**NAME: THAKKAR MALAY TUSHARBHAI**

**ENROLLMENT NO.:20012011169**

**DEPARTMENT: B. Tech. CE**

**Batch: CEIT-B\_6AB4**

**SUBJECT: 2CEIT602: ARTIFICIAL INTELLIGENCE**

**COLLEGE: U. V. PATEL COLLEGE OF ENGINEERING**

**Practical-1**

**AIM: Write a program to implement simple Chat bot using Python**

**(without using any libraries or packages of python).**

**• It should accept any case like lower case and upper case**

**• It should accept any symbol like !,?. etc. by entering user**

**• It should have at least 20 questions.**

* **Program:**

print("Malay Thakkar(20012011169)")

question = {"how are you": "fine",

            "name of your college": "uvpce",

            "which type of course available": "btech,bsc,bba,etc",

            "fees in btech": "57000",

            "how many semester in btech": "8",

            "fees in bsc": "25000",

            "how many semester in bsc": "6",

            "how many department in btech": "ce,it,mechanical,etc",

            "is it good college": "yes",

            "what is university name": "guni",

            "is bsc available": "yes",

            "is bba available": "yes",

            "is bpharm available": "no",

            "what is bpharm fees": "40000",

            "which is last date for admission": "30 june",

            "how far from ahmedabad": "60km",

            "is ce good": "yes depend on interest",

            "is ce hard": "depend on interest",

            "course duration of btech": "4 years",

            "course duration of bsc": "3 years",

            }

def chatbot():

    while True:

        qs = input("Enter Question: ").lower()

        symbols = {'?','!', '@', '#', '$', '%', '^', '&', '\*', '(', ')', '-'}

        message = ""

        for i in qs:

            if i not in symbols:

                message = message+i

        if message in ["quit",'bye']:

            print("College-bot: Bye-bye")

            break;

        elif message in question:

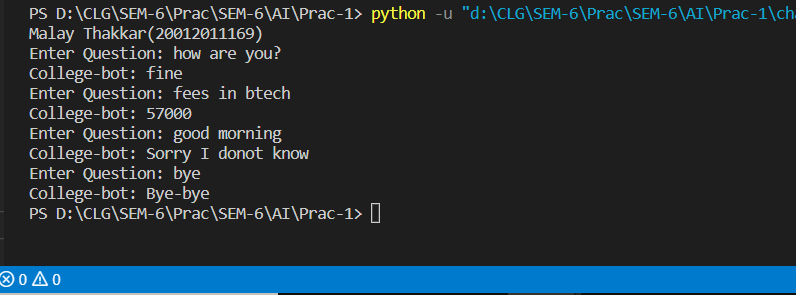
            print("College-bot: "+question[message])

        else:

            print("College-bot: Sorry I donot know")

chatbot()

* **Output:**

****

**Practical-2**

**AIM: Write a program to implement Breadth first search Traversal on Tree using Python (without using any libraries or packages of python)**

**• Use class concept of python (Tree Class, Node Class)**

**• Use class to implement Data structure to be used in program**

**• Tree & Output should look like below:**

* **Program:**

class Node:

    def \_\_init\_\_(self, data, parent, childlist):

        self.data = data

        self.parent = parent

        self.childlist = childlist

        self.level = 0

        if self.parent != None:

            self.level = self.parent.level + 1

    def add\_child(self, child):

        self.childlist.append(child)

    def space\_count(self):

        str\_parent = ""

        if self.parent == None:

            str\_parent = str(self.parent)

        else:

            str\_parent = "None"

            temp = self.parent

            while temp != None:

                str\_parent += "->" + temp.data

                temp = temp.parent

        return len(str\_parent)

    def \_\_repr\_\_(self):

        str\_parent = ""

        if self.parent == None:

            str\_parent = str(self.parent)

        else:

            str\_parent = str(self.parent.data)

        str\_return = "\n"

        str\_return += " " \* self.space\_count()

        str\_return += "->" + str(self.data) + " " + \

            ' '.join(map(str, self.childlist))

        return str\_return

class Tree:

    def \_\_init\_\_(self, root):

        self.root = root

    def insert\_node(self, data, parent):

        node = Node(data, parent, [])

        parent.add\_child(node)

        return node

    def \_\_repr\_\_(self):

        return str(self.root)

def bfs(tree, search\_string):

    queue = []

    queue.append(tree.root)

    node = None

    while queue:

        temp = queue.pop(0)

        if temp.data == search\_string:

            node = temp

            break

        queue.extend(temp.childlist)

    return node

def draw\_path(node):

    list = []

    temp = node

    while temp != None:

        list.append(temp.data)

        temp = temp.parent

        if temp == None:

            break

    list.reverse()

    print("Path: ")

    print(\*list, sep="->")

    print("Path Cost = " + str(len(list)-1))

tree = Tree(Node("India", None, []))

gujarat = tree.insert\_node("Gujarat", tree.root)

ahmedabad = tree.insert\_node("Ahmedabad", gujarat)

mehsana = tree.insert\_node("Mehsana", gujarat)

gandhinagar = tree.insert\_node("Gandhinagar", gujarat)

rajasthan = tree.insert\_node("Rajasthan", tree.root)

jaipur = tree.insert\_node("Jaipur", rajasthan)

jodhpur = tree.insert\_node("Jodhpur", rajasthan)

ajmer = tree.insert\_node("Ajmer", rajasthan)

kota = tree.insert\_node("Kota", rajasthan)

maharashtra = tree.insert\_node("Maharashtra", tree.root)

mumbai = tree.insert\_node("Mumbai", maharashtra)

bandra = tree.insert\_node("Bandra", mumbai)

juhu = tree.insert\_node("Juhu", mumbai)

nashik = tree.insert\_node("Nashik", maharashtra)

pune = tree.insert\_node("Pune", maharashtra)

nagpur = tree.insert\_node("Nagpur", maharashtra)

thane = tree.insert\_node("Thane", maharashtra)

print("Malay Thakkar (20012011169)")

print(tree)

search\_string = "Bandra"

print("Search String = " + search\_string)

node = bfs(tree, search\_string)

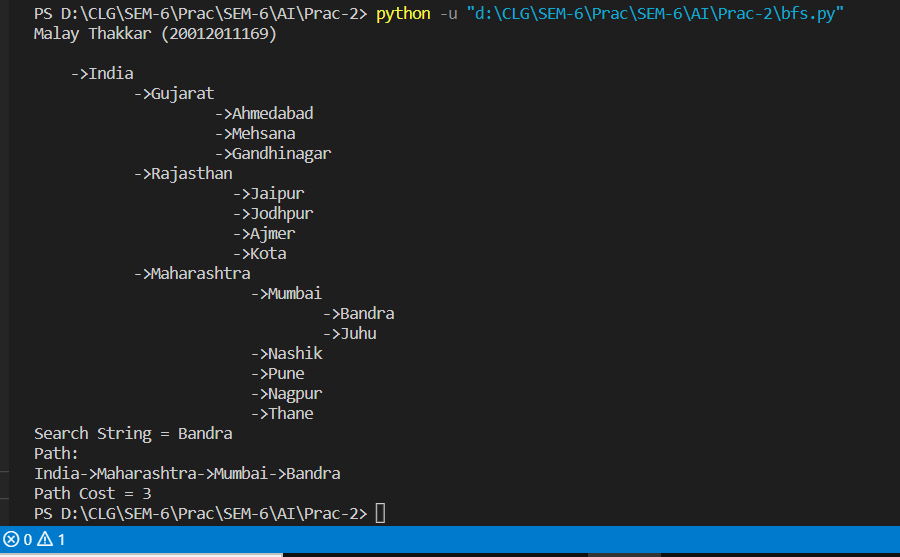
if node == None:

    print(search\_string + "String can't be found in tree")

else:

    draw\_path(node)

* **Output:**



**Practical-3**

**AIM: Write a program to implement a Water Jug Problem using Python**

**and to solve a Water Jug Problem by using BFS (without using any**

**libraries or packages of python).**

* **CODE:**

import time

import random

class node:

    def \_\_init\_\_(self, data):

        self.x = 0

        self.y = 0

        self.parent = data

    def \_\_cmp\_\_(self,other):

      if(other == None):

        return False

      return self.x == other.x and self.y == other.y

    def \_\_eq\_\_(self,other):

      if(other == None):

        return False

      return self.x == other.x and self.y == other.y

    def \_\_repr\_\_(self):

      return "("+str(self.x)+", "+str(self.y)+")"

def operation(cnode, rule):

        x = cnode.x

        y = cnode.y

        if rule == 1:

            if x < maxjug1:

                x = maxjug1

            else:

                return None

        elif rule==2:

            if y < maxjug2:

                y = maxjug2

            else:

                return None

        elif rule==3:

            if x > 0:

                x = 0

            else:

                return None

        elif rule==4:

            if y > 0:

                y = 0

            else:

                return None

        elif rule==5:

            if x+y >= maxjug1:

                y=y-(maxjug1-x)

                x = maxjug1

            else:

                return None

        elif rule==6:

            if x+y >= maxjug2:

                x = x-(maxjug2-y)

                y = maxjug2

            else:

                 return None

        elif rule==7:

            if x+y < maxjug1:

                x = x+y

                y = 0

            else:

                return None

        elif rule==8:

            if x+y < maxjug2:

                y = x+y

                x = 0

            else:

                return None

        if(x==cnode.x and y==cnode.y):

            return None

        nextnode=node(cnode)

        nextnode.x=x

        nextnode.y=y

        nextnode.parent=cnode

        return nextnode

class BFS:

  def \_\_init\_\_(self,initNode,goalNode):

    self.initNode = initNode

    self.goalNode = goalNode

    self.q = []

    self.q.append(initNode)

  def pushList(self,list1):

    self.q.extend(list1)

  def popNode(self):

    return self.q.pop(0)

  def isNotEmpty(self):

    return len(self.q)>0

  def generateAllSuccessor(self,cnode):

    list1 = []

    for i in range(1,9):

      nextNode = operation(cnode,i)

      if(nextNode != None):

        list1.append(nextNode)

    return list1

  def execution(self):

    while self.isNotEmpty():

      cnode = self.popNode()

      #print("Pop Node:"+str(cnode))

      if cnode.x == self.goalNode.x:

        return cnode

      list1 = self.generateAllSuccessor(cnode)

      self.pushList(list1)

    return None

class DFS:

  def \_\_init\_\_(self,initNode,goalNode):

    self.initNode = initNode

    self.goalNode = goalNode

    self.q = []

    self.q.append(initNode)

    self.popList = []

  def pushList(self,list1):

    self.q.extend(list1)

  def popNode(self):

    return self.q.pop()

  def isNotEmpty(self):

    return len(self.q)>0

  def generateAllSuccessor(self,cnode):

    list1 = []

    for i in range(1,9):

      nextNode = operation(cnode,i)

      if(nextNode != None):

        list1.append(nextNode)

    return list1

  def generateAllSuccessorByRandom(self,cnode):

    list1 = []

    ruleList = []

    while(len(ruleList)!= 8):

      i = random.randint(1,8)

      if(i not in ruleList):

        ruleList.append(i)

    for i in ruleList:

      nextNode = operation(cnode,i)

      if(nextNode != None):

        list1.append(nextNode)

    return list1

  def generateAllSuccessorByRandomPopList(self,cnode):

    list1 = []

    ruleList = []

    while(len(ruleList)!= 8):

      i = random.randint(1,8)

      if(i not in ruleList):

        ruleList.append(i)

    for i in ruleList:

      nextNode = operation(cnode,i)

      if(nextNode != None and nextNode not in self.popList):

        list1.append(nextNode)

    return list1

  def execution(self):

    while self.isNotEmpty():

      cnode = self.popNode()

      self.popList.append(cnode)

      #print("Pop Node:"+str(cnode))

      if cnode.x == self.goalNode.x:

        return cnode

      list1 = self.generateAllSuccessorByRandomPopList(cnode)

      self.pushList(list1)

    return None

def printPath(cnode):

  temp = cnode

  retStr = ""

  pathCost = 0

  while(temp!=None):

    retStr = str(temp)+"\n"+retStr

    temp = temp.parent

    pathCost += 1

  print(retStr)

  print("Path Cost="+str(pathCost-1))

print("Malay Thakkar-20012011169")

maxjug1=int(input("Enter value of maxjug1:"))

maxjug2=int(input("Enter value of maxjug2:"))

initialNode=node(None)

initialNode.x=0

initialNode.y=0

initialNode.parent=None

GoalNode=node(None)

GoalNode.x=int(input("Enter value of goal in jug1:"))

GoalNode.y=0

GoalNode.parent=None

print("BFS Algorithm")

startTime = time.time()

bfsSolNode = BFS(initialNode,GoalNode).execution()

endTime = time.time()

diffTime = endTime - startTime

if(bfsSolNode != None):

  print("Got Solution:")

  printPath(bfsSolNode)

  print("Execution Time="+str(diffTime\*1000)+"ms")

else:

  print("No Solution")

print("DFS Algorithm")

startTime = time.time()

dfsSolNode = DFS(initialNode,GoalNode).execution()

endTime = time.time()

diffTime = endTime - startTime

if(dfsSolNode != None):

  print("Got Solution:")

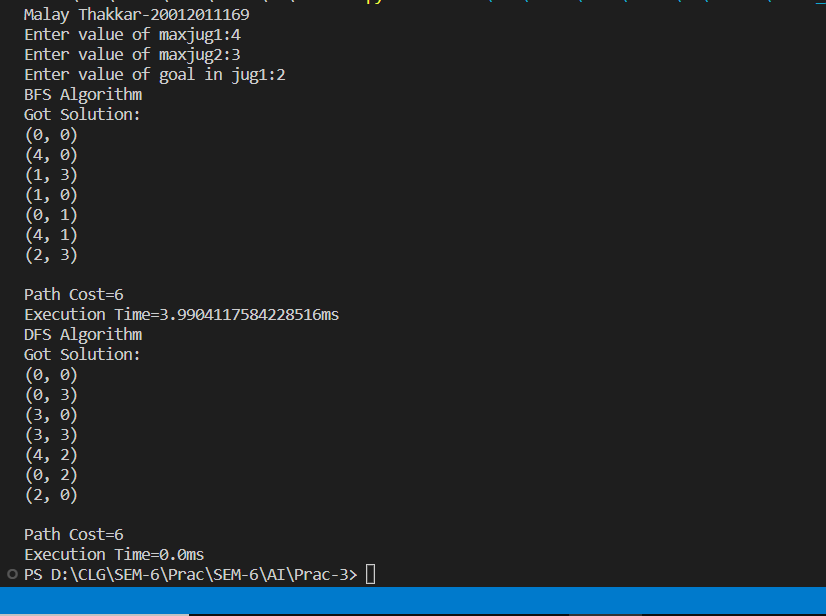
  printPath(dfsSolNode)

  print("Execution Time="+str(diffTime\*1000)+"ms")

else:

  print("No Solution")

* **OUTPUT:**

****

**Practical-4**

**AIM: Write a program to solve 8 puzzle problem using the Best First search algorithm and also find Execution time, completeness of algorithm, etc.**

**Consider following steps to create a program in python:**

**1. Create Enum named “Action” for this problem**

**2. Create Node class with support of compare node & sort node**

**3. Choose appropriate heuristic function to solve this problem and create in Node class**

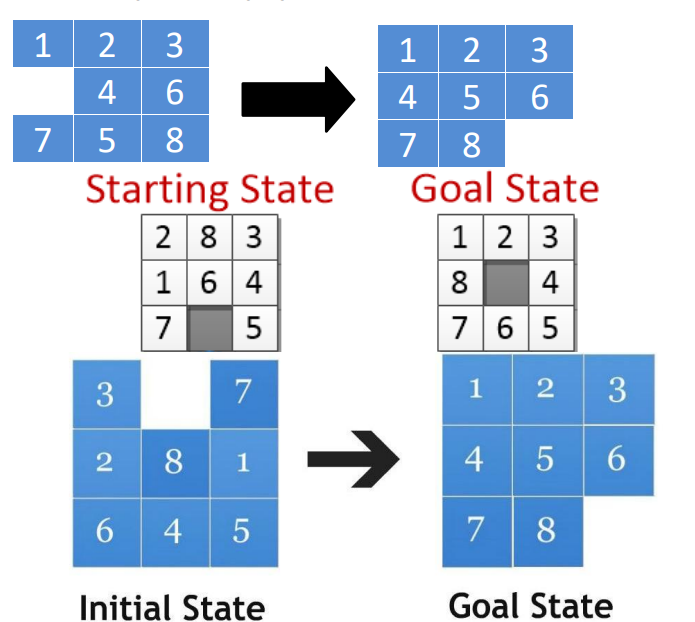
**4. Create BestFirstSearch class with “execution” method**

**5. Output should be according to given image**

**6. Print execution time & number of steps needed to reach goal state**

**7. Don’t use any libraries or packages of python**

**8. Test Program according to given below test cases**



import enum

import time

class Action(enum.Enum):

    MoveDown = 0

    MoveUp = 1

    MoveLeft = 2

    MoveRight = 3

    noAction = 4

class Node:

    def \_\_init\_\_(self, position, action=Action.noAction, parent=None):

        self.position = position

        self.action = action

        self.parent = parent

        self.h = 0

        self.f = 0

    def printNode(self):

        print("Position : ", self.position, "\n", "Action : ", self.action, "\n", "Parent : ", self.parent, "\n", )

    def \_\_eq\_\_(self, other):

        return self.position == other.position

    def \_\_lt\_\_(self, other):

        return self.f < other.f

    def \_\_gt\_\_(self, other):

        return self.f > other.f

    def \_\_repr\_\_(self):

        return '\n'.join(

            ['\n', str(self.action), str(self.position[:3]), str(self.position[3:6]), str(self.position[6:])]).replace(

            '[', '').replace(']', '').replace(',', '').replace('0', '\_')

    # heuristic value

    def \_h(self, goal):

        return sum([1 if self.position[i] != goal[i] else 0 for i in range(9)])

    def generateValue(self, goal):

        self.h = self.\_h(goal)

        self.f = self.h

    # Possible Moves

    def possibleMoves(self):

        successor = []

        i = self.position.index(0)

        # MoveDown

        if i in [3, 4, 5, 6, 7, 8]:

            newValue = self.position[:]

            newValue[i], newValue[i - 3] = newValue[i - 3], newValue[i]

            successor.append(Node(position=newValue, parent=self, action=Action.MoveDown))

        # MoveUp

        if i in [0, 1, 2, 3, 4, 5]:

            newValue = self.position[:]

            newValue[i], newValue[i + 3] = newValue[i + 3], newValue[i]

            successor.append(Node(position=newValue, parent=self, action=Action.MoveUp))

        # MoveLeft

        if i in [0, 1, 3, 4, 6, 7]:

            newValue = self.position[:]

            newValue[i], newValue[i + 1] = newValue[i + 1], newValue[i]

            successor.append(Node(position=newValue, parent=self, action=Action.MoveLeft))

        # MoveRight

        if i in [1, 2, 4, 5, 7, 8]:

            newValue = self.position[:]

            newValue[i], newValue[i - 1] = newValue[i - 1], newValue[i]

            #             successor.append(Node(newValue,self,Action.MoveDown))

            successor.append(Node(position=newValue, parent=self, action=Action.MoveRight))

        return successor

def push(list1, node):

    list1.append(node)

def pop(list1):

    a = list1[0]

    del list1[0]

    return a

def not\_empty(list1):

    if len(list1) != 0:

        return True

    else:

        return False

# PrintPath

def printpath(node, iniState):

    list3 = []

    while (node != iniState):

        list3.append(node)

        node = node.parent

    reversed\_list = [list3[-(i + 1)] for i in range(len(list3))]

    print('The path :\n ')

    for i in range(len(reversed\_list)):

        print('Action No:', i + 1, reversed\_list[i])

    print('\nThe Cost :', len(reversed\_list))

def can\_add\_to\_openlist(openList, successor):

    for node in openList:

        if successor == node and successor.f >= node.f:

            return False

    return True

def EightPuzzle(initialState, goalState):

    iniState = Node(initialState)

    iniState.generateValue(goalState)

    openList = []

    closedList = []

    find = 1

    openList.append(iniState)

    while (not\_empty(openList)):

        openList.sort()

        currentNode = pop(openList)

        #         print(type(currentNode))

        closedList.append(currentNode)

        if currentNode.position == goalState:

            find = 1

            printpath(currentNode, iniState)

            break

        else:

            successors = currentNode.possibleMoves()

        for succ in successors:

            if succ in closedList:

                continue

            else:

                succ.generateValue(goalState)

                if can\_add\_to\_openlist(openList, succ):

                    openList.append(succ)

    if find == 1:

        print("Solution Found...")

    else:

        print("Solution UnFound....")

if \_\_name\_\_ == '\_\_main\_\_':

    print("20012011169 Malay Thakkar")

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

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

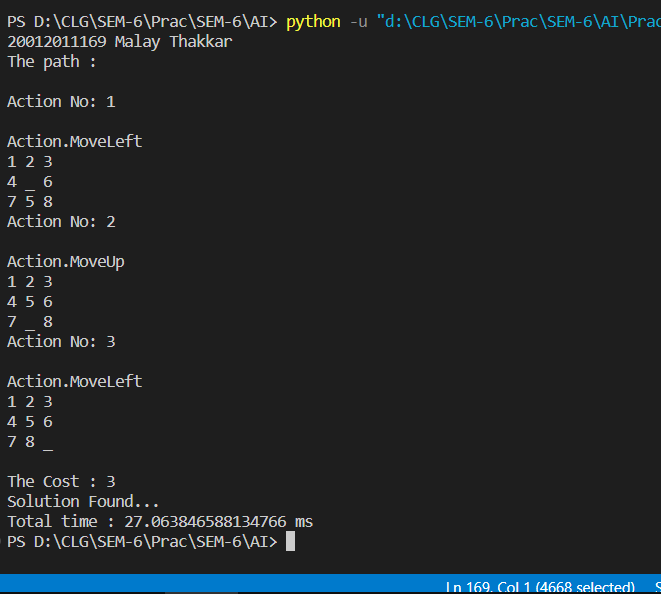
    startTime = time.time()

    EightPuzzle(initialState, goalState)

    endTime = time.time()

    print("Total time :", (endTime - startTime) \* 1000, "ms")

**OUTPUT:**

****

**Practical-5**

**Aim: Write a program to solve N-Queen problem using the A\* search**

**algorithm with Priority Queue and also find Execution time,**

**completeness of algorithm, etc.**

**CODE:**

import time

import queue

import random

import numpy as np

import matplotlib.pyplot as plt

from heapq import heappush, heappop, heapify

N = int(input("Enter the number of Queen You want to place: "))

a = queue.Queue()

class PriorityQueue:

    def \_\_init\_\_(self):

        self.pq = []

    def add(self, item):

        heappush(self.pq, item)

    def poll(self):

        return heappop(self.pq)

    def peek(self):

        return self.pq[0]

    def remove(self, item):

        value = self.pq.remove(item)

        heapify(self.pq)

        return value is not None

    def \_\_len\_\_(self):

        return len(self.pq)

class queen:

    def \_\_init\_\_(self):

        self.row = -1

        self.col = -1

    def \_\_cmp\_\_(self, other):

        return self.row == other.row and self.cok == other.col

    def \_\_eq\_\_(self, other):

        return self.\_\_cmp\_\_(other)

    def \_\_hash\_\_(self):

        return hash(str(self.list\_()))

    def list\_(self):

        return [self.row,self.col]

class state:

    def \_\_init\_\_(self, data):

        self.nQueen = [queen() for i in range(N)]

        if(data != None):

            self.moves = data.moves + 1

            self.heuristicVal = data.heuristicVal

            for i in range(N):

                self.nQueen[i].row = data.nQueen[i].row

                self.nQueen[i].col = data.nQueen[i].col

        else:

            self.moves = 0

            self.initQueens()

        self.parent = data

    def getConflictCount(self,row,col):

        count = 0

        conflictCount = 0

        ConflictSet = []

        for i in range(N):

            if(self.nQueen[i].row == row):

                count+=1

                ConflictSet.append(self.nQueen[i])

        for i in range(N):

            if(self.nQueen[i].col == col):

                count+=1

                ConflictSet.append(self.nQueen[i])

        for i in range(N):

            if(abs(self.nQueen[i].row - row) == abs(self.nQueen[i].col -col)):

                count+=1

                ConflictSet.append(self.nQueen[i])

        for obj in ConflictSet:

            if(not(obj.row == row and obj.col == col)):

                conflictCount+=1

        return conflictCount

    def placeQueen(self,row,col):

        if(row >= N or col >= N):

            return

        if(self.nQueen[col].row == row and self.nQueen[col].col == col):

            return

        self.nQueen[col].row = row

        self.nQueen[col].col = col

        self.heuristicVal = self.getHeuristicCost()

    def printQueen(self):

        for i in range(N):

            for j in range(N):

                if(self.nQueen[j].row == i):

                    print("1", end=" ")

                else:

                    print("0", end=" ")

            print()

        print()

    def drawQueens(self):

        board = self.getMatrix()

        matrix = np.zeros ((N, N))

        matrix = matrix.astype(str)

        for i in range(N):

            for j in range (N):

                if board[i][j] == 1:

                    matrix[i][j] = 'Q'

                else:

                    matrix[i][j] =' '

        w = 5

        h = 5

        plt.figure(1, figsize=(w, h))

        tb = plt.table(cellText=matrix, loc=(0, 0), cellLoc='center')

        for i in range(N):

            for j in range(N):

                if board[i][j] ==1:

                    tb.\_cells[(i, j)].\_text.set\_color('#960018')

                    tb.\_cells[(i, j)].\_text.set\_weight('extra bold')

                if ((i + j) % 2) == 0:

                    tb.\_cells[(i, j)].set\_facecolor('#CD853F')

                else:

                    tb.\_cells[(i, j)].set\_facecolor('#FADFAD')

                tb.\_cells[(i, j)].set\_height(1.0 / N)

                tb.\_cells[(i, j)].set\_width(1.0 / N)

        ax = plt.gca()

        ax.set\_xticks([])

        ax.set\_yticks([])

        plt.show()

    def getMatrix(self):

        board = np.zeros((N, N))

        board.astype(int)

        for j in range(N):

            for i in range(N):

                if(self.nQueen[i].row == j):

                    board[i][j] = 1

                else:

                    board[i][j] = 0

        return board

    def initQueens(self):

        for col in range(N):

            row = random.randint(0,N-1)

            self.placeQueen(row, col)

        self.moves = 0

        self.heuristicVal = self.getHeuristicCost()

    def getHeuristicCost(self):

        count = 0

        for i in range(N):

            count = count + self.getConflictCount(self.nQueen[i].row, self.nQueen[i].col)

        return count

    def score(self):

        return self.\_h() + self.\_g()

    def \_h(self):

        return self.heuristicVal

    def \_g(self):

        return self.moves

    def \_\_cmp\_\_(self, other):

        if(other == None):

            return False

        return self.nQueen == other.nQueen

    def \_\_eq\_\_(self, other):

        return self.\_\_cmp\_\_(other)

    def \_\_hash\_\_(self):

        return hash(str(self.nQueen))

    def \_\_lt\_\_(self, other):

        return self.score() < other.score()

    def nextAllState(self):

        list1 = []

        row = self.moves

        for i in range(N):

            if(not(self.nQueen[i].row == row and self.nQueen[i].col == i)):

                nextState = state(self)

                nextState.placeQueen(row, i)

                list1.append(nextState)

        return list1

def solve(initial\_state):

    openset = PriorityQueue()

    openset.add(initial\_state)

    closed = set()

    moves = 0

    print("Trying to solve:")

    print(openset.peek().printQueen(),'\n\n')

    start = time.time()

    while openset:

        current = openset.poll()

        if current.heuristicVal == 0:

            end = time.time()

            print('I found a solution')

            current.printQueen()

            current.drawQueens()

            print('I found the solution in %2.f milliseconds'% float((end - start)\*1000))

            break

        moves += 1

        for state in current.nextAllState():

            if state not in closed:

                openset.add(state)

        closed.add(current)

    else:

        print('I couldn''t solve it!')

def main():

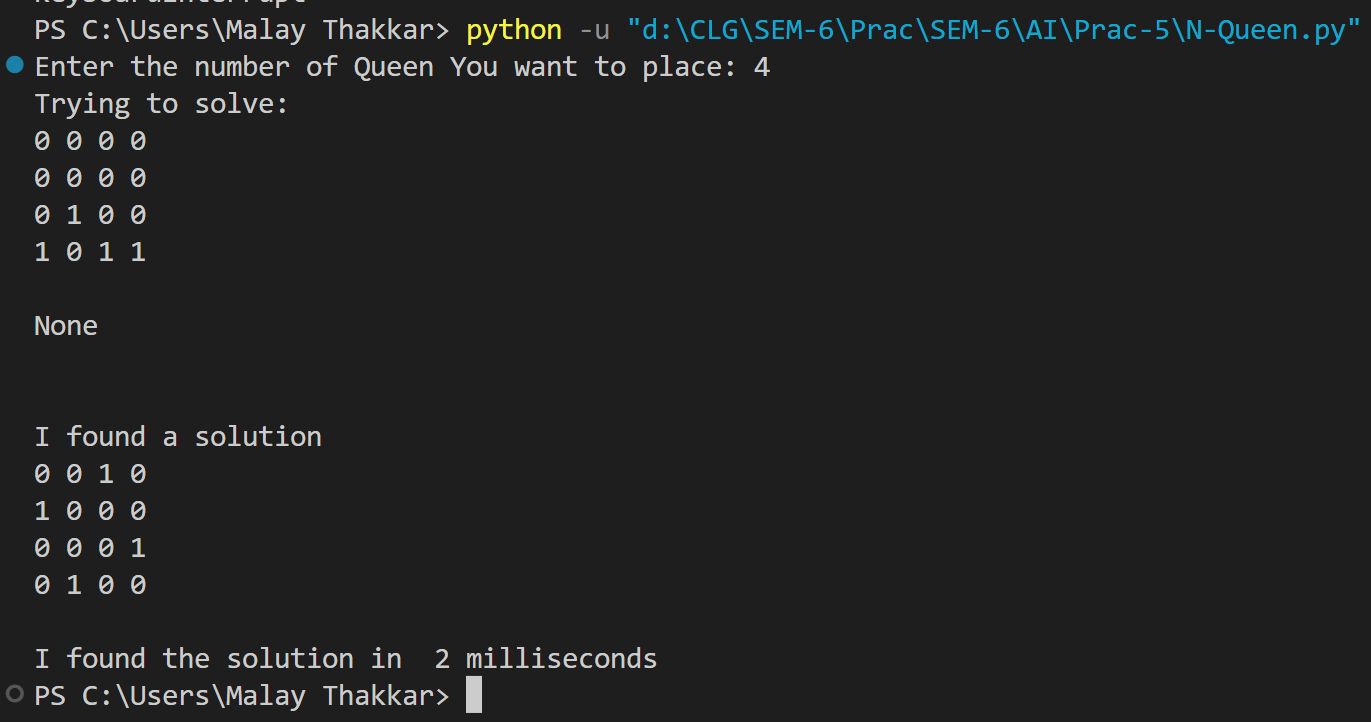
    initial\_state = state(None)

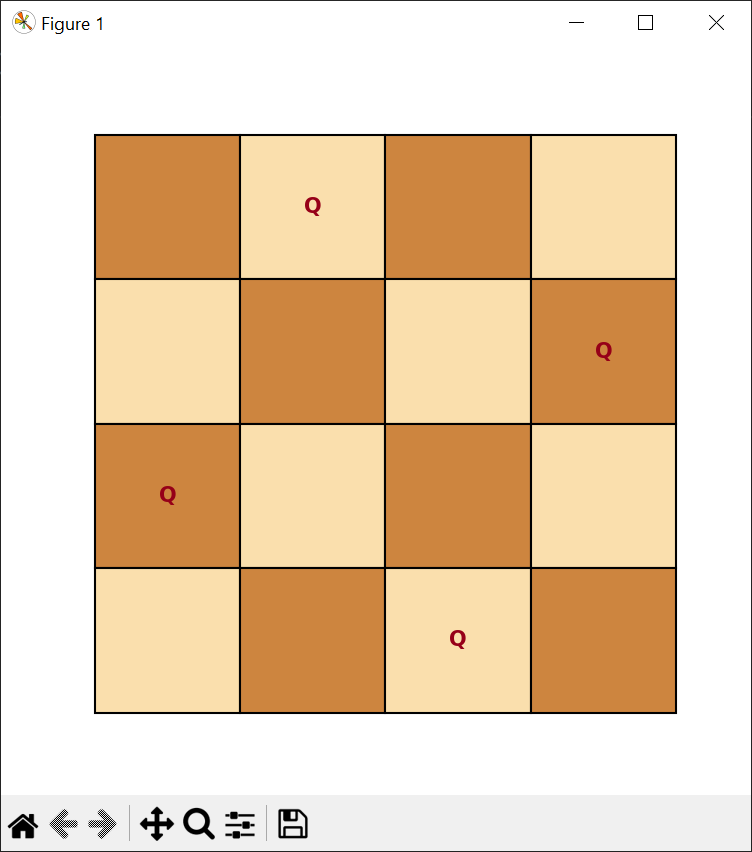
    solve(initial\_state)

if \_\_name\_\_ == '\_\_main\_\_':

    main()

**OUTPUT:**

****



**Practical-6**

**AIM: Write a program to create tic-tac-toe game using the alpha-**

**beta algorithm.**

* **CODE:**

from math import inf as infinity

from random import choice

import platform

import time

from os import system

import numpy as np

import matplotlib.pyplot as plt

from matplotlib.colors import ListedColormap

player = -1

computer = +1

steps = 1

turn = ""

status = "RUNNING..."

class stateNode:

    def \_\_init\_\_(self):

        self.board = [

            [0, 0, 0],

            [0, 0, 0],

            [0, 0, 0],

        ]

    def evaluate(self):

        if self.wins(computer):

            score = +1

        elif self.wins(player):

            score = -1

        else:

            score = 0

        return score

    def game\_over(self):

        return self.wins(player) or self.wins(computer)

    def empty\_cells(self):

        cells = []

        for x, row in enumerate(self.board):

            for y, cell in enumerate(row):

                if cell == 0:

                    cells.append([x, y])

        return cells

    def valid\_move(self, x, y):

        if [x, y] in self.empty\_cells():

            return True

        else:

            return False

    def set\_move(self, x, y, player):

        if self.valid\_move(x, y):

            self.board[x][y] = player

            return True

        else:

            return False

    def wins(self, player):

        state = self.board

        win\_state = [

            [state[0][0], state[0][1], state[0][2]],

            [state[1][0], state[1][1], state[1][2]],

            [state[2][0], state[2][1], state[2][2]],

            [state[0][0], state[1][0], state[2][0]],

            [state[0][1], state[1][1], state[2][1]],

            [state[0][2], state[1][2], state[2][2]],

            [state[0][0], state[1][1], state[2][2]],

            [state[2][0], state[1][1], state[0][2]],

        ]

        if [player, player, player] in win\_state:

            return True

        else:

            return False

    def render(self, computer\_choice, player\_choice):

        global steps, turn, status

        chars = {-1: player\_choice, +1: computer\_choice, 0: ""}

        str\_line = "---------------"

        print("\n" + str\_line)

        for row in self.board:

            for cell in row:

                symbol = chars[cell]

                print(f"| {symbol} |", end="")

            print("\n" + str\_line)

        arr = np.zeros((3, 3), dtype=int)

        arr[1::2, 0::2] = 1

        arr[0::2, 1::2] = 1

        image = arr.reshape((3, 3))

        colors = ["blue", "yellow", "red", "green", "k", "#550011", "black", "orange"]

        cmap = ListedColormap(colors)

        plt.matshow(image, cmap=cmap)

        i, j = 0, 0

        for row in self.board:

            j = 0

            for cell in row:

                symbol = chars[cell]

                plt.text(

                    j,

                    i,

                    symbol,

                    va="center",

                    ha="center",

                    color="blue" if (i - j) % 2 == 0 else "green",

                    fontsize=30,

                )

                j += 1

            i += 1

        plt.xlabel(

            "step no.:-[{}] Turn:-[{}]\nchoice:- player[{}] computer[{}]".format(

                steps, turn, player\_choice, computer\_choice

            )

        )

        plt.ylabel("Status:- {}".format(status))

        plt.show()

        steps += 1

def minimax(state, depth, player):

    if player == computer:

        best = [-1, -1, -infinity]

    else:

        best = [-1, -1, +infinity]

    if depth == 0 or state.game\_over():

        score = state.evaluate()

        return [-1, -1, score]

    for cell in state.empty\_cells():

        x, y = cell[0], cell[1]

        state.board[x][y] = player

        score = minimax(state, depth - 1, -player)

        state.board[x][y] = 0

        score[0], score[1] = x, y

        if player == computer:

            if score[2] > best[2]:

                best = score  # max value

        else:

            if score[2] < best[2]:

                best = score  # min value

    return best

def clean():

    os\_name = platform.system().lower()

    if "windows" in os\_name:

        system("cls")

    else:

        system("clear")

def computer\_turn(state, computer\_choice, player\_choice):

    global turn

    turn = "COMPUTER"

    depth = len(state.empty\_cells())

    if depth == 0 or state.game\_over():

        return

    # clean()

    print(f"Computer turn[{computer\_choice}]")

    state.render(computer\_choice, player\_choice)

    if depth == 9:

        x = choice([0, 1, 2])

        y = choice([0, 1, 2])

    else:

        move = minimax(state, depth, computer)

        x, y = move[0], move[1]

    state.set\_move(x, y, computer)

    time.sleep(1)

def player\_turn(state, computer\_choice, player\_choice):

    global turn

    turn = "player"

    depth = len(state.empty\_cells())

    if depth == 0 or state.game\_over():

        return

    move = -1

    moves = {

        1: [0, 0],

        2: [0, 1],

        3: [0, 2],

        4: [1, 0],

        5: [1, 1],

        6: [1, 2],

        7: [2, 0],

        8: [2, 1],

        9: [2, 2],

    }

    print(f"player turn [{player\_choice}]")

    state.render(computer\_choice, player\_choice)

    while move < 1 or move > 9:

        try:

            move = int(input("Enter Any Number (1..9): "))

            coord = moves[move]

            can\_move = state.set\_move(coord[0], coord[1], player)

            if not can\_move:

                print("Bad move")

                move = -1

        except (EOFError, KeyboardInterrupt):

            print("Bye")

            exit()

        except (KeyError, ValueError):

            print("Bad choice")

def main():

    player\_choice = ""

    computer\_choice = ""

    first = ""

    state = stateNode()

    while player\_choice != "O" and player\_choice != "X":

        try:

            player\_choice = input("::Choose 'X' or 'O'::\nYour Choice: ").upper()

            print("")

        except (EOFError, KeyboardInterrupt):

            print("Program End")

            exit()

        except (KeyError, ValueError):

            print("Bad choice")

        if player\_choice == "X":

            computer\_choice = "O"

        else:

            computer\_choice = "X"

    while first != "Y" and first != "N":

        try:

            first = input("Do you want to start first? [Y/N]: ").upper()

        except (EOFError, KeyboardInterrupt):

            print("Program End")

            exit()

        except (KeyError, ValueError):

            print("Bad choice")

    while len(state.empty\_cells()) > 0 and not state.game\_over():

        if first == "N":

            computer\_turn(state, computer\_choice, player\_choice)

            first = ""

        player\_turn(state, computer\_choice, player\_choice)

        computer\_turn(state, computer\_choice, player\_choice)

    global status

    if state.wins(player):

        print(f"player turn [{player\_choice}]")

        status = "player WINS!"

        state.render(computer\_choice, player\_choice)

        print(status)

    elif state.wins(computer):

        print(f"Computer turn [{computer\_choice}]")

        status = "COMPUTER WINS"

        state.render(computer\_choice, player\_choice)

        print(status)

    else:

        status = "DRAW!"

        state.render(computer\_choice, player\_choice)

        print(status)

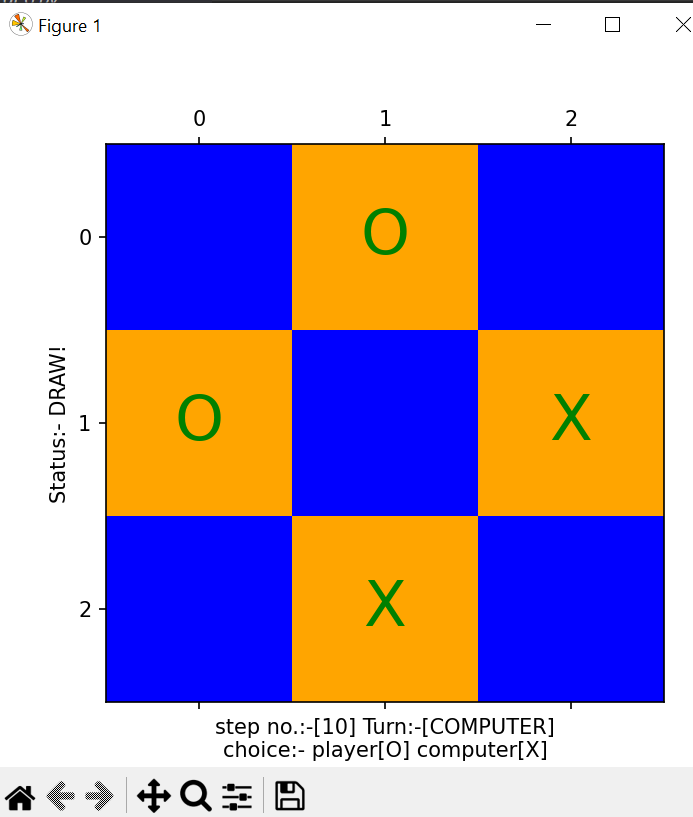
    exit()

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

    main()

* **OUTPUT:**

|  |  |
| --- | --- |
|  |  |

****

**Practical-7**

**Aim: Write a program to build Multi-layer Perceptron to implement any Boolean functions as mentioned below without using any python packages.**

* **CODE:**

from numpy import dot

class perceptronNeuron:

    def \_\_init\_\_(self, x, w, w0):

        l = [ww for ww in x]

        l.insert(0, 1)

        self.x = l

        self.y = 0

        l = [ww for ww in w]

        l.insert(0, w0)

        self.w = l

        #print("L:"+str(l)+", w:"+str(w)+"self.w:"+str(self.w))

    def \_\_repr\_\_(self):

        return "Input:"+str(self.x)+", Weight:"+str(self.w)

    def activationFunction(self):

        self.y = 1 if dot(self.x, self.w) >= 0 else 0

        return self.y

    def dot(x, W):

        if len(x) != len(W):

            return 0

        return sum(i[0] \* i[1] for i in zip(X, W))

class multiLayerPerceptron:

    def \_\_init\_\_(self, a0, a1, dimension, inputBias, weight, functionName, s0, s1):

        self.n = dimension

        self.a0 = a0

        self.a1 = a1

        self.inputBias = inputBias

        self.hidden = []

        self.weight = weight

        self.funcName = functionName

        self.s0 = s0

        self.s1 = s1

    def binaryCombinations(self, a0, a1, n):

        list1 = []

        for i in range(1 << n):

            s = bin(i)[2:]

            s = '0'\*(n-len(s))+s

            l = list(map(int, list(s)))

            l = [a0 if item == 0 else a1 for item in l]

            list1.append(l)

        return list1

    def generateHiddenLayer(self, input):

        allPossibleList = self.binaryCombinations(self.a0, self.a1, self.n)

        self.hidden = [perceptronNeuron(input, weight, self.inputBias)

                       for weight in allPossibleList]

        return self.hidden

    def outputActivationFun(self, hiddenLayer):

        return self.a1 if dot(hiddenLayer, self.weight) >= 0 else self.a0

    def generateOutput(self, xStr):

        allPossibleInputs = self.binaryCombinations(self.a0, self.a1, self.n)

        output = []

        strheader = "\n"+self.funcName+"\n"

        for i in range(self.n):

            strheader += (xStr+str(i+1)+"\t")

        strheader += ("Output")

        print(strheader)

        for input in allPossibleInputs:

            allhiddenOutput = self.generateHiddenLayer(input)

            # print(allhiddenOutput)

            o = [hiddenPerceptron.activationFunction()

                 for hiddenPerceptron in allhiddenOutput]

            o.insert(0, 1)

            # print(str(o))

            o = self.outputActivationFun(o)

            output.append(o)

            print(str([self.s0 if item == self.a0 else self.s1 for item in input]).replace("[", "").replace(

                "]", "").replace(",", "\t").replace("'", "")+"\t"+str(self.s0 if o == self.a0 else self.s1))

# return output

true = 1

false = -1

initialBias = -2

outputBias = -1

dimension = 2

s0 = "0"

s1 = "1"

xStr = "x"

andMLP = multiLayerPerceptron(false, true, dimension, initialBias, [

                              outputBias, false, false, false, true], "AND Function", s0, s1)

orMLP = multiLayerPerceptron(false, true, dimension, initialBias, [

                             outputBias, false, true, true, true], "OR Function", s0, s1)

xorMLP = multiLayerPerceptron(false, true, dimension, initialBias, [

                              outputBias, false, true, true, false], "XOR Function", s0, s1)

xnorMLP = multiLayerPerceptron(false, true, dimension, initialBias, [

                               outputBias, true, false, false, true], "XNOR Function", s0, s1)

norMLP = multiLayerPerceptron(false, true, dimension, initialBias, [

                              outputBias, true, false, false, false], "NOR Function", s0, s1)

nandMLP = multiLayerPerceptron(false, true, dimension, initialBias, [

                               outputBias, true, true, true, false], "NAND Function", s0, s1)

notInput1MLP = multiLayerPerceptron(false, true, dimension, initialBias, [

                                    outputBias, true, true, false, false], "Not"+xStr+"1 Function", s0, s1)

notInput2MLP = multiLayerPerceptron(false, true, dimension, initialBias, [

                                    outputBias, true, false, true, false], "Not"+xStr+"2 Function", s0, s1)

nullMLP = multiLayerPerceptron(false, true, dimension, initialBias, [

                               outputBias, false, false, false, false], "NULL Function", s0, s1)

identityMLP = multiLayerPerceptron(false, true, dimension, initialBias, [

                                   outputBias, true, true, true, true], "Identity Function", s0, s1)

inhibition1MLP = multiLayerPerceptron(false, true, dimension, initialBias, [

                                      outputBias, false, false, true, false], "Inhibition x1^~x2 Function", s0, s1)

inhibition2MLP = multiLayerPerceptron(false, true, dimension, initialBias, [

                                      outputBias, false, true, false, false], "Inhibition x2^~x1 Function", s0, s1)

transferX1MLP = multiLayerPerceptron(false, true, dimension, initialBias, [

                                     outputBias, false, false, true, true], "Transfer x1 Function", s0, s1)

transferX2MLP = multiLayerPerceptron(false, true, dimension, initialBias, [

                                     outputBias, false, true, false, true], "Transfer x2 Function", s0, s1)

implication1MLP = multiLayerPerceptron(false, true, dimension, initialBias, [

                                       outputBias, true, false, true, true], "Implication x1V~x2 Function", s0, s1)

implication2MLP = multiLayerPerceptron(false, true, dimension, initialBias, [

                                       outputBias, true, true, false, true], "Implication x2V~x1 Function", s0, s1)

andMLP.generateOutput(xStr)

orMLP.generateOutput(xStr)

xorMLP.generateOutput(xStr)

norMLP.generateOutput(xStr)

xnorMLP.generateOutput(xStr)

nandMLP.generateOutput(xStr)

notInput1MLP.generateOutput(xStr)

notInput2MLP.generateOutput(xStr)

nullMLP.generateOutput(xStr)

identityMLP.generateOutput(xStr)

inhibition1MLP.generateOutput(xStr)

inhibition2MLP.generateOutput(xStr)

transferX1MLP.generateOutput(xStr)

transferX2MLP.generateOutput(xStr)

implication1MLP.generateOutput(xStr)

implication2MLP.generateOutput(xStr)

* **OUTPUT:**

|  |  |
| --- | --- |
|  |  |