**AI Python Lab Programs**

1a) Problem statement for A\*: ABC has to reach Mumbai from Bangalore as there are multiple paths, help ABC reach the destination using the **shortest path by applying A\* algo.**

1b) Problem statement for Uniform Cost Search: We have the Map of Romania. In this, 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.**

1c) Problem statement for Depth Limited Search: Design and develop a prog in Python to print all the nodes reachable from a given starting node in a graph using **Depth Limited Search method**. Repeat the experiment for different graphs.

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

3) Write a program to implement **Alpha Beta Pruning** in Python. The algo 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.

4) Assume that you are organizing a party for N people and have been given a list L of people who should not sit at the same table. Furthermore, assume that you have C tables that are infinitally large. **Write a function layout (N,C,L) that can give a table placement ( a number from 0 to C-1) for each guest. 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.

5) Implementation of **Tic Tac Toe game**, the player needs to take turns marking the spaces in a 3x3 grid with their own marks, if 3 consecutive marks are formed then the player who owns those moves wins.

6) Write a python program to implement the **Wumpus world problem**. The problem deals with an AI robot navigating its way through a 4x4 puzzle to try and find gold. The robot must safely navigate its way around bottomless pits of death and evil Wumpus creatures to locate the gold hidden on the board. After it has successfully found the gold, it must safely navigate its way back to the starting point. The robot must use its light sensors and the signals sent to it at each square to determine which way to properly navigate to reach its goal.

7) Write a program to implement **McCulloch-Pitts algo**.

8) Implement the **Perceptron Learning Single Layer** algo by initializing the weights and the threshold calculate the error of a prediction plot the dataset to see, that they are linearly separable execute the code and check, how many iterations are needed, until all samples are classified right.

9) Write a python program to implement **neural networks with backpropagation for XOR using one hidden layer**.

10) Suppose we have some pattern of arbitrary dimensions, however we need them in one dimension or two dimensions. Then the process of feature mapping would be very useful to convert the wide pattern space into a typical feature space. Write a program to implement Kohonen algo.

11) Simulate the **Mexican Hat Neural Network** **with fixed weight competitive net**. *External signals (S), number of iterations (t\_max), Radius of region with positive reinforcement (R1), Radius of region with negative reinforcement (R2), vector of activations (X) are the inputs.*

1. **a**

class Node():

    def \_\_init\_\_(self, parent=None, position=None):

        self.parent = parent

        self.position = position

        self.g = 0

        self.h = 0

        self.f = 0

    def \_\_eq\_\_(self, other):

        return self.position == other.position

def astar(maze, start, end):

    start\_node = Node(None, start)

    start\_node.g = start\_node.h = start\_node.f = 0

    end\_node = Node(None, end)

    end\_node.g = end\_node.h = end\_node.f = 0

    open\_list = []

    closed\_list = []

    open\_list.append(start\_node)

    while len(open\_list) > 0:

        current\_node = open\_list[0]

        current\_index = 0

        for index, item in enumerate(open\_list):

            if item.f < current\_node.f:

                current\_node = item

                current\_index = index

        open\_list.pop(current\_index)

        closed\_list.append(current\_node)

        if current\_node == end\_node:

            path = []

            current = current\_node

            while current is not None:

                path.append(current.position)

                current = current.parent

            return path[::-1]

        children = []

        for new\_position in [(0, -1), (0, 1), (-1, 0), (1, 0), (-1, -1), (-1, 1), (1, -1), (1, 1)]:

            node\_position = (current\_node.position[0] + new\_position[0], current\_node.position[1] + new\_position[1])

            if node\_position[0] > (len(maze) - 1) or node\_position[0] < 0 or node\_position[1] > (

                    len(maze[len(maze) - 1]) - 1) or node\_position[1] < 0:

                continue

            if maze[node\_position[0]][node\_position[1]] != 0:

                continue

            new\_node = Node(current\_node, node\_position)

            children.append(new\_node)

        for child in children:

            for closed\_child in closed\_list:

                if child == closed\_child:

                    continue

            child.g = current\_node.g + 1

            child.h = ((child.position[0] - end\_node.position[0]) \*\* 2) + (

                    (child.position[1] - end\_node.position[1]) \*\* 2)

            child.f = child.g + child.h

            for open\_node in open\_list:

                if child == open\_node and child.g > open\_node.g:

                    continue

            open\_list.append(child)

def main():

    maze = [[0, 0, 0, 0, 1, 0],

            [0, 0, 0, 0, 1, 0],

            [0, 0, 0, 0, 1, 0],

            [0, 0, 0, 0, 1, 0],

            [0, 0, 0, 0, 1, 0],

            [0, 0, 0, 0, 0, 0]]

    graph = [[0, 1, 0, 0, 0, 0],

             [1, 0, 1, 0, 1, 0],

             [0, 1, 0, 0, 0, 1],

             [0, 0, 0, 0, 1, 0],

             [0, 1, 0, 1, 0, 0],

             [0, 0, 1, 0, 0, 0]

             ]

    start = (0, 0)

    end = (5, 5)

    end1 = (5, 5)

    path = astar(maze, start, end)

    print(path)

    path1 = astar(graph, start, end1)

    print(path1)

if \_\_name\_\_ == '\_\_main\_\_':

    main()

1. **.**

# Function to find minimum and maximum position in list

def maxminpos(A, n):

   # inbuilt function to find the position of minimum

   minpos = A.index(min(A))

   # inbuilt function to find the position of maximum

   maxpos = A.index(max(A))

   print ("The maximum is at position::", maxpos and + 1)

   print ("The minimum is at position::", minpos + 1)

# Driver code

A=list()

n=int(input("Enter the size of the List ::"))

print("Enter the Element ::")

for i in range(int(n)):

   k=int(input(""))

   A.append(k)

maxminpos (A,n)

1. .

**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

**// Calling the function for the first time.**

minimax(0, 0, true, -INFINITY, +INFINITY)

# Initial values of Alpha and Beta

MAX, MIN = 1000, -1000

# Returns optimal value for current player

#(Initially called for root and maximizer)

def minimax(depth, nodeIndex, maximizingPlayer,

            values, alpha, beta):

    # Terminating condition. i.e

    # leaf node is reached

    if depth == 3:

        return values[nodeIndex]

    if maximizingPlayer:

        best = MIN

        # Recur for left and right children

        for i in range(0, 2):

            val = minimax(depth + 1, nodeIndex \* 2 + i,

                          False, values, alpha, beta)

            best = max(best, val)

            alpha = max(alpha, best)

            # Alpha Beta Pruning

            if beta <= alpha:

                break

        return best

    else:

        best = MAX

        # Recur for left and

        # right children

        for i in range(0, 2):

            val = minimax(depth + 1, nodeIndex \* 2 + i,

                            True, values, alpha, beta)

            best = min(best, val)

            beta = min(beta, best)

            # Alpha Beta Pruning

            if beta <= alpha:

                break

        return best

# Driver Code

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

    values = [3, 5, 6, 9, 1, 2, 0, -1]

    print("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))

1. **G**
2. **.**

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

def main():

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')

# FUNCTION TO RETURN GAME STATUS, 1 FOR GAME IS OVER WITH RESULT, -1 FOR GAME IS IN PROGRESS, O GAME IS OVER AND NO RESULT

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

# FUNCTION TO DRAW BOARD OF TIC TAC TOE WITH PLAYERS MARK

def board():

print('\n\n\tTic Tac Toe\n\n')

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

print()

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(' | | ' )

main()

import os

import time

board = [' ',' ',' ',' ',' ',' ',' ',' ',' ',' ']

player = 1

########win Flags##########

Win = 1

Draw = -1

Running = 0

Stop = 1

###########################

Game = Running

Mark = 'X'

#This Function Draws Game Board

def DrawBoard():

print(" %c | %c | %c " % (board[1],board[2],board[3]))

print("\_\_\_|\_\_\_|\_\_\_")

print(" %c | %c | %c " % (board[4],board[5],board[6]))

print("\_\_\_|\_\_\_|\_\_\_")

print(" %c | %c | %c " % (board[7],board[8],board[9]))

print(" | | ")

#This Function Checks position is empty or not

def CheckPosition(x):

if(board[x] == ' '):

return True

else:

return False

#This Function Checks player has won or not

def CheckWin():

global Game

#Horizontal winning condition

if(board[1] == board[2] and board[2] == board[3] and board[1] != ' '):

Game = Win

elif(board[4] == board[5] and board[5] == board[6] and board[4] != ' '):

Game = Win

elif(board[7] == board[8] and board[8] == board[9] and board[7] != ' '):

Game = Win

#Vertical Winning Condition

elif(board[1] == board[4] and board[4] == board[7] and board[1] != ' '):

Game = Win

elif(board[2] == board[5] and board[5] == board[8] and board[2] != ' '):

Game = Win

elif(board[3] == board[6] and board[6] == board[9] and board[3] != ' '):

Game=Win

#Diagonal Winning Condition

elif(board[1] == board[5] and board[5] == board[9] and board[5] != ' '):

Game = Win

elif(board[3] == board[5] and board[5] == board[7] and board[5] != ' '):

Game=Win

#Match Tie or Draw Condition

elif(board[1]!=' ' and board[2]!=' ' and board[3]!=' ' and board[4]!=' ' and board[5]!=' ' and board[6]!=' ' and board[7]!=' ' and board[8]!=' ' and board[9]!=' '):

Game=Draw

else:

Game=Running

print("Tic-Tac-Toe Game Designed By Sourabh Somani")

print("Player 1 [X] --- Player 2 [O]\n")

print()

print()

print("Please Wait...")

time.sleep(3)

while(Game == Running):

os.system('cls')

DrawBoard()

if(player % 2 != 0):

print("Player 1's chance")

Mark = 'X'

else:

print("Player 2's chance")

Mark = 'O'

choice = int(input("Enter the position between [1-9] where you want to mark : "))

if(CheckPosition(choice)):

board[choice] = Mark

player+=1

CheckWin()

os.system('cls')

DrawBoard()

if(Game==Draw):

print("Game Draw")

elif(Game==Win):

player-=1

if(player%2!=0):

print("Player 1 Won")

else:

print("Player 2 Won")

1. **G**

classMP\_Neuron:

# firing threshold for the neuron

threshold = 0

# weights for the neuron

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

defiterate\_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

ifself.check\_combination():

return True

return False

defcheck\_combination(self):

valid = True

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

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

valid = False

return valid

defcompare\_target(self, x1, x2, target):

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

return True

else:

return False

defneuron\_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],

]

defneuron\_calculate(mp):

ifmp.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)

1. **G**

import numpy as np

teta = 1 #Threshold

epoch = 3 #Epoch value, change to repeat the learning till ntwk converge

class Perceptron(object):

def \_\_init\_\_(self, no\_of\_inputs, learning\_rate=0.2 ):

self.learning\_rate = learning\_rate

self.weights = np.zeros(no\_of\_inputs + 1) #Initialize weights & bias as zero, weights[0] is used as bias

def predict(self, inputs):

return (np.dot(inputs, self.weights[1:]) + self.weights[0])

# Returns the net\_in

def train(self, training\_inputs, t,weights): #Calculate the weights for each input

for inputs, label in zip(training\_inputs, t):

net\_in = self.predict(inputs)

if net\_in > teta: #Find the activation output Y

y\_out = 1

elif net\_in < -teta:

y\_out = -1

else:

y\_out = 0

if y\_out != label: #If t != y , update the weights, otherwise W(new)=W(old)

self.weights[1:] += self.learning\_rate \* label \* inputs #W(new)

self.weights[0] += self.learning\_rate \* label #b(new)

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

def disp(self):

print(self.weights)

training\_inputs = [] #Holds vector of inputs for AND/OR logic

training\_inputs.append(np.array([1, 1]))

training\_inputs.append(np.array([1, -1]))

training\_inputs.append(np.array([-1, 1]))

training\_inputs.append(np.array([-1, -1]))

t = np.array([1, 1, 1, -1]) #Target output

perceptron = Perceptron(2) #Create 2 input object vector for i in range(epoch):

print("Epoch",i)

print("X1 X2","Net"," T"," Y", "B Weights") # For output display pattern

weights = perceptron.weights

print("Initial Weights",weights)

perceptron.train(training\_inputs, t,weights)

#Train the model & display weights

# In[ ]:

1. **G**
2. **G**
3. **G**