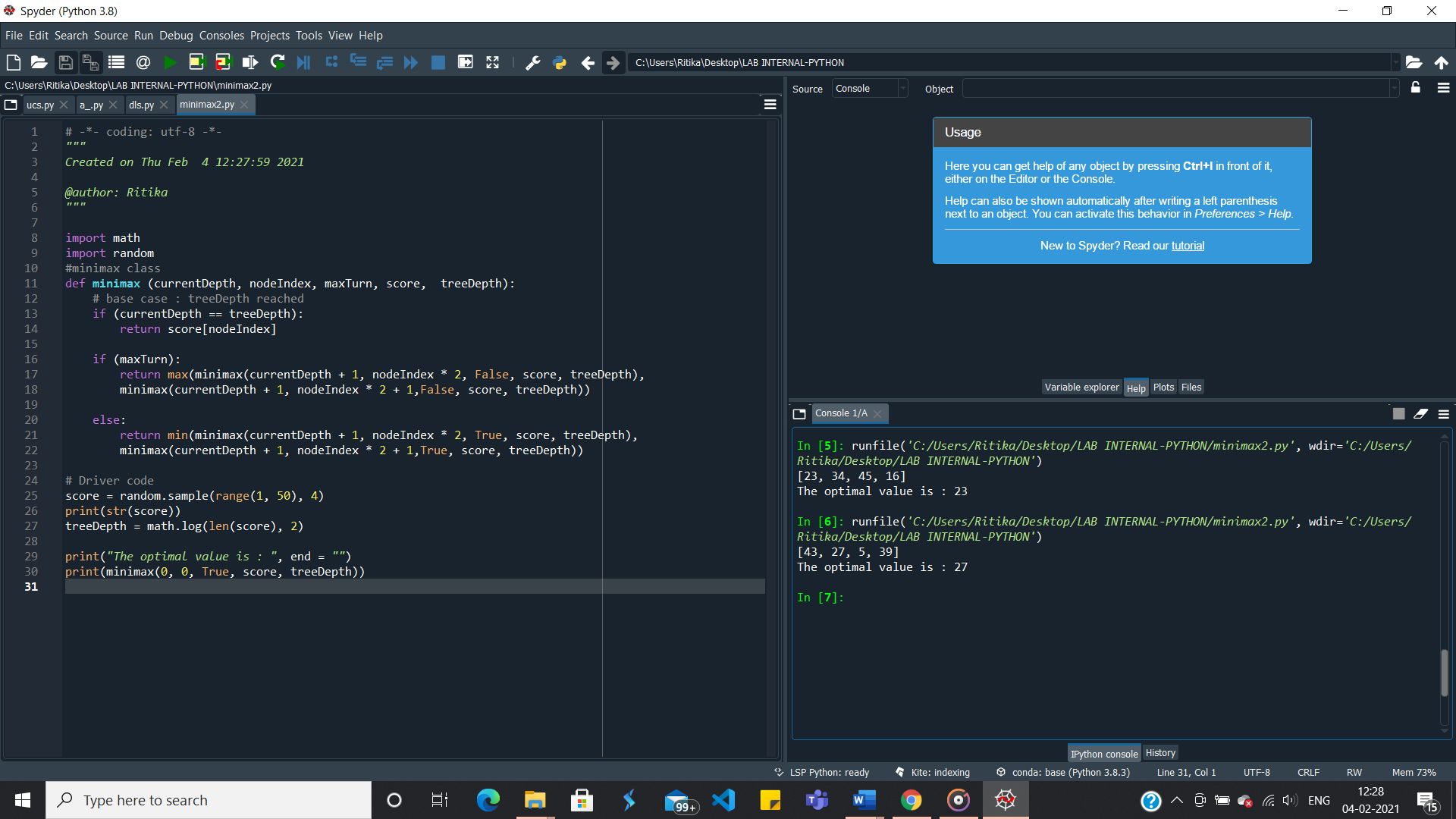
treeDepth = math.log(len(score), 2)

print("The optimal value is : ", end = "")

print(minimax(0, 0, True, score, treeDepth))

**---------------------------------------------------------------------------------------------------------**

**Output:**



**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

1. Write a program to implement Alpha Beta pruning in Python. The algorithm can be applied to any depth of tree by not only pruning the tree leaves but also the entire subtree. Order the nodes in the tree such that the best nodes are checked first from the shallowest node.

**SOLUTION:**

**Algorithm:**

Alpha-Beta pruning is not actually a new algorithm, rather an optimization technique for minimax algorithm. It reduces the computation time by a huge factor. This allows us to search much faster and even go into deeper levels in the game tree. It cuts off branches in the game tree which need not be searched because there already exists a better move available. It is called Alpha-Beta pruning because it passes 2 extra parameters in the minimax function, namely alpha and beta. Alpha is the best value that the maximizer currently can guarantee at that level or above. Beta is the best value that the minimizer currently can guarantee at that level or above.

Pseudo code –

function minimax(node, depth, isMaximizingPlayer, alpha, beta):

if node is a leaf node :

return value of the node

if isMaximizingPlayer :

bestVal = -INFINITY

for each child node :

value = minimax(node, depth+1, false, alpha, beta)

bestVal = max( bestVal, value)

alpha = max( alpha, bestVal)

if beta <= alpha:

break

return bestVal

else :

bestVal = +INFINITY

for each child node :

value = minimax(node, depth+1, true, alpha, beta)

bestVal = min( bestVal, value)

beta = min( beta, bestVal)

if beta <= alpha:

break

return bestVal

--------------------------------------------------------------------------------------------------------------------------

**Program:**

import math

MIN,MAX= -1000,1000

def MINMAX(depth,nodeIndex,maximizingPlayer,values,alpha,beta):

if depth==math.ceil(math.log(len(values),2)):

return values[nodeIndex]

if maximizingPlayer:

best=MIN

for i in range(0,math.ceil(math.log(len(values),2))-1):

val = MINMAX(depth+1,nodeIndex\*2+i,False,values,alpha,beta)

best=max(best,val)

alpha=max(alpha,best)

if beta<=alpha:

break

return best

else:

best=MAX

for i in range(0,math.ceil(math.log(len(values),2))-1):

val = MINMAX(depth+1,nodeIndex\*2+i,True,values,alpha,beta)

best=min(best,val)

alpha=min(alpha,best)

if beta<=alpha:

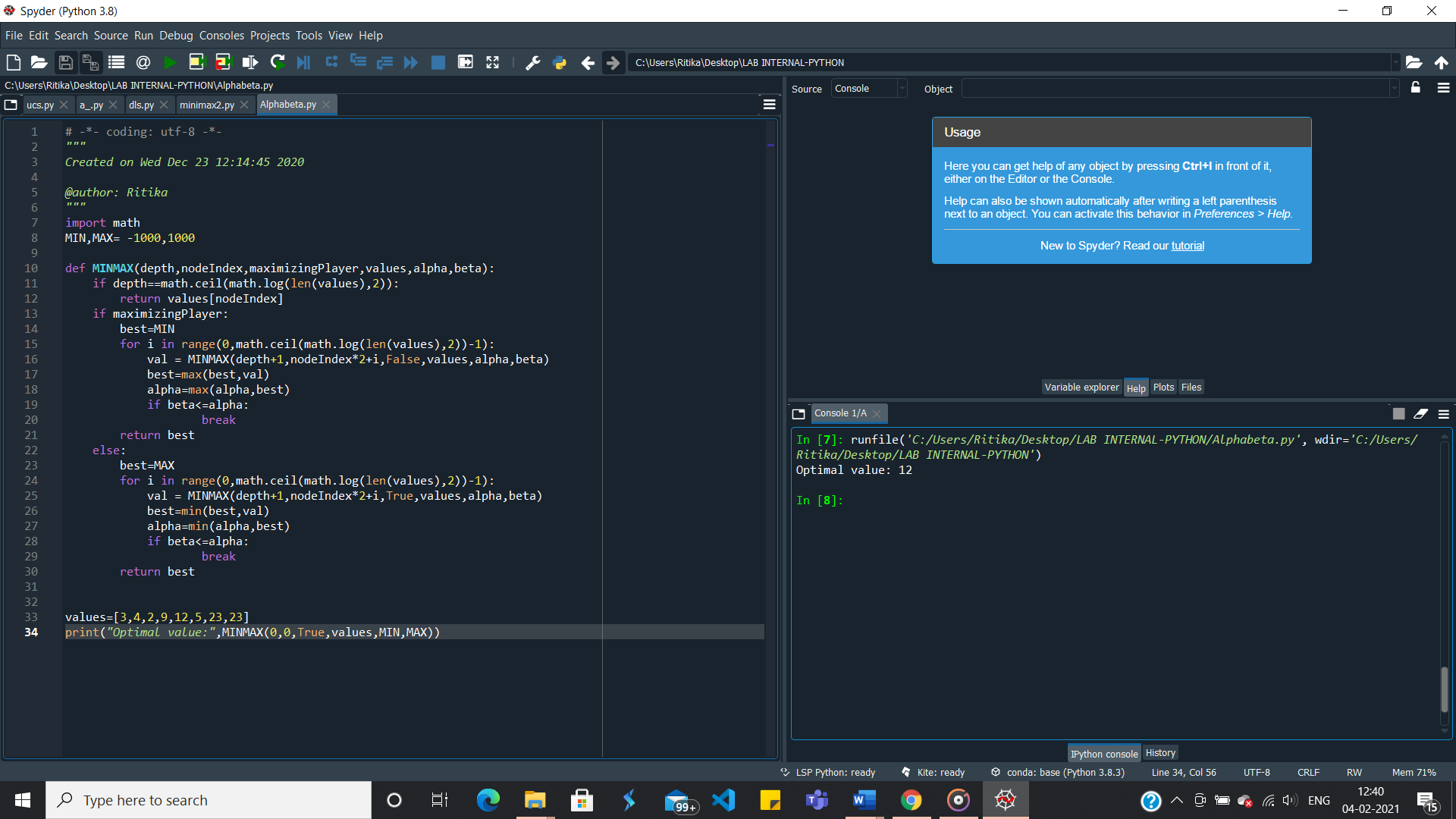
break

return best

values=[3,4,2,9,12,5,23,23]

print("Optimal value:",MINMAX(0,0,True,values,MIN,MAX))

---------------------------------------------------------------------------------------------------------

**Output:**

1. Assume that you are organizing a party for N people and have been given a list L of people who, for social reasons, should not sit at the same table. Furthermore, assume that you have C tables (that are infinitely large). Write a function layout(N,C,L) that can give a table placement (ie. a number from 0 . . . C − 1) for each guest such that there will be no social mishaps.

For simplicity we assume that you have a unique number 0 . . . N − 1 for each guest and that the list of restrictions is of the form [(X,Y), ...] denoting guests X, Y that are not allowed to sit together.

Answer with a dictionary mapping each guest into a table assignment, if there are no possible layouts of the guests you should answer False.

**SOLUTION:**

**Program:**

def backtrack(x,enemy\_list,domain,assigned):

if -1 not in assigned:

return x

v = 999

for i in range(len(domain)):

if v>len(domain[i]) and assigned[i]!=1:

v = i

order=[]

for i in domain[v]:

mini = 1000

for j in enemy\_list[v]:

temp = len(domain[j])

if i in domain[j]:

temp-=1

if temp<mini:

mini = temp

order.append((i,mini))

order = sorted(order,key=lambda x:x[1],reverse=True)

ordered = [i[0] for i in order]

for i in ordered:

newdomain = [ [j for j in i] for i in domain]

for j in enemy\_list[v]:

if i == x[j]:

continue

x[v] = i

assigned[v] = 1

newdomain[v] = [z for z in newdomain[v] if z==i]

temp = []

for j in range(len(newdomain)):

if j!=v and j in enemy\_list[v]:

newdomain[j] = [z for z in newdomain[j] if z!=i]

res = backtrack(x,enemy\_list,newdomain,assigned)

if res!=0:

return res

x[v] = ""

assigned[v] = -1

return 0

people = int(input("Enter the number of people"))

tables = int(input("enter the number of tables"))

edges = []

line = input("enter elements of list L(people who should not sit together) till an empty newline character. ").split()

while(line):

edges.append((int(line[0]),int(line[1])))

line = input().split()

x = ["" for i in range(people)]

enemy\_list = [[] for i in range(people)]

for i in edges:

enemy\_list[i[0]].append(i[1])

enemy\_list[i[1]].append(i[0])

for i in range(people):

j = list(set(enemy\_list[i]))

enemy\_list[i] = j

assigned = [-1 for i in range(people)]

domain = [[x for x in range(tables)] for i in range(people)]

res = backtrack(x,enemy\_list,domain,assigned)

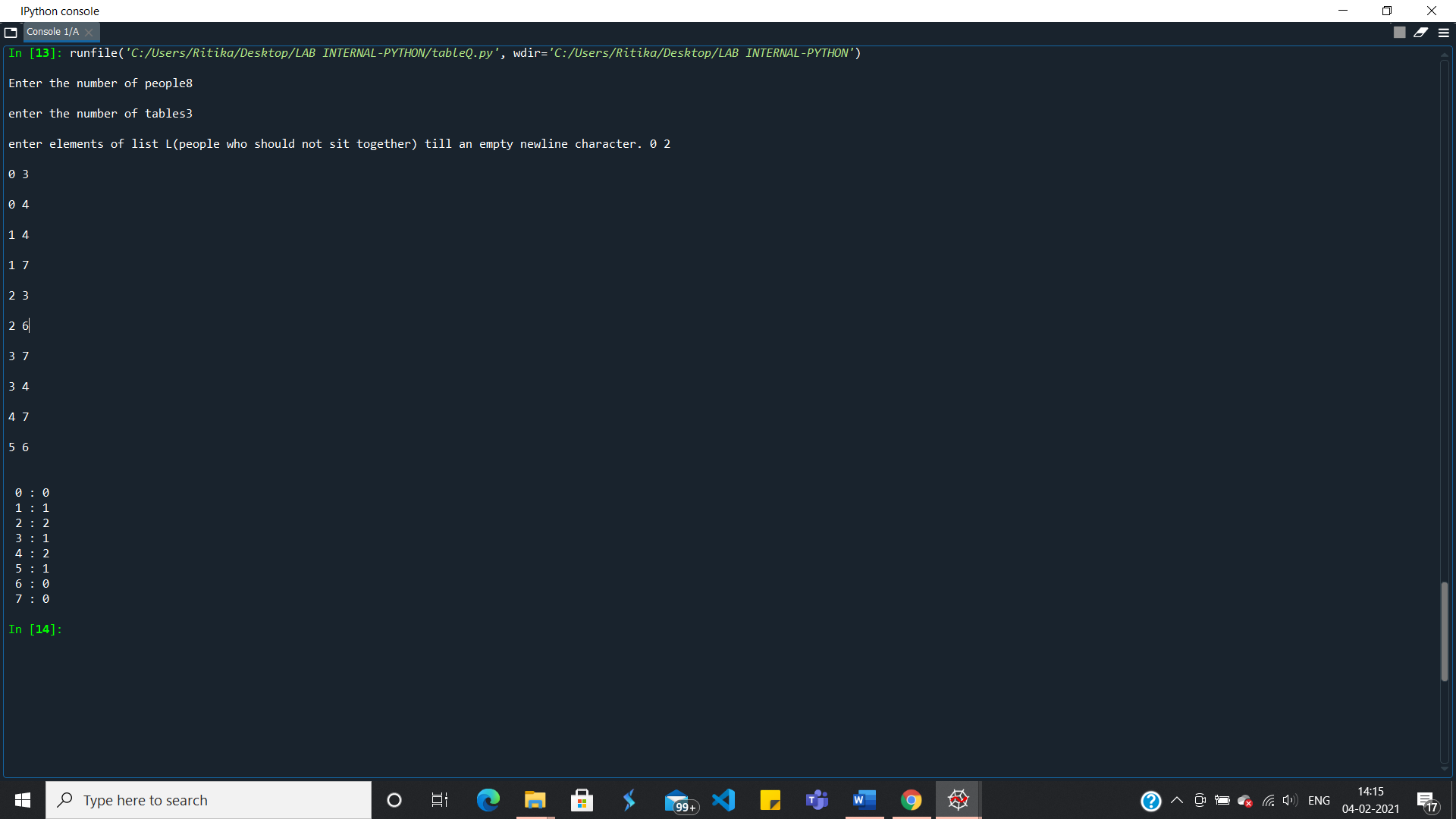
if res == 0:

print('False')

else:

for i in range(len(res)):

print(' {} :'.format(i),res[i])

**Output:**

1. Implementation of Tic Tac Toe game here, the player needs to take turns marking the spaces in a 3x3 grid with their own marks, if 3 consecutive marks (Horizontal, Vertical,Diagonal) are formed then the player who owns these moves get won. Noughts and Crosses or X’s and O’s abbreviations can be used to play.

**SOLUTION:**

**Algorithm/Explanation:**

Tic-tac-toe is a two-player game. It contains 3\*3 board where each player takes turn and select a block which is not marked already and marks it with ‘x’ and ‘o’ for player 1 and 2 respectively.

if 3 consecutive marks (Horizontal, Vertical, Diagonal) are formed then the player who owns these moves get won.

In the program,

1.The board function is called to display the board

2.The game status function is called to check if there is a winner always after a player turn.

**Program:**

square=[0,1,2,3,4,5,6,7,8,9]

def board():

print('\n\tTic Tac Toe')

print('Player 1 (X) - Player 2 (O)' )

print(' | | ' )

print(' ' ,square[1] ,' | ' ,square[2] ,' | ' ,square[3] )

print('\_\_\_\_\_|\_\_\_\_\_|\_\_\_\_\_' )

print(' | | ' )

print(' ' ,square[4] ,' | ' ,square[5] ,' | ' ,square[6] )

print('\_\_\_\_\_|\_\_\_\_\_|\_\_\_\_\_' )

print(' | | ' )

print(' ' ,square[7] ,' | ' ,square[8] ,' | ' ,square[9] )

print(' | | ' )

def game\_status():

if square[1] == square[2] and square[2] == square[3]:

return 1

elif square[4] == square[5] and square[5] == square[6]:

return 1

elif square[7] == square[8] and square[8] == square[9]:

return 1

elif square[1] == square[4] and square[4] == square[7]:

return 1

elif square[2] == square[5] and square[5] == square[8]:

return 1

elif square[3] == square[6] and square[6] == square[9]:

return 1

elif square[1] == square[5] and square[5] == square[9]:

return 1

elif square[3] == square[5] and square[5] == square[7]:

return 1

elif square[1] != 1 and square[2] != 2 and square[3] != 3 and square[4] != 4 and square[5] != 5 and square[6] != 6 and square[7] != 7 and square[8] != 8 and square[9] != 9:

return 0

else:

return -1

player = 1

status = -1

while status== -1:

board()

if player%2 == 1:

player = 1

else:

player = 2

print('\nPlayer', player)

choice = int(input('Enter a number:'))

if player == 1:

mark = 'X'

else:

mark = 'O'

if choice == 1 and square[1] == 1:

square[1] = mark

elif choice == 2 and square[2] == 2:

square[2] = mark

elif choice == 3 and square[3] == 3:

square[3] = mark

elif choice == 4 and square[4] == 4:

square[4] = mark

elif choice == 5 and square[5] == 5:

square[5] = mark

elif choice == 6 and square[6] == 6:

square[6] = mark

elif choice == 7 and square[7] == 7:

square[7] = mark

elif choice == 8 and square[8] == 8:

square[8] = mark

elif choice == 9 and square[9] == 9:

square[9] = mark

else:

print('Invalid move ')

player -= 1

status = game\_status()

player += 1

print('RESULT')

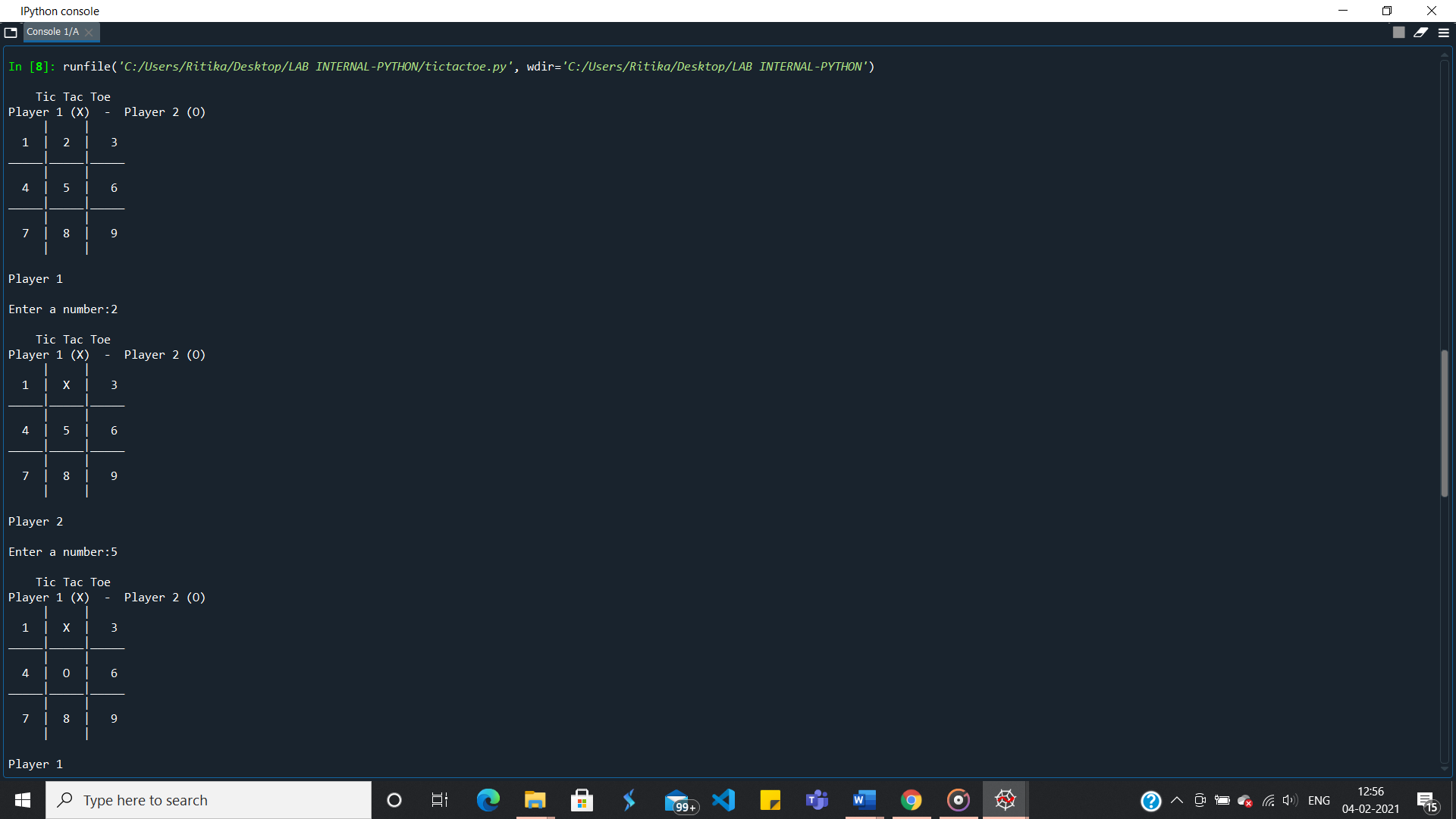
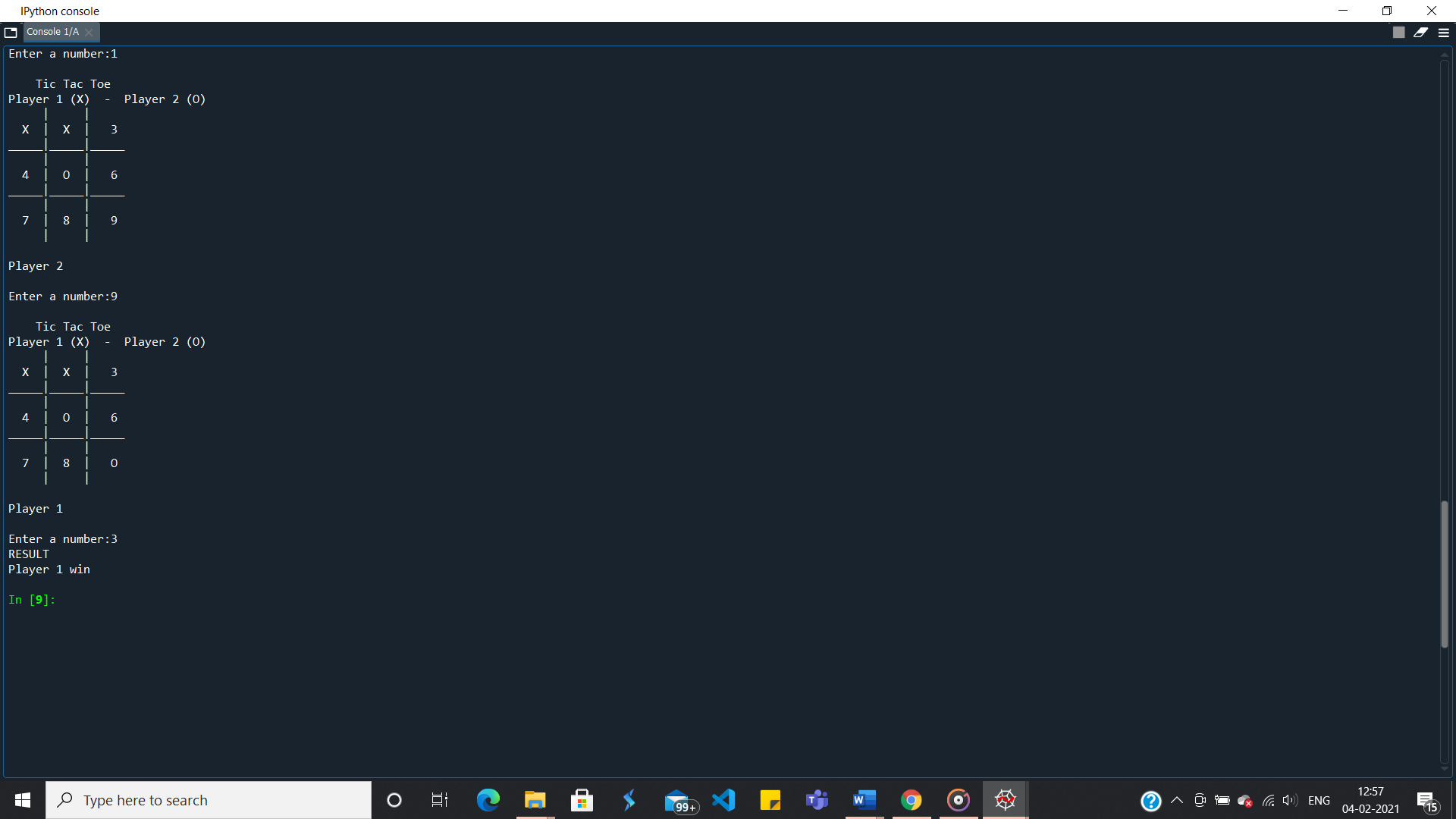
if status == 1:

print('Player',player-1,'win')

else:

print('Game draw')

-------------------------------------------------------------------------------------------------------

**Output:**

1. Write a program to implement McCulloh-Pitts algorithms, for realizing the AND/OR/XOR/ANDNOT logic functions.

**SOLUTION:**

**Explanation:**

The model allows only binary states. Neurons are connected by directed weighted path Neuron is associated with a threshold value. Neuron fires if the net input is greater than the threshold.

The threshold is set so that the inhibition is absolute because non-zero inhibitory input will prevent the neuron from firing.

----------------------------------------------------------------------------------------------------------

**Program:**

class MP\_Neuron:

threshold = 0

w1 = 0

w2 = 0

possible\_w1\_vals = [-1, 1]

possible\_w2\_vals = [-1, 1]

possible\_thresh\_vals = [-2, -1, 0, 1, 2]

def \_\_init\_\_(self, input\_matrix):

self.input\_matrix = input\_matrix

def iterate\_all\_values(self):

for w1 in self.possible\_w1\_vals:

self.w1 = w1

for w2 in self.possible\_w2\_vals:

self.w2 = w2

for threshold in self.possible\_thresh\_vals:

self.threshold = threshold

if self.check\_combination():

return True

return False

def check\_combination(self):

valid = True

for (x1, x2, y) in self.input\_matrix:

if not self.compare\_target(x1, x2, y):

valid = False

return valid

def compare\_target(self, x1, x2, target):

if self.neuron\_activate(x1, x2) == target:

return True

else:

return False

def neuron\_activate(self, x1, x2):

output = self.w1\*x1 + self.w2\*x2

if output >= self.threshold:

return 1

else:

return 0

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

AND\_Matrix = [[-1, -1, 0],[-1, 1, 0],[ 1, -1, 0],[ 1, 1, 1],]

OR\_Matrix = [[-1, -1, 0],[-1, 1, 1],[ 1, -1, 1],[ 1, 1, 1],]

NAND\_Matrix = [[-1, -1, 1],[-1, 1, 1],[ 1, -1, 1],[ 1, 1, 0],]

XOR\_Matrix = [[-1, -1, 0],[-1, 1, 1],[ 1, -1, 1],[ 1, 1, 0],]

def neuron\_calculate(mp):

if mp.iterate\_all\_values():

print("Weights are : {}, {}".format(mp.w1, mp.w2))

print("Threshold is {}".format(mp.threshold))

else:

print("Not linearly separable.")

print()

print("++ AND Gate ++")

mp\_AND = MP\_Neuron(AND\_Matrix)

neuron\_calculate(mp\_AND)

print("++ OR Gate ++")

mp\_OR = MP\_Neuron(OR\_Matrix)

neuron\_calculate(mp\_OR)

print("++ NAND Gate ++")

mp\_NAND = MP\_Neuron(NAND\_Matrix)

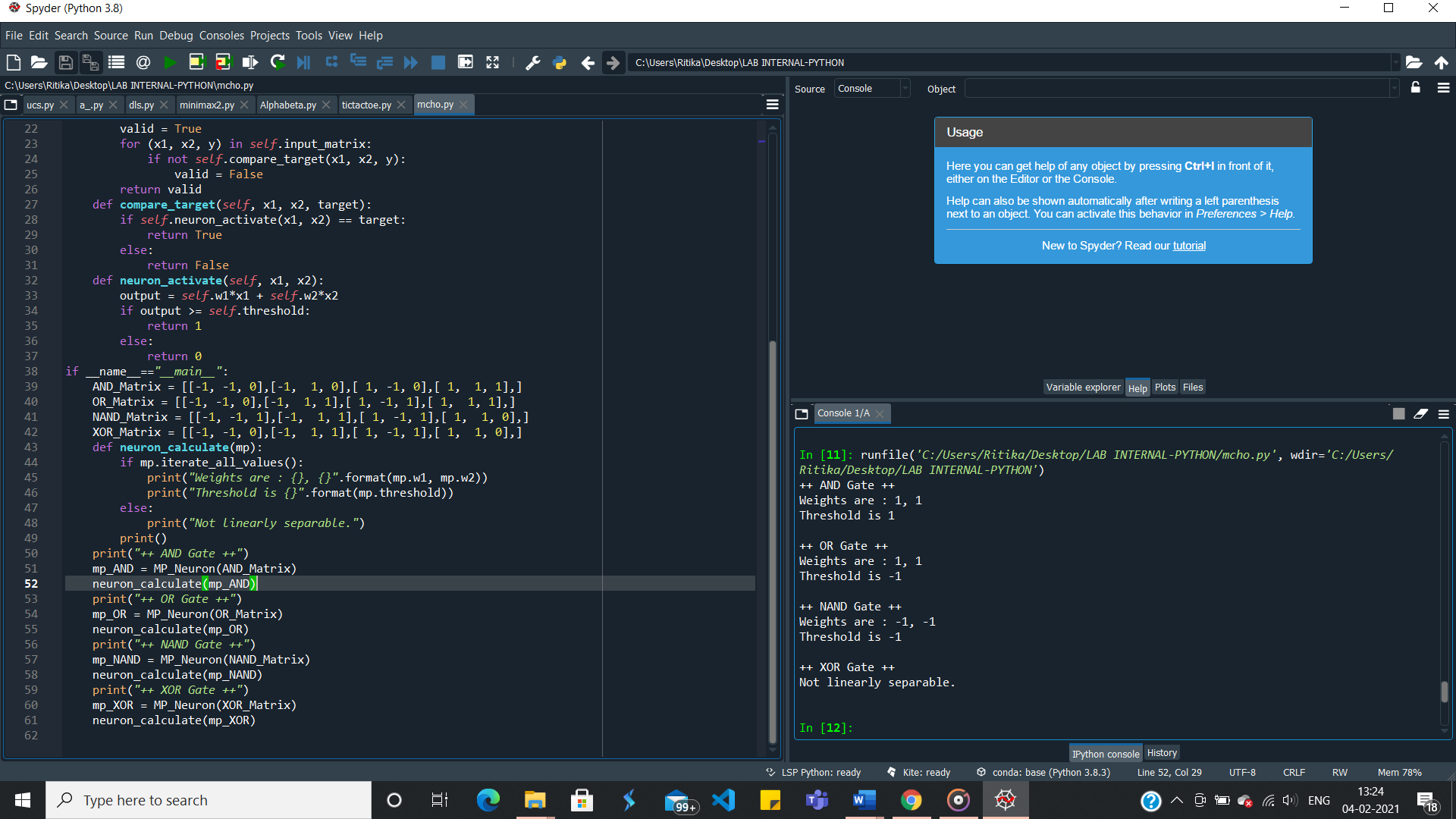
neuron\_calculate(mp\_NAND)

print("++ XOR Gate ++")

mp\_XOR = MP\_Neuron(XOR\_Matrix)

neuron\_calculate(mp\_XOR)

---------------------------------------------------------------------------------------------------

**Output:**

1. Implement the perceptron learning single layer algorithm by initializing the weights and threshold. Execute the code and check, how many iterations are needed, until the network coverage.

**SOLUTION:**

**Explanation:**

Perceptron consist of four parts-

a. Input values or one input layer: The input layer of a perceptron is made of artificial input neurons and brings the initial data into the system for further processing.

b. Weights: Weight represents the strength or dimension of the connection between units. If the weight from node 1 to node 2 has the greater quantity, then neuron 1 has greater influence over neuron 2. How much influence of the input will have on the output, is determined by weight.

c. Bias is similar to the intercept added in a linear equation. It is an additional parameter which task is to adjust the output along with the weighted sum of the inputs to the neuron.

d. Activation Function: A neuron should be activated or not, determined by an activation function. It calculates a weighted sum and further adds bias to the given result.

**Program:**

import numpy as np

theta = 1

epoch = 3

class Perceptron(object):

def \_\_init\_\_(self, input\_size, learning\_rate=0.2):

self.learning\_rate = learning\_rate

self.weights = np.zeros(input\_size + 1) # zero init for weights and bias

def predict(self, x):

return (np.dot(x, self.weights[1:]) + self.weights[0]) # X.W + B

def train(self, x, y, weights):

for inputs, label in zip(x, y):

net\_in = self.predict(inputs)

if net\_in > theta:

y\_out = 1

elif net\_in < -theta:

y\_out = -1

else:

y\_out = 0

if y\_out != label: # updating the net on incorrect prediction

self.weights[1:] += self.learning\_rate \* label \* inputs # W = alpha \* Y \* X

self.weights[0] += self.learning\_rate \* label # B = alpha \* Y

print(inputs, net\_in, label, y\_out, self.weights)

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

x = []

x.append(np.array([1, 1]))

x.append(np.array([1, -1]))

x.append(np.array([-1, 1]))

x.append(np.array([-1, -1]))

y = np.array([1, -1, -1, -1])

perceptron = Perceptron(2)

for i in range(epoch):

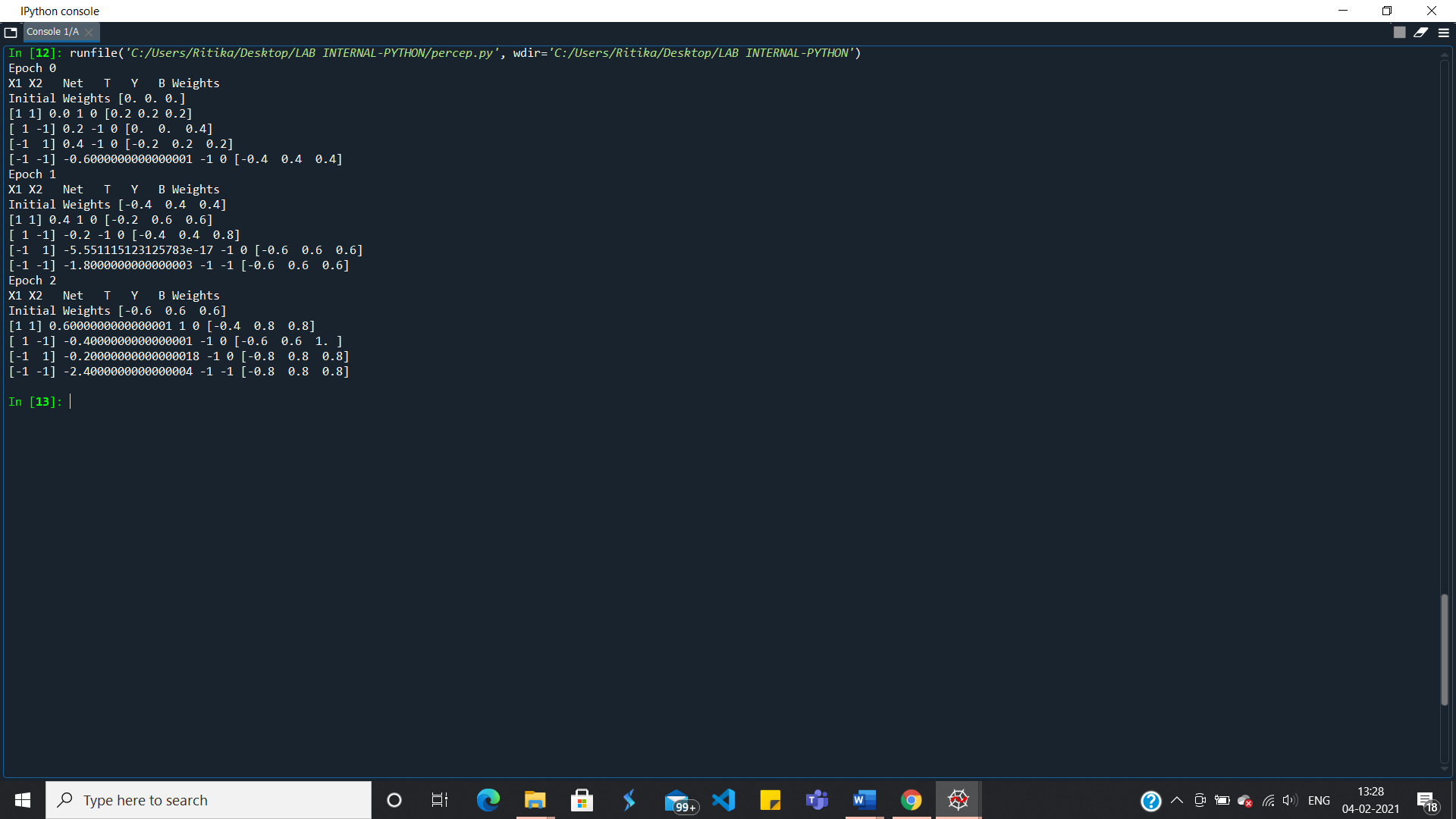
print("Epoch",i)

print("X1 X2 ", " Net ", " T ", " Y ", " B Weights")

weights = perceptron.weights

print("Initial Weights", weights)

perceptron.train(x, y, weights)

**Output:**