18CSE365J   ARTIFICIAL INTELLIGENCE

<u>SEMESTER VI</u>

DEPARTMENT OF DATA SCIENCE AND BUSINESS SYSTEMS

NAME - AYSHWARYA S

REG NO - RA1911042010048

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

(Under SECTION 3 of the UGC Act, 1956)

S.R.M NAGAR, KATTANKULATHUR – 603203

KANCHEEPURAM DISTRICT

FACULTY OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

(Under SECTION 3 of the UGC Act, 1956)

S.R.M NAGAR, KATTANKULATHUR – 603203

KANCHEEPURAM DISTRICT

BONAFIDE CERTIFICATE

Register No RA1911042010048

Certified to be the bonafide record of work done by AYSHWARYA S of CSBS-R1 B. Tech Degree course in the practical of 18CSE365J ARTIFICIAL INTELLIGENCE in **SRM INSTITUTE OF SCIENCE AND TECHNOLOGY, KATTANKULATHUR** during the academic year 2022

Lab Incharge

Date: **Head of the Department**

Submitted for the University examination held on _____________ at SRM Institute of Science and Technology, Kattankulathur, Chennai - 603203.

Date: ---------------- ------------------

Examiner - 1 **Examiner - 2**

INDEX

Date: **Expt.No: 01**	**IMPLEMENTATION OF TIC-TAC-TOE PROBLEM**

AIM:-

To implement toy problem using python.

Algorithm:-

Step 1: Now our board looks like 000 000 000 (tenary number) convert it into decimal no.The decimal no is 0

Step 2: Use the computed number ie 0 as an index into the move table and access the vector stored in

Step 3: New Board Position.

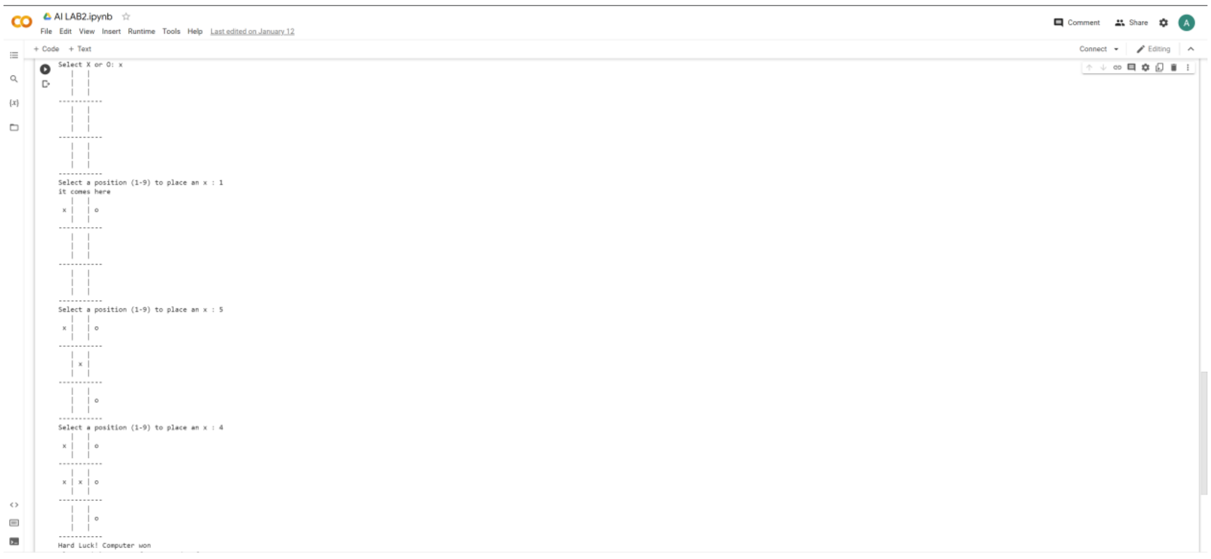
Step 4: The new board position is 000 010 000

Step 5: The vector selected in step 2(000 010 000) represents the way the board will look after the move that should be made. So set board equal to that vector.

Code:-

```
import random
def select_letter():
    let=""
    auto_let=""
    while(let != "x" and let != "o"):
        let=input("Select X or O: ").replace(" ","").strip().lower()
        if let == "x":
            auto_let="o"
        else:
            auto_let="x"
    return let, auto_let
def clean_board():
    brd=[" " for x in range(10)]
    return brd

def is_board_full(board):
    return board.count(" ")==0

def insert_letter(board,letter,pos):
    board[pos]=letter

def computer_move(board,letter):
    computer_letter=letter
    possible_moves=[]
    available_corners=[]
    available_edges=[]
    available_center=[]
    position=-1

    for i in range(1,len(board)):
        if board[i] ==" ":
            possible_moves.append(i)
    for let in ["x","o"]:
        for i in possible_moves:
            board_copy=board[:]
            board_copy[i] = let
            if is_winner(board_copy,let):
                position=i

    if position == -1:
        for i in range(len(board)):
            if board[i]==" ":
                if i in [1,3,7,9]:
                    available_corners.append(i)
                if i is 5:
                    available_center.append(i)
                if i in [2,4,6,8]:
                    available_edges.append(i)
        if len(available_corners)>0:
            print("it comes here")
            position=random.choice(available_corners)
        elif len(available_center)>0:
            position=available_center[0]
        elif len(available_edges)>0:
            position=random.choice(available_edges)
    board[position]=computer_letter
def draw_board(board):
    board[0]=-1
    print("   |   |   ")
    print(" "+board[1]+" | "+board[2]+" | "+board[3]+" ")
    print("   |   |   ")
    print("-"*11)
    print("   |   |   ")
    print(" "+board[4]+" | "+board[5]+" | "+board[6]+" ")
    print("   |   |   ")
    print("-"*11)
    print("   |   |   ")
    print(" "+board[7]+" | "+board[8]+" | "+board[9]+" ")
    print("   |   |   ")
    print("-"*11)
    return board
def is_winner(board,letter):
    return (board[1] == letter and board[2] == letter and board[3] == letter) or \
    (board[4] == letter and board[5] == letter and board[6] == letter) or \
    (board[7] == letter and board[8] == letter and board[9] == letter) or \
    (board[1] == letter and board[4] == letter and board[7] == letter) or \
    (board[2] == letter and board[5] == letter and board[8] == letter) or \
    (board[3] == letter and board[6] == letter and board[9] == letter) or \
    (board[1] == letter and board[5] == letter and board[9] == letter) or \
    (board[3] == letter and board[5] == letter and board[7] == letter)

def repeat_game():

    repeat=input("Play again? Press y for yes and n for no: ")
    while repeat != "n" and repeat != "y":
        repeat=input("PLEASE, press y for yes and n for no: ")
    return repeat

def play_game():

    letter, auto_letter= select_letter()
    board=clean_board()
    board=draw_board(board)
    while is_board_full(board) == False:
        try:
            position=int(input("Select a position (1-9) to place an "+letter+" : " ))

        except:
            position=int(input("PLEASE enter position using only NUMBERS from 1-9: "))

        if position not in range(1,10):
            position=int(input("Please, choose another position to place an "+letter+" from 1 to 9 :"))

        if board[position] != " ":
            position=int(input("Please, choose an empty position to place an "+letter+" from 1 to 9: "))

        insert_letter(board,letter,position)
        computer_move(board,auto_letter)
        board=draw_board(board)

        if is_winner(board,letter):
            print("Congratulations! You Won.")
            return repeat_game()
        elif is_winner(board,auto_letter):
            print("Hard Luck! Computer won")
            return repeat_game()
    if is_board_full(board):
        print("Tie Game :)")
        return repeat_game()
print("Welcome to Tic Tac Toe.")
repeat="y"
while(repeat=="y"):
    repeat=play_game()
```

Output:-



```
Select X or O: x
   |   |
   |   |
   |   |
-----------
   |   |
   |   |
   |   |
-----------
   |   |
   |   |
   |   |
-----------
Select a position (1-9) to place an x : 1
it comes here
   |   |
 x |   | o
   |   |
-----------
   |   |
   |   |
   |   |
-----------
   |   |
   |   |
   |   |
-----------
Select a position (1-9) to place an x : 5
   |   |
 x |   | o
   |   |
-----------
   |   |
   | x |
   |   |
-----------
   |   |
   |   | o
   |   |
-----------
Select a position (1-9) to place an x : 4
   |   |
 x |   | o
   |   |
-----------
   |   |
 x | x | o
   |   |
-----------
   |   |
   |   | o
   |   |
-----------
Hard Luck! Computer won
```

Results:-

We have successfully implemented the toy problem.

Date: **Expt.No: 02**	**IMPLEMENTATION OF 8 PUZZLE PROBLEM**

AIM:-

To implement 8-puzzle problem using python.

Algorithm:-

Step 1: Randomly move or alter the state

Step 2: Assess the energy of the new state using an objective function

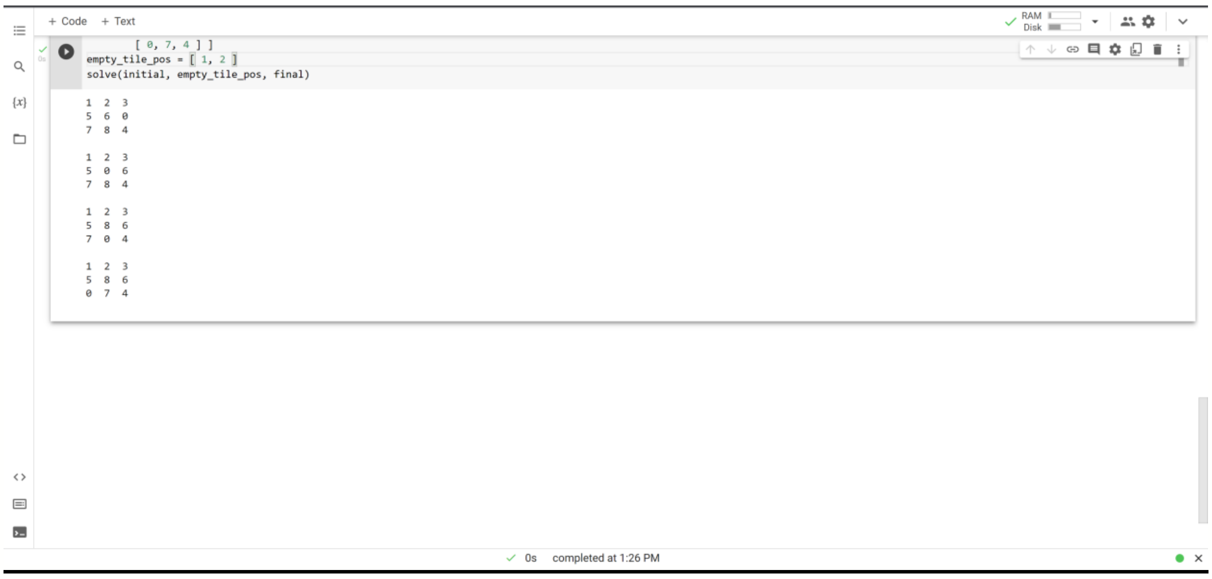
Step 3: Compare the energy to the previous state and decide whether to accept the new solution or reject it based on the current temperature.

Step 4: Repeat until you have converged on an acceptable answer

Code:-

```
import copy
from heapq import heappush, heappop
n = 3
row = [ 1, 0, -1, 0 ]
col = [ 0, -1, 0, 1 ]
class priorityQueue:
   def __init__(self):
      self.heap = []
   def push(self, k):
      heappush(self.heap, k)
   def pop(self):
      return heappop(self.heap)
   def empty(self):
      if not self.heap:
         return True
      else:
         return False
class node:

   def __init__(self, parent, mat, empty_tile_pos,
            cost, level):
      self.parent = parent
      self.mat = mat
      self.empty_tile_pos = empty_tile_pos
      self.cost = cost
      self.level = level
   def __lt__(self, nxt):
      return self.cost < nxt.cost
def calculateCost(mat, final) -> int:

   count = 0
   for i in range(n):
        for j in range(n):
            if ((mat[i][j]) and
                (mat[i][j] != final[i][j])):
                count += 1

    return count

def newNode(mat, empty_tile_pos, new_empty_tile_pos,
            level, parent, final) -> node:
    new_mat = copy.deepcopy(mat)
    x1 = empty_tile_pos[0]
    y1 = empty_tile_pos[1]
    x2 = new_empty_tile_pos[0]
    y2 = new_empty_tile_pos[1]
    new_mat[x1][y1], new_mat[x2][y2] = new_mat[x2][y2], new_mat[x1][y1]
    cost = calculateCost(new_mat, final)

    new_node = node(parent, new_mat, new_empty_tile_pos,
                    cost, level)
    return new_node
def printMatrix(mat):

    for i in range(n):
        for j in range(n):
            print("%d " % (mat[i][j]), end = " ")

        print()
def isSafe(x, y):

    return x >= 0 and x < n and y >= 0 and y < n
def printPath(root):

    if root == None:
        return

    printPath(root.parent)
    printMatrix(root.mat)
    print()
def solve(initial, empty_tile_pos, final):
    pq = priorityQueue()
    cost = calculateCost(initial, final)
    root = node(None, initial,empty_tile_pos, cost, 0)
    pq.push(root)
    while not pq.empty():
        minimum = pq.pop()
        if minimum.cost == 0:
            printPath(minimum)
            return
        for i in range(n):
            new_tile_pos = [
                minimum.empty_tile_pos[0] + row[i],
                minimum.empty_tile_pos[1] + col[i], ]

            if isSafe(new_tile_pos[0], new_tile_pos[1]):
                child = newNode(minimum.mat,
                                minimum.empty_tile_pos,
                                new_tile_pos,
                                minimum.level + 1,
                                minimum, final,)
                pq.push(child)
initial = [ [ 1, 2, 3 ],
            [ 5, 6, 0 ],
            [ 7, 8, 4 ] ]
final = [ [ 1, 2, 3 ],
        [ 5, 8, 6 ],
        [ 0, 7, 4 ] ]
empty_tile_pos = [ 1, 2 ]
solve(initial, empty_tile_pos, final)
```

OUTPUT:-



```
      [ 0, 7, 4 ] ]
empty_tile_pos = [ 1, 2 ]
solve(initial, empty_tile_pos, final)

1  2  3
5  6  0
7  8  4

1  2  3
5  0  6
7  8  4

1  2  3
5  8  6
7  0  4

1  2  3
5  8  6
0  7  4
```

✓ 0s completed at 1:26 PM

RESULT:-
We have successfully implemented the 8-puzzle problem.

Date: Expt.No: 03	DEVELOPING AGENT PROBLEMS FOR REAL WORLD PROBLEMS

AIM:-

To implement developing agent programs for real world problems using python.

Algorithm:-

Step 1: Enter LOCATION A/B in captial letters where A and B are the two adjacent rooms respectively.

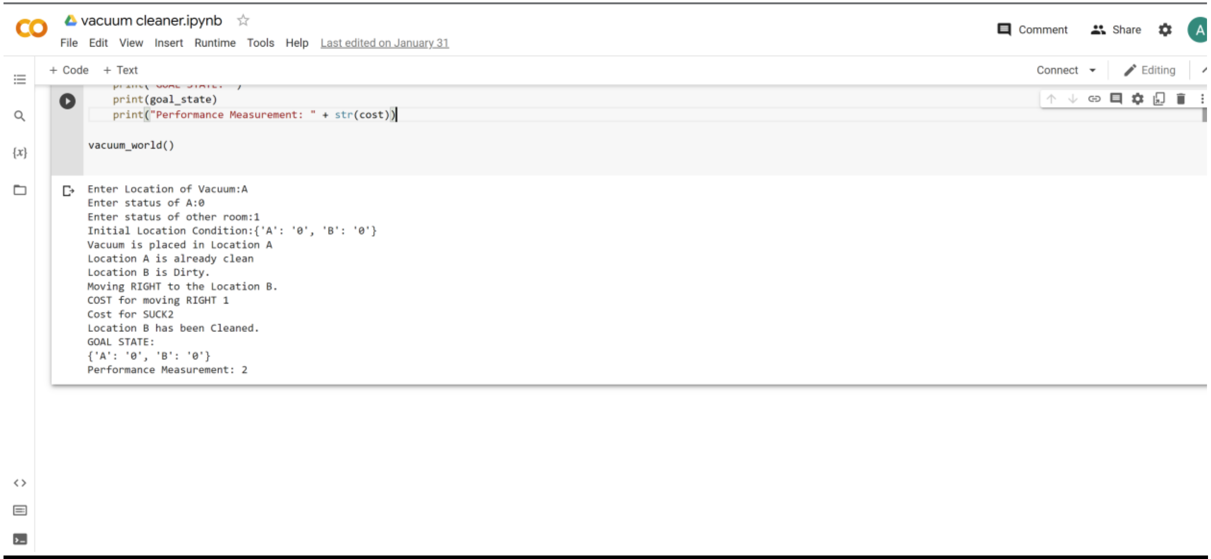
Step 2: Enter Status O/1 accordingly where 0 means CLEAN and 1 means DIRTY.

Step 3: Vacuum Cleaner senses the status of the other room before performing any action, also known as Environment sensing.

Code:-

```
def vacuum_world():
    goal_state = {'A': '0', 'B': '0'}
    cost = 0

    location_input = input("Enter Location of Vacuum:")
    status_input = input("Enter status of " + location_input+":")
    status_input_complement = input("Enter status of other room:")
    print("Initial Location Condition:" + str(goal_state))

    if location_input == 'A':

        print("Vacuum is placed in Location A")
        if status_input == '1':
            print("Location A is Dirty.")

            goal_state['A'] = '0'
            cost += 1
            print("Cost for CLEANING A " + str(cost))
            print("Location A has been Cleaned.")

            if status_input_complement == '1':

                print("Location B is Dirty.")
                print("Moving right to the Location B. ")
                cost += 1
                print("COST for moving RIGHT" + str(cost))

                goal_state['B'] = '0'
                cost += 1
                print("COST for SUCK " + str(cost))
                print("Location B has been Cleaned. ")
            else:
                print("No action" + str(cost))
            print("Location B is already clean.")

        if status_input == '0':
            print("Location A is already clean ")
            if status_input_complement == '1':
                print("Location B is Dirty.")
                print("Moving RIGHT to the Location B. ")
                cost += 1
                print("COST for moving RIGHT " + str(cost))

                goal_state['B'] = '0'
                cost += 1
                print("Cost for SUCK" + str(cost))
                print("Location B has been Cleaned. ")
            else:
                print("No action " + str(cost))
                print(cost)

                print("Location B is already clean.")

    else:
        print("Vacuum is placed in location B")
        if status_input == '1':
            print("Location B is Dirty.")
            goal_state['B'] = '0'
            cost += 1
            print("COST for CLEANING " + str(cost))
            print("Location B has been Cleaned.")

            if status_input_complement == '1':

                print("Location A is Dirty.")
                print("Moving LEFT to the Location A. ")
                cost += 1
                print("COST for moving LEFT" + str(cost))

                goal_state['A'] = '0'
                cost += 1
                print("COST for SUCK " + str(cost))
                print("Location A has been Cleaned.")

        else:
            print(cost)
            print("Location B is already clean.")

            if status_input_complement == '1':
                print("Location A is Dirty.")
                print("Moving LEFT to the Location A. ")
                cost += 1
                print("COST for moving LEFT " + str(cost))
                goal_state['A'] = '0'
                cost += 1
                print("Cost for SUCK " + str(cost))
                print("Location A has been Cleaned. ")
            else:
                print("No action " + str(cost))
                print("Location A is already clean.")
    print("GOAL STATE: ")
    print(goal_state)
    print("Performance Measurement: " + str(cost))

vacuum_world()
```

<u>Output:-</u>



```
    print(goal_state)
    print("Performance Measurement: " + str(cost))

vacuum_world()
```

```
Enter Location of Vacuum:A
Enter status of A:0
Enter status of other room:1
Initial Location Condition:{'A': '0', 'B': '0'}
Vacuum is placed in Location A
Location A is already clean
Location B is Dirty.
Moving RIGHT to the Location B.
COST for moving RIGHT 1
Cost for SUCK2
Location B has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 2
```

<u>Results:-</u>

We have successfully implemented vacuum cleaner problem.

Date: **Expt.No: 04**	**IMPLEMENTATION AND ANALYSIS OF BFS FOR AN APPLICATION**

AIM:-

To implement analysis of bfs for an application using python.

Algorithm:-

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Enqueue the starting node A and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until QUEUE is empty

Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).

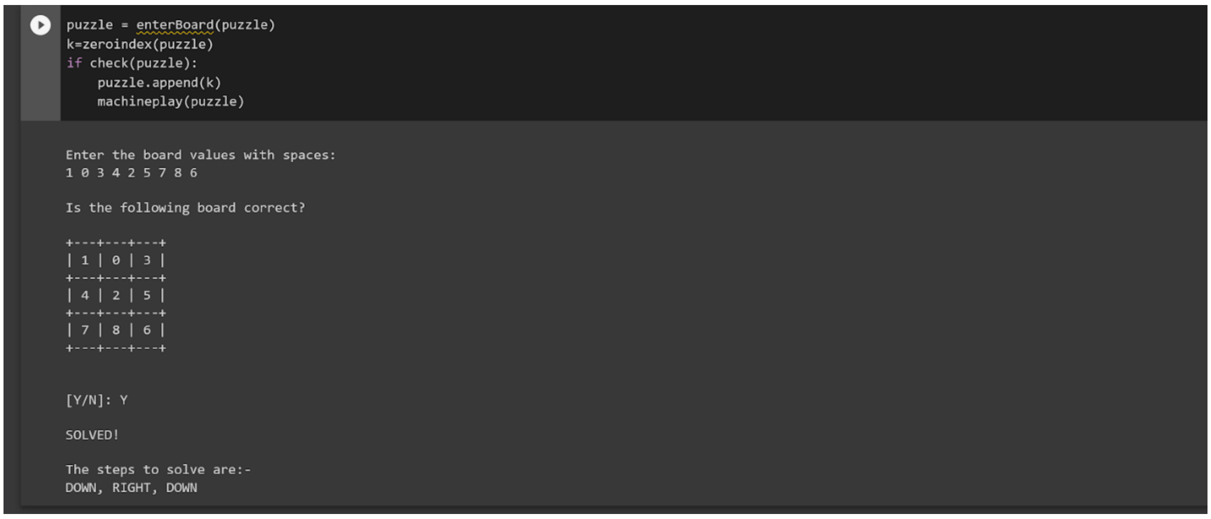
Step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state) [END OF LOOP]

Step 6: EXIT

Code:-

```
import time as t

puzzle=[]
solved=[1, 2, 3, 4, 5, 6, 7, 8, 0]
def zeroindex(puzzle):
    for i in range(9):
        if puzzle[i] == 0:
            return i
            break

def check(puzzle):
    count=0
    for i in range(9):
        for j in range(i+1, 9):
            if j==9:
                break
            if puzzle[i]>puzzle[j] and puzzle[i]!=0 and puzzle[j]!=0:
                count+=1
    if (not count%2):
        return True
    else:
        return False

def heuristic(puzzle):         #finds the heursitic value
    man_dist=sum(abs((val-1)%3 - i%3) + abs((val-1)//3 - i//3) for i, val in enumerate(puzzle) if val)
    return man_dist

def min_heuristics(lists):     #finds the minimum heuristic among a list of heurisitc values and returns i
t's position
    min=999999
    for i in range(len(lists)):
        if lists[i]<min:
            min=lists[i]
            index=i
    return(index)

def machineplay(puzzle):
    openlist=[]
    openLIST=[]
    closedlist=[]
    heuristicval=[]
    openlist.append(puzzle)
    x=[]
    x=openlist.pop(0)
    a=x[9]                                          #stores the index of 0
    while x[:9]!=solved:
        if a%3!=0:                                  #left
            statespace1=x.copy()
            temp=statespace1[a]
            statespace1[a]=statespace1[a-1]
            statespace1[a-1]=temp
            statespace1[9]=a-1
            statespace1.append("LEFT")

            if statespace1[:9] == solved:
                print("SOLVED!")
                print("The steps to solve are:- \n")
                print(", ".join(statespace1[10:]))
                break
            else:
                if statespace1[:9] not in closedlist and statespace1[:9] not in openLIST:
                    openlist.append(statespace1)    #for printing the steps
                    openLIST.append(statespace1[:9]) #to prevent loops
                    heuristicval.append(heuristic(statespace1[:9]))


        if a%3!=2:                                  #right
            statespace2=x.copy()
            temp=statespace2[a]
            statespace2[a]=statespace2[a+1]
            statespace2[a+1]=temp
            statespace2[9]=a+1
            statespace2.append("RIGHT")

            if statespace2[:9] == solved:
                print("SOLVED!")
                print("The steps to solve are:- \n")
                print(", ".join(statespace2[10:]))
                break
            else:
                if statespace2[:9] not in closedlist and statespace2[:9] not in openLIST:
                    openlist.append(statespace2)
                openLIST.append(statespace2[:9])
                heuristicval.append(heuristic(statespace2[:9]))

        if a!=0 and a!=1 and a!=2:                      #up
            statespace3=x.copy()
            temp=statespace3[a]
            statespace3[a]=statespace3[a-3]
            statespace3[a-3]=temp
            statespace3[9]=a-3
            statespace3.append("UP")

            if statespace3[:9] == solved:
                print("SOLVED!")
                print("The steps to solve are:- \n")
                print(", ".join(statespace3[10:]))
                break
            else:
                if statespace3[:9] not in closedlist and statespace3[:9] not in openLIST:
                    openlist.append(statespace3)
                    openLIST.append(statespace3[:9])
                    heuristicval.append(heuristic(statespace3[:9]))

        if a!=6 and a!=7 and a!=8:                      #down
            statespace4=x.copy()
            temp=statespace4[a]
            statespace4[a]=statespace4[a+3]
            statespace4[a+3]=temp
            statespace4[9]=a+3
            statespace4.append("DOWN")
            if statespace4[:9] == solved:
                print("\nSOLVED!")
                print("\nThe steps to solve are:- ")
                print(", ".join(statespace4[10:]))
                break
            else:
                if statespace4[:9] not in closedlist and statespace4[:9] not in openLIST:
                    openlist.append(statespace4)
                    openLIST.append(statespace4[:9])
                    heuristicval.append(heuristic(statespace4[:9]))
        closedlist.append(x[:9])
        y=min_heuristics(heuristicval)
        tem=heuristicval.pop(y)
        x=openlist.pop(y)
        a=x[9]

    # print("SOLVED!")
    # print("CLOSED LIST:", len(closedlist), "nodes")

def show_board(puzzle):
            print("""\n+---+---+---+
| {} | {} | {} |
+---+---+---+
| {} | {} | {} |
+---+---+---+
| {} | {} | {} |
+---+---+---
+""".format(puzzle[0], puzzle[1], puzzle[2], puzzle[3], puzzle[4], puzzle[5], puzzle[6], puzzle[7], puz
zle[8]))

def enterBoard(puzzle):
    hmm = "n"
    while(hmm=="n"):
        print("\nEnter the board values with spaces: ")
        puzzle = list(map(int, input().split()))
        print("\nIs the following board correct?")
        show_board(puzzle)
        print('\n')
        hmm = input("[Y/N]: ").lower()
    return puzzle

puzzle = enterBoard(puzzle)
k=zeroindex(puzzle)
if check(puzzle):
    puzzle.append(k)
    # start_time=t.time()
    machineplay(puzzle)
```

<u>Output:-</u>



```
puzzle = enterBoard(puzzle)
k=zeroindex(puzzle)
if check(puzzle):
    puzzle.append(k)
    machineplay(puzzle)
```

```
Enter the board values with spaces:
1 0 3 4 2 5 7 8 6

Is the following board correct?

+---+---+---+
| 1 | 0 | 3 |
+---+---+---+
| 4 | 2 | 5 |
+---+---+---+
| 7 | 8 | 6 |
+---+---+---+


[Y/N]: Y

SOLVED!

The steps to solve are:-
DOWN, RIGHT, DOWN
```

<u>Results:-</u>

We have successfully implemented BFS.

Date: **Expt.No: 05**	**IMPLEMENTATION AND ANALYSIS OF DFS FOR AN APPLICATION**

AIM:-

To implement analysis of dfs for an application using python.

Algorithm:-

```
Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until STACK is empty

Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)

Step 5: Push on the stack all the neighbours of N that are in the ready state (whose STATUS= 1)and
set their STATUS=2(waitingstate) [END OF LOOP]

Step 6: EXIT
```

Code:-

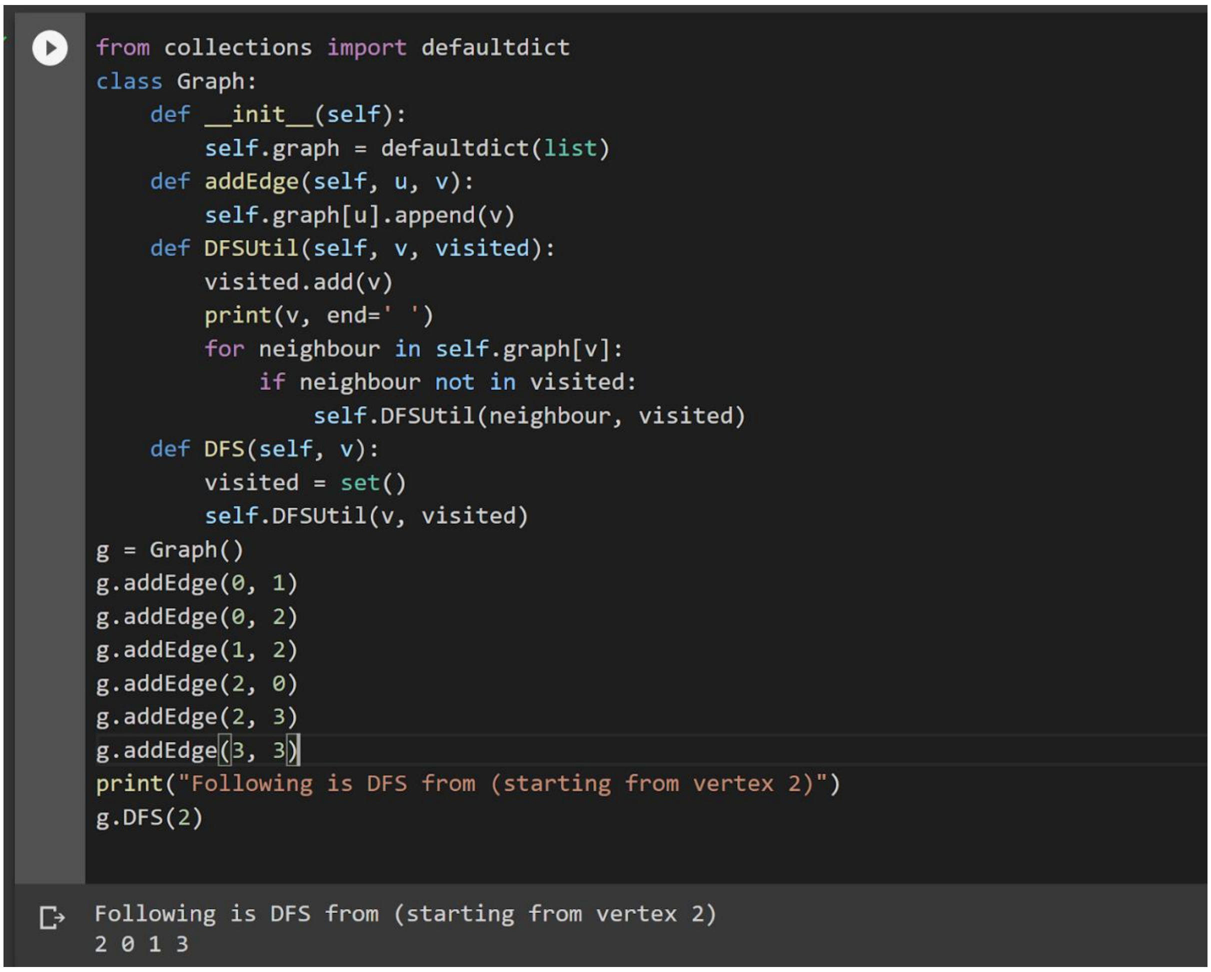
```
from collections import defaultdict

class Graph:

    def __init__(self):

        self.graph = defaultdict(list)

    def addEdge(self, u, v):

        self.graph[u].append(v)

    def DFSUtil(self, v, visited):

        visited.add(v)

        print(v, end=' ')

        for neighbour in self.graph[v]:

            if neighbour not in visited:

                self.DFSUtil(neighbour, visited)

    def DFS(self, v):

        visited = set()

        self.DFSUtil(v, visited)

g = Graph()

g.addEdge(0, 1)

g.addEdge(0, 2)

g.addEdge(1, 2)

g.addEdge(2, 0)

g.addEdge(2, 3)
```

g.addEdge(3, 3)

print("Following is DFS from (starting from vertex 2)")

g.DFS(2)

Output:-



```
from collections import defaultdict
class Graph:
    def __init__(self):
        self.graph = defaultdict(list)
    def addEdge(self, u, v):
        self.graph[u].append(v)
    def DFSUtil(self, v, visited):
        visited.add(v)
        print(v, end=' ')
        for neighbour in self.graph[v]:
            if neighbour not in visited:
                self.DFSUtil(neighbour, visited)
    def DFS(self, v):
        visited = set()
        self.DFSUtil(v, visited)
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)
print("Following is DFS from (starting from vertex 2)")
g.DFS(2)
```

```
Following is DFS from (starting from vertex 2)
2 0 1 3
```

Results:-

We have successfully implemented DFS.

Date: **Expt.No: 06**	**A* ALGORITHM IN REAL WORLD PROBLEMS**

AIM:-

To implement analysis of A* Algorithm for an application using python.

Algorithm:-

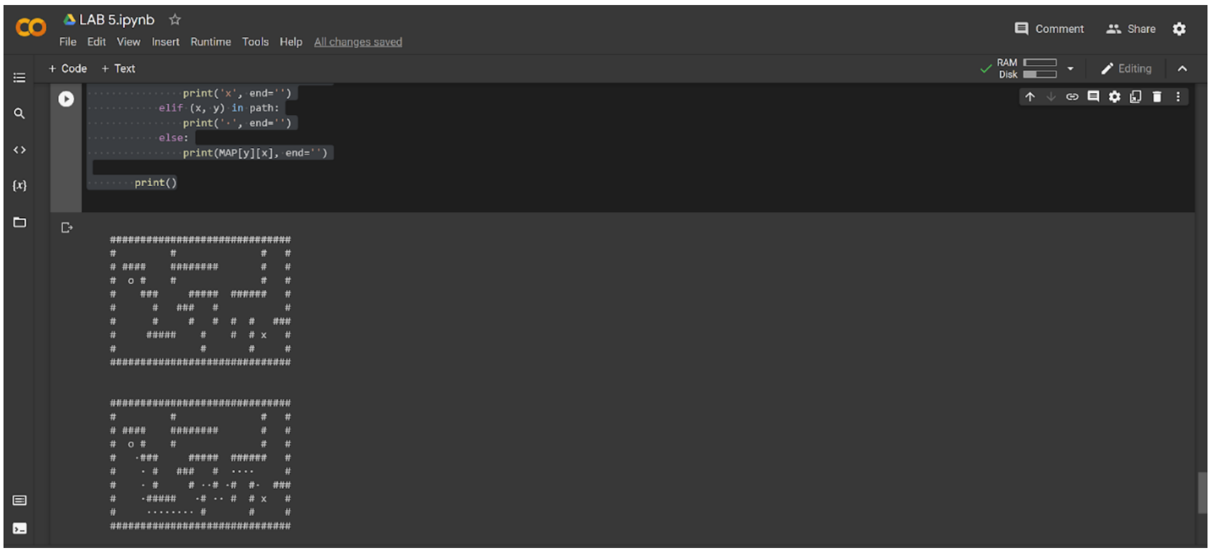
```
Step 1. Initialize the open list

Step 2. Initialize the closed list put the starting node on the open list (you can leave its f at zero)

Step 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)  push q on the closed list end (while loop)
    e)  stop
```

Code:-

A*ALGORITHM

```
import math
from simpleai.search import SearchProblem, astar
class MazeSolver(SearchProblem):
    def __init__(self, board):
        self.board = board
        self.goal = (0, 0)

        for y in range(len(self.board)):
            for x in range(len(self.board[y])):
                if self.board[y][x].lower() == "o":
                    self.initial = (x, y)
                elif self.board[y][x].lower() == "x":
                    self.goal = (x, y)

        super(MazeSolver, self).__init__(initial_state=self.initial)
    def actions(self, state):
        actions = []
        for action in COSTS.keys():
            newx, newy = self.result(state, action)
            if self.board[newy][newx] != "#":
                actions.append(action)

        return actions

    def result(self, state, action):
        x, y = state

        if action.count("up"):
            y -= 1
        if action.count("down"):
            y += 1
        if action.count("left"):
            x -= 1
        if action.count("right"):
            x += 1

        new_state = (x, y)

        return new_state

    def is_goal(self, state):
        return state == self.goal

    def cost(self, state, action, state2):
        return COSTS[action]

    def heuristic(self, state):
        x, y = state
        gx, gy = self.goal

        return math.sqrt((x - gx) ** 2 + (y - gy) ** 2)

if __name__ == "__main__":

    MAP = """
    ##############################
    #         #              #   #
    # ####    ########       #   #
    #  o #    #              #   #
    #    ###     #####  ######   #
    #      #   ###   #           #
    #      #     #   #  #  #   ###
    #     #####    #    #  # x   #
    #              #       #     #
    ##############################
    """

    print(MAP)
    MAP = [list(x) for x in MAP.split("\n") if x]

    cost_regular = 1.0
    cost_diagonal = 1.7

    COSTS = {
        "up": cost_regular,
        "down": cost_regular,
        "left": cost_regular,
        "right": cost_regular,
        "up left": cost_diagonal,
        "up right": cost_diagonal,
        "down left": cost_diagonal,
        "down right": cost_diagonal,
    }
    problem = MazeSolver(MAP)
    result = astar(problem, graph_search=True)
    path = [x[1] for x in result.path()]
    print()
    for y in range(len(MAP)):
        for x in range(len(MAP[y])):
            if (x, y) == problem.initial:
                print('o', end='')
            elif (x, y) == problem.goal:
                print('x', end='')
            elif (x, y) in path:
                print('·', end='')
            else:
                print(MAP[y][x], end='')
        print()
```

OUTPUT:-



```
                print('x', end='')
            elif (x, y) in path:
                print('·', end='')
            else:
                print(MAP[y][x], end='')

        print()
```

```
##############################
#         #              #   #
# ####    ########       #   #
#  o #    #              #   #
#    ###     #####  ######   #
#      #   ###   #           #
#      #     #   #  #  #   ###
#     #####    #    #  # x   #
#              #       #     #
##############################


##############################
#         #              #   #
# ####    ########       #   #
#  o #    #              #   #
#   ·###     #####  ######   #
#    · #   ###   #  ····     #
#    · #     # ··# ·#  #·  ###
#    ·#####   ·# ·· #  # x   #
#     ········ #       #     #
##############################
```

Result:-
We have successfully implemented the experiments.

Date: Expt.No: 07	IMPLEMENTATION OF CONSTRAINT SATIATION PROBLEMS

AIM:-

To implement of constraint satiation problems using python.

Algorithm:-

Step 1. Graph / map colouring problems are those where the nodes are assigned colors such that the adjacent connected nodes / regions don't have the same color assigned.

Step 2. At the same time, it is required to use the minimum number of colors possible – called the chromatic number.

Step 3. Start by coloring the first node with a color.

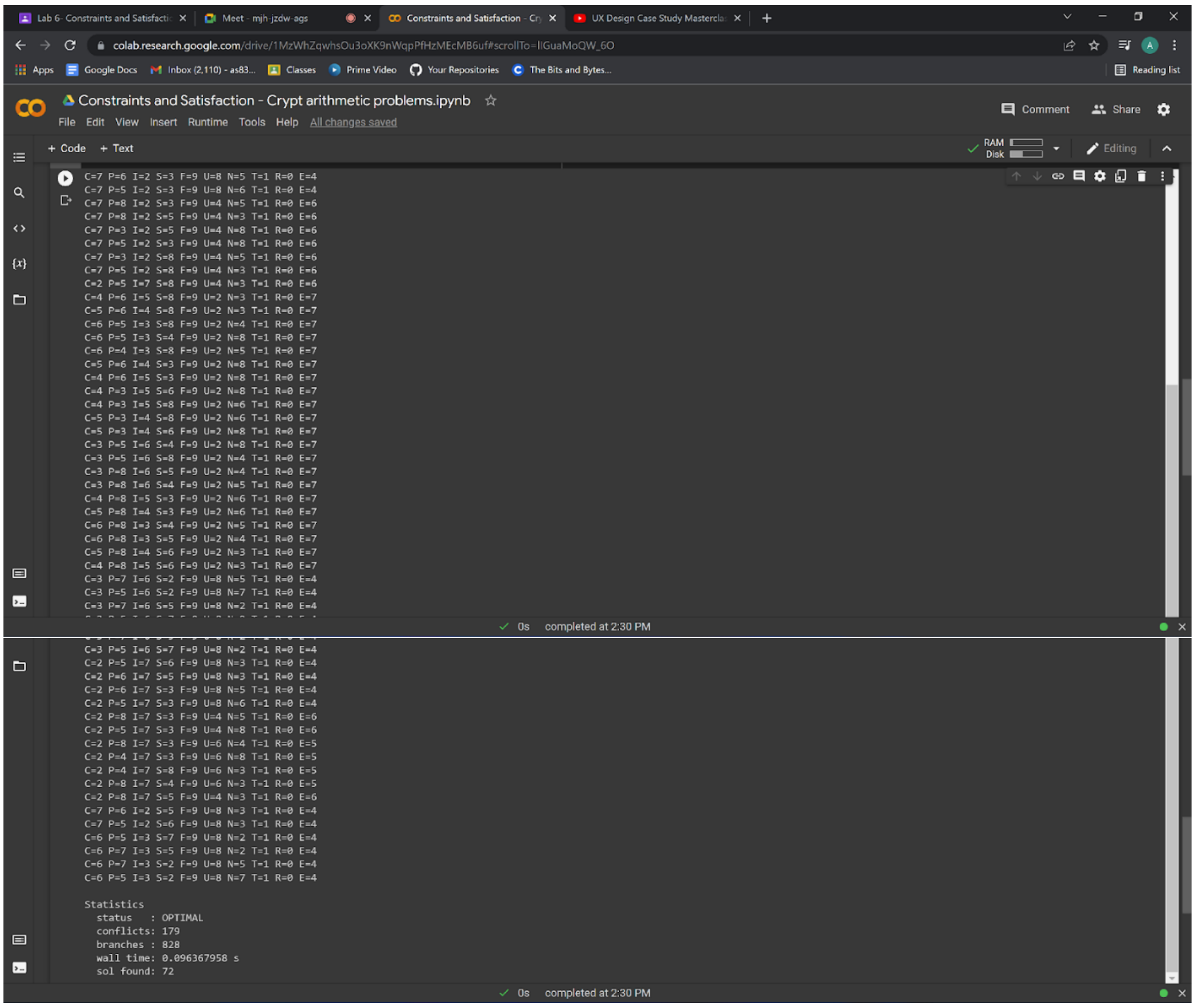
Step 4. Color the subsequent connected nodes with a different color.

Step 5. Check at every step that it satisfies the condition.

Code:-

```
from ortools.sat.python import cp_model
class VarArraySolutionPrinter(cp_model.CpSolverSolutionCallback):
    def __init__(self, variables):
        cp_model.CpSolverSolutionCallback.__init__(self)
        self.__variables = variables
        self.__solution_count = 0
    def on_solution_callback(self):
        self.__solution_count += 1
        for v in self.__variables:
            print('%s=%i' % (v, self.Value(v)), end=' ')
        print()
    def solution_count(self):
        return self.__solution_count
def main():
    model = cp_model.CpModel()
    base = 10
    c = model.NewIntVar(1, base - 1, 'C')
    p = model.NewIntVar(0, base - 1, 'P')
    i = model.NewIntVar(1, base - 1, 'I')
    s = model.NewIntVar(0, base - 1, 'S')
    f = model.NewIntVar(1, base - 1, 'F')
    u = model.NewIntVar(0, base - 1, 'U')
    n = model.NewIntVar(0, base - 1, 'N')
    t = model.NewIntVar(1, base - 1, 'T')
    r = model.NewIntVar(0, base - 1, 'R')
    e = model.NewIntVar(0, base - 1, 'E')

    # We need to group variables in a list to use the constraint AllDifferent.
    letters = [c, p, i, s, f, u, n, t, r, e]
    assert base >= len(letters)
    model.AddAllDifferent(letters)
    model.Add(c * base + p + i * base + s + f * base * base + u * base +
              n == t * base * base * base + r * base * base + u * base + e)
    solver = cp_model.CpSolver()
    solution_printer = VarArraySolutionPrinter(letters)
    # Enumerate all solutions.
    solver.parameters.enumerate_all_solutions = True
    # Solve.
    status = solver.Solve(model, solution_printer)
    # Statistics.
    print('\nStatistics')
    print(f'  status   : {solver.StatusName(status)}')
    print(f'  conflicts: {solver.NumConflicts()}')
    print(f'  branches : {solver.NumBranches()}')
    print(f'  wall time: {solver.WallTime()} s')
    print(f'  sol found: {solution_printer.solution_count()}')
if __name__ == '__main__':
    main()
```

OUTPUT:-


Constraints and Satisfaction - Crypt arithmetic problems.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
Comment Share
Editing
0s completed at 2:30 PM

```
C=7 P=6 I=2 S=3 F=9 U=8 N=5 T=1 R=0 E=4
C=7 P=5 I=2 S=3 F=9 U=8 N=6 T=1 R=0 E=4
C=7 P=8 I=2 S=3 F=9 U=4 N=5 T=1 R=0 E=6
C=7 P=8 I=2 S=5 F=9 U=4 N=3 T=1 R=0 E=6
C=7 P=3 I=2 S=5 F=9 U=4 N=8 T=1 R=0 E=6
C=7 P=5 I=2 S=3 F=9 U=4 N=8 T=1 R=0 E=6
C=7 P=3 I=2 S=8 F=9 U=4 N=5 T=1 R=0 E=6
C=7 P=5 I=2 S=8 F=9 U=4 N=3 T=1 R=0 E=6
C=2 P=5 I=7 S=8 F=9 U=4 N=3 T=1 R=0 E=6
C=4 P=6 I=5 S=8 F=9 U=2 N=3 T=1 R=0 E=7
C=5 P=6 I=4 S=8 F=9 U=2 N=3 T=1 R=0 E=7
C=6 P=5 I=3 S=8 F=9 U=2 N=4 T=1 R=0 E=7
C=6 P=5 I=3 S=4 F=9 U=2 N=8 T=1 R=0 E=7
C=6 P=4 I=3 S=8 F=9 U=2 N=5 T=1 R=0 E=7
C=5 P=6 I=4 S=3 F=9 U=2 N=8 T=1 R=0 E=7
C=4 P=6 I=5 S=3 F=9 U=2 N=8 T=1 R=0 E=7
C=4 P=3 I=5 S=6 F=9 U=2 N=8 T=1 R=0 E=7
C=4 P=3 I=5 S=8 F=9 U=2 N=6 T=1 R=0 E=7
C=5 P=3 I=4 S=8 F=9 U=2 N=6 T=1 R=0 E=7
C=5 P=3 I=4 S=6 F=9 U=2 N=8 T=1 R=0 E=7
C=3 P=5 I=6 S=4 F=9 U=2 N=8 T=1 R=0 E=7
C=3 P=5 I=6 S=8 F=9 U=2 N=4 T=1 R=0 E=7
C=3 P=8 I=6 S=5 F=9 U=2 N=4 T=1 R=0 E=7
C=3 P=8 I=6 S=4 F=9 U=2 N=5 T=1 R=0 E=7
C=4 P=8 I=5 S=3 F=9 U=2 N=6 T=1 R=0 E=7
C=5 P=8 I=4 S=3 F=9 U=2 N=6 T=1 R=0 E=7
C=6 P=8 I=3 S=4 F=9 U=2 N=5 T=1 R=0 E=7
C=6 P=8 I=3 S=5 F=9 U=2 N=4 T=1 R=0 E=7
C=5 P=8 I=4 S=6 F=9 U=2 N=3 T=1 R=0 E=7
C=4 P=8 I=5 S=6 F=9 U=2 N=3 T=1 R=0 E=7
C=3 P=7 I=6 S=2 F=9 U=8 N=5 T=1 R=0 E=4
C=3 P=5 I=6 S=2 F=9 U=8 N=7 T=1 R=0 E=4
C=3 P=7 I=6 S=5 F=9 U=8 N=2 T=1 R=0 E=4
```

```
C=3 P=5 I=6 S=7 F=9 U=8 N=2 T=1 R=0 E=4
C=2 P=5 I=7 S=6 F=9 U=8 N=3 T=1 R=0 E=4
C=2 P=6 I=7 S=5 F=9 U=8 N=3 T=1 R=0 E=4
C=2 P=6 I=7 S=3 F=9 U=8 N=5 T=1 R=0 E=4
C=2 P=5 I=7 S=3 F=9 U=8 N=6 T=1 R=0 E=4
C=2 P=8 I=7 S=3 F=9 U=4 N=5 T=1 R=0 E=6
C=2 P=5 I=7 S=3 F=9 U=4 N=8 T=1 R=0 E=6
C=2 P=8 I=7 S=3 F=9 U=6 N=4 T=1 R=0 E=5
C=2 P=4 I=7 S=3 F=9 U=6 N=8 T=1 R=0 E=5
C=2 P=4 I=7 S=8 F=9 U=6 N=3 T=1 R=0 E=5
C=2 P=8 I=7 S=4 F=9 U=6 N=3 T=1 R=0 E=5
C=2 P=8 I=7 S=5 F=9 U=4 N=3 T=1 R=0 E=6
C=7 P=6 I=2 S=5 F=9 U=8 N=3 T=1 R=0 E=4
C=7 P=5 I=2 S=6 F=9 U=8 N=3 T=1 R=0 E=4
C=6 P=5 I=3 S=7 F=9 U=8 N=2 T=1 R=0 E=4
C=6 P=7 I=3 S=5 F=9 U=8 N=2 T=1 R=0 E=4
C=6 P=7 I=3 S=2 F=9 U=8 N=5 T=1 R=0 E=4
C=6 P=5 I=3 S=2 F=9 U=8 N=7 T=1 R=0 E=4

Statistics
  status   : OPTIMAL
  conflicts: 179
  branches : 828
  wall time: 0.096367958 s
  sol found: 72
```

Result:-
We have successfully implemented the experiments.

Date: **Expt.No: 08**	**IMPLEMENTATION OF MINMAX ALGORITHM FOR AN APPLICATION**

AIM:-

To implement minmax algorithm for an application using python.

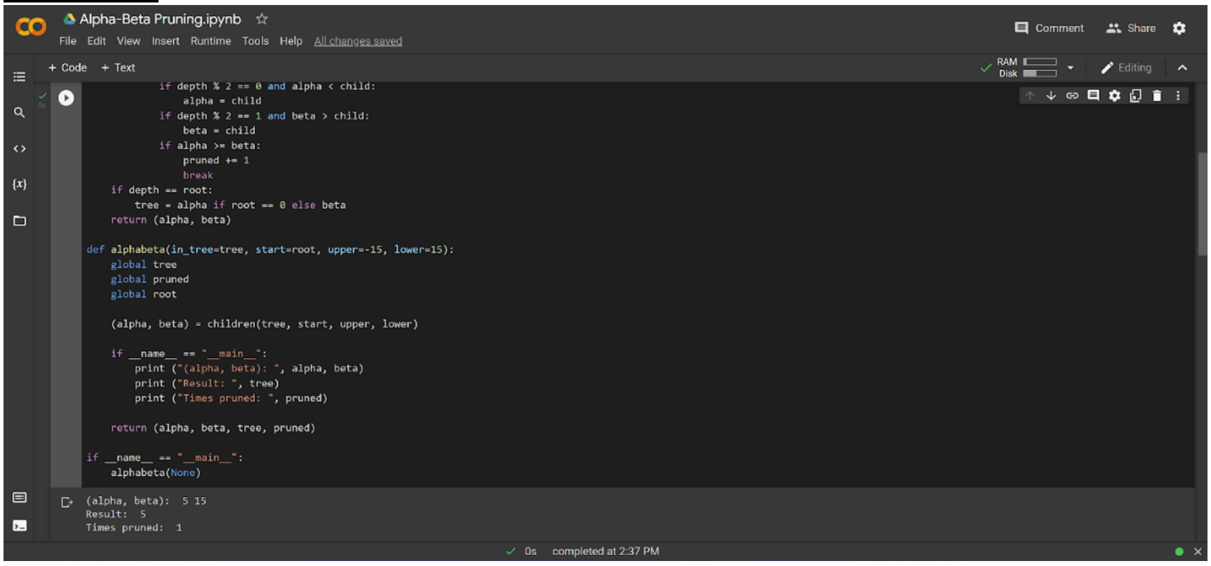
Algorithm:-

Step 1. Minimax algorithm, also known as Maximin, is a technique in AI to minimize the maximum loss or maximize the minimum gain while decision making.
Step 2. A recursive algorithm, used in two player games (predominantly) where a tree structure is used to denote the position of the player.
Step 3. The goal is to obtain the maximum possible score without knowing the decision of the other players in the game.
Step 4. The levels of the tree are alternatively names as max and min, where the root level takes max

Code:-

```
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 alphabeta(in_tree=tree, start=root, upper=-15, lower=15):
    global tree
    global pruned
    global root
    (alpha, beta) = children(tree, start, upper, lower)

    if __name__ == "__main__":
        print ("(alpha, beta): ", alpha, beta)
        print ("Result: ", tree)
        print ("Times pruned: ", pruned)

    return (alpha, beta, tree, pruned)

if __name__ == "__main__":
    alphabeta(None)
```

OUTPUT:-



```
            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 alphabeta(in_tree=tree, start=root, upper=-15, lower=15):
    global tree
    global pruned
    global root

    (alpha, beta) = children(tree, start, upper, lower)

    if __name__ == "__main__":
        print ("(alpha, beta): ", alpha, beta)
        print ("Result: ", tree)
        print ("Times pruned: ", pruned)

    return (alpha, beta, tree, pruned)

if __name__ == "__main__":
    alphabeta(None)
```

```
(alpha, beta):  5 15
Result:  5
Times pruned:  1
```

Result:-

We have successfully implemented the experiments.

Date:- **Expt. No:- 09**	**IMPLEMENTATION OF UNIFICATION AND RESOLUTION FOR REAL WORLD**

AIM:-

To implement unification and resolution for real world using python.

Algorithm:-

Unification

Step. 1: If Ψ 1 or Ψ 2 is a variable or constant, then:

1. If Ψ 1 or Ψ 2 are identical, then return NIL.
2. Else if Ψ 1 is a variable, a. then if Ψ 1 occurs in Ψ 2 , then return FAILURE b. Else return { (Ψ 2 / Ψ 1)}.
3. Else if Ψ 2 is a variable, a. If Ψ 2 occurs in Ψ 1 then return FAILURE, b. Else return {(Ψ 1 / Ψ 2)}.
4. Else return FAILURE.

Step.2: If the initial Predicate symbol in Ψ 1 and Ψ 2 are not same, then return FAILURE.

Step. 3: IF Ψ 1 and Ψ 2 have a different number of arguments, then return FAILURE.

Step. 4: Set Substitution set(SUBST) to NIL.

Step. 5: For i=1 to the number of elements in Ψ 1 . a) Call Unify function with the ith element of Ψ 1 and ith element of Ψ 2 , and put the result into S. b) If S = failure then returns Failure c) If S ≠ NIL then do, a. Apply S to the remainder of both L1 and L2. b. SUBST= APPEND(S, SUBST).

Step.6: Return SUBST

Code:-

Unification

```
def unify(E1, E2):
    constants = [chr(i) for i in range(ord('a'), ord('w') + 1)]
    variables = [chr(i) for i in range(ord('A'), ord('Z') + 1)]
    variables.extend(['x', 'y', 'z'])
    if (E1 in constants and E2 in constants) or (E1 is None and E2 is None):  # base case
        if E1 == E2:
            return None
        else:
            return "FAIL"

    elif E1 in variables:
        if E1 in E2:
            return "FAIL - E1 occurs in E2"
        else:
            return (E2 + "/" + E1)

    elif E2 in variables:
        if E2 in E1:
            return "FAIL - E2 occurs in E1"
        else:
            return (E1 + "/" + E2)
    else:
        if ('(' in E1 and '(' not in E2):
            return "FAIL - E1 is a function and E2 is a variable/constant"
        elif ('(' not in E1 and '(' in E2):
            return "FAIL - E1 is a variable/constant and E2 is a function"

print("Enter the Expressions (without spaces):")
s1 = input()
s2 = input()
E1 = s1[2:len(s1)-1].split(',')
E2 = s2[2:len(s2)-1].split(',')
if s1[0] != s2[0]:
    print("FAIL - Initial Predicate Symbols in E1 and E2 are not identical")
elif len(E1) != len(E2):
    print("FAIL - E1 and E2 have different number of arguments")
else:
    n = len(E1)
    s = []   # General Unifiers
    print("--------------------------------------------------------------------------")
    for i in range(n):
        print("E1:", E1[i])
        print("E2:", E2[i])
        print("Result:", unify(E1[i],E2[i]))
        print("--------------------------------------------------------------------------")
        if "FAIL" not in unify(E1[i],E2[i]):
            s.append(unify(E1[i],E2[i]))

    if len(s) == n:
        print("General Unifiers: { ", end = "")
        for i in range(len(s)):
            if i != len(s)-1:
                print(s[i] + ", ", end = "")
            else:
                print(s[i] + " }", end = "")
```

Resolution:-
Algorithm:-

Step-1: Conversion of Facts into FOL

Step-2: Conversion of FOL into CNF

- Eliminate all implication (→) and rewrite
- Move negation (¬)inwards and rewrite
- Rename variables or standardize variables
- Eliminate existential instantiation quantifier by elimination.
- Drop Universal quantifiers

Step-3: Negate the statement to be proved

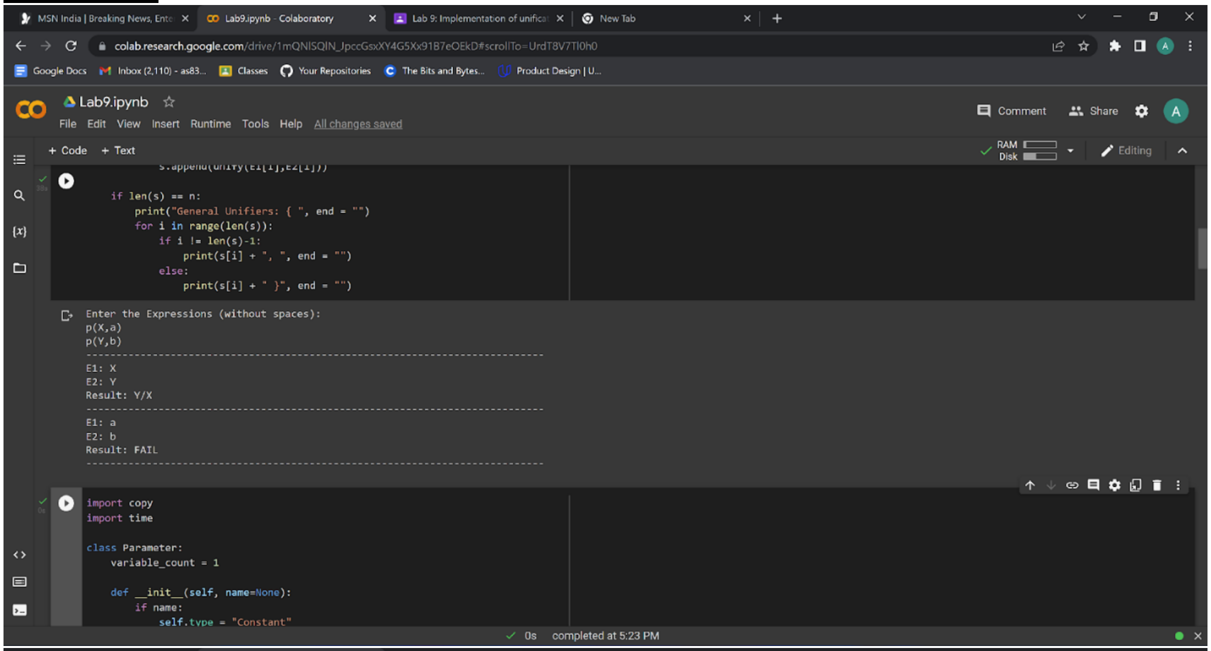
Step-4: Draw Resolution graph:

Code:-

```
import copy
import time

class Parameter:
    variable_count = 1

    def __init__(self, name=None):
        if name:
            self.type = "Constant"
            self.name = name
        else:
            self.type = "Variable"
            self.name = "v" + str(Parameter.variable_count)
            Parameter.variable_count += 1

    def isConstant(self):
        return self.type == "Constant"

    def unify(self, type_, name):
        self.type = type_
        self.name = name

    def __eq__(self, other):
        return self.name == other.name

    def __str__(self):
        return self.name

class Predicate:
    def __init__(self, name, params):
        self.name = name
        self.params = params

    def __eq__(self, other):
        return self.name == other.name and all(a == b for a, b in zip(self.params, other.params))

    def __str__(self):
        return self.name + "(" + ",".join(str(x) for x in self.params) + ")"

    def getNegatedPredicate(self):
        return Predicate(negatePredicate(self.name), self.params)

class Sentence:
    sentence_count = 0

    def __init__(self, string):
        self.sentence_index = Sentence.sentence_count
        Sentence.sentence_count += 1
        self.predicates = []
        self.variable_map = {}
        local = {}

        for predicate in string.split("|"):
            name = predicate[:predicate.find("(")]
            params = []

            for param in predicate[predicate.find("(") + 1: predicate.find(")")].split(","):
                if param[0].islower():
                    if param not in local:  # Variable
                        local[param] = Parameter()
                        self.variable_map[local[param].name] = local[param]
                    new_param = local[param]
                else:
                    new_param = Parameter(param)
                    self.variable_map[param] = new_param

                params.append(new_param)

            self.predicates.append(Predicate(name, params))

    def getPredicates(self):
        return [predicate.name for predicate in self.predicates]

    def findPredicates(self, name):
        return [predicate for predicate in self.predicates if predicate.name == name]

    def removePredicate(self, predicate):
        self.predicates.remove(predicate)
        for key, val in self.variable_map.items():
            if not val:
                self.variable_map.pop(key)

    def containsVariable(self):
        return any(not param.isConstant() for param in self.variable_map.values())

    def __eq__(self, other):
        if len(self.predicates) == 1 and self.predicates[0] == other:
            return True
        return False

    def __str__(self):
        return "".join([str(predicate) for predicate in self.predicates])

class KB:
    def __init__(self, inputSentences):
        self.inputSentences = [x.replace(" ", "") for x in inputSentences]
        self.sentences = []
        self.sentence_map = {}

    def prepareKB(self):
        self.convertSentencesToCNF()
        for sentence_string in self.inputSentences:
            sentence = Sentence(sentence_string)
            for predicate in sentence.getPredicates():
                self.sentence_map[predicate] = self.sentence_map.get(predicate, []) + [sentence]

    def convertSentencesToCNF(self):
        for sentenceIdx in range(len(self.inputSentences)):
            if "=>" in self.inputSentences[sentenceIdx]:  # Do negation of the Premise and add them as lite
ral
                self.inputSentences[sentenceIdx] = negateAntecedent(self.inputSentences[sentenceIdx])

    def askQueries(self, queryList):
        results = []

        for query in queryList:
            negatedQuery = Sentence(negatePredicate(query.replace(" ", "")))
            negatedPredicate = negatedQuery.predicates[0]
            prev_sentence_map = copy.deepcopy(self.sentence_map)
            self.sentence_map[negatedPredicate.name] = self.sentence_map.get(negatedPredicate.name, []
) + [negatedQuery]
            self.timeLimit = time.time() + 40

            try:
                result = self.resolve([negatedPredicate], [False]*(len(self.inputSentences) + 1))
            except:
                result = False

            self.sentence_map = prev_sentence_map

            if result:
                results.append("TRUE")
            else:
                results.append("FALSE")

        return results

    def resolve(self, queryStack, visited, depth=0):
        if time.time() > self.timeLimit:
            raise Exception
        if queryStack:
            query = queryStack.pop(-1)
            negatedQuery = query.getNegatedPredicate()
            queryPredicateName = negatedQuery.name
            if queryPredicateName not in self.sentence_map:
                return False
            else:
                queryPredicate = negatedQuery
                for kb_sentence in self.sentence_map[queryPredicateName]:
                    if not visited[kb_sentence.sentence_index]:
                        for kbPredicate in kb_sentence.findPredicates(queryPredicateName):
                            canUnify, substitution = performUnification(copy.deepcopy(queryPredicate), copy.deepcopy(kbPredicate))

                            if canUnify:
                                newSentence = copy.deepcopy(kb_sentence)
                                newSentence.removePredicate(kbPredicate)
                                newQueryStack = copy.deepcopy(queryStack)

                                if substitution:
                                    for old, new in substitution.items():
                                        if old in newSentence.variable_map:
                                            parameter = newSentence.variable_map[old]
                                            newSentence.variable_map.pop(old)
                                            parameter.unify("Variable" if new[0].islower() else "Constant", new)
                                            newSentence.variable_map[new] = parameter

                                    for predicate in newQueryStack:
                                        for index, param in enumerate(predicate.params):
                                            if param.name in substitution:
                                                new = substitution[param.name]
                                                predicate.params[index].unify("Variable" if new[0].islower() else "Constant", new)

                                for predicate in newSentence.predicates:
                                    newQueryStack.append(predicate)

                                new_visited = copy.deepcopy(visited)
                                if kb_sentence.containsVariable() and len(kb_sentence.predicates) > 1:
                                    new_visited[kb_sentence.sentence_index] = True

                                if self.resolve(newQueryStack, new_visited, depth + 1):
                                    return True
                return False
        return True

def performUnification(queryPredicate, kbPredicate):
    substitution = {}
    if queryPredicate == kbPredicate:
        return True, {}
    else:
        for query, kb in zip(queryPredicate.params, kbPredicate.params):
            if query == kb:
                continue
            if kb.isConstant():
                if not query.isConstant():
                    if query.name not in substitution:
                        substitution[query.name] = kb.name
                    elif substitution[query.name] != kb.name:
                        return False, {}
                    query.unify("Constant", kb.name)
                else:
```

```
                return False, {}
            else:
                if not query.isConstant():
                    if kb.name not in substitution:
                        substitution[kb.name] = query.name
                    elif substitution[kb.name] != query.name:
                        return False, {}
                    kb.unify("Variable", query.name)
                else:
                    if kb.name not in substitution:
                        substitution[kb.name] = query.name
                    elif substitution[kb.name] != query.name:
                        return False, {}
    return True, substitution

def negatePredicate(predicate):
    return predicate[1:] if predicate[0] == "~" else "~" + predicate

def negateAntecedent(sentence):
    antecedent = sentence[:sentence.find("=>")]
    premise = []

    for predicate in antecedent.split("&"):
        premise.append(negatePredicate(predicate))

    premise.append(sentence[sentence.find("=>") + 2:])
    return "|".join(premise)

def getInput(filename):
    with open(filename, "r") as file:
        noOfQueries = int(file.readline().strip())
        inputQueries = [file.readline().strip() for _ in range(noOfQueries)]
        noOfSentences = int(file.readline().strip())
        inputSentences = [file.readline().strip() for _ in range(noOfSentences)]
        return inputQueries, inputSentences

def printOutput(filename, results):
    print(results)
    with open(filename, "w") as file:
        for line in results:
            file.write(line)
            file.write("\n")
    file.close()

if __name__ == '__main__':
    inputQueries_, inputSentences_ = getInput("input.txt")
    knowledgeBase = KB(inputSentences_)
    knowledgeBase.prepareKB()
    results_ = knowledgeBase.askQueries(inputQueries_)
printOutput("output.txt", results_)
```

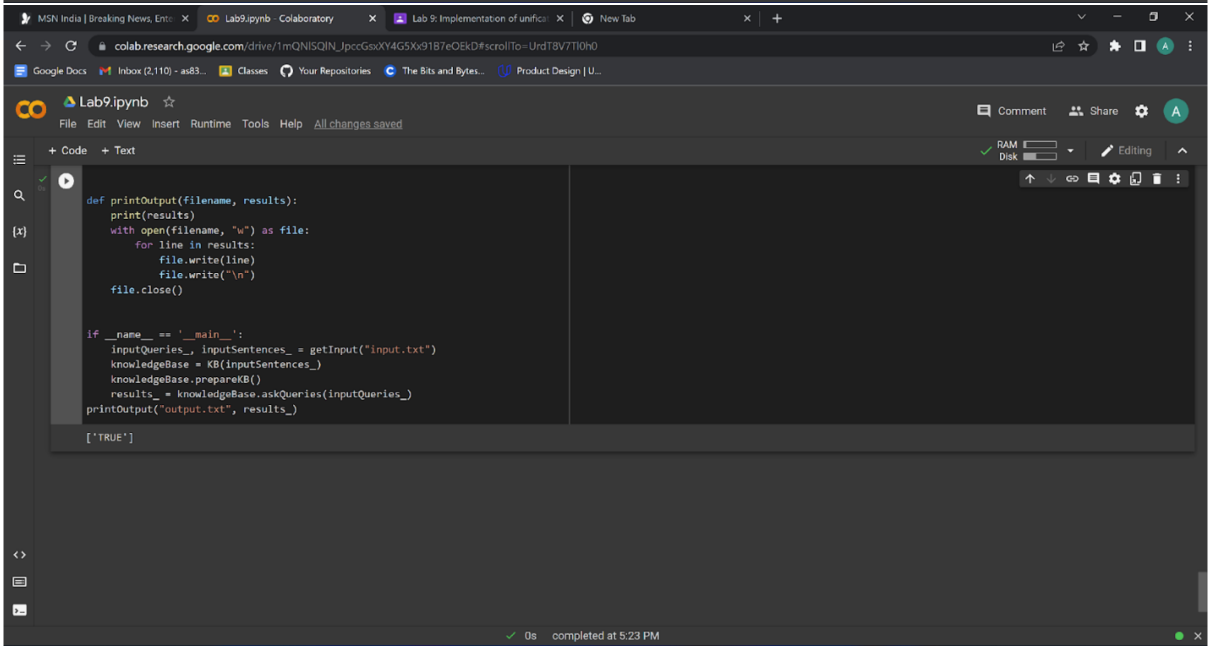
OUTPUT:-



```
        if len(s) == n:
            print("General Unifiers: { ", end = "")
            for i in range(len(s)):
                if i != len(s)-1:
                    print(s[i] + ", ", end = "")
                else:
                    print(s[i] + " }", end = "")
```

```
Enter the Expressions (without spaces):
p(X,a)
p(Y,b)
--------------------------------------------------------------------------
E1: X
E2: Y
Result: Y/X
--------------------------------------------------------------------------
E1: a
E2: b
Result: FAIL
--------------------------------------------------------------------------
```

```
import copy
import time

class Parameter:
    variable_count = 1

    def __init__(self, name=None):
        if name:
            self.type = "Constant"
```



```
def printOutput(filename, results):
    print(results)
    with open(filename, "w") as file:
        for line in results:
            file.write(line)
            file.write("\n")
    file.close()


if __name__ == '__main__':
    inputQueries_, inputSentences_ = getInput("input.txt")
    knowledgeBase = KB(inputSentences_)
    knowledgeBase.prepareKB()
    results_ = knowledgeBase.askQueries(inputQueries_)
printOutput("output.txt", results_)
```

```
['TRUE']
```

Result:-

We have successfully implemented the experiments.

Date: **Expt.No: 10**	**IMPLEMENTATION OF BLOCK WORLD PROBLEM**

AIM:-

To implement block world problem using python.

Algorithm:-

Step 1: start

Step 2: Move Totable(a,b,c)

Step 3: Construct Preconditions: On(a,b), Clear(a), Clear(c)

Step 4: Effects: On(a,c), Clear(b), NOT(On(a,b)), NOT(Clear(c))

Step 5: Stop

Code:-

Main.py

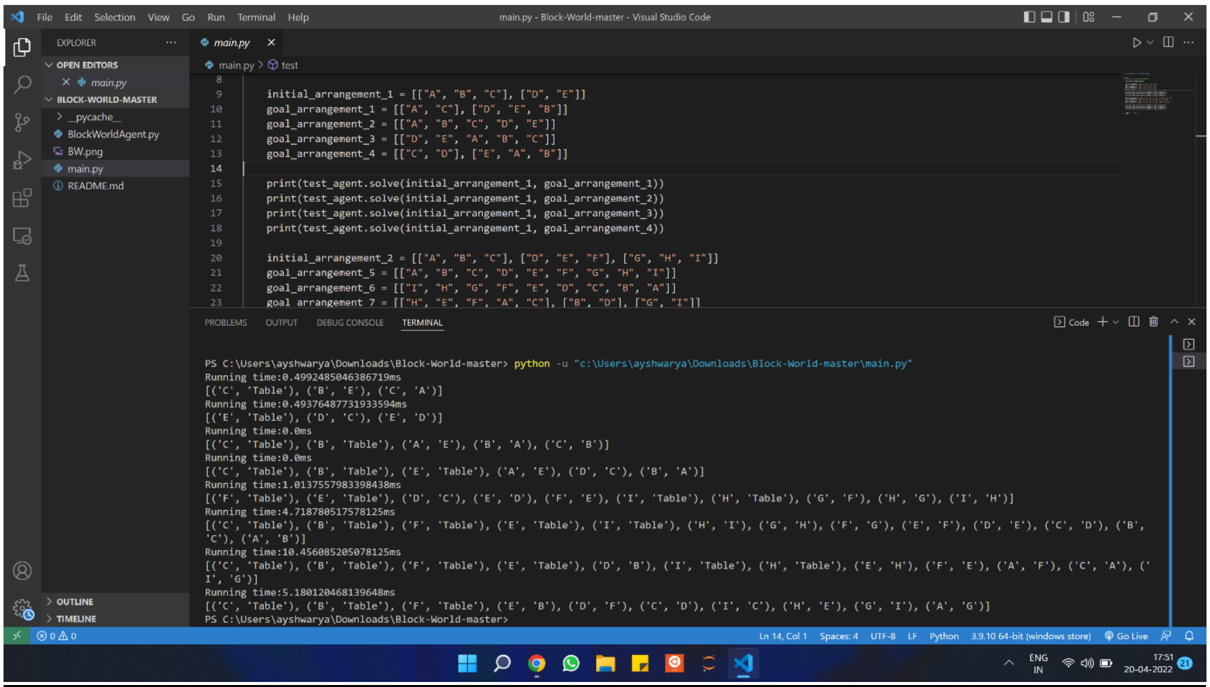
```
from BlockWorldAgent import BlockWorldAgent
def test():
    # This will test your BlockWorldAgent
    # with eight initial test cases.
    test_agent = BlockWorldAgent()

    initial_arrangement_1 = [["A", "B", "C"], ["D", "E"]]
    goal_arrangement_1 = [["A", "C"], ["D", "E", "B"]]
    goal_arrangement_2 = [["A", "B", "C", "D", "E"]]
    goal_arrangement_3 = [["D", "E", "A", "B", "C"]]
    goal_arrangement_4 = [["C", "D"], ["E", "A", "B"]]

    print(test_agent.solve(initial_arrangement_1, goal_arrangement_1))
    print(test_agent.solve(initial_arrangement_1, goal_arrangement_2))
    print(test_agent.solve(initial_arrangement_1, goal_arrangement_3))
    print(test_agent.solve(initial_arrangement_1, goal_arrangement_4))

    initial_arrangement_2 = [["A", "B", "C"], ["D", "E", "F"], ["G", "H", "I"]]
    goal_arrangement_5 = [["A", "B", "C", "D", "E", "F", "G", "H", "I"]]
    goal_arrangement_6 = [["I", "H", "G", "F", "E", "D", "C", "B", "A"]]
    goal_arrangement_7 = [["H", "E", "F", "A", "C"], ["B", "D"], ["G", "I"]]
    goal_arrangement_8 = [["F", "D", "C", "I", "G", "A"], ["B", "E", "H"]]

    print(test_agent.solve(initial_arrangement_2, goal_arrangement_5))
    print(test_agent.solve(initial_arrangement_2, goal_arrangement_6))
    print(test_agent.solve(initial_arrangement_2, goal_arrangement_7))
    print(test_agent.solve(initial_arrangement_2, goal_arrangement_8))

if __name__ == "__main__":
    test()
```

Block.py

```
import copy
import time
class BlockWorldAgent:

    def __init__(self):
        pass

    def solve(self, initial_arrangement, goal_arrangement):
        start = time.time()

        class State:
            def __init__(self, first_stack, second_stack, total_num, moves=None):
                if moves is None:
                    moves = []
                self.first_stack = first_stack
                self.second_stack = second_stack
                self.total_num = total_num
                self.moves = moves

            def __eq__(self, other):
                return (self.first_stack == other.first_stack and self.second_stack == other.second_stack
                        and self.total_num == other.total_num and self.moves == other.moves)

            def goal_state_move(self):
                while self.difference() != 0:
                    self = self.select_move()
                return self.moves

            def select_move(self):  # will select and return the best move
                # first try moving the top block to a stack, if the diff is not reduced, then move it to the
temp_table
                for index, stack in enumerate(self.first_stack):
                    for index2, stack2 in enumerate(self.first_stack):
                        if index != index2:  # don't move to itself stack
                            curr_table, move = self.valid_state_move(self.first_stack, index, index2)
                            new_state = State(curr_table, self.second_stack, self.total_num,
copy.copy(self.moves))
                            new_state.moves.append(move)
                            if new_state.difference() < self.difference():
                                return new_state

                # move the top block to the temp_table, skip if it is already on the table (itself alone on a
table)
                for index, stack in enumerate(self.first_stack):
                    if len(stack) > 1:  # not it self alone
                        curr_table, move = self.valid_state_move(self.first_stack, index, -1)  # -1 means table
                        new_state = State(curr_table, self.second_stack, self.total_num,
copy.copy(self.moves))
                        new_state.moves.append(move)
                        if new_state.difference() <= self.difference():
```

```
                return new_state

        def valid_state_move(self, table, start_index, end_index):
            temp_table = copy.deepcopy(table)
            left = temp_table[start_index]
            top_block = left.pop()
            right = []

            if end_index < 0:  # move to table (-1)
                temp_table.append(right)
                move = (top_block, 'Table')
            else:  # move to stack
                right = temp_table[end_index]
                move = (top_block, right[-1])
            right.append(top_block)

            if len(left) == 0:
                temp_table.remove(left)
            return temp_table, move

        def difference(self):
            same_num = 0
            # compare each stack on two stacks
            for left in self.first_stack:
                for right in self.second_stack:
                    index = 0
                    while index < len(left) and index < len(right):
                        if left[index] == right[index]:
                            same_num += 1
                            index += 1
                        else:
                            break
            diff = self.total_num - same_num
            return diff

    total_num = 0
    for ls in initial_arrangement:
        for e in ls:
            total_num += 1
    state = State(initial_arrangement, goal_arrangement, total_num)
    solution = state.goal_state_move()

    end = time.time()
    run_time = str((end - start) * 1000)
    print("Running time:" + run_time + "ms")
    return solution
```

OUTPUT:-



```python
    initial_arrangement_1 = [["A", "B", "C"], ["D", "E"]]
    goal_arrangement_1 = [["A", "C"], ["D", "E", "B"]]
    goal_arrangement_2 = [["A", "B", "C", "D", "E"]]
    goal_arrangement_3 = [["D", "E", "A", "B", "C"]]
    goal_arrangement_4 = [["C", "D"], ["E", "A", "B"]]

    print(test_agent.solve(initial_arrangement_1, goal_arrangement_1))
    print(test_agent.solve(initial_arrangement_1, goal_arrangement_2))
    print(test_agent.solve(initial_arrangement_1, goal_arrangement_3))
    print(test_agent.solve(initial_arrangement_1, goal_arrangement_4))

    initial_arrangement_2 = [["A", "B", "C"], ["D", "E", "F"], ["G", "H", "I"]]
    goal_arrangement_5 = [["A", "B", "C", "D", "E", "F", "G", "H", "I"]]
    goal_arrangement_6 = [["I", "H", "G", "F", "E", "D", "C", "B", "A"]]
    goal_arrangement_7 = [["H", "E", "F", "A", "C"], ["B", "D"], ["G", "I"]]
```

```
PS C:\Users\ayshwarya\Downloads\Block-World-master> python -u "c:\Users\ayshwarya\Downloads\Block-World-master\main.py"
Running time:0.4992485046386719ms
[('C', 'Table'), ('B', 'E'), ('C', 'A')]
Running time:0.49376487731933594ms
[('E', 'Table'), ('D', 'C'), ('E', 'D')]
Running time:0.0ms
[('C', 'Table'), ('B', 'Table'), ('A', 'E'), ('B', 'A'), ('C', 'B')]
Running time:0.0ms
[('C', 'Table'), ('B', 'Table'), ('E', 'Table'), ('A', 'E'), ('D', 'C'), ('B', 'A')]
Running time:1.0137557983398438ms
[('F', 'Table'), ('E', 'Table'), ('D', 'C'), ('E', 'D'), ('F', 'E'), ('I', 'Table'), ('H', 'Table'), ('G', 'F'), ('H', 'G'), ('I', 'H')]
Running time:4.718780517578125ms
[('C', 'Table'), ('B', 'Table'), ('F', 'Table'), ('E', 'Table'), ('I', 'Table'), ('H', 'I'), ('G', 'H'), ('F', 'G'), ('E', 'F'), ('D', 'E'), ('C', 'D'), ('B',
'C'), ('A', 'B')]
Running time:10.456085205078125ms
[('C', 'Table'), ('B', 'Table'), ('F', 'Table'), ('E', 'Table'), ('D', 'B'), ('I', 'Table'), ('H', 'Table'), ('E', 'H'), ('F', 'E'), ('A', 'F'), ('C', 'A'), ('
I', 'G')]
Running time:5.180120468139648ms
[('C', 'Table'), ('B', 'Table'), ('F', 'Table'), ('E', 'B'), ('D', 'F'), ('C', 'D'), ('I', 'C'), ('H', 'E'), ('G', 'I'), ('A', 'G')]
PS C:\Users\ayshwarya\Downloads\Block-World-master>
```

main.py - Block-World-master - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
OPEN EDITORS
main.py
BLOCK-WORLD-MASTER
__pycache__
BlockWorldAgent.py
BW.png
main.py
README.md
OUTLINE
TIMELINE
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
Ln 14, Col 1 Spaces: 4 UTF-8 LF Python 3.9.10 64-bit (windows store) Go Live
ENG IN 17:51 20-04-2022

Result:-

We have successfully implemented the experiments.