**AI WITH PYTHON LAB PROGRAMS**

**(18CSL58)**

1) Problems in Search

**a.** Problem Statement for A \* : ABC has to reach to Mumbai from Bangalore. As there are multiple paths to reach Mumbai help ABC to reach the destination using the shortest path by applying A\* Algorithm

**b.** Problem Statement for uniform cost search : 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 uniform cost search.

Ans.

#PQueue() functions

class PQueue():

    def \_\_init\_\_(self):

        self.dict = {}

        self.keys = []

        self.sorted = False

    #push fuction is used to push the keys into the stack with the given values. The push library is used

    def push(self, k, v):

        self.dict[k] = v

        self.sorted = False

    #sort fuction is used to sort the keys with the given values. The sort library is used

    def \_sort(self):

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

        self.sorted = True

    #pop fuction is used to pop the keys from the stack with the given values after sorting

    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

# Heuristics function is used in uniform cost search and finds the most promissing path.

# #It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal.

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

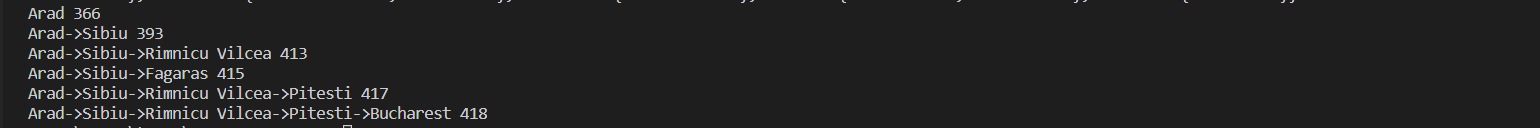
            new\_cost = cost + weight + h[node]

            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'))



2) 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.

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'))

Heuristics.txt

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

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

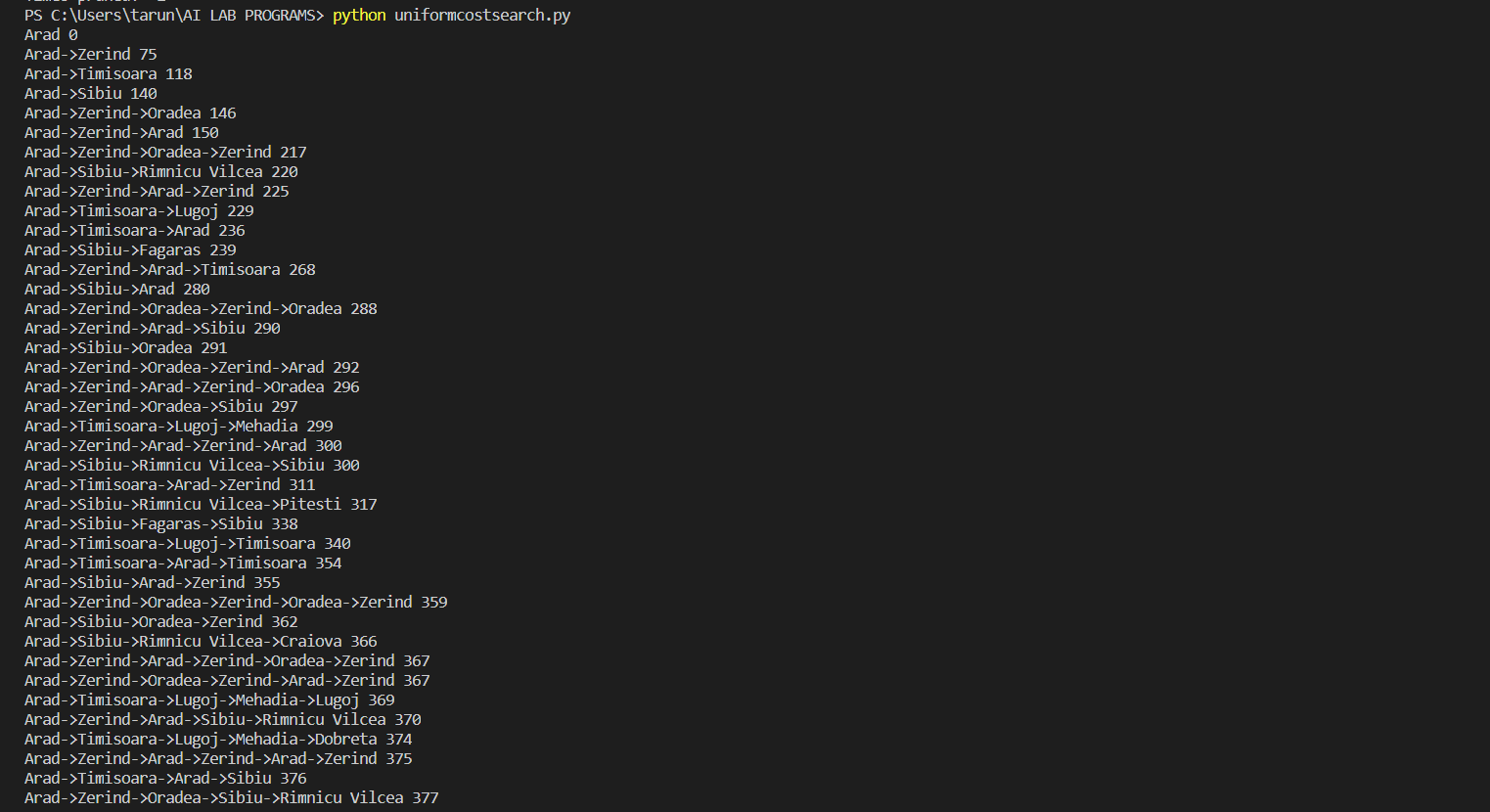
Urziceni, Vaslui, 142

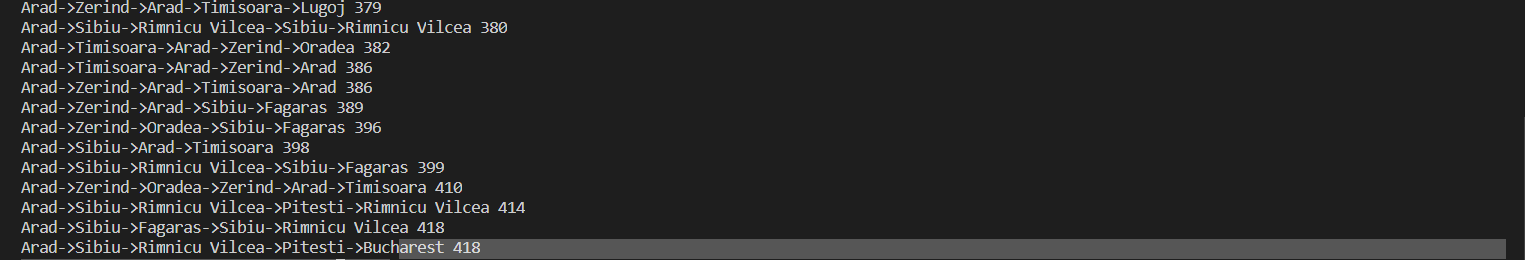
Hirsova, Eforie, 86

Vaslui, Lasi, 92

Lasi, Neamt, 87

Output:-





3) 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.

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}")

Heuristics.txt

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

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

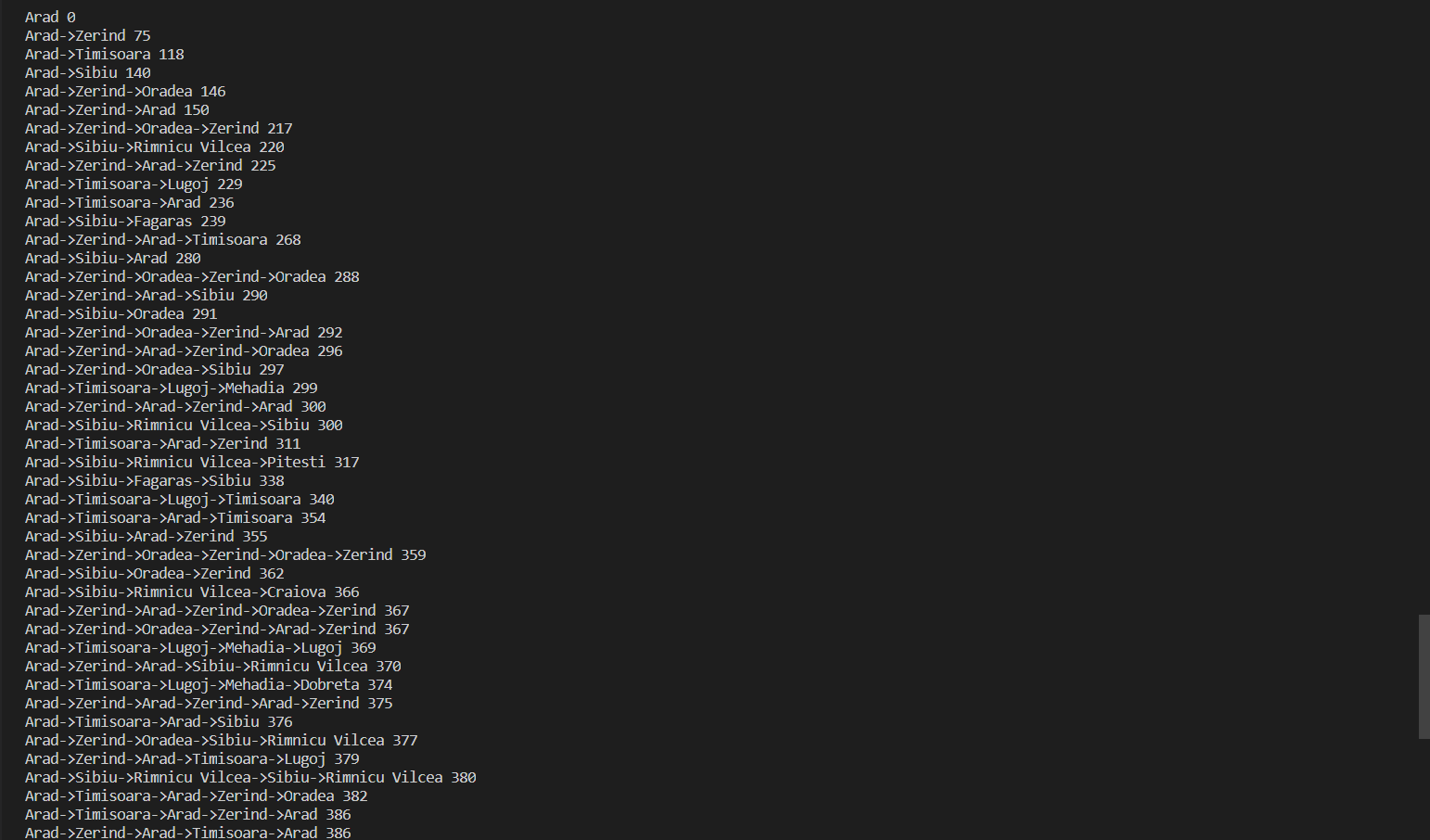
Urziceni, Vaslui, 142

Hirsova, Eforie, 86

Vaslui, Lasi, 92

Lasi, Neamt, 87

Outupt:-

4) 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.

Ans

import math

import random

#minimax class

def minimax (currentDepth, nodeIndex,

             maxTurn, score,

             tarDepth):

    # base case : tarDepth reached

    if (currentDepth == tarDepth):

        return score[nodeIndex]

    if (maxTurn):

        return max(minimax(currentDepth + 1, nodeIndex \* 2,

                    False, score, tarDepth),

                   minimax(currentDepth + 1, nodeIndex \* 2 + 1,

                    False, score, tarDepth))

    else:

        return min(minimax(currentDepth + 1, nodeIndex \* 2,

                     True, score, tarDepth),

                   minimax(currentDepth + 1, nodeIndex \* 2 + 1,

                     True, score, tarDepth))

# Driver code

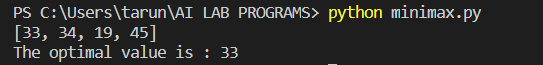
score = random.sample(range(1, 50), 4)

print(str(score))

treeDepth = math.log(len(score), 2)

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

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



5) 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. Use domain knowledge while finding the best move. Ex: for Chess, try order: captures first, then threats, then forward moves, backward moves.

Ans

tree = [[[5, 1, 2], [8, -8, -9]], [[9, 4, 5], [-3, 4, 3]]]

root = 0

pruned = 0

def children(branch, depth, alpha, beta):

    global tree

    global root

    global pruned

    i = 0

    for child in branch:

        if type(child) is list:

            (nalpha, nbeta) = children(child, depth + 1, alpha, beta)

            if depth % 2 == 1:

                beta = nalpha if nalpha < beta else beta

            else:

                alpha = nbeta if nbeta > alpha else alpha

            branch[i] = alpha if depth % 2 == 0 else beta

            i += 1

        else:

            if depth % 2 == 0 and alpha < child:

                alpha = child

            if depth % 2 == 1 and beta > child:

                beta = child

            if alpha >= beta:

                pruned += 1

                break

    if depth == root:

        tree = alpha if root == 0 else beta

    return (alpha, beta)

def alpha\_beta(in\_tree=tree, start=root, up=-15, low=15):

    global tree

    global pruned

    global root

    (alpha, beta) = children(tree, start, up, low)

    return (alpha, beta, tree, pruned)

if \_\_name\_\_ == "\_\_main\_\_":

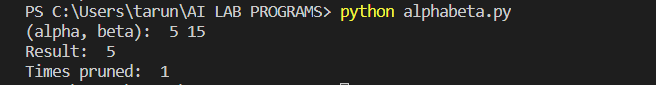
    res=[]

    (alpha, beta, tree, pruned)=alpha\_beta(None)

    print ("(alpha, beta): ", alpha, beta)

    print ("Result: ", tree)

    print ("Times pruned: ", pruned)



6) 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.

Ans

def backtrack(x, enemies, domain, assigned):

    if -1 not in assigned: # checking for unassigned people

        return x

    v = 999

    for i in range(len(domain)):

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

            v = i

    order = []

    for i in domain[v]:

        min = 1000

        for j in enemies[v]:

            temp = len(domain[j])

            if i in domain[j]:

                temp -= 1

            if temp < min:

                min = temp

        order.append((i, min))

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

    ordered = [i[0] for i in order]

    for i in ordered:

        new\_d = [[j for j in i] for i in domain]

        for j in enemies[v]:

            if i == x[j]:

                continue

        x[v] = i

        assigned[v] = 1

        new\_d[v] = [z for z in new\_d[v] if z==i]

        temp = []

        for j in range(len(new\_d)):

            if j!=v and j in enemies[v]:

                new\_d[j] = [z for z in new\_d[j] if z!=i]

        res = backtrack(x, enemies, new\_d, assigned)

        if res!=0:

            return res

    x[v] = ""

    assigned[v] = -1

    return 0

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

    people = int(input("Number of people = "))

    tables = int(input("Number of tables = "))

    edges = []

    rows = input("People who should not sit together = ").split()

    while(rows):

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

        rows = input().split()

    x = ["" for i in range(people)]

    # filling out the enemies matrix

    enemies = [[] for i in range(people)]

    for i in edges:

        enemies[i[0]].append(i[1])

        enemies[i[1]].append(i[0])

    for i in range(people):

        j = list(set(enemies[i])) # deduplicating the each row

        enemies[i] = j

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

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

    res = backtrack(x, enemies, domain, assigned)

    if res == 0:

        print("Tables could not be assigned")

    else:

        for i in range(len(res)):

            print(f"{i} : {res[i]}")

7 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.

Ans

import os

turn = 'X'

win = False

spaces = 9

def draw(board):

    for i in range(6, -1, -3):

        print(' ' + board[i] + '|' +

              board[i+1] + '|' + board[i+2])

def takeinput(board, spaces, turn):

    pos = -1

    print(turn + "'s turn:")

    while pos == -1:

        try:

            print("Pick position 1-9:")

            pos = int(input())

            if(pos < 1 or pos > 9):

                pos = -1

            elif board[pos - 1] != ' ':

                pos = -1

        except:

            print("enter a valid position")

    spaces -= 1

    board[pos - 1] = turn

    if turn == 'X':

        turn = 'O'

    else:

        turn = 'X'

    return board, spaces, turn

def checkwin(board):

    # could probably make this better

    for i in range(0, 3):

        # rows

        r = i\*3

        if board[r] != ' ':

            if board[r] == board[r+1] and board[r+1] == board[r+2]:

                return board[r]

        # columns

        if board[i] != ' ':

            if board[i] == board[i+3] and board[i] == board[i+6]:

                return board[i]

    # diagonals

    if board[0] != ' ':

        if (board[0] == board[4] and board[4] == board[8]):

            return board[0]

    if board[2] != ' ':

        if (board[2] == board[4] and board[4] == board[6]):

            return board[2]

    return 0

board = [' ']\*9

while not win and spaces:

    draw(board)

    board, spaces, turn = takeinput(board, spaces, turn)

    win = checkwin(board)

    os.system('cls')

draw(board)

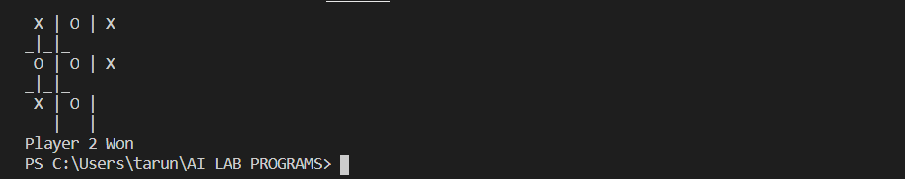
if not win and not spaces:

    print("draw")

elif win:

    print(f'{win} wins')

    input()



9) 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 converge.

Ans

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)