DEPARTMENT OF NETWORKING AND COMMUNICATIONLAB MANUAL

ACADEMIC YEAR: 2021-22 EVEN SEMESTER

Programme (UG/PG)	**: UG**
Semester	**: VI**
Course Code	**: 18CSC305J**
Course Title	**: Artificial Intelligence Lab**
Section	**: H2**
Year	**: Third**

Submitted by

RAVI MYTRESH(RA1911003011019)

In partial fulfillment of the requirements for the degree of

BACHELOR OF TECHNOLOGY

FACULTY OF ENGINEERING AND TECHNOLOGYSRM UNIVERSITY

(Under section 3 of UGC Act, 1956)
SRM Nagar, Kattankulathur- 603203
Kancheepuram District

VISION OF THE INSTITUTE

To emerge as a World - Class University in creating and disseminating knowledge, and providing students a unique learning experience in Science, Technology, Medicine, Management, and other areas of scholarship that will best serve the world and betterment of mankind

MISSION OF THE INSTITUTE

- MOVE UP through international alliances and collaborative initiatives to achieve global excellence.

- ACCOMPLISH A PROCESS to advance knowledge in a rigorous academic and research environment.

- ATTRACT AND BUILD PEOPLE in a rewarding and inspiring environment by fostering freedom, empowerment, creativity and innovation.

VISION OF THE DEPARTMENT

To Nurture as a globally recognizable department in imparting students high-quality education and providing high confidence, unique knowledge, and research experience in the field of networking, cyber security, forensics, information technology, cognitive computing, and the internet of things.

MISSION OF THE DEPARTMENT

- To provide world-class IT professionals with appropriate industry and research-based curriculum
- To train the students in such a way that leads to entrepreneurship and develops societal need-based industries
- To nourish the students as socially responsible professionals by providing them training in personality development, ethics and leadership program.

Registration Number_____

LABORATORY RECORD

Course Code:

Name of the Course:

Programme:

It is certified that this is a Bonafede record of the work carried out by _____ of_____class during the year 2021 -2022

Faculty In-Charge _____ HoD_____

Internal Examiner _____

Date of the Examination_____

LIST OF EXPERIMENTS & SCHEDULE

COURSE CODE:

COURSE TITLE:

Exp No.	Name of the Experiment	Page No	Date	Signature
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				

Faculty In charge **HOD**

HARDWARE AND SOFTWARE REQUIREMENTS

HARDWARE REQUIREMENTS

- Thin Client
- Broadband – Internet Connection

SOFTWARE REQUIREMENT

- AWS Login (Cloud9 Service)
- Google Colab

Prepared By

Dr. S. Prabakeran

(Assistant Professor, Department of Networking and Communications)

MARK SPLIT UP

<table>
<tr><td>1</td><td>Lab 1: Implementation of toy problems</td><td rowspan="3">CLAP1:5 marks

3 Exp= 2.5 marks
Viva= 2.5 marks</td></tr>
<tr><td>2</td><td>Lab 2: Developing agent programs for real world problems</td></tr>
<tr><td>3</td><td>Lab 3: Implementation of constraint satisfaction problems</td></tr>
<tr><td>4</td><td>Lab4: Implementation and Analysis of DFS andBFS for same application</td><td rowspan="4">CLAP2:7.5 marks

4 Exp= 4 marks
Hackerrank= 3.5 marks
(3 medium- 3 marks + 1 difficult – 0.5 mark)</td></tr>
<tr><td>5</td><td>Lab 5: Developing Best first search and A* Algorithm for real world problems</td></tr>
<tr><td>6</td><td>Lab 6: Implementation of uncertain methods foran application (Fuzzy logic/ Dempster Shafer Theory)</td></tr>
<tr><td>7</td><td>Lab 7: Implementation of unification and resolution for real world problems.</td></tr>
<tr><td>8</td><td>Lab 8: Implementation of learning algorithms for an application</td><td rowspan="3">CLAP3:7.5 marks

3 Exp= 5 marks

Hackerrank= 2.5 marks
(2 difficult + 1 Advanced level)</td></tr>
<tr><td>9</td><td>Lab 9:Implementation of NLP programs</td></tr>
<tr><td>10</td><td>Lab 10: Applying deep learning methods to solve an application</td></tr>
<tr><td></td><td>Course Project:
Problem statement/objective with technical depth : 2 marks
Execution and Github upload : 2 marks
Purpose of the problem statement (societal benefit) : 1 mark
Team Members : upto 4 max (expected 20-25 projects per faculty member in a section)</td><td>CLAP4: 5 marks</td></tr>
</table>

Experiment No:-1
Date:-04-01-2022

Date : 4-01-2022

TOY PROBLEM

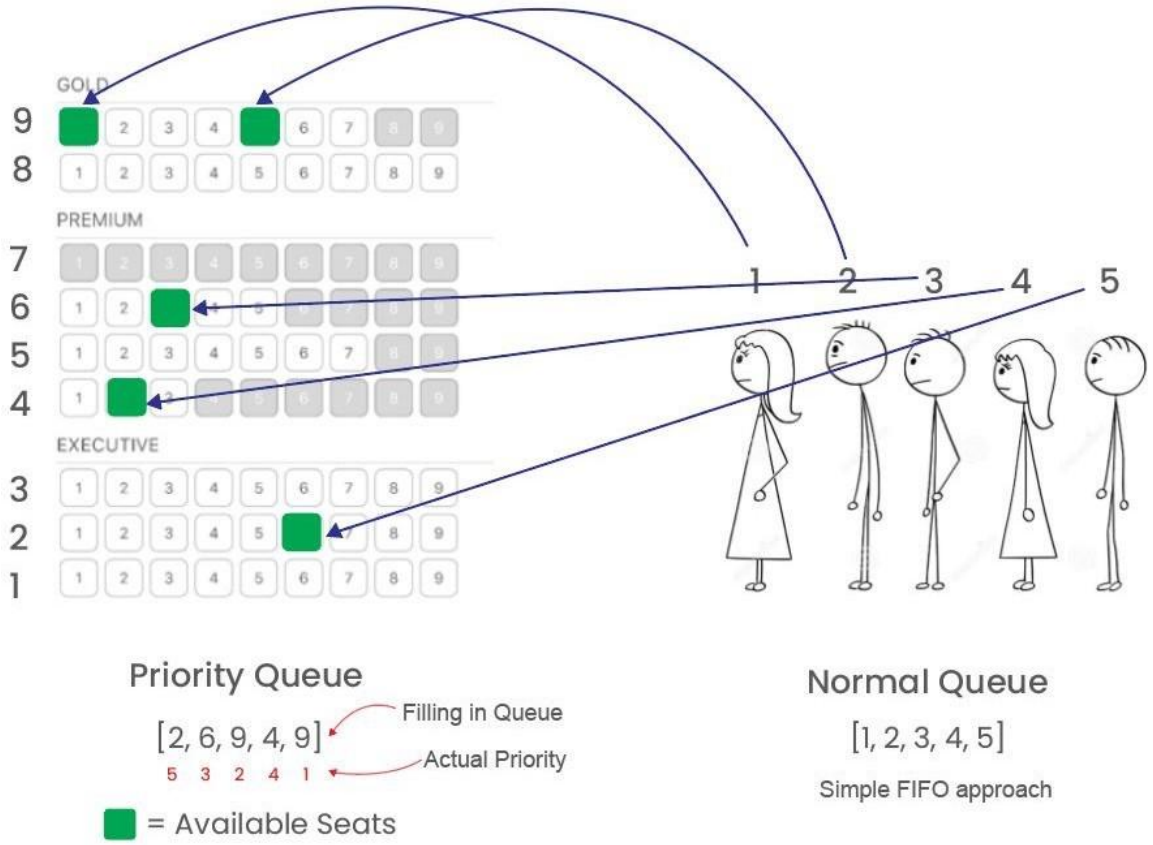
Problem Statement : Given an integer N and an array of seats[] where N is the number of people standing in a line to buy a movie ticket and seat[i] is the number of empty seats in the ith row of the movie theater. The task is to find the maximum amount a theater owner can make by selling movie tickets to N people. Price of a ticket is equal to the maximum number of empty seats among all the rows.

Algorithm :

1. Initialize queue q insert all seats array elements to the queue.
2. Tickets sold and the amount generated to be set to 0.
3. If tickets sold < N (People in the queue) and q top > 0
4. Then remove top element from queue and update total amount
5. Repeat step 3 and 4 until tickets sold = number of people in the queue.

Optimization technique : This problem can be solved by using a priority queue that will store the count of empty seats for every row and the maximum among them will be available at the top.

1. Create an empty priority_queue q and traverse the seats[] array and insert all elements into the priority_queue.
2. Initialize two integer variable ticketSold = 0 and ans = 0 that will store the number of tickets sold and the total collection of the amount so far.
3. Now check while ticketSold < N and q.top() > 0 then remove the top element from the priority_queue and update ans by adding top element of the priority queue. Also store this top value in a variable temp and insert temp – 1 back to the priority_queue.
4. Repeat these steps until all the people have been sold the tickets and print the final result.



Tool : jupyter notebook

Programming code :

```
def maxAmount(M, N, seats):
    q = []     for i in
range(M):
        q.append(seats[i])
ticketSold = 0     ans =
0
    q.sort(reverse = True)     while
(ticketSold < N and q[0] > 0):
        ans = ans + q[0]
temp = q[0]         q =
q[1:]
        q.append(temp - 1)
        q.sort(reverse = True)
ticketSold += 1     return
ans
```

```
if __name__ == '__main__':
    seats = []     rows = int(input("Enter number of rows
available : "))     for i in range(0, rows):         empty =
int(input())         seats.append(empty)
print(seats) M
= len(seats)
N = int(input("Enter the number of People standing in the queue : ")) print("Maximum
Profit generated = ", maxAmount(N, M, seats))
```

Output screen shots :

```
Enter number of rows available : 4
2
3
5
3
[2, 3, 5, 3]
Enter the number of People standing in the queue : 4
Maximum Profit generated =  15
```

Result : Successfully found out the maximum amount the theater owner can make by selling movie tickets to N people for a movie.

Experiment No:-1B
Date:- -01-2022

TIC TAC TOE PROBLEM

Problem Statement :

Two players, named ‘player1’ and ‘player2’, play a tic-tac-toe game on a grid of size ‘3 x 3’. Given an array ‘moves’ of size ‘n’, where each element of the array is a tuple of the form (row, column) representing a position on the grid. Players place their characters alternatively in the sequence of positions given in ‘moves’. Consider that ‘player1’ makes the first move. Your task is to return the winner of the game, i.e., the winning player’s name. If there is no winner and some positions remain unmarked, return ‘uncertain’. Otherwise, the game ends in a draw, i.e., when all positions are marked without any winner, return ‘draw’.

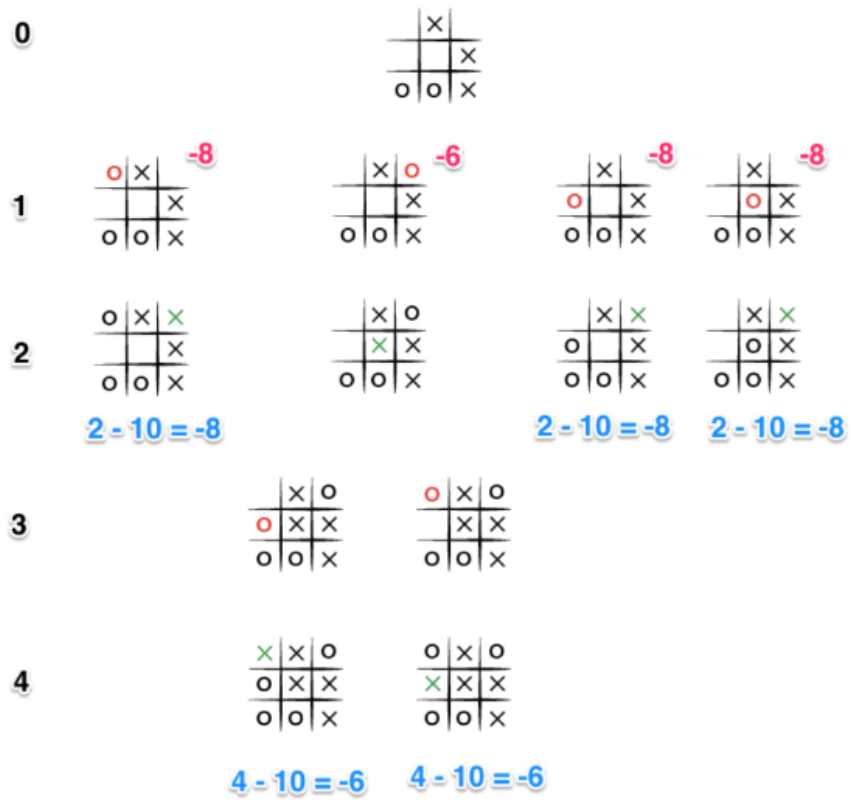
Algorithm :

- The game is to be played between two people (in this program between HUMAN and COMPUTER).
- One of the player chooses ‘O’ and the other ‘X’ to mark their respective cells.
- The game starts with one of the players and the game ends when one of the players has one whole row/ column/ diagonal filled with his/her respective character (‘O’ or ‘X’).
- If no one wins, then the game is said to be draw.

Optimization technique : The key is to use Minimax algorithm .A back and forth between the two players, where the player whose "turn it is" desires to pick the move with the maximum score. In turn, the scores for each of the available moves are determined by the opposing player deciding which of its available moves has the minimum score. And the scores for the opposing players moves are again determined by the turn-taking player trying to maximize its score and so on all the way down the move tree to an end state.

A description for the algorithm, assuming X is the "turn taking player,"

- If the game is over, return the score from X's perspective.
- Otherwise get a list of new game states for every possible move
- Create a scores list
- For each of these states add the minimax result of that state to the scores list
- If it's X's turn, return the maximum score from the scores list
- If it's O's turn, return the minimum score from the scores list



Tool : VS Code and Python 3.9.0

Programming code :

```
import random
import copy as cp
# NOTE use cp.deepcopy() so the temp variable isn't linked with the other

class Cell:
    def __init__(self, position, location, max_val, min_val):
        self.position = position
        self.location = location # NOTE this is a list, [0] is row info and [1] is col info
        self.min_val = min_val
        self.max_val = max_val

def generate_cells(board):
    uboard = cp.deepcopy(board)
    for i in range(len(uboard)):
        for j in range(len(uboard[i])):
            if uboard[i][j] != 'X' and uboard[i][j] != 'O':
```

```
            uboard[i][j] = Cell(uboard[i][j], [i,j], 0, 0)
            uboard[i][j].max_val = max_val(board, [i, j])
            uboard[i][j].min_val = min_val(board, [i, j])
            # NOTE convert uboard[i][j] into list of maxval and minval from objects
            uboard[i][j] = [uboard[i][j].position, uboard[i][j].max_val, uboard[i][j].min_val]
    return uboard

 def max_val(board, location): # NOTE only generates one max_val by a given location, not
entire board
    maxval = 0
    if board[location[0]][location[1]] != 'O' and board[location[0]][location[1]] != 'X':
        maxval += check_horizontal(board, location[0], 'max') # need row
        maxval += check_vertical(board, location[1], 'max') # need column
        # NOTE diagonal check is splitted into left and right diagonal for convienence
        maxval += left_diagonal(board, location[0], location[1], 'max')
        maxval += right_diagonal(board, location[0], location[1], 'max')

    return maxval

 def min_val(board, location):
    minval = 0
    if board[location[0]][location[1]] != 'O' and board[location[0]][location[1]] != 'X':
        minval -= check_horizontal(board, location[0], 'min')
        minval -= check_vertical(board, location[1], 'min')
        minval -= left_diagonal(board, location[0], location[1], 'min')
        minval -= right_diagonal(board, location[0], location[1], 'min')

    return minval

 def check_horizontal(board, row, u_type):
    opposed = 'X'
    sign = 'O'
    if u_type == 'min':
        opposed = 'O'
        sign = 'X'

    v = 0
    unfilled = 0
    for i in range(3): # 3 == len(board)'s row
        if board[row][i] != opposed:
            unfilled += 1
            if board[row][i] == sign:
                v += 1

    if unfilled == 3:
        if v == 2:
            v = 10
        else:
```

```
            v += 1
    elif unfilled < 3:
        v = 0
    return v

def check_vertical(board, col, u_type):
    opposed = 'X'
    sign = 'O'
    if u_type == 'min':
        opposed = 'O'
        sign = 'X'

    v = 0
    unfilled = 0
    for i in range(3): # 3 == len(board)'s column
        if board[i][col] != opposed:
            unfilled += 1
            if board[i][col] == sign:
                v += 1

    if unfilled == 3:
        if v == 2:
            v = 10
        else:
            v += 1
    elif unfilled < 3:
        v = 0
    return v

def left_diagonal(board, row, col, u_type): # NOTE top_left to bottom_right diagonal check
    opposed = 'X'
    sign = 'O'
    if u_type == 'min':
        opposed = 'O'
        sign = 'X'

    v = 0
    unfilled = 0
    if row == col:
        for i in range(3):
            if board[i][i] != opposed:
                unfilled += 1
                if board[i][i] == sign:
                    v += 1

    if unfilled == 3:
        if v == 2:
            v = 10
```

```
        else:
            v += 1
    elif unfilled < 3:
        v = 0
    return v

def right_diagonal(board, row, col, u_type):
    opposed = 'X'
    sign = 'O'
    if u_type == 'min':
        opposed = 'O'
        sign = 'X'

    v = 0
    unfilled = 0
    state = False
    for i in range(len(board)):
        if board[i][abs(i-2)] == board[row][col]:
            state = True
        if board[i][abs(i-2)] != opposed:
            unfilled += 1
            if board[i][abs(i-2)] == sign:
                v +=1

    if unfilled == 3 and state == True:
        if v == 2:
            v = 10
        else:
            v += 1
    elif unfilled < 3:
        v = 0
    return v

def dispUboard(uboard):
    print('\n')
    count = 0
    print("Utility Board:\n")
    for i in range(len(uboard)):
        for j in range(len(uboard[i])):
            count += 1
            if uboard[i][j] == 'O' or uboard[i][j] == 'X':
                print('  ',uboard[i][j],end='     ')
            else:
                print(uboard[i][j],end='  ')
            if count%3 == 0:
                print('\n')
```

```
def checkWin(board, sign):
    if checkHorizontal(board, sign) == True:
        return True
    if checkVertical(board, sign) == True:
        return True
    if checkDiagonal(board, sign) == True:
        return True
    return False

def checkTie(board):
    filled = 0
    for i in range(len(board)):
        for j in range(len(board[i])):
            if board[i][j] == 'O' or board[i][j] == 'X':
                filled += 1
    if filled == 9:
        return True
    return False

def checkDiagonal(board, sign):
    for i in range(len(board)):
        filled = 0
        if board[0][0] == sign:
            for j in range(len(board[i])):
                if board[j][j] == sign:
                    filled += 1
        elif board[0][2] == sign:
            for j in range(len(board[i])):
                if board[0+j][2-j] == sign:
                    filled += 1
    if filled == 3:
        return True
    return False

def checkHorizontal(board, sign): # NOTE BUGGY so fix it
    for i in range(len(board)):
        if board[i][0] == sign:
            filled = 0
            for j in range(len(board[i])):
                if board[i][j] == sign:
                    filled += 1
            if filled == 3:
                return True
    return False

def checkVertical(board, sign):
    for i in range(len(board)):
        if board[0][i] == sign:
```

```
            filled = 0
            for j in range(len(board[i])):
                if board[j][i] == sign:
                    filled += 1
            if filled == 3:
                return True
    return False

def dispboard(board):
    print('\n')
    count = 0
    print('Tictactoe Board:\n')
    for i in range(len(board)):
        for j in range(len(board[i])):
            count += 1
            print(board[i][j],end='  ')
            if count%3 == 0:
                print('\n')

def checkCompatible(board, move, sign):
    i = 2
    if move <= 2:
        i = 0
    elif move >= 3 and move <= 5:
        i = 1

    loc = [i,(move-(i*3))]

    if board[loc[0]][loc[1]] == move:
        board[loc[0]][loc[1]] = sign
        return True
    else:
        print("Please select an empty spot and try again.")
        return False

def computerDecision(board):
    while (checkTie(board) == False) and (checkWin(board, 'X') == False):
        uboard = generate_cells(board)
        dispUboard(uboard)
        dispboard(board)

        # TODO run minimax algorithm here
        computer_decision = minimax_algorithm(uboard)
        computer_decision = int(computer_decision)

        if checkCompatible(board, computer_decision, 'O') == True:
            if checkTie(board) == True:
                dispboard(board)
```

```
            play_again = input("\nThis is a tie game, to play again enter any key, otherwise enter 'q' to quit.\nYour decision: ")
            if play_again == 'q':
                return
            else:
                board = [[0, 1, 2],[3, 4, 5],[6, 7, 8]]
                GameInitializer(board)

        elif checkWin(board, 'O') == True:
            dispboard(board)
            print("The computer won!")
            return
        else:
            playerDecision(board)
    else:
        computerDecision(board)

def playerDecision(board):
    while (checkTie(board) == False) and (checkWin(board, 'O') == False):
        dispboard(board)
        player_decision = input("\n(The player's turn) Enter the empty position you want to place your 'X': ")
        player_decision = int(player_decision)

        if checkCompatible(board, player_decision, 'X') == True:
            if checkTie(board) == True:
                dispboard(board)
                play_again = input("\nThis is a tie game, if you want to play again enter 'p', to quit enter any key.\nYour decision: ")
                if play_again == 'q':
                    return
                else:
                    board = [[0, 1, 2],[3, 4, 5],[6, 7, 8]]
                    GameInitializer(board)

            elif checkWin(board, 'X') == True:
                dispboard(board)
                print("The player won!")
                return
            else:
                computerDecision(board)
        else:
            playerDecision(board)

def GameInitializer(board):
    choice = input("\nDo you want to go first or the computer goes first?\nEnter 'c' for computer first, or 'p' if you would like to go first\nYour Choice: ")
    if choice == 'c':
```

```
        computerDecision(board)
    elif choice == 'p':
        playerDecision(board)
    else:
        print("\nPlease enter 'c' or 'p' and try again.")
        GameInitializer(board)

def minimax_algorithm(ub): # should return a pos, such as 4, not index[1,1]
    optimal = 0
    options = []
    redundant_optimal = [] # This adds the random feature for the computer decision.
    for i in range(len(ub)):
        for j in range(len(ub[i])):
            if ub[i][j] != 'X' and ub[i][j] != 'O':
                # NOTE uboard[i][j's 0 is position, 1 is maxval, 2 is minval
                if ub[i][j][1] >= 10:
                    return ub[i][j][0]
                elif ub[i][j][2] <= -10:
                    return ub[i][j][0]
                else:
                    if abs(ub[i][j][1]) == abs(ub[i][j][2]):
                        # NOTE if abs of max = abs of min, add 1 to their sum. Why? because we want to
win more more than limiting the enemy
                        options.append([abs(ub[i][j][1]) + abs(ub[i][j][2])+1, ub[i][j][0]])
                    else: # NOTE, [0] is the total val of abs(max + min). [1] is the index
                        options.append([abs(ub[i][j][1]) + abs(ub[i][j][2]), ub[i][j][0]])

    optimal = max(options) # NOTE for redundant_optimal, [0] is index, [1] is val
    for i in range(len(options)):
        if options[i][0] == optimal[0]:
            redundant_optimal.append(options[i][1])

    redundant_optimal.append(optimal[1])
    randnum = random.randint(0,len(redundant_optimal)-1)
    return redundant_optimal[randnum]


# NOTE play game here

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

GameInitializer(init_board)
```

Output:-

```
Do you want to go first or the computer goes first?
Enter 'c' for computer first, or 'p' if you would like to go first
Your Choice: P

Please enter 'c' or 'p' and try again.

Do you want to go first or the computer goes first?
Enter 'c' for computer first, or 'p' if you would like to go first
Your Choice: p


Tictactoe Board:

0  1  2

3  4  5

6  7  8

(The player's turn) Enter the empty position you want to place your 'X': 4

Utility Board:

[0, 2, -5]  [1, 1, -4]  [2, 2, -4]

[3, 1, -4]     X     [5, 1, -4]

[6, 2, -4]  [7, 1, -4]  [8, 2, -5]


Tictactoe Board:

0  1  2

3  X  5
```

```
6  7  8


Tictactoe Board:

O  1  2

3  X  5

6  7  8

(The player's turn) Enter the empty position you want to place your 'X': 2

Utility Board:

   O     [1, 0, -4]     X

[3, 2, -4]     X     [5, 0, -6]

[6, 3, -11]  [7, 1, -5]  [8, 1, -5]


Tictactoe Board:

O  1  X

3  X  5

6  7  8
```

```
Tictactoe Board:

O  1  X

3  X  5

O  7  8

(The player's turn) Enter the empty position you want to place your 'X': 3

Utility Board:

   O     [1, 0, -2]     X

   X        X     [5, 0, -12]

   O     [7, 2, -2]  [8, 2, -2]
```

```
Tictactoe Board:

O  1  X

X  X  5

O  7  8


Tictactoe Board:

O  1  X

X  X  O

O  7  8
```

```
(The player's turn) Enter the empty position you want to place your 'X': 8

Utility Board:

   O     [1, 0, -2]     X

   X        X        O

   O     [7, 0, -2]     X


Tictactoe Board:

O  1  X

X  X  O
```

```
Tictactoe Board:

O  1  X

X  X  O

O  O  X

(The player's turn) Enter the empty position you want to place your 'X': 1

Tictactoe Board:

O  X  X

X  X  O

O  O  X
```

```
O  O  X

(The player's turn) Enter the empty position you want to place your 'X': 1

Tictactoe Board:

O  X  X

X  X  O

O  O  X


This is a tie game, if you want to play again enter 'p', to quit enter any key.
Your decision:
```

Result : The Tic Tac Toe problem was implemented successfully using minmax algorithm to evaluate the best moves with the highest score.

Experiment No:-2
Date:-24-01-2022

GRAPH COLORING PROBLEM

Problem Statement: graph coloring is a special case of graph labeling; it is an assignment of labels traditionally called "colors" to elements of a graph subject to certain constraints. The problem is, given m colors, find a way of coloring the vertices of a graph such that no two adjacent vertices are colored using same color. The other graph coloring problem is ***Edge Coloring*** (No vertex is incident to two edges of same color).

Algorithm :

1.Color first vertex with first color.

2. Do following for remaining V-1 vertices.

a) Consider the currently picked vertex and color it with the lowest numbered color that has not been used on any previously colored vertices adjacent to it.

b) If all previously used colors appear on vertices adjacent to v, assign a new color to it.

Optimization technique : The idea is to assign colors one by one to different vertices, starting from the vertex 0. Before assigning a color, check for safety by considering already assigned colors to the adjacent vertices i.e check if the adjacent vertices have the same color or not. If there is any color assignment that does not violate the conditions, mark the color assignment as part of the solution. If no assignment of color is possible then backtrack and return false.

Algorithm:

1. Create a recursive function that takes the graph, current index, number of vertices, and output color array.
2. If the current index is equal to the number of vertices. Print the color configuration in output array.
3. Assign a color to a vertex (1 to m).

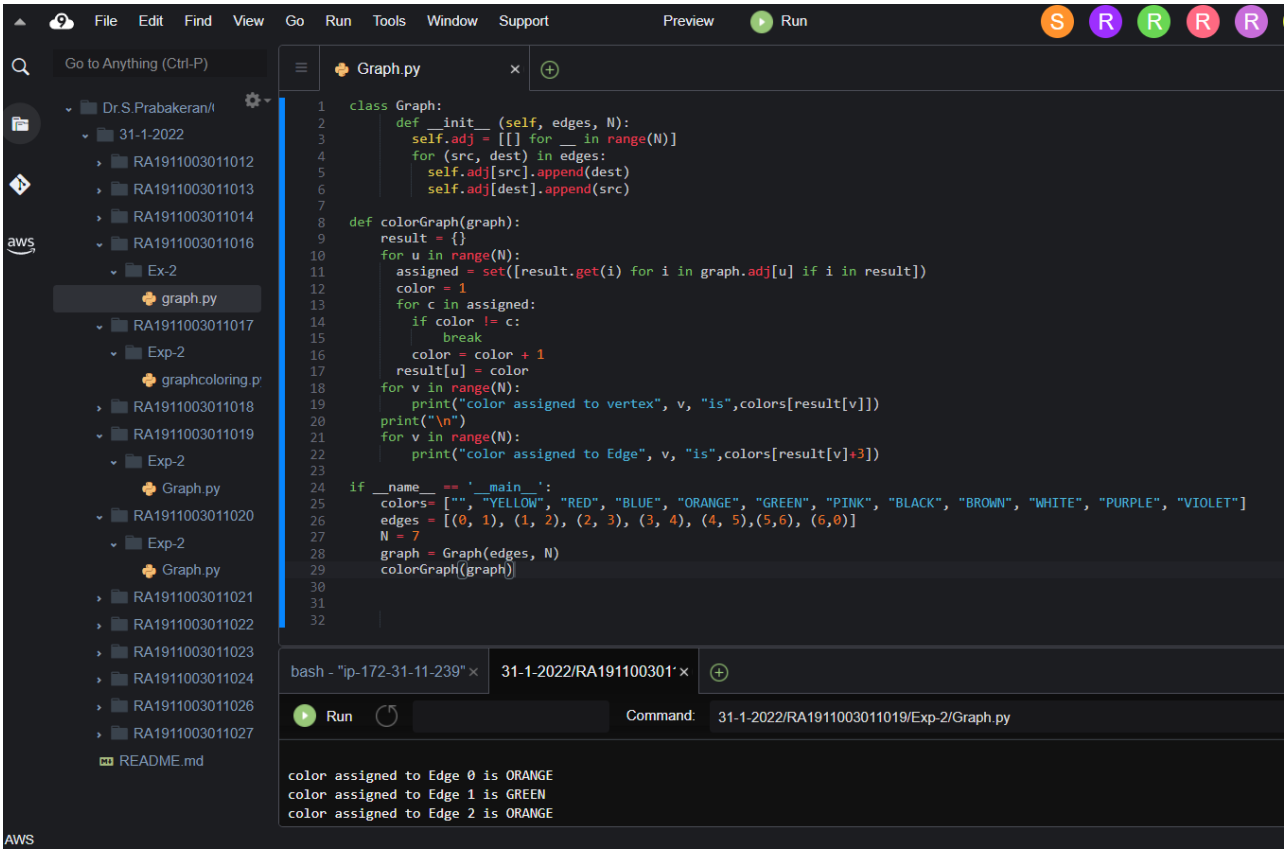
4. For every assigned color, check if the configuration is safe, (i.e. check if the adjacent vertices do not have the same color) recursively call the function with next index and number of vertices
5. If any recursive function returns true break the loop and return true.
6. If no recursive function returns true then return false.

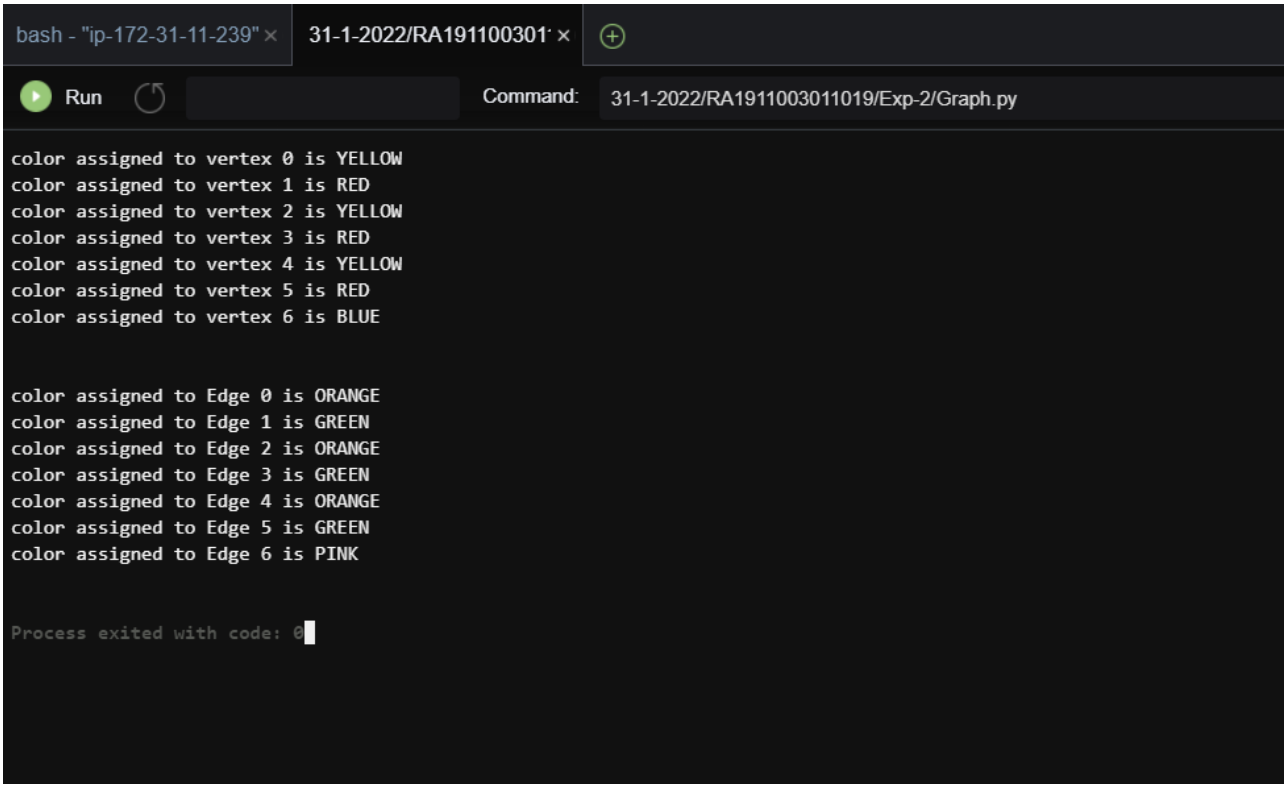
Tool : Cloud9 ide and Python 3.9.0

Programming code :

```
class Graph:
    def _init_ (self, edges, N):
      self.adj = [[] for _ in range(N)]
      for (src, dest) in edges:
        self.adj[src].append(dest)
        self.adj[dest].append(src)


def colorGraph(graph):
   result = {}
   for u in range(N):
     assigned = set([result.get(i) for i in graph.adj[u] if i in result])
     color = 1
     for c in assigned:
       if color != c:
          break
       color = color + 1
     result[u] = color
   for v in range(N):
      print("color assigned to vertex", v, "is",colors[result[v]])
   print("\n")
   for v in range(N):
      print("color assigned to Edge", v, "is",colors[result[v]+3])

if _name_ == '_main_':
   colors= ["", "YELLOW", "RED", "BLUE", "ORANGE", "GREEN", "PINK", "BLACK",
"BROWN", "WHITE", "PURPLE", "VIOLET"]
edges=[(0,1),(1,2),(2,3),(3,4),(4,5),(5,6),(6,0)]
   N = 7
   graph = Graph(edges, N)
   colorGraph(graph)
```

Output screen shots:





Result : A unique color was successfully assigned to each vertex and edge of the graph.

Experiment No:-3
Date:-01-02-2022

Implementation of constraint satisfaction problem (Cryptarithmetic Problem)

Algorithm :

- First, create a list of all the characters that need assigning to pass to Solve
- If all characters are assigned, return true if puzzle is solved, false otherwise
- Otherwise, consider the first unassigned character
- for (every possible choice among the digits not in use)

make that choice and then recursively try to assign the rest of the characters
if recursion successful, return true
if !successful, unmake assignment and try another digit

Optimization technique : The algorithm above actually has a lot in common with the permutations algorithm, it pretty much just creates all arrangements of the mapping from characters to digits and tries each until one works or all have been successfully tried. For a large puzzle, this could take a while.
A smarter algorithm could take into account the structure of the puzzle and avoid going down dead-end paths. For example, if we assign the characters starting from the one's place and moving to the left, at each stage, we can verify the correctness of what we have so far before we continue onwards. This definitely complicates the code but leads to a tremendous improvement in efficiency, making it much more feasible to solve large puzzles.

- Start by examining the rightmost digit of the topmost row, with a carry of 0
- If we are beyond the leftmost digit of the puzzle, return true if no carry, false otherwise
- If we are currently trying to assign a char in one of the addends
  If char already assigned, just recur on the row beneath this one, adding value into the sum
  If not assigned, then

    - for (every possible choice among the digits not in use)
      make that choice and then on row beneath this one, if successful, return true
      if !successful, unmake assignment and try another digit
    - return false if no assignment worked to trigger backtracking
- Else if trying to assign a char in the sum
- If char assigned & matches correct,
  recur on next column to the left with carry, if success return true,
- If char assigned & doesn't match, return false
- If char unassigned & correct digit already used, return false
- If char unassigned & correct digit unused,
  assign it and recur on next column to left with carry, if success return true
- return false to trigger backtracking.

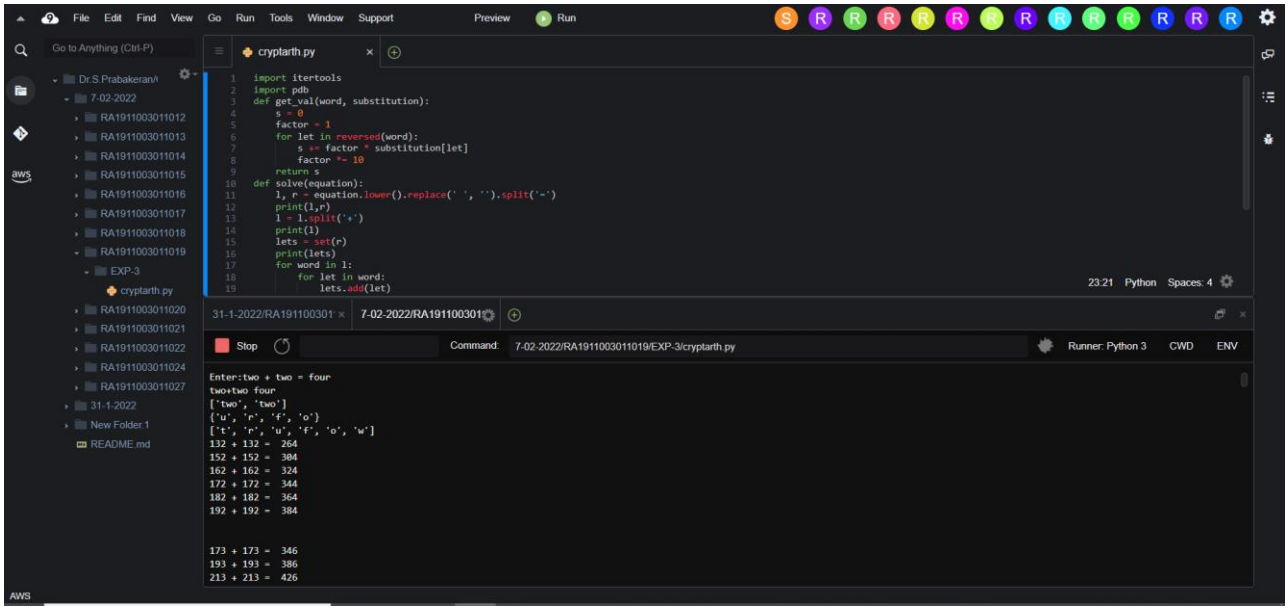
Tool : aws cloud9 and Python 3.9.0

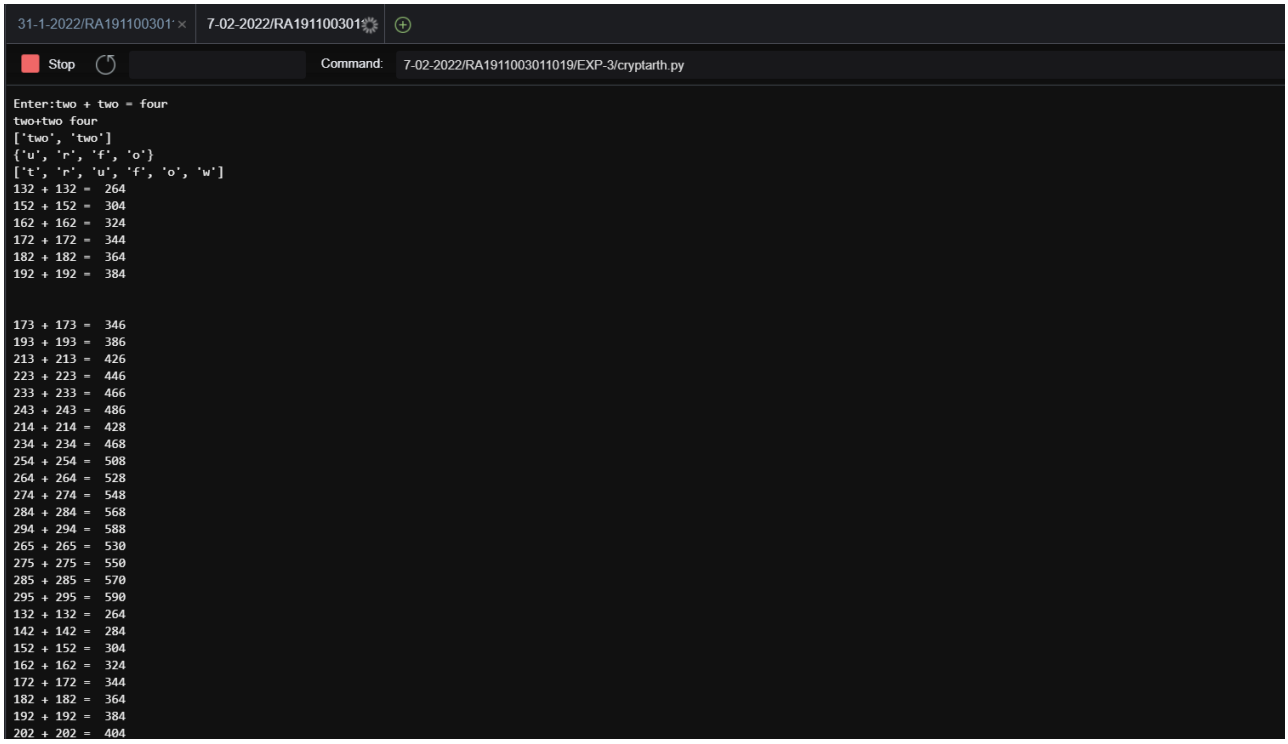
Programming code :

```
import itertools

import pdb

def get_val(word, substitution):

    s = 0

    factor = 1

    for let in reversed(word):

        s += factor * substitution[let]

        factor *= 10

    return s

def solve(equation):

    l, r = equation.lower().replace(' ', '').split('=')

    print(l,r)
```

```
    l = l.split('+')
    print(l)
    lets = set(r)
    print(lets)
    for word in l:
        for let in word:
            lets.add(let)
    lets = list(lets)
    print(lets)
    digits = range(20)
    for perm in itertools.permutations(digits, len(lets)):
        sol = dict(zip(lets, perm))

        if sum(get_val(word, sol) for word in l) == get_val(r, sol):
            print(' + '.join(str(get_val(word, sol)) for word in l) + " = ",get_val(r, sol))



equation = input("Enter:")
solve(equation)
```

Output screen shots :





Result : Successfully solved the given constraint satisfaction problem.

Experiment No:-4
Date:-08-02-2022

Implementation And Analysis Of DFS And BFS

Breadth-First Search : Breadth-First Search (BFS) is an algorithm used for traversing graphs or trees. Traversing means visiting each node of the graph. Breadth-First Search is a recursive algorithm to search all the vertices of a graph or a tree. BFS in python can be implemented by using data structures like a dictionary and lists. Breadth-First Search in tree and graph is almost the same. The only difference is that the graph may contain cycles, so we may traverse to the same node again.

Algorithm :

1. Start by putting any one of the graph's vertices at the back of the queue.
2. Now take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add those which are not within the visited list to the rear of the queue.
4. Keep continuing steps two and three till the queue is empty.

The Depth-First Search : The Depth-First Search is a recursive algorithm that uses the concept of backtracking. It involves thorough searches of all the nodes by going ahead if potential, else by backtracking. Here, the word backtrack means once you are moving forward and there are not any more nodes along the present path, you progress backward on an equivalent path to seek out nodes to traverse. All the nodes are progressing to be visited on the current path until all the unvisited nodes are traversed after which subsequent paths are going to be selected.

Algorithm :

1. We will start by putting any one of the graph's vertex on top of the stack.
2. After that take the top item of the stack and add it to the visited list of the vertex.
3. Next, create a list of that adjacent node of the vertex. Add the ones which aren't in the visited list of vertexes to the top of the stack.
4. Lastly, keep repeating steps 2 and 3 until the stack is empty.

Tool : Aws Cloud9 and Python 3.9.0

Programming code :

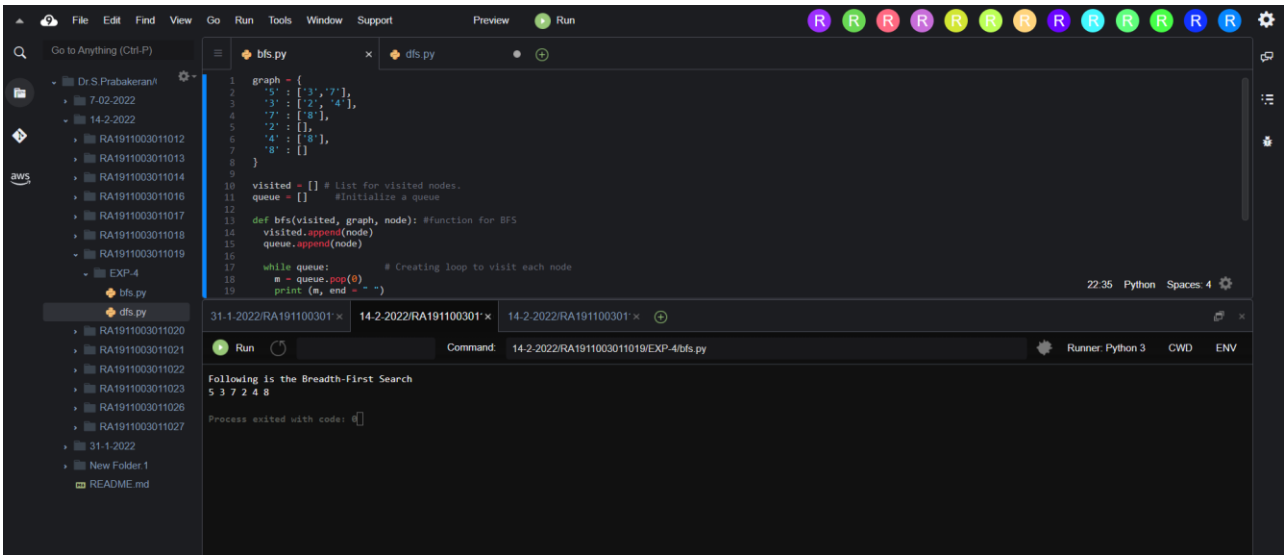
BFS:

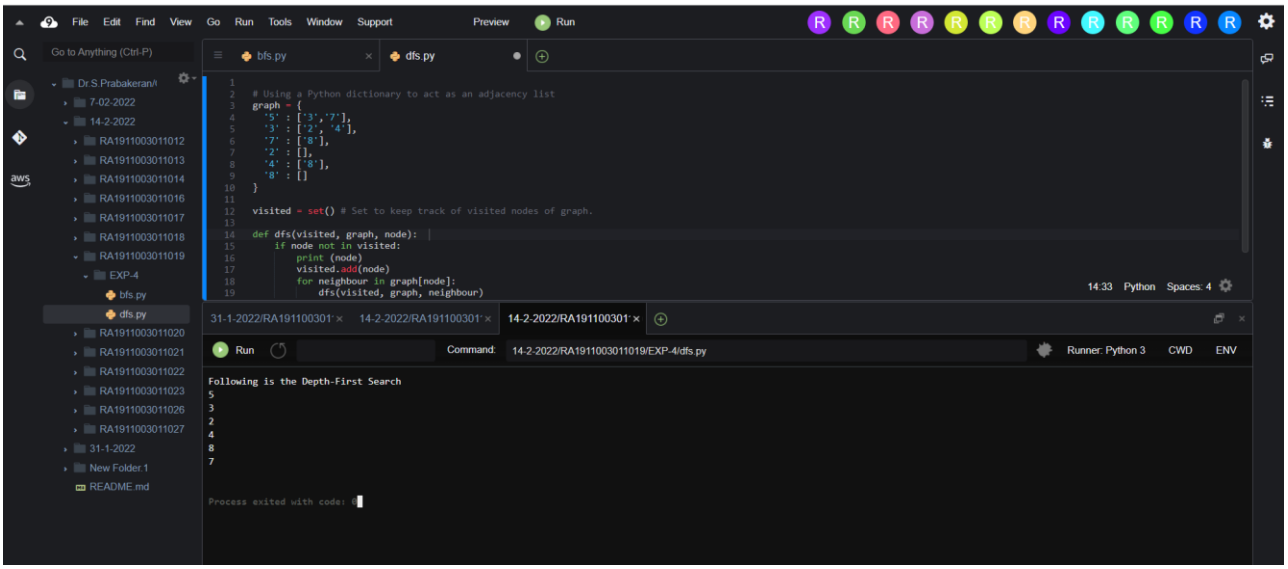
```
graph = {
'5' : ['3','7'],
'3' : ['2', '4'],
'7' : ['8'],
'2' : [],
'4' : ['8'],
'8' : []
}

visited = [] # List for visited nodes.
queue = []     #Initialize a queue

defbfs(visited, graph, node): #function for BFS
  visited.append(node)
  queue.append(node)

while queue:          # Creating loop to visit each node
    m = queue.pop(0)
print (m, end =" ")

for neighbour in graph[m]:
if neighbour notin visited:
        visited.append(neighbour)
        queue.append(neighbour)

# Driver Code
print("Following is the Breadth-First Search")
bfs(visited, graph, '5')    # function calling
```

DFS:

```
# Using a Python dictionary to act as an adjacency list
graph = {
'5' : ['3','7'],
'3' : ['2', '4'],
'7' : ['8'],
'2' : [],
'4' : ['8'],
'8' : []
}

visited =set() # Set to keep track of visited nodes of graph.

defdfs(visited, graph, node):  #function for dfs
if node notin visited:
print (node)
        visited.add(node)
for neighbour in graph[node]:
            dfs(visited, graph, neighbour)

# Driver Code
print("Following is the Depth-First Search")
dfs(visited, graph, '5')
```

Output screen shots :





Result : Successfully Implemented BFS and DFS.

Experiment No:-5
Date:-15-02-2022

Best First Search (Informed Search)

In BFS and DFS, when we are at a node, we can consider any of the adjacent as next node. So both BFS and DFS blindly explore paths without considering any cost function. The idea of Best First Search is to use an evaluation function to decide which adjacent is most promising and then explore. Best First Search falls under the category of Heuristic Search or Informed Search.

We use a priority queue to store costs of nodes. So the implementation is a variation of BFS, we just need to change Queue to Priority Queue.

Algorithm :

```
1) Create an empty PriorityQueue
      PriorityQueue pq;
2) Insert "start" in pq.
      pq.insert(start)
   3) Until PriorityQueue is empty
u = PriorityQueue.DeleteMin
        If u is the goal
          Exit
        Else
          Foreach neighbor v of u
            If v "Unvisited"
               Mark v "Visited"
pq.insert(v)
          Mark u "Examined"
End procedure
```

A* Algorithm

A heuristic algorithm sacrifices optimality, with precision and accuracy for speed, to solve problems faster and more efficiently.

All graphs have different nodes or points which the algorithm has to take, to reach the final node. The paths between these nodes all have a numerical value, which is considered as the weight of the path. The total of all paths transverse gives you the cost of that route.

Initially, the Algorithm calculates the cost to all its immediate neighboring nodes,n, and chooses the one incurring the least cost. This process repeats until no new nodes can be

chosen and all paths have been traversed. Then, you should consider the best path among them. If f(n) represents the final cost, then it can be denoted as :
f(n) = g(n) + h(n), where :

g(n) = cost of traversing from one node to another. This will vary from node to node

h(n) = heuristic approximation of the node's value. This is not a real value but an approximation cost

Algorithm

- Make an open list containing starting node ○ If it reaches the destination node : ○ Make a closed empty list
  - If it does not reach the destination node, then consider a node with the lowest f-score in the open list

We are finished

- Else :

Put the current node in the list and check its neighbors

- For each neighbor of the current node :
  - If the neighbor has a lower g value than the current node and is in the closed list:

Replace neighbor with this new node as the neighbor's parent

- Else If (current g is lower and neighbor is in the open list):

Replace neighbor with the lower g value and change the neighbor's parent to the current node.

- Else If the neighbor is not in both lists:

Add it to the open list and set its g

Tool : VS Code and Python 3.9.0

Programming code :

A-star

```
# graph class class Graph:

   # init class     def __init__(self, graph_dict=None,
directed=True):
       self.graph_dict = graph_dict or {}
       self.directed = directed         if
not directed:
           self.make_undirected()

   # create undirected graph by adding symmetric edges     def
make_undirected(self):
       for a in list(self.graph_dict.keys()):             for (b,
dist) in self.graph_dict[a].items():
self.graph_dict.setdefault(b, {})[a] = dist

   # add link from A and B of given distance, and also add the inverse link if the graph is undirected
def connect(self, A, B, distance=1):
       self.graph_dict.setdefault(A, {})[B] = distance         if
not self.directed:
           self.graph_dict.setdefault(B, {})[A] = distance

   # get neighbors or a neighbor     def
get(self, a, b=None):
       links = self.graph_dict.setdefault(a, {})
if b is None:             return links         else:
           return links.get(b)

   # return list of nodes in the graph     def nodes(self):
s1 = set([k for k in self.graph_dict.keys()])         s2 =
set([k2 for v in self.graph_dict.values() for k2, v2 in
v.items()])         nodes = s1.union(s2)
       return list(nodes)

# node class class
Node:

   # init class     def __init__(self, name:str,
parent:str):         self.name = name
self.parent = parent         self.g = 0 # distance to
```

```
start node        self.h = 0 # distance to goal
node
        self.f = 0 # total cost

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

    # sort nodes     def
__lt__(self, other):          return
self.f < other.f

    # print node     def
__repr__(self):
        return ('({0},{1})'.format(self.name, self.f))

# A* search def astar_search(graph, heuristics, start,
end):

    # lists for open nodes and closed nodes
    open = []
    closed = []

    # a start node and an goal node     start_node =
Node(start, None)     goal_node = Node(end, None)

    # add start node
    open.append(start_node)

    # loop until the open list is empty     while
len(open) > 0:


        open.sort()                          # sort open list to get the node with the lowest cost first
        current_node = open.pop(0)                  # get node with the lowest cost
closed.append(current_node)               # add current node to the closed list

        # check if we have reached the goal, return the path         if current_node
== goal_node:             path = []             while current_node != start_node:
path.append(current_node.name + ': ' + str(current_node.g))
current_node = current_node.parent
            path.append(start_node.name + ': ' + str(start_node.g))             return
path[::-1]


        neighbors = graph.get(current_node.name)    # get neighbours

        # loop neighbors         for key, value in
neighbors.items():
            neighbor = Node(key, current_node)      # create neighbor node             if(neighbor in
closed):                # check if the neighbor is in the closed list                 continue

            # calculate full path cost
```

```
        neighbor.g = current_node.g + graph.get(current_node.name, neighbor.name)
neighbor.h = heuristics.get(neighbor.name)          neighbor.f = neighbor.g + neighbor.h

        # check if neighbor is in open list and if it has a lower f value
if(add_to_open(open, neighbor) == True):

            # everything is green, add neighbor to open list              open.append(neighbor)

   # return None, no path is found     return
None

# check if a neighbor should be added to open list def
add_to_open(open, neighbor):     for node in open:         if
(neighbor == node and neighbor.f > node.f):
        return False     return
True

# create a graph graph =
Graph()  # create graph
connections (Actual
distance)
graph.connect('S', 'A', 2) graph.connect('S', 'G',
20) graph.connect('A', 'C', 7) graph.connect('C',
'G', 8) graph.connect('C', 'D', 9)
graph.connect('D', 'G', 10)
# make graph undirected, create symmetric connections graph.make_undirected()
# create heuristics (straight-line distance, air-travel distance)
heuristics = {} heuristics['A'] = 5 heuristics['C'] = 8 heuristics['G'] =
7 heuristics['D'] = 6 heuristics['S'] = 9 # run the search algorithm path
= astar_search(graph, heuristics, 'S', 'G') print("Path:", path)
```

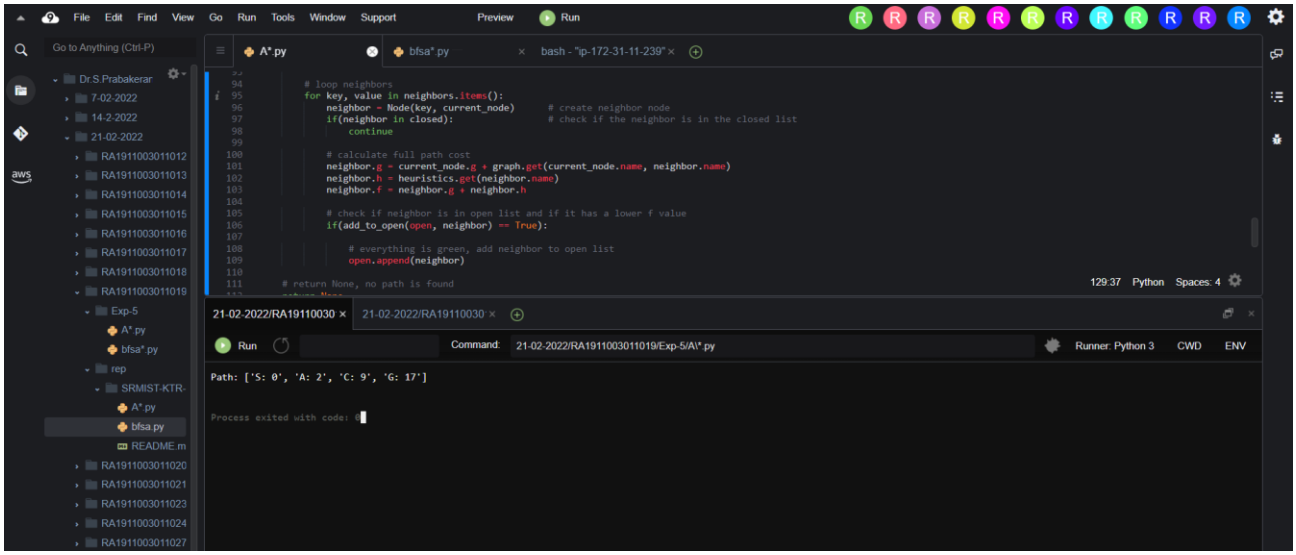
Best First Search

```
from queue import PriorityQueue v = 5 graph = [[] for i in range(v)] def best_first_search(source, target, n):
  visited = [0] * n   visited[0] =
True   pq = PriorityQueue()
pq.put((0, source))   while
pq.empty() == False:
   u = pq.get()[1]
```

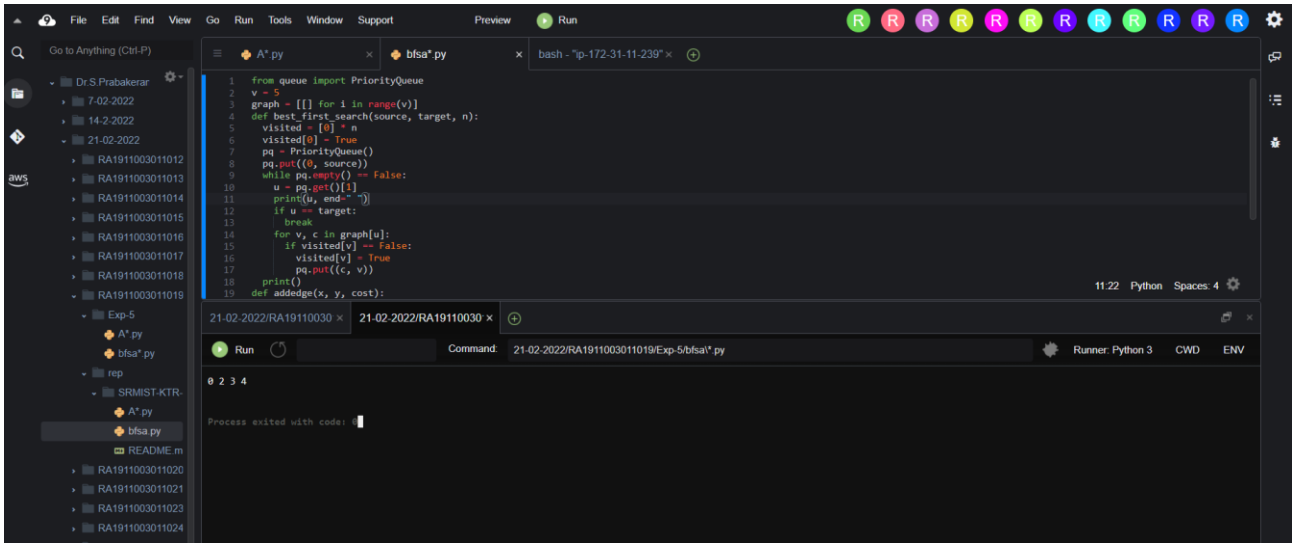
```
print(u, end=" ")     if u
== target:
     break     for v, c in graph[u]:
if visited[v] == False:
visited[v] = True
pq.put((c, v))   print() def
addedge(x, y, cost):
graph[x].append((y, cost))
graph[y].append((x, cost))
addedge(0, 1, 5) addedge(0, 2,
1) addedge(2, 3, 2) addedge(1,
4, 1) addedge(3, 4, 2) source =
0 target = 4
best_first_search(source, target, v)
```

Output screen shots :

A*



BFSA :



Result : A* and Best first search algorithms were implemented successfully.

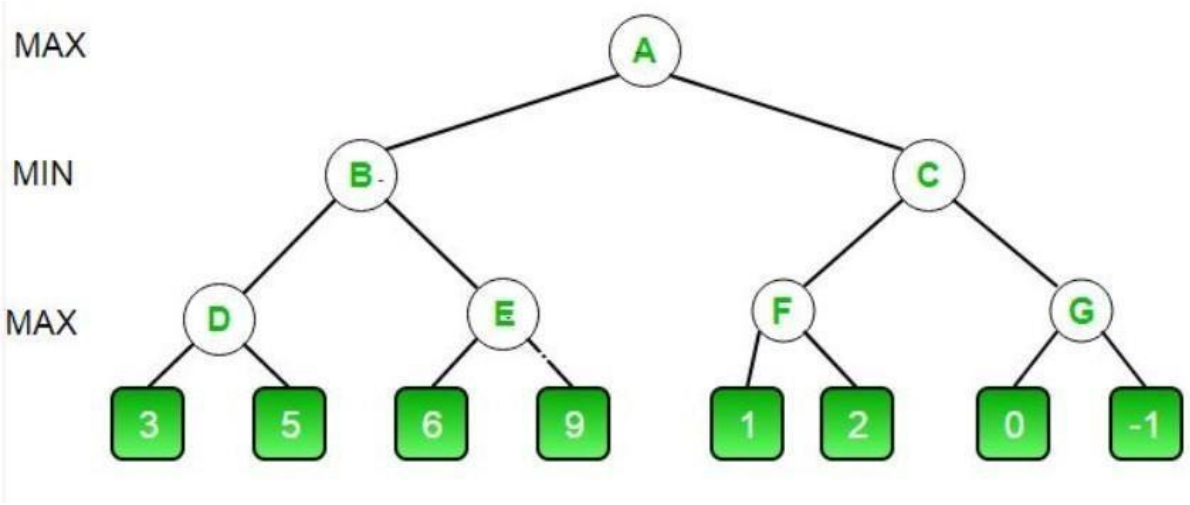
Experiment No: 6

Date : 22-02-2022

MINIMAX ALGORITHM

AIM : Developing a mini max algorithm for real world problems.

PROBLEM : Find the optimal value in the given tree of integer values in the most optimal way possible under the time complexity O(B^D).



ALGORITHM MINIMAX APPROACH :

1. Start traversing the given tree in top to bottom manner.
2. If node is a leaf node then return the value of the node.
3. If isMaximizingPlayer exist then bestVal = -INFINITY
4. For each child node, value = minimax(node, depth+1, false, alpha, beta)
5. bestVal = max(bestVal, value) and alpha = max(alpha, bestVal)
6. If beta <= alpha then stop traversing and return bestVal
7. Else, bestVal = +INFINITY
8. For each child node, value = minimax(node, depth+1, true, alpha, beta)
9. bestVal = min(bestVal, value) and beta = min(beta, bestVal)

10. if beta <= alpha the stop traversing and return bestVal

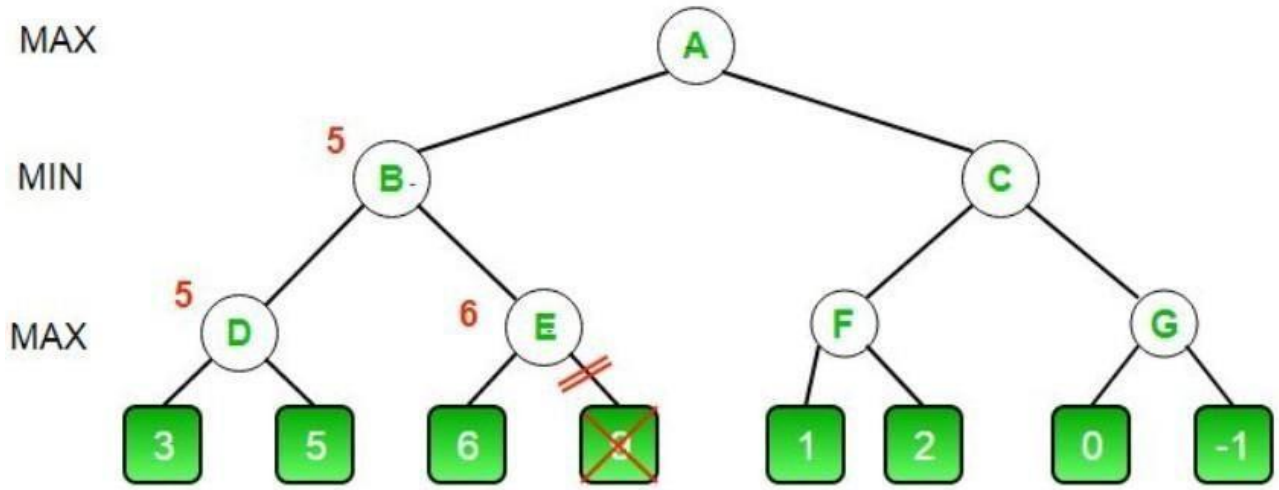
OPTIMIZATION TECHNIQUE :

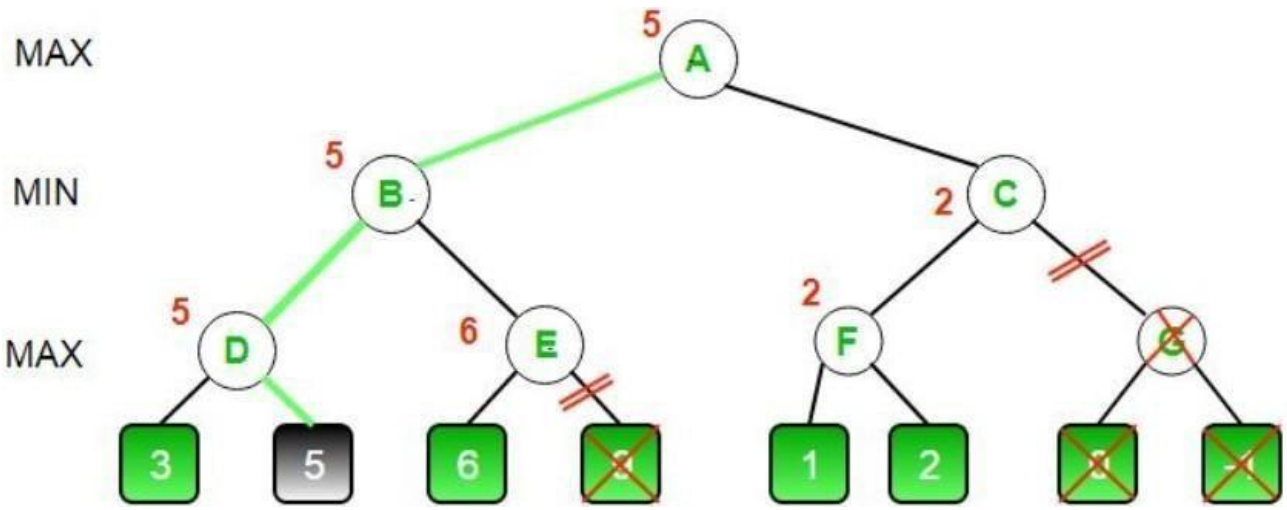
Alpha-Beta pruning is not actually a new algorithm, rather an optimization technique for minimax algorithms. It reduces the computation time by a huge factor. This allows us to search much faster and even go into deeper levels in the game tree. It cuts off branches in the game tree which need not be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

Let's define the parameters alpha and beta.

Alpha is the best value that the **maximizer** currently can guarantee at that level or above.

Beta is the best value that the **minimizer** currently can guarantee at that level or above.



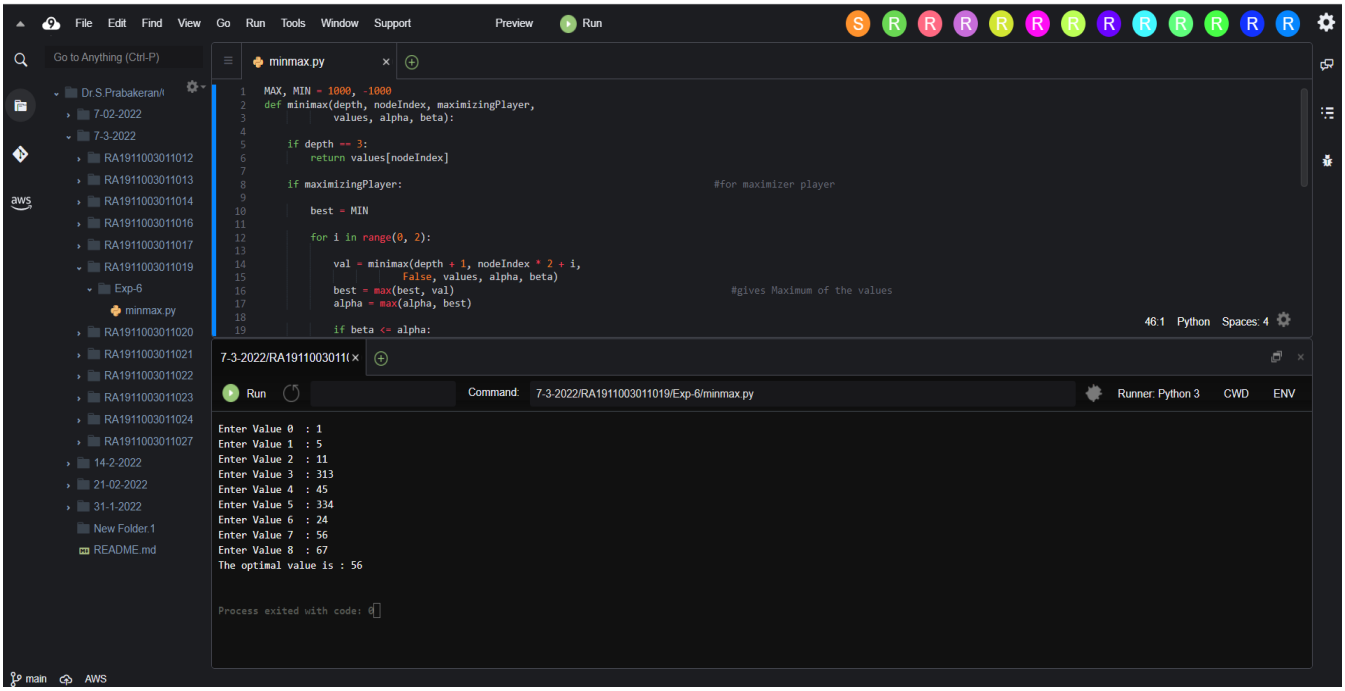


CODE (MINIMAX ALGORITHM) :

```
MAX, MIN = 1000, -1000  def
minimax(depth, nodeIndex,
maximizingPlayer,
values, alpha, beta):

        if depth == 3:
                return values[nodeIndex]
        if
maximizingPlay
er:

                best = MIN
                for i
in range(0, 2):

                        val = minimax(depth + 1, nodeIndex * 2 + i,
                                                False, values,
alpha, beta)                    best = max(best, val)
                        alpha = max(alpha, best)
                        if
beta <= alpha:
                                break

                return best



        else:
```

```
                best =
MAX             for i
in range(0, 2):
        val =
minimax(depth + 1,
nodeIndex * 2 + i,
                                                    True,
values, alpha, beta)                    best = min(best, val)
        beta = min(beta, best)                  if beta <= alpha:
                        break

                return best

if __name__ == "__main__":

   values
= []
for i in
range(0,
8):

      x = int(input(f"Enter Value {i}  : "))
values.append(x)
   print ("The optimal value is :", minimax(0, 0, True, values, MIN,
MAX))
```

OUTPUT :



RESULT : The Optimal value of the given tree successfully found using Minimax Algorithm with Alpha Beta Pruning in time complexity O(B^D).

Experiment No : 7

Date:- 14-03-2022

IMPLEMENTATION OF UNCERTAIN METHODS OF AN APPLICATION

Problem Statement:

To implement Fuzzy logic using matplotlib in python and find the graph of temperature, humidity and speed in different conditions.

Algorithm:

1. Locate the input, output, and state variables of the plane under consideration.
2. Split the complete universe of discourse spanned by each variable into a number of fuzzy subsets, assigning each with a linguistic label. The subsets include all the elements in the universe.
3. Obtain the membership function for each fuzzy subset.
4. Assign the fuzzy relationships between the inputs or states of fuzzy subsets on one side and the output of fuzzy subsets on the other side, thereby forming the rule base.
5. Choose appropriate scaling factors for the input and output variables for normalizing the variables between [0, 1] and [-1, I] interval.
6. Carry out the fuzzification process.
7. Identify the output contributed from each rule using fuzzy approximate reasoning.
8. Combine the fuzzy outputs obtained from each rule.
9. Finally, apply defuzzification to form a crisp output.

Optimization Technique:

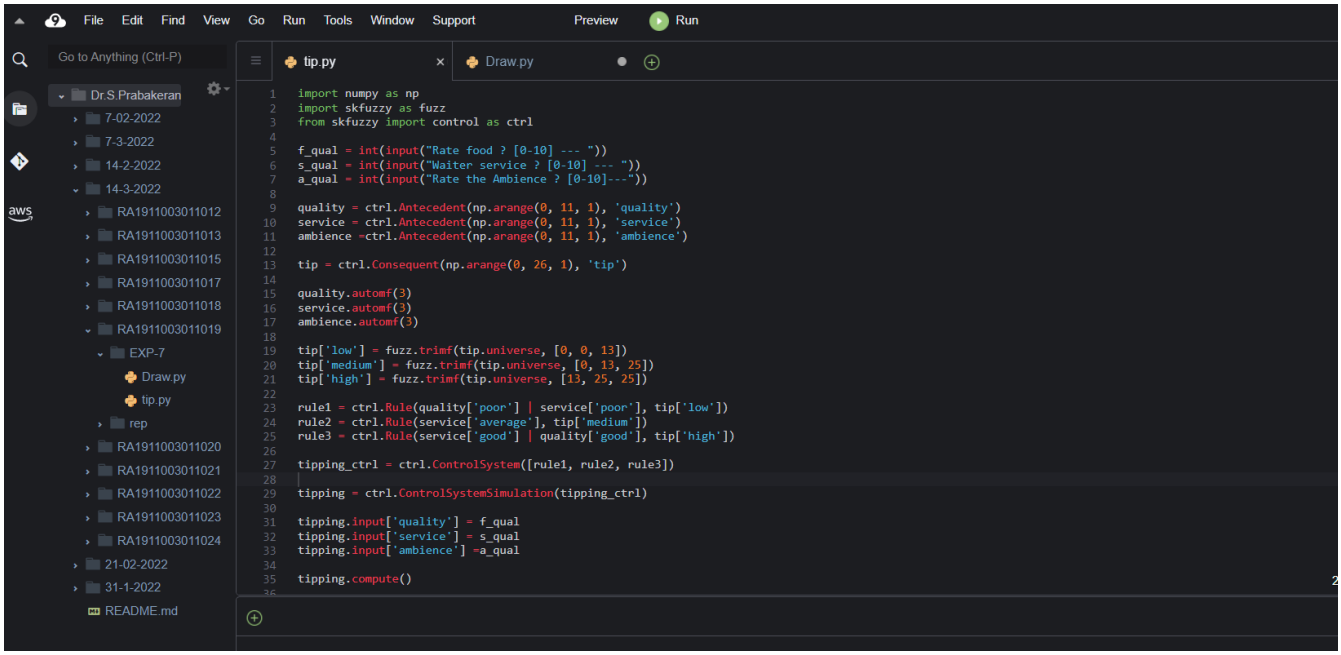
1. Decomposing the large-scale system into a collection of various subsystems.
2. Varying the plant dynamics slowly and linearizing the nonlinear plane dynamics about a set of operating points.
3. Organizing a set of state variables, control variables, or output features for the system under consideration.
4. Designing simple P, PD, PID controllers for the subsystems. Optimal controllers can also be designed.

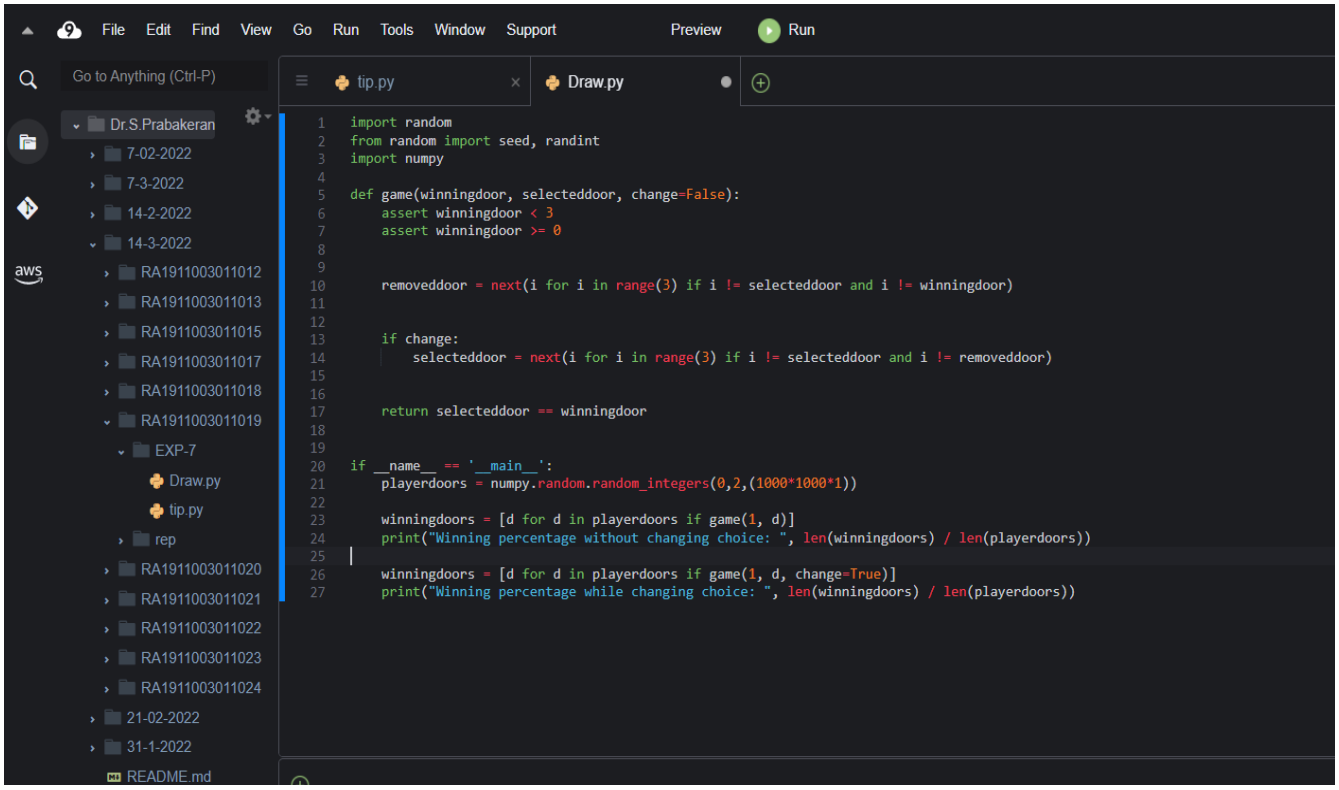
Uncertainty In this problem : Fuzzy Logic - Temperature, Humidity.

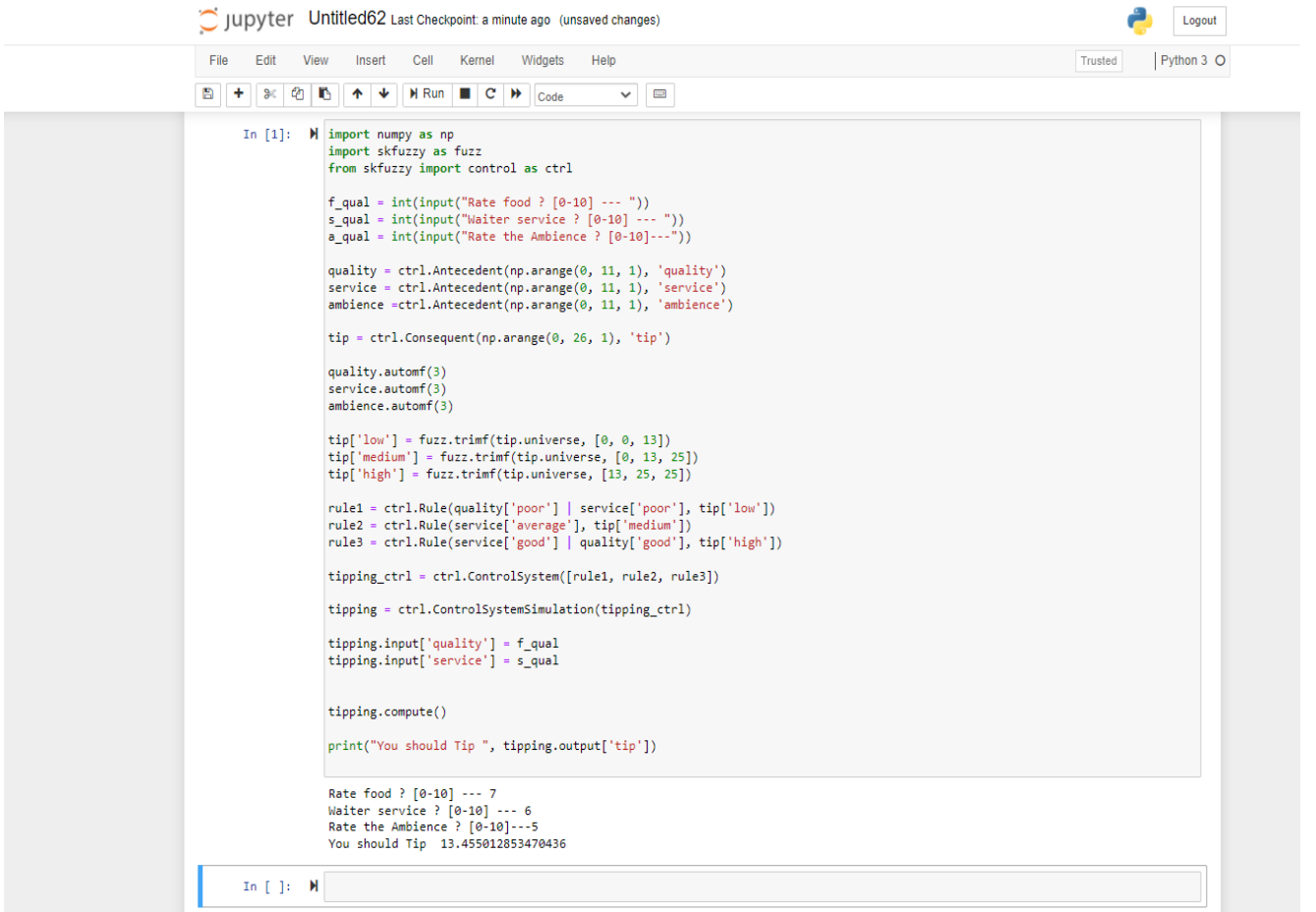
CODE :
```
M = 9
def puzzle(a):
for i in range(M):
for j in range(M):
print(a[i][j],end =
" ")
        print() def
solve(grid, row, col,
num):
    for x in
range(9):         if
grid[row][x] ==
num:
            return False

    for x in
range(9):         if
grid[x][col] ==
num:
            return False


    startRow = row - row % 3     startCol
= col - col % 3     for i in range(3):
for j in range(3):              if grid[i +
startRow][j + startCol] == num:
                return
False     return
True

def Suduko(grid, row, col):

    if (row == M - 1 and col == M):
        return True     if col ==
M:         row += 1         col =
0     if grid[row][col] > 0:
return Suduko(grid, row, col
+ 1)     for num in range(1,
M + 1, 1):

        if solve(grid, row, col, num):

            grid[row][col] =
num              if
Suduko(grid, row, col + 1):
                return
True
grid[row][col] = 0
return False

'"0 means the cells where no value is assigned"'
```

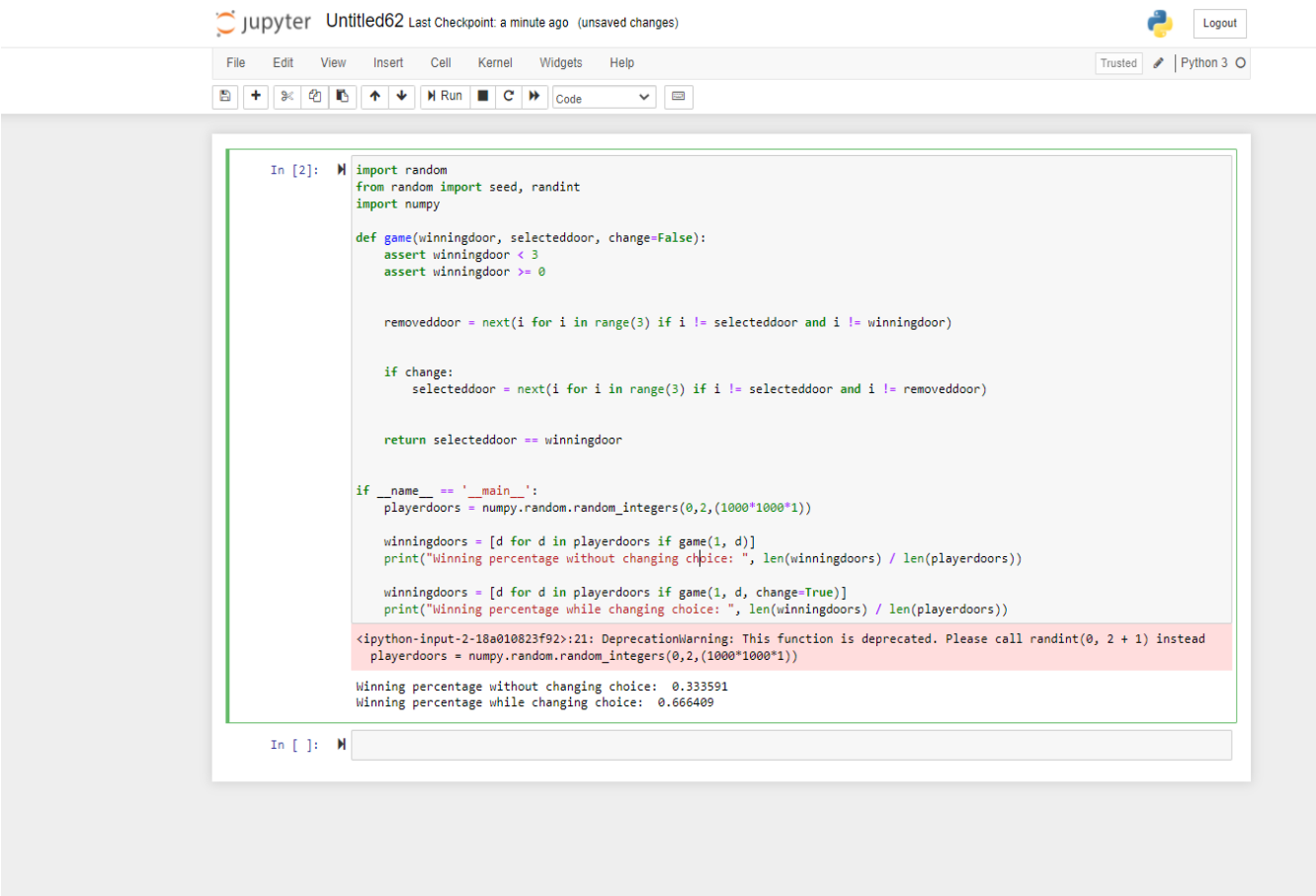
```
grid = [[2, 5, 0, 0, 3, 0, 9, 0, 1],
        [0, 1, 0, 0, 0, 4, 0, 0, 0],
    [4, 0, 7, 0, 0, 0, 2, 0, 8],
    [0, 0, 5, 2, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 9, 8, 1, 0, 0],
    [0, 4, 0, 0, 0, 3, 0, 0, 0],
    [0, 0, 0, 3, 6, 0, 0, 7, 2],
    [0, 7, 0, 0, 0, 0, 0, 0, 3],
    [9, 0, 3, 0, 0, 0, 6,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

if
(Suduko(gri
d, 0, 0)):
puzzle(grid
) else:
    print("Solution does not exist:(")
```

OUTPUT :




File Edit Find View Go Run Tools Window Support Preview Run
Go to Anything (Ctrl-P)
tip.py
Draw.py
Dr.S.Prabakeran
7-02-2022
7-3-2022
14-2-2022
14-3-2022
RA1911003011012
RA1911003011013
RA1911003011015
RA1911003011017
RA1911003011018
RA1911003011019
EXP-7
Draw.py
tip.py
rep
RA1911003011020
RA1911003011021
RA1911003011022
RA1911003011023
RA1911003011024
21-02-2022
31-1-2022
README.md
import random
from random import seed, randint
import numpy
def game(winningdoor, selecteddoor, change=False):
assert winningdoor < 3
assert winningdoor >= 0
removeddoor = next(i for i in range(3) if i != selecteddoor and i != winningdoor)
if change:
selecteddoor = next(i for i in range(3) if i != selecteddoor and i != removeddoor)
return selecteddoor == winningdoor
if __name__ == '__main__':
playerdoors = numpy.random.random_integers(0,2,(1000*1000*1))
winningdoors = [d for d in playerdoors if game(1, d)]
print("Winning percentage without changing choice: ", len(winningdoors) / len(playerdoors))
winningdoors = [d for d in playerdoors if game(1, d, change=True)]
print("Winning percentage while changing choice: ", len(winningdoors) / len(playerdoors))


jupyter Untitled62 Last Checkpoint: a minute ago (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code
In [1]:
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl
f_qual = int(input("Rate food ? [0-10] --- "))
s_qual = int(input("Waiter service ? [0-10] --- "))
a_qual = int(input("Rate the Ambience ? [0-10]---"))
quality = ctrl.Antecedent(np.arange(0, 11, 1), 'quality')
service = ctrl.Antecedent(np.arange(0, 11, 1), 'service')
ambience =ctrl.Antecedent(np.arange(0, 11, 1), 'ambience')
tip = ctrl.Consequent(np.arange(0, 26, 1), 'tip')
quality.automf(3)
service.automf(3)
ambience.automf(3)
tip['low'] = fuzz.trimf(tip.universe, [0, 0, 13])
tip['medium'] = fuzz.trimf(tip.universe, [0, 13, 25])
tip['high'] = fuzz.trimf(tip.universe, [13, 25, 25])
rule1 = ctrl.Rule(quality['poor'] | service['poor'], tip['low'])
rule2 = ctrl.Rule(service['average'], tip['medium'])
rule3 = ctrl.Rule(service['good'] | quality['good'], tip['high'])
tipping_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])
tipping = ctrl.ControlSystemSimulation(tipping_ctrl)
tipping.input['quality'] = f_qual
tipping.input['service'] = s_qual
tipping.compute()
print("You should Tip ", tipping.output['tip'])
Rate food ? [0-10] --- 7
Waiter service ? [0-10] --- 6
Rate the Ambience ? [0-10]---5
You should Tip 13.455012853470436
In [ ]:



Result: We have successfully implemented fuzzy uncertainty problem using matplotlib and output is received

Experiment No:-8
Date:-08-03-2022

UNIFICATION AND RESOLUTION

AIM: Developing an algorithm for implementation of **UNIFICATION AND RESOLUTION**

PROBLEM STATEMENT: Developing an optimized technique using an appropriate artificial intelligence algorithm to solve the Unification and Resolution.

ALGORITHM :

1. function PL-RESOLUTION (KB, Q) returns true or false inputs: KB,
2. the knowledge base, group of sentences/facts in propositional logic
3. Q, the query, a sentence in propositional logic
4. clauses → the set of clauses in the CNF representation of KB ^ Q new → { }
5. loop do for each Ci, Cj in clauses do
6. resolvents → PL-RESOLVE (Ci, Cj)
7. if resolvents contains the empty clause the return true
8. new → new union resolvents
9. if new is a subset of clauses then return false
10. clauses → clauses union true

OPTIMIZATION TECHNIQUE:

Resolution basically works by using the principle of proof by contradiction. To find the conclusion we should negate the conclusion. Then the resolution rule is applied to the resulting clauses. Each clause that contains complementary literals is resolved to produce a2. new clause, which can be added to the set of facts (if it is not already present). This process continues until one of the two things happen:•There are no new clauses that can be added. An application of the resolution rule derives the empty clauseAn empty clause shows that the negation of the conclusion is a complete contradiction,hence the negation of the conclusion is invalid or false or the assertion is completely valid or true.

1. Convert the given statements in Predicate/Propositional Logic
2. Convert these statements into Conjunctive Normal Form
3. Negate the Conclusion (Proof by Contradiction)
4. Resolve using a Resolution Tree (Unification)

UNIFICATION CODE:

```
def get_index_comma(string):
    index_list = list()
    par_count = 0
```

```
    for i in range(len(string)):
        if string[i] == ',' and par_count == 0:
            index_list.append(i)
        elif string[i] == '(':
            par_count += 1
        elif string[i] == ')':
            par_count -= 1

    return index_list


def is_variable(expr):
    for i in expr:
        if i == '(' or i == ')':
            return False

    return True


def process_expression(expr):
    expr = expr.replace(' ', '')
    index = None
    for i in range(len(expr)):
        if expr[i] == '(':
            index = i
            break
    predicate_symbol = expr[:index]
    expr = expr.replace(predicate_symbol, '')
    expr = expr[1:len(expr) - 1]
    arg_list = list()
    indices = get_index_comma(expr)

    if len(indices) == 0:
        arg_list.append(expr)
    else:
        arg_list.append(expr[:indices[0]])
        for i, j in zip(indices, indices[1:]):
            arg_list.append(expr[i + 1:j])
        arg_list.append(expr[indices[len(indices) - 1] + 1:])

    return predicate_symbol, arg_list


def get_arg_list(expr):
    _, arg_list = process_expression(expr)

    flag = True
    while flag:
        flag = False
```

```
        for i in arg_list:
            if not is_variable(i):
                flag = True
                _, tmp = process_expression(i)
                for j in tmp:
                    if j not in arg_list:
                        arg_list.append(j)
                arg_list.remove(i)

    return arg_list


def check_occurs(var, expr):
    arg_list = get_arg_list(expr)
    if var in arg_list:
        return True

    return False


def unify(expr1, expr2):

    if is_variable(expr1) and is_variable(expr2):
        if expr1 == expr2:
            return 'Null'
        else:
            return False
    elif is_variable(expr1) and not is_variable(expr2):
        if check_occurs(expr1, expr2):
            return False
        else:
            tmp = str(expr2) + '/' + str(expr1)
            return tmp
    elif not is_variable(expr1) and is_variable(expr2):
        if check_occurs(expr2, expr1):
            return False
        else:
            tmp = str(expr1) + '/' + str(expr2)
            return tmp
    else:
        predicate_symbol_1, arg_list_1 = process_expression(expr1)
        predicate_symbol_2, arg_list_2 = process_expression(expr2)

        # Step 2
        if predicate_symbol_1 != predicate_symbol_2:
            return False
        # Step 3
        elif len(arg_list_1) != len(arg_list_2):
```

```
            return False
        else:
            # Step 4: Create substitution list
            sub_list = list()

            # Step 5:
            for i in range(len(arg_list_1)):
                tmp = unify(arg_list_1[i], arg_list_2[i])

                if not tmp:
                    return False
                elif tmp == 'Null':
                    pass
                else:
                    if type(tmp) == list:
                        for j in tmp:
                            sub_list.append(j)
                    else:
                        sub_list.append(tmp)

            # Step 6
            return sub_list


if __name__ == '__main__':

    f1 = 'Q(a, g(x, a), f(y))'
    f2 = 'Q(a, g(f(b), a), x)'
    # f1 = input('f1 : ')
    # f2 = input('f2 : ')

    result = unify(f1, f2)
    if not result:
        print('The process of Unification failed!')
    else:
        print('The process of Unification successful!')
        print(result)
```

RESOLUTION 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:
```

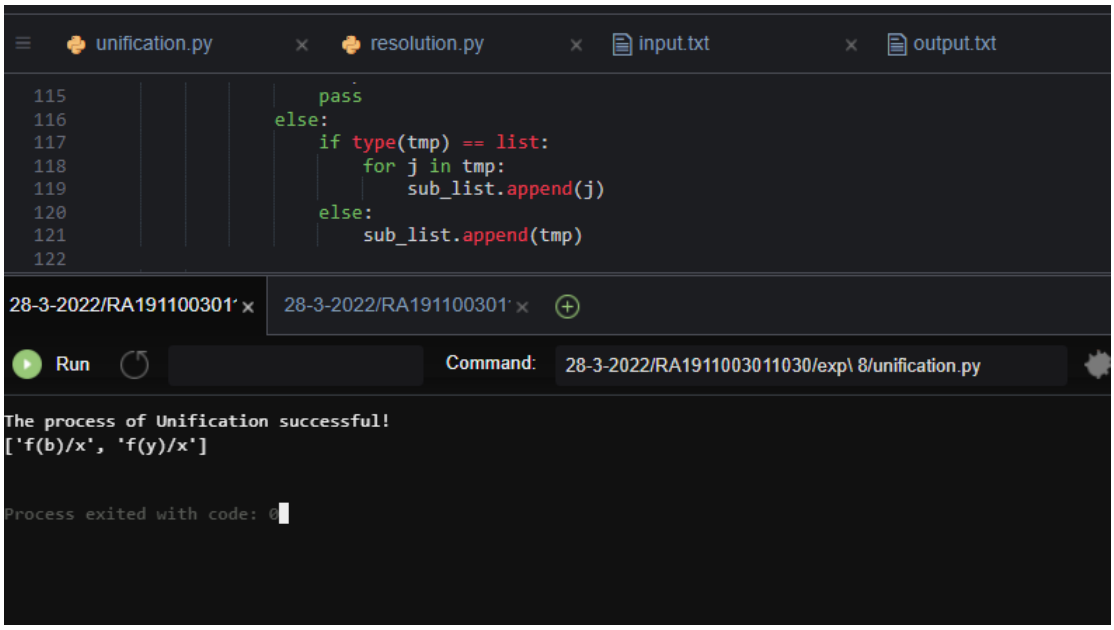
```
        sentence = Sentence(sentence_string)
        for predicate in sentence.getPredicates():
            self.sentence_map[predicate] = self.sentence_map.get(
                predicate, []) + [sentence]

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

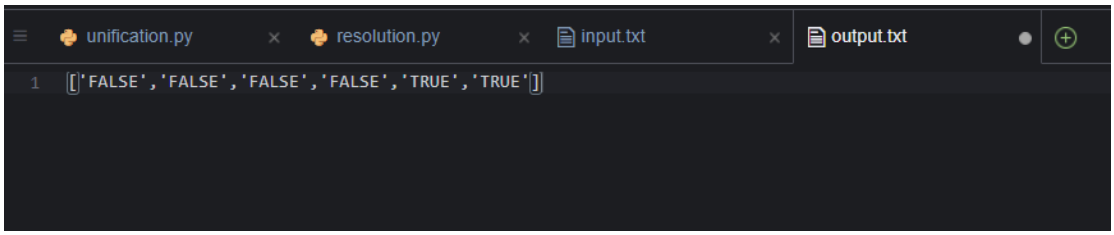
```
            if kb.isConstant():
                if not query.isConstant():
                    if query.name not in substitution:
                        substitution[query.name] = kb.name
                    elif substitution[query.name] != kb.name:
                        return False, {}
                    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_)
```

UNIFICATION OUTPUT:



RESOLUTION OUTPUT:



RESULT:

Developed Unification and Resolution Algorithm in Python for solving logical problems.

Experiment No:-9
Date:-21-03-2022

Implementation of learning algorithms for an application

Aim:

a) Implementation of Linear Regression algorithm to predict students score using the given dataset.
b) Implementation of Support Vector Classification algorithm to classify the cases of breast cancer
using the given dataset.
c) Implementation of K-means clustering algorithm to group the customers based on their demographic detail using the given dataset.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn import metrics
%matplotlib inline
Import required modules and packages
dataset = pd.read_csv('....\student_scores.csv')
dataset.head()
Import data set
Choose the right path for the dataset
dataset.describe() Descriptive statistics of the attributes
available in the dataset
dataset.plot(x='Hours', y='Scores', style='o')
plt.title('Hours vs Percentage')
plt.xlabel('Hours Studied')
plt.ylabel('Percentage Score')
```

plt.show()
Visualize the data.
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, 1].values
Identify the independent (X) and
dependent variables (y) in the data set
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=0)
print('X train shape: ', X_train.shape)
print('Y train shape: ', Y_train.shape)
print('X test shape: ', X_test.shape)
Splitting the given data in to training set
(80%) and testing set (20%)
Beginners Level
Lab 11 - Implementation of Learning Algorithms for an Application
18CSC305J - ARTIFICIAL INTELLIGENCE Page 4
print('Y test shape: ', Y_test.shape)
regressor = LinearRegression()
Model instantiation
regressor.fit(X_train, y_train) Model Training
print(regressor.intercept_)
print(regressor.coef_)
Finding out the coefficient (a) and
intercept (b) value of linear model
(y=aX+b)
y_pred = regressor.predict(X_test)
df = pd.DataFrame({'Actual': y_test, 'Predicted': y_pred})
print(df)
Testing the model
print('Mean Absolute Error:',
metrics.mean_absolute_error (y_test, y_pred))
print('Mean Squared Error:',
metrics.mean_squared_error (y_test, y_pred))
print('Root Mean Squared Error:',
np.sqrt(metrics.mean_squared_error (y_test, y_pred)))
MAE, MSE, RMSE – Evaluation metrics
of Model
Discussion:

```
[ ]  dataset = pd.read_csv('C:\\Users\DELL\Desktop\student_scores.csv')
     dataset.shape()
```

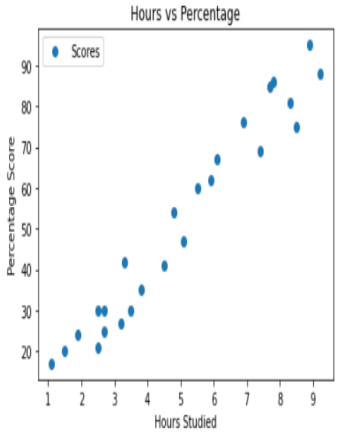
	Hours	Scores
0	2.5	21
1	5.1	47
2	3.2	27
3	8.5	75
4	3.5	30

```
dataset.shape
```

```
(25, 2)
```

```
dataset.describe()
```

	Hours	Scores
count	25.000000	25.000000
mean	5.012000	51.480000
std	2.525094	25.286887
min	1.100000	17.000000
25%	2.700000	30.000000
50%	4.800000	47.000000
75%	7.400000	75.000000
max	9.200000	95.000000



```
[ ] print(regressor.coef_)

    [9.91065648]

[ ] y_pred = regressor.predict(X_test)
    df = pd.DataFrame({'Actual': y_test, 'Predicted': y_pred})
    df
```

	Actual	Predicted
0	20	16.884145
1	27	33.732261
2	69	75.357018
3	30	26.794801
4	62	60.491033

```
[ ] print('Mean Absolute Error:', metrics.mean_absolute_error(y_test, y_pred))
    print('Mean Squared Error:', metrics.mean_squared_error(y_test, y_pred))
    print('Root Mean Squared Error:', np.sqrt(metrics.mean_squared_error(y_test, y_pred)))

    Mean Absolute Error: 4.183859899002975
    Mean Squared Error: 21.5987693072174
    Root Mean Squared Error: 4.6474476121003665
```

B)

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import confusion_matrix,
classification_report
Import required modules and packages
dataset = pd.read_csv('....\diabetes data.csv')
print(dataset.head())
Import data set
Choose the right path for the dataset
def diagnosis(x):
if x=='M' :
return 1
if x=='B' :
return 0
dataset['diagnosis'] = dataset['diagnosis'].apply(diagnosis)
print(dataset)
Data cleaning process. Converting
categorical value in to numerical value.
M = malignant, B = benign
print("Any missing sample in data set:",
dataset.isnull().values.any(), "\n")
Check for any missing values in the data
set
dataset = dataset.replace([np.inf, -np.inf], np.nan)
dataset= dataset.fillna(dataset.mean())
dataset
Replace the missing value with its mean
value of the respective attribute
dataset= dataset.drop(columns=["Unnamed: 32"]) drop this column because it's not
necessary (null)
```

Y = dataset['diagnosis']
X = dataset.drop(columns=['diagnosis'])
X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
test_size=0.2, random_state=9)
print('X train shape: ', X_train.shape)
print('Y train shape: ', Y_train.shape)
print('X test shape: ', X_test.shape)
print('Y test shape: ', Y_test.shape)
Splitting the given data in to training set
(80%) and testing set (20%)
svc_classifier= SVC(kernel='poly') Model instantiation. Apply SVM with
different kernels 'linear', 'poly', 'rbf',
'sigmoid' and verify the accuracy of the
model
svc_classifier.fit(X_train,Y_train) Model Training
y_pred=svc_classifier.predict(X_test) Testing the model
print(confusion_matrix(Y_test,y_pred))
print(classification_report(Y_test,y_pred))
Evaluation metrics to measure the
performance of the model

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import confusion_matrix,classification_report
%matplotlib inline
```

```
dataset = pd.read_csv('C:\\Users\DELL\Desktop\data.csv')
dataset.head()
```

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	concave points_mean	...	texture_worst	perimeter
0	842302	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	...	17.33	
1	842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	...	23.41	
2	84300903	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	...	25.53	
3	84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	...	26.50	
4	84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	...	16.67	

5 rows × 33 columns

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	concave points_mean	...	texture_worst	perimete
0	842302	1	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.30010	0.14710	...	17.33	
1	842517	1	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.08690	0.07017	...	23.41	
2	84300903	1	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.19740	0.12790	...	25.53	
3	84348301	1	11.42	20.38	77.58	386.1	0.14250	0.28390	0.24140	0.10520	...	26.50	
4	84358402	1	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.19800	0.10430	...	16.67	
...	...	...	...	...	...	...	...	...	...	...	...	...	
564	926424	1	21.56	22.39	142.00	1479.0	0.11100	0.11590	0.24390	0.13890	...	26.40	
565	926682	1	20.13	28.25	131.20	1261.0	0.09780	0.10340	0.14400	0.09791	...	38.25	
566	926954	1	16.60	28.08	108.30	858.1	0.08455	0.10230	0.09251	0.05302	...	34.12	
567	927241	1	20.60	29.33	140.10	1265.0	0.11780	0.27700	0.35140	0.15200	...	39.42	
568	92751	0	7.76	24.54	47.92	181.0	0.05263	0.04362	0.00000	0.00000	...	30.37	

569 rows × 33 columns

```
] svc_classifier= SVC(kernel='rbf')
  svc_classifier

  SVC()

] svc_classifier=svc_classifier.fit(X_train,Y_train)

] y_pred=svc_classifier.predict(X_test)

] print(confusion_matrix(Y_test,y_pred))

  [[74  0]
   [40  0]]

] print(classification_report(Y_test,y_pred))

              precision    recall  f1-score   support

           0       0.65      1.00      0.79        74
           1       0.00      0.00      0.00        40

    accuracy                           0.65       114
   macro avg       0.32      0.50      0.39       114
weighted avg       0.42      0.65      0.51       114
```

(c) Implementation of K-means clustering algorithm to group the customers based on their demographic detail using the given dataset.

Problem: Client is owing a supermarket mall and through membership cards, client have some basic data

about your customers like Customer ID, age, gender, annual income and spending score. Help the client
to understand the customers like who are the target customers so that the sense can be given to marketing

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
%matplotlib inline
```

```
data=pd.read_csv('C:\\Users\DELL\Desktop\mall_customers.csv')
print(data.head())
```

```
   CustomerID   Genre  Age  Annual Income (k$)  Spending Score (1-100)
0           1    Male   19                  15                      39
1           2    Male   21                  15                      81
2           3  Female   20                  16                       6
3           4  Female   23                  16                      77
4           5  Female   31                  17                      40
```

```
inVsout=data.iloc[:,[3,4]]
inVsout
```

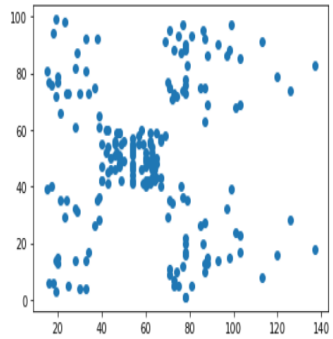
```
] inVsout=data.iloc[:,[3,4]]
  inVsout
```

	Annual Income (k$)	Spending Score (1-100)
0	15	39
1	15	81
2	16	6
3	16	77
4	17	40
...	...	...
195	120	79
196	126	28
197	126	74
198	137	18
199	137	83

200 rows × 2 columns

```
plt.scatter(inVsout.iloc[:,0],inVsout.iloc[:,1])
```

```
<matplotlib.collections.PathCollection at 0x1fa26e7ca90>
```

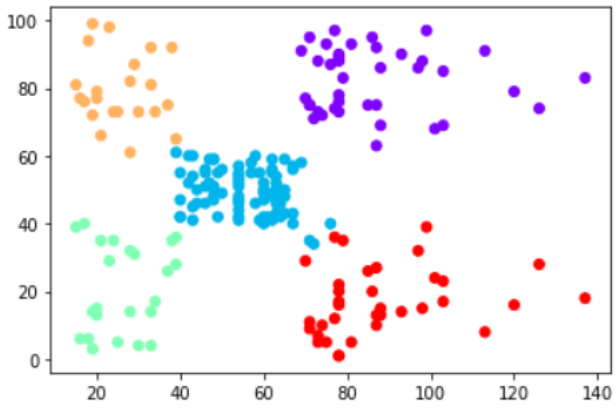



```
kmeans=KMeans(n_clusters=5)
kmeans.fit(inVsout)
```

```
KMeans(n_clusters=5)
```

```
plt.scatter(inVsout.iloc[:,0],inVsout.iloc[:,1], c=kmeans.labels_, cmap='rainbow')
plt.show()
```

```
plt.scatter(inVsout.iloc[:,0],inVsout.iloc[:,1], c=kmeans.labels_, cmap='rainbow')
plt.show()
```



	Annual Income (k$)	Spending Score (1-100)
0	15	39
2	16	6
4	17	40
6	18	6
8	19	3
10	19	14
12	20	15
14	20	13
16	21	35
18	23	29
20	24	35
22	25	5
24	28	14
26	28	32
28	29	31
30	30	4
32	33	4
34	33	14

```
silhouette_score(inVsout,kmeans.labels_)
```

```
0.553931997444648
```

Experiment No:-10
Date:-10-04-2022

To Implement NLP programs

Aim:-To Implement NLP programs

NLP stands for Natural Language Processing, which is a part of Computer Science, Human language, and Artificial Intelligence. It is the technology that is used by machines to understand, analyse, manipulate, and interpret human's languages. It helps developers to organize knowledge for performing tasks such as translation, automatic summarization, Named Entity Recognition (NER), speech recognition, relationship extraction, and topic segmentation.

code:-

```
!pip install -q wordcloud
import wordcloud

import nltk
nltk.download('stopwords')
nltk.download('wordnet')
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')

import pandas as pd
import matplotlib.pyplot as plt
import io
import unicodedata
import numpy as np
import re
```

```
import string
# Constants
# POS (Parts Of Speech) for: nouns, adjectives, verbs and adverbs
DI_POS_TYPES = {'NN':'n', 'JJ':'a', 'VB':'v', 'RB':'r'}
POS_TYPES = list(DI_POS_TYPES.keys())

# Constraints on tokens
MIN_STR_LEN = 3
RE_VALID = '[a-zA-Z]'
# Upload from google drive
from google.colab import files
uploaded = files.upload()
print("len(uploaded.keys():", len(uploaded.keys()))

for fn in uploaded.keys():
  print('User uploaded file "{name}" with length {length} bytes'.format(name=fn,
length=len(uploaded[fn])))

# Get list of quotes
df_quotes = pd.read_csv(io.StringIO(uploaded['quotes.txt'].decode('utf-8')), sep='\t')

# Display
print("df_quotes:")
print(df_quotes.head().to_string())
print(df_quotes.describe())

# Convert quotes to list
li_quotes = df_quotes['Quote'].tolist()
print()
print("len(li_quotes):", len(li_quotes)
# Get stopwords, stemmer and lemmatizer
```

```
stopwords = nltk.corpus.stopwords.words('english')
stemmer = nltk.stem.PorterStemmer()
lemmatizer = nltk.stem.WordNetLemmatizer()

# Remove accents function
def remove_accents(data):
    return ''.join(x for x in unicodedata.normalize('NFKD', data) if x in
string.ascii_letters or x == ' ')

# Process all quotes
li_tokens = []
li_token_lists = []
li_lem_strings = []

for i,text in enumerate(li_quotes):
    # Tokenize by sentence, then by lowercase word
    tokens = [word.lower() for sent in nltk.sent_tokenize(text) for word in
nltk.word_tokenize(sent)]

    # Process all tokens per quote
    li_tokens_quote = []
    li_tokens_quote_lem = []
    for token in tokens:
        # Remove accents
        t = remove_accents(token)

        # Remove punctuation
        t = str(t).translate(string.punctuation)
        li_tokens_quote.append(t)

        # Add token that represents "no lemmatization match"
```

```
        li_tokens_quote_lem.append("-") # this token will be removed if a lemmatization
match is found below

        # Process each token
        if t not in stopwords:
            if re.search(RE_VALID, t):
                if len(t) >= MIN_STR_LEN:
                    # Note that the POS (Part Of Speech) is necessary as input to the
lemmatizer
                    # (otherwise it assumes the word is a noun)
                    pos = nltk.pos_tag([t])[0][1][:2]
                    pos2 = 'n'  # set default to noun
                    if pos in DI_POS_TYPES:
                      pos2 = DI_POS_TYPES[pos]

                    stem = stemmer.stem(t)
                    lem = lemmatizer.lemmatize(t, pos=pos2)  # lemmatize with the correct
POS

                    if pos in POS_TYPES:
                        li_tokens.append((t, stem, lem, pos))

                        # Remove the "-" token and append the lemmatization match
                        li_tokens_quote_lem = li_tokens_quote_lem[:-1]
                        li_tokens_quote_lem.append(lem)

    # Build list of token lists from lemmatized tokens
    li_token_lists.append(li_tokens_quote)

    # Build list of strings from lemmatized tokens
    str_li_tokens_quote_lem = ' '.join(li_tokens_quote_lem)
```

```
    li_lem_strings.append(str_li_tokens_quote_lem)

# Build resulting dataframes from lists
df_token_lists = pd.DataFrame(li_token_lists)

print("df_token_lists.head(5):")
print(df_token_lists.head(5).to_string())

# Replace None with empty string
for c in df_token_lists:
    if str(df_token_lists[c].dtype) in ('object', 'string_', 'unicode_'):
        df_token_lists[c].fillna(value='', inplace=True)

df_lem_strings = pd.DataFrame(li_lem_strings, columns=['lem quote'])

print()
print("")
print("df_lem_strings.head():")
print(df_lem_strings.head().to_string())
# Add counts
print("Group by lemmatized words, add count and sort:")
df_all_words = pd.DataFrame(li_tokens, columns=['token', 'stem', 'lem', 'pos'])
df_all_words['counts'] = df_all_words.groupby(['lem'])['lem'].transform('count')
df_all_words = df_all_words.sort_values(by=['counts', 'lem'], ascending=[False,
True]).reset_index()

print("Get just the first row in each lemmatized group")
df_words = df_all_words.groupby('lem').first().sort_values(by='counts',
ascending=False).reset_index()
print("df_words.head(10):")
print(df_words.head(10))
```

```
df_words = df_words[['lem', 'pos', 'counts']].head(200)
for v in POS_TYPES:
    df_pos = df_words[df_words['pos'] == v]
    print()
    print("POS_TYPE:", v)
    print(df_pos.head(10).to_string())
li_token_lists_flat = [y for x in li_token_lists for y in x]  # flatten the list of token lists to a single list
print("li_token_lists_flat[:10]:", li_token_lists_flat[:10])

di_freq = nltk.FreqDist(li_token_lists_flat)
del di_freq['']
li_freq_sorted = sorted(di_freq.items(), key=lambda x: x[1], reverse=True)  # sorted list
print(li_freq_sorted)

di_freq.plot(30, cumulative=False)
li_lem_words = df_all_words['lem'].tolist()
di_freq2 = nltk.FreqDist(li_lem_words)
li_freq_sorted2 = sorted(di_freq2.items(), key=lambda x: x[1], reverse=True)  # sorted list
print(li_freq_sorted2)

di_freq2.plot(30, cumulative=False)
```

Output:-

```
Group by lemmatized words, add count and sort:
Get just the first row in each lemmatized group
df_words.head(10):
          lem  index      token    stem pos  counts
0     always     50     always   alway  RB      10
1    nothing    116    nothing    noth  NN       6
2       life     54       life    life  NN       6
3        man     74        man     man  NN       5
4       give     39       gave    gave  VB       5
5       fact    106       fact    fact  NN       5
6      world    121      world   world  NN       5
7  happiness    119  happiness   happi  NN       4
8       work    297       work    work  NN       4
9     theory    101     theory  theori  NN       4
```

POS_TYPE: NN

	lem	pos	counts
1	nothing	NN	6
2	life	NN	6
3	man	NN	5
5	fact	NN	5
6	world	NN	5
7	happiness	NN	4
8	work	NN	4
9	theory	NN	4
10	woman	NN	4
17	holmes	NN	3

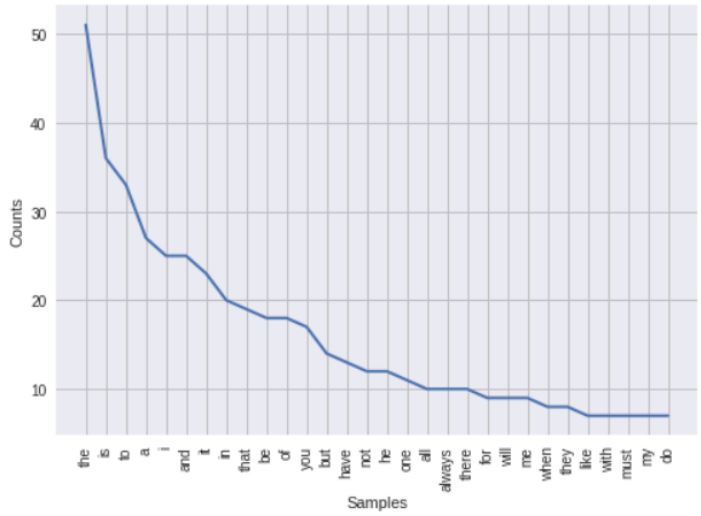
POS_TYPE: JJ

	lem	pos	counts
11	impossible	JJ	4
15	certain	JJ	3
18	curious	JJ	3
34	nice	JJ	2
43	little	JJ	2
48	good	JJ	2
61	improbable	JJ	2
62	best	JJ	2
72	philosophical	JJ	1
81	possible	JJ	1

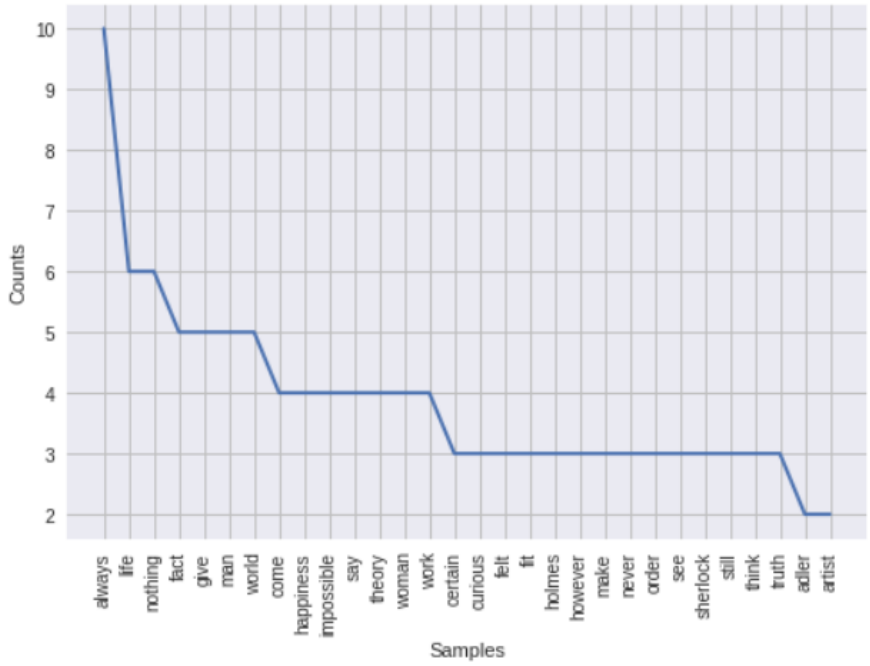
POS_TYPE: VB

	lem	pos	counts
4	give	VB	5
12	say	VB	4
13	come	VB	4
22	see	VB	3
23	make	VB	3
26	think	VB	3

```
li_token_lists_flat[:10]: ['i', 'like', 'living', '', 'i', 'have', 'sometimes', 'been', 'wildly', '']
[('the', 51), ('is', 36), ('to', 33), ('a', 27), ('i', 25), ('and', 25), ('it', 23), ('in', 20), ('that
```




```
[('always', 10), ('life', 6), ('nothing', 6), ('fact', 5), ('give', 5), ('man', 5),
```



Result:-Thus the NPL program was implemented

Experiment No:-11
Date:-06-04-2022

Deep Learning

AIM: Implementation of Deep Learning

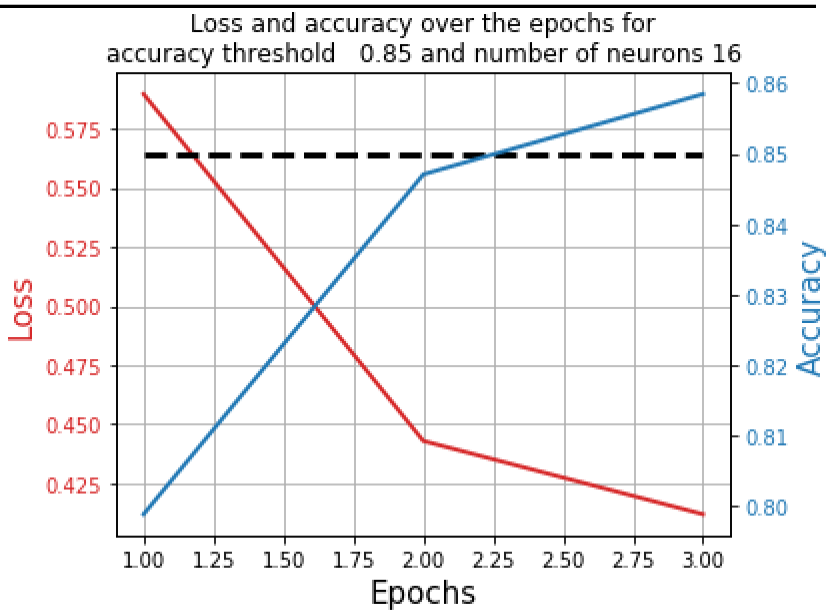
CODE:

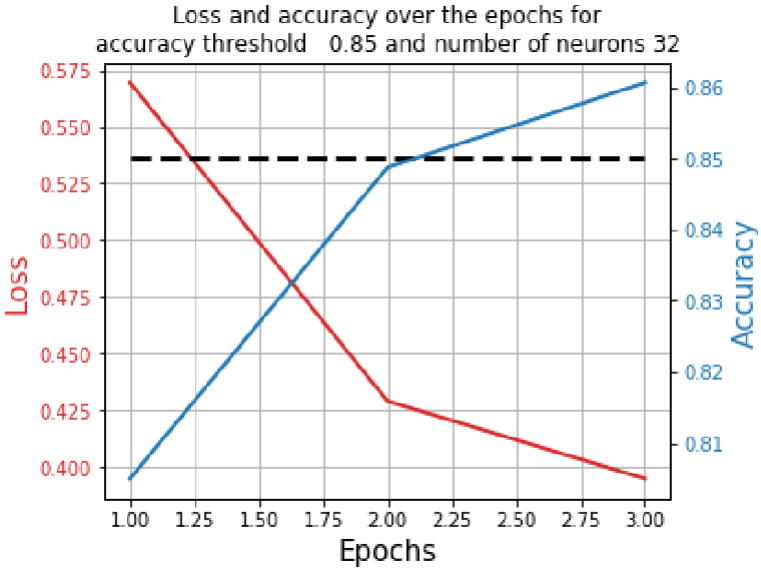
```
import tensorflow as tf
from utils.DL_utils import myCallback, build_model,
compile_train_model, plot_loss_acc
from itertools import product

accuracy_desired = [0.85,0.9,0.95]
num_neurons = [16,32,64,128]

cases = list(product(accuracy_desired,num_neurons)) print("So, the cases we

are considering are as follows...\n")
for i,c in enumerate(cases):
  print("Accuracy target {}, number of neurons:
{}".format(c[0],c[1]))
  if (i+1)%4==0 and (i+1)!=len(cases):
       print("-"*50)
  for c in cases:
  # Create a mycallback class with the specific accuracy target
  callbacks = myCallback(c[0], print_msg=False)

  # Build a model with a specific number of neurons
  model = build_model(num_layers=1,architecture=[c[1]])

  # Compile and train the model passing on the callback
class,choose suitable batch size and a max epoch limit
  model = compile_train_model(model,
x_train,y_train,callbacks=callbacks,
                              batch_size=32,epochs=30)

  # Construct a suitable title string for displaying the results
properly
  title = "Loss and accuracy over the epochs for\naccuracythreshold \
  {} and number of neurons {}".format(c[0],c[1])
```

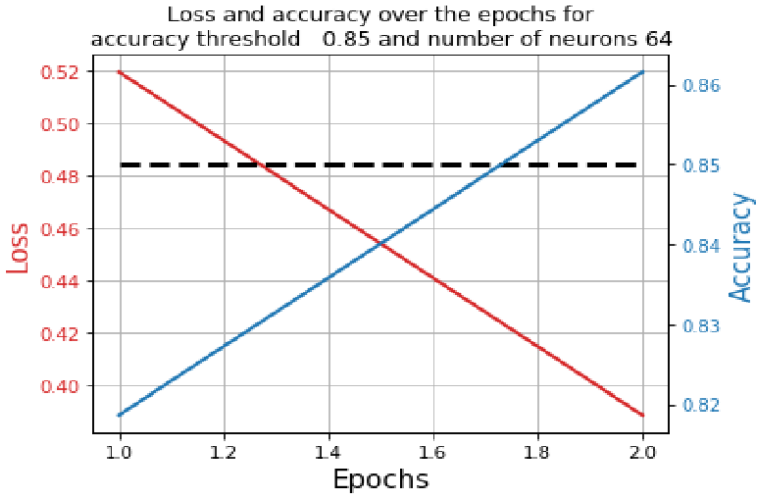
 # Use the plotting utility function, pass on the accuracy target,# trained model, and the custom title string
plot_loss_acc(model,target_acc=c[0],title=title)

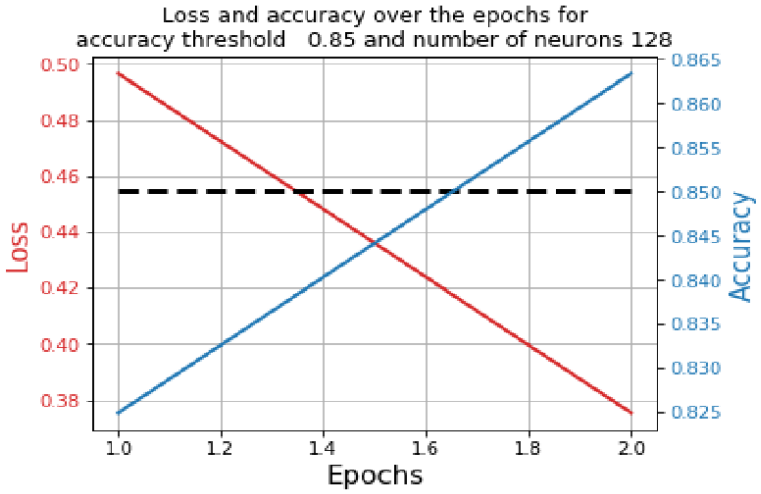
OUTPUT:

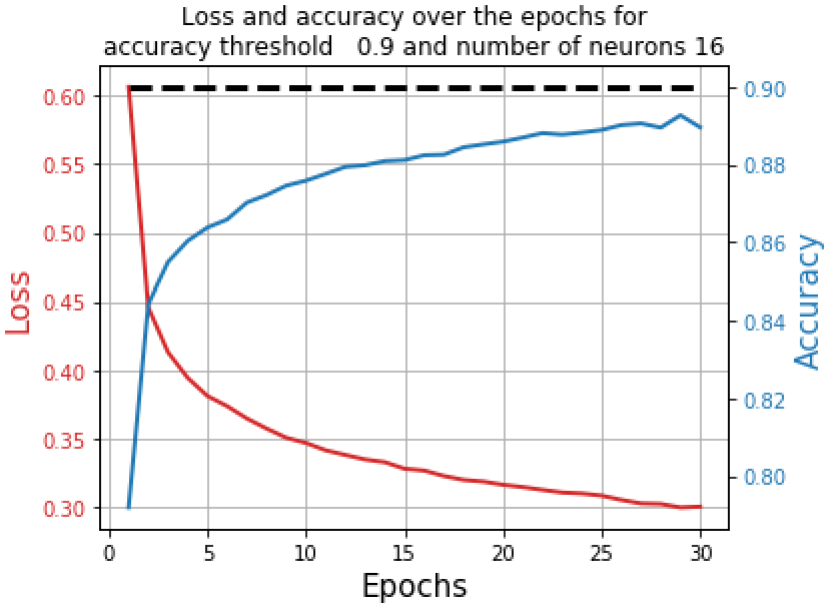
```
So, the cases we are considering are as follows...

Accuracy target 0.85, number of neurons: 16
Accuracy target 0.85, number of neurons: 32
Accuracy target 0.85, number of neurons: 64
Accuracy target 0.85, number of neurons: 128
--------------------------------------------------
Accuracy target 0.9, number of neurons: 16
Accuracy target 0.9, number of neurons: 32
Accuracy target 0.9, number of neurons: 64
Accuracy target 0.9, number of neurons: 128
--------------------------------------------------
Accuracy target 0.95, number of neurons: 16
Accuracy target 0.95, number of neurons: 32
Accuracy target 0.95, number of neurons: 64
Accuracy target 0.95, number of neurons: 128
```




Loss and accuracy over the epochs for
accuracy threshold 0.85 and number of neurons 32
Loss
Accuracy
Epochs
0.575
0.550
0.525
0.500
0.475
0.450
0.425
0.400
0.86
0.85
0.84
0.83
0.82
0.81
1.00
1.25
1.50
1.75
2.00
2.25
2.50
2.75
3.00


Loss and accuracy over the epochs for
accuracy threshold 0.85 and number of neurons 64
Loss
Accuracy
Epochs
0.52
0.50
0.48
0.46
0.44
0.42
0.40
0.86
0.85
0.84
0.83
0.82
1.0
1.2
1.4
1.6
1.8
2.0


Loss and accuracy over the epochs for
accuracy threshold 0.85 and number of neurons 128
Loss
Accuracy
Epochs
0.50
0.48
0.46
0.44
0.42
0.40
0.38
0.865
0.860
0.855
0.850
0.845
0.840
0.835
0.830
0.825
1.0
1.2
1.4
1.6
1.8
2.0



RESULT: Deep Learning Model Implemented