VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

S M MRUNALINI(1BM22CS228)

in partial fulfillment for the award of the degree of
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Sep-2024 to Jan-2025

B.M.S. College of Engineering,
Bull Temple Road, Bangalore 560019
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by **S M MRUNALINI(1BM22CS228),** who is a bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

Sneha P Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

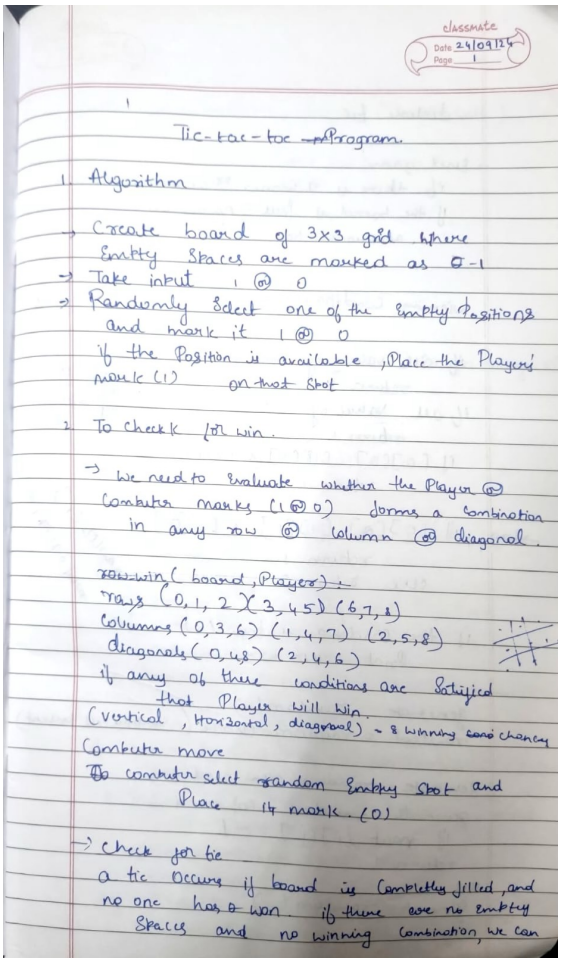
Sl. No.	Date	Experiment Title	Page No.
1	24-9-2024	Implement Tic –Tac –Toe Game	1-10
2	1-10-2024	Implement vacuum cleaner agent	10-18
3	8-10-2024	puzzle problems using Depth First Search (DFS)	18-25
4	15-10-2024	Implement Iterative deepening search algorithm Implement A* search algorithm	25-33
5	22-10-2024	Simulated Annealing to Solve 8-Queens problem	33-41
6	29-10-2024	Implement Hill Climbing search algorithm to solve N-Queens problem	41-49
7	12-11-2024	Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.	49-56
8	19-11-2024	Implement unification in first order logic Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.	56-63
9	3-12-2024	Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning. Implement Alpha-Beta Pruning. minmax(tic-tac-toe)	63-71

Github Link:
https://github.com/Mrunalinishettar/mrunalini-1BM22CS228-AI-LAB

Program 1

Implement Tic –Tac –Toe Game

Algorithm:



Tic-tac-toe Program.

1. Algorithm

→ Create board of 3×3 grid where Emtty Spaces are marked as 0-1

→ Take input 1 @ 0

→ Randomly Select one of the Emtpty Positions and mark it 1 @ 0

if the Position is available, Place the Players mark (1) on that Slot

2. To check for win.

→ We need to Evaluate whether the Player @ Computer marks (1 @ 0) forms a Combination in any row @ column @ diagonal.

row-win (board, Player) :-

rows (0, 1, 2) (3, 4, 5) (6, 7, 8)

Columns (0, 3, 6) (1, 4, 7) (2, 5, 8)

diagonals (0, 4, 8) (2, 4, 6)

if any of these conditions are Satisfied that Player will win.

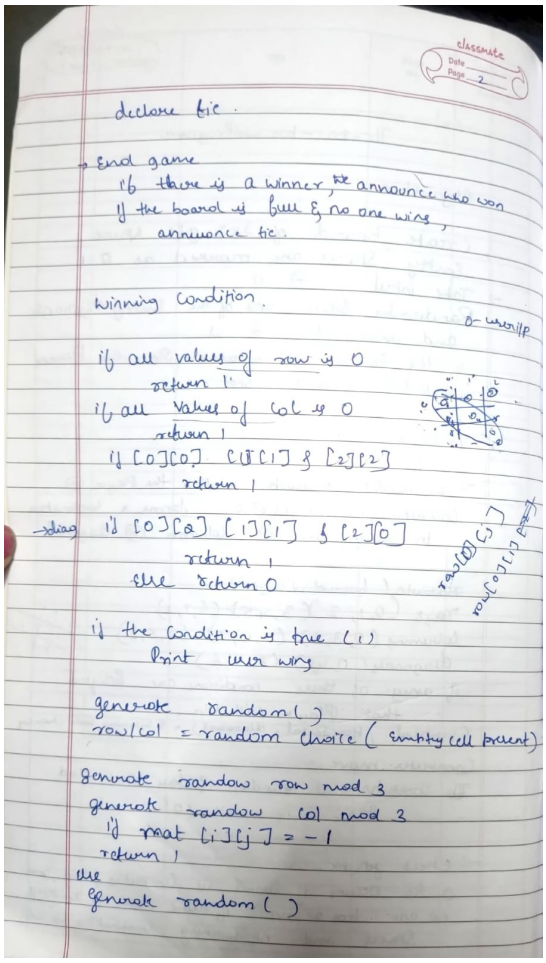
(Vertical, Horizontal, diagonal) - 8 winning combi chances

Computer move

The computer select random Emtpy Slot and Place it mark. (0)

→ Check for tie

a tie Occurs if board is Completely filled, and no one has won. if there are no Emtpy Spaces and no winning Combination, we can



declare tie.

→ End game

if there is a winner, announce who won

if the board is full & no one wins, announce tie.

winning condition.

if all values of row is 0

return 1

if all values of col is 0

return 1

if [0][0] [1][1] & [2][2]

return 1

if [0][2] [1][1] & [2][0]

return 1

else return 0

if the condition is true (1)

Print user wins

Generate random ( )

row/col = random choice (Emtpty cell present)

Generate random row mod 3

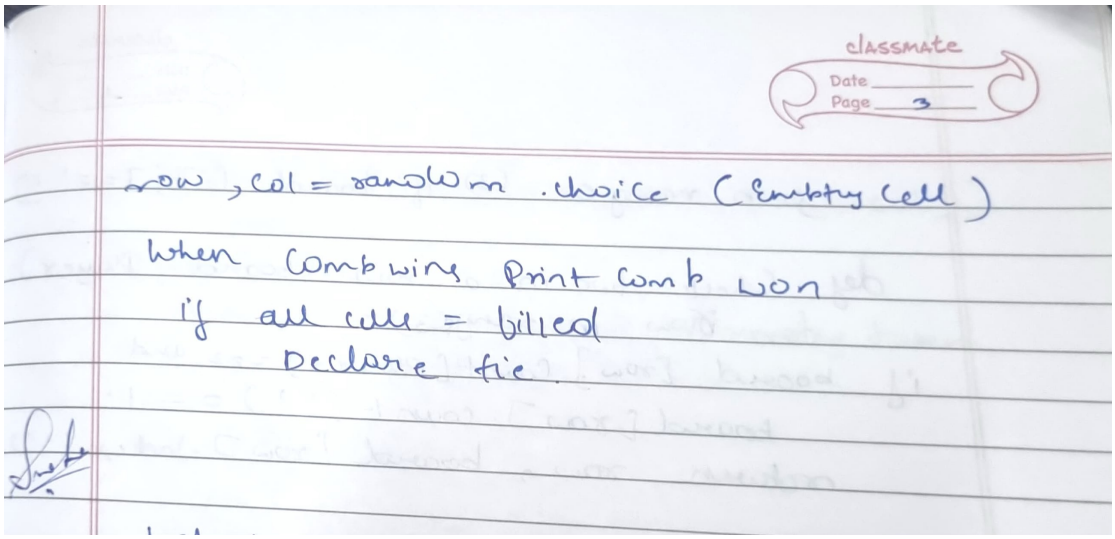
Generate random col mod 3

if mat [i][j] = -1

return 1

else

Generate random ( )



row, col = random.choice (Emtpty cell)

When Comb wins Print Comb won

if all cell = filled

Declare tie.

Code:

```
import numpy as np

def create_board():

    return np.array([[" ", " ", " "],

                     [" ", " ", " "],

                     [" ", " ", " "]])

def coordinates(board, player):

    i, j, cn = (-1, -1, 0)

    player_symbol = 'X' if player == 1 else 'O'

    while (i < 0 or i > 2 or j < 0 or j > 2) or (board[i][j] != " "):

        if cn > 0:

            print("Wrong Input! Try Again")

        print("Player {}'s turn ({})".format(player, player_symbol))

        i = int(input("x-coordinates (1-3): ")) - 1

        j = int(input("y-coordinates (1-3): ")) - 1

        cn += 1

    board[i][j] = player_symbol

    return board

def row_win(board, player):

    player_symbol = 'X' if player == 1 else 'O'

    for x in range(len(board)):

        if all(board[x, y] == player_symbol for y in range(len(board))):

            return True

    return False
```

```
def col_win(board, player):
    player_symbol = 'X' if player == 1 else 'O'
    for x in range(len(board)):
        if all(board[y][x] == player_symbol for y in range(len(board))):
            return True
    return False


def diag_win(board, player):
    player_symbol = 'X' if player == 1 else 'O'
    if all(board[x][x] == player_symbol for x in range(len(board))):
        return True
    if all(board[x][len(board)-1-x] == player_symbol for x in
range(len(board))):
        return True
    return False


def evaluate(board):
    winner = 0
    for player in [1, 2]:
        if (row_win(board, player) or
                col_win(board, player) or
                diag_win(board, player)):
            winner = player
```

```
    if np.all(board != " ") and winner == 0:
        winner = -1
    return winner


def play_game():
    board, winner, counter = create_board(), 0, 1
    print(board)
    while winner == 0:
        for player in [1, 2]:
            board = coordinates(board, player)
            print("Board after " + str(counter) + " move:")
            print(board)
            counter += 1
            winner = evaluate(board)
            if winner != 0:
                break
    return winner
winner = play_game()
if winner == -1:
    print("It's a draw!")
else:
    print("Winner is Player {} ({})".format(winner, 'X' if winner == 1 else
'O'))
```

Output:

```
[[' ' ' ' ' ']
 [' ' ' ' ' ']
 [' ' ' ' ' ']]
Player 1's turn (X)
x-coordinates (1-3): 4
y-coordinates (1-3): 1
Wrong Input! Try Again
Player 1's turn (X)
x-coordinates (1-3):
```

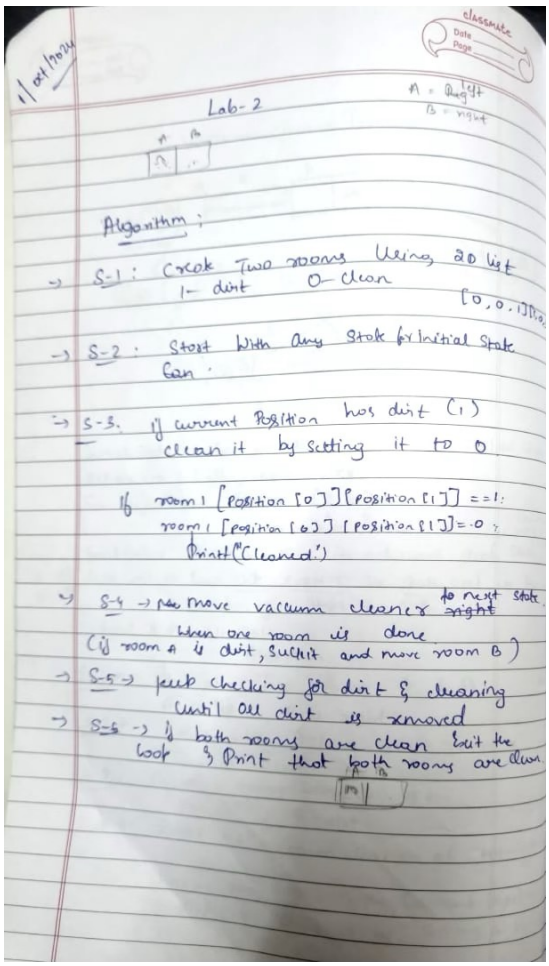
```
[[' ' ' ' ' ']
 [' ' ' ' ' ']
 [' ' ' ' ' ']]
Player 1's turn (X)
x-coordinates (1-3): 1
y-coordinates (1-3): 1
Board after 1 move:
[['X' ' ' ' ']
 [' ' ' ' ' ']
 [' ' ' ' ' ']]
Player 2's turn (O)
x-coordinates (1-3): 1
y-coordinates (1-3): 2
Board after 2 move:
[['X' 'O' ' ']
 [' ' ' ' ' ']
 [' ' ' ' ' ']]
Player 1's turn (X)
x-coordinates (1-3): 2
y-coordinates (1-3): 2
Board after 3 move:
[['X' 'O' ' ']
 [' ' 'X' ' ']
 [' ' ' ' ' ']]
Player 2's turn (O)
x-coordinates (1-3): 1
y-coordinates (1-3): 3
Board after 4 move:
[['X' 'O' 'O']
 [' ' 'X' ' ']
 [' ' ' ' ' ']]
Player 1's turn (X)
x-coordinates (1-3): 3
y-coordinates (1-3): 3
Board after 5 move:
[['X' 'O' 'O']
 [' ' 'X' ' ']
 [' ' ' ' 'X']]
Winner is Player 1 (X)
```

```
[[' ' ' ' ' ']
 [' ' ' ' ' ']
 [' ' ' ' ' ']]
Player 1's turn (X)
x-coordinates (1-3): 1
y-coordinates (1-3): 2
Board after 1 move:
[[' ' 'X' ' ']
 [' ' ' ' ' ']
 [' ' ' ' ' ']]
Player 2's turn (O)
x-coordinates (1-3): 1
y-coordinates (1-3): 1
Board after 2 move:
[['O' 'X' ' ']
 [' ' ' ' ' ']
 [' ' ' ' ' ']]
Player 1's turn (X)
x-coordinates (1-3): 2
y-coordinates (1-3): 2
Board after 3 move:
[['O' 'X' ' ']
 [' ' 'X' ' ']
 [' ' ' ' ' ']]
Player 2's turn (O)
x-coordinates (1-3): 3
y-coordinates (1-3): 2
Board after 4 move:
[['O' 'X' ' ']
 [' ' 'X' ' ']
 [' ' 'O' ' ']]
Player 1's turn (X)
x-coordinates (1-3): 3
y-coordinates (1-3): 3
Board after 5 move:
[['O' 'X' ' ']
 [' ' 'X' ' ']
 [' ' 'O' 'X']]
Player 2's turn (O)
x-coordinates (1-3): 1
y-coordinates (1-3): 3
Board after 6 move:
[['O' 'X' 'O']
 [' ' 'X' ' ']
 [' ' 'O' 'X']]
Player 1's turn (X)
x-coordinates (1-3): 2
y-coordinates (1-3): 1
Board after 7 move:
[['O' 'X' 'O']
 ['X' 'X' ' ']
 [' ' 'O' 'X']]
Player 2's turn (O)
```

```
Board after 6 move:
[['O' 'X' 'O']
 [' ' 'X' ' ']
 [' ' 'O' 'X']]
Player 1's turn (X)
x-coordinates (1-3): 2
y-coordinates (1-3): 1
Board after 7 move:
[['O' 'X' 'O']
 ['X' 'X' ' ']
 [' ' 'O' 'X']]
Player 2's turn (O)
x-coordinates (1-3): 2
y-coordinates (1-3): 3
Board after 8 move:
[['O' 'X' 'O']
 ['X' 'X' 'O']
 [' ' 'O' 'X']]
Player 1's turn (X)
x-coordinates (1-3): 3
y-coordinates (1-3): 1
Board after 9 move:
[['O' 'X' 'O']
 ['X' 'X' 'O']
 ['X' 'O' 'X']]
It's a draw!
```

Implement vacuum cleaner agent

Algorithm:


Lab-2

A = left
B = right

Algorithm:

→ S-1: Create Two rooms Using 2D list
1- dirt 0- clean
[0,0,1]

→ S-2: Start With Any State for Initial State Gen.

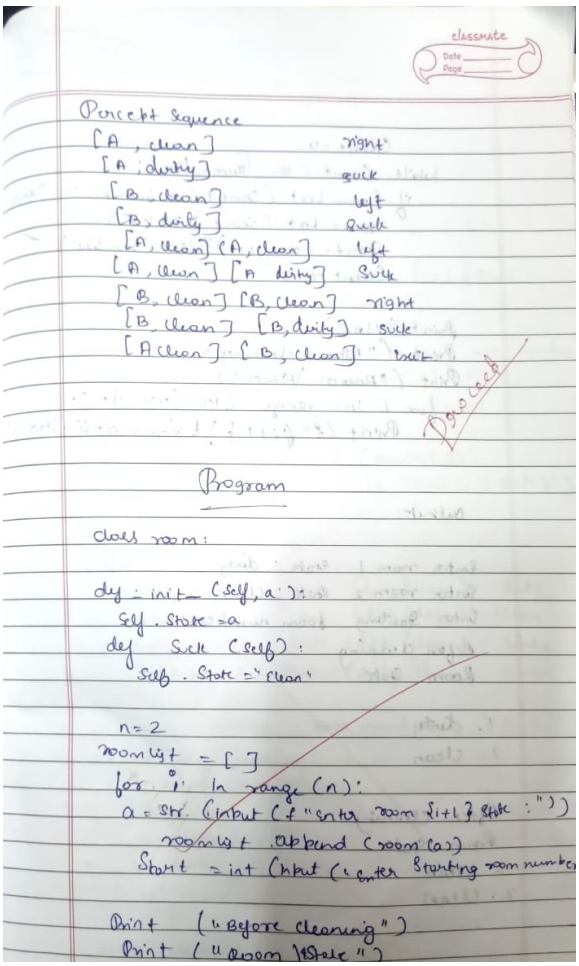
→ S-3. if current Position has dirt (1) clean it by setting it to 0

if room1 [position[0]][position[1]] == 1:
room1 [position[0]][position[1]] = 0
Print("Cleaned")

→ S-4 → Move vacuum cleaner to next state right when one room is done
(if room A is dirt, Suck it and move room B)

→ S-5 → Keep checking for dirt & cleaning until all dirt is removed

→ S-6 → if both rooms are clean Exit the loop & Print that both rooms are clean


Percept Sequence

[A, clean]	right
[A, dirty]	suck
[B, clean]	left
[B, dirty]	suck
[A, clean] [A, clean]	left
[A, clean] [A dirty]	Suck
[B, clean] [B, clean]	right
[B, clean] [B, dirty]	suck
[A clean] [B, clean]	Exit

Program

class room:

```
def __init__(self, a):
    self.state = a
def Suck(self):
    self.state = "clean"

n = 2
roomlist = []
for i in range(n):
    a = str(input(f"Enter room {i+1} State :"))
    roomlist.append(room(a))
Start = int(input("Enter Starting room number
Print("Before cleaning")
Print("Room State")
```

Code:

```
#Enter LOCATION A/B in captial letters

#Enter Status O/1 accordingly where 0 means CLEAN and 1 means DIRTY


def vacuum_world():
```

```
    # initializing goal_state

    # 0 indicates Clean and 1 indicates Dirty

  goal_state = {'A': '0', 'B': '0'}

    cost = 0



  location_input = input("Enter Location of Vacuum") #user_input of
location vacuum is placed

  status_input = input("Enter status of " + location_input) #user_input if
location is dirty or clean

  status_input_complement = input("Enter status of other room")

    print("Initial Location Condition" + str(goal_state))



    if location_input == 'A':

    # Location A is Dirty.

    print("Vacuum is placed in Location A")

    if status_input == '1':

          print("Location A is Dirty.")

          # suck the dirt  and mark it as clean

           goal_state['A'] = '0'

           cost += 1                    #cost for suck

          print("Cost for CLEANING A " + str(cost))

          print("Location A has been Cleaned.")


          if status_input_complement == '1':
```

```
            # if B is Dirty
            print("Location B is Dirty.")
            print("Moving right to the Location B. ")
            cost += 1                    #cost for moving right
            print("COST for moving RIGHT" + str(cost))
            # suck the dirt and mark it as clean
            goal_state['B'] = '0'
            cost += 1                    #cost for suck
            print("COST for SUCK " + str(cost))
            print("Location B has been Cleaned. ")
        else:
            print("No action" + str(cost))
            # suck and mark clean
            print("Location B is already clean.")


    if status_input == '0':
        print("Location A is already clean ")
        if status_input_complement == '1':# if B is Dirty
            print("Location B is Dirty.")
            print("Moving RIGHT to the Location B. ")
            cost += 1                    #cost for moving right
            print("COST for moving RIGHT " + str(cost))
            # suck the dirt and mark it as clean
            goal_state['B'] = '0'
```

```
            cost += 1                        #cost for suck
            print("Cost for SUCK" + str(cost))
            print("Location B has been Cleaned. ")
      else:
            print("No action " + str(cost))
            print(cost)
            # suck and mark clean
            print("Location B is already clean.")


else:
print("Vacuum is placed in location B")
# Location B is Dirty.
if status_input == '1':
      print("Location B is Dirty.")
      # suck the dirt  and mark it as clean
       goal_state['B'] = '0'
       cost += 1  # cost for suck
      print("COST for CLEANING " + str(cost))
      print("Location B has been Cleaned.")


      if status_input_complement == '1':
            # if A is Dirty
            print("Location A is Dirty.")
            print("Moving LEFT to the Location A. ")
```

```
            cost += 1  # cost for moving right
            print("COST for moving LEFT" + str(cost))
            # suck the dirt and mark it as clean
            goal_state['A'] = '0'
            cost += 1  # cost for suck
            print("COST for SUCK " + str(cost))
            print("Location A has been Cleaned.")

    else:
        print(cost)
        # suck and mark clean
        print("Location B is already clean.")

        if status_input_complement == '1':  # if A is Dirty
            print("Location A is Dirty.")
            print("Moving LEFT to the Location A. ")
            cost += 1  # cost for moving right
            print("COST for moving LEFT " + str(cost))
            # suck the dirt and mark it as clean
            goal_state['A'] = '0'
            cost += 1  # cost for suck
            print("Cost for SUCK " + str(cost))
            print("Location A has been Cleaned. ")
        else:
```

```
                print("No action " + str(cost))

                # suck and mark clean

                print("Location A is already clean.")



    # done cleaning

    print("GOAL STATE: ")

    print(goal_state)

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




vacuum_world()
```

Output:

```
Enter Location of VacuumA
Enter status of A0
Enter status of other room0
Initial Location Condition{'A': '0', 'B': '0'}
Vacuum is placed in Location A
Location A is already clean
No action 0
0
Location B is already clean.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 0
```

```
Enter Location of VacuumA
Enter status of A1
Enter status of other room1
Initial Location Condition{'A': '0', 'B': '0'}
Vacuum is placed in Location A
Location A is Dirty.
Cost for CLEANING A 1
Location A has been Cleaned.
Location B is Dirty.
Moving right to the Location B.
COST for moving RIGHT2
COST for SUCK 3
Location B has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 3
```

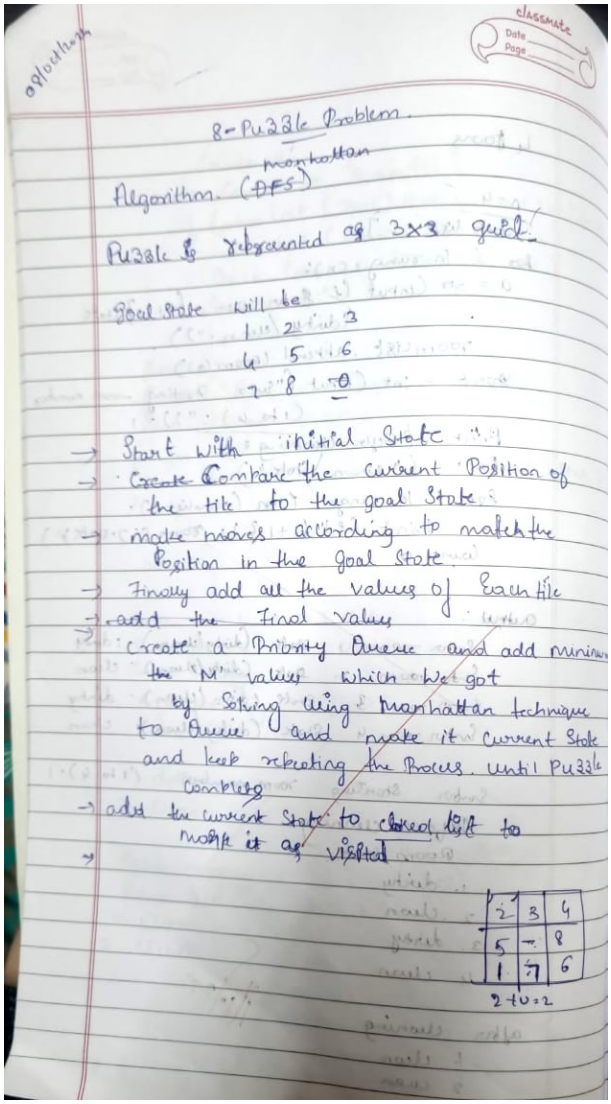
```
Enter Location of VacuumA
Enter status of A1
Enter status of other room0
Initial Location Condition{'A': '0', 'B': '0'}
Vacuum is placed in Location A
Location A is Dirty.
Cost for CLEANING A 1
Location A has been Cleaned.
No action1
Location B is already clean.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 1
```

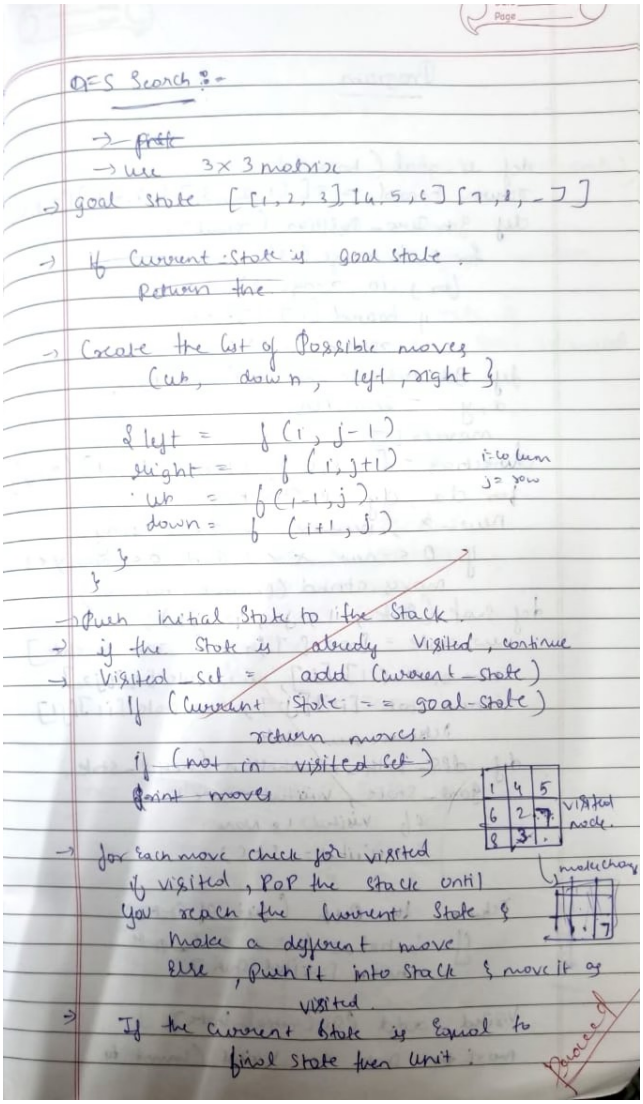
```
Enter Location of VacuumA
Enter status of A0
Enter status of other room1
Vacuum is placed in Location A
Location A is already clean
Location B is Dirty.
Moving RIGHT to the Location B.
COST for moving RIGHT 1
Cost for SUCK2
Location B has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 2
```

Program 2

Implement 8 puzzle problems using Depth First Search (DFS)

Algorithm:





Code:

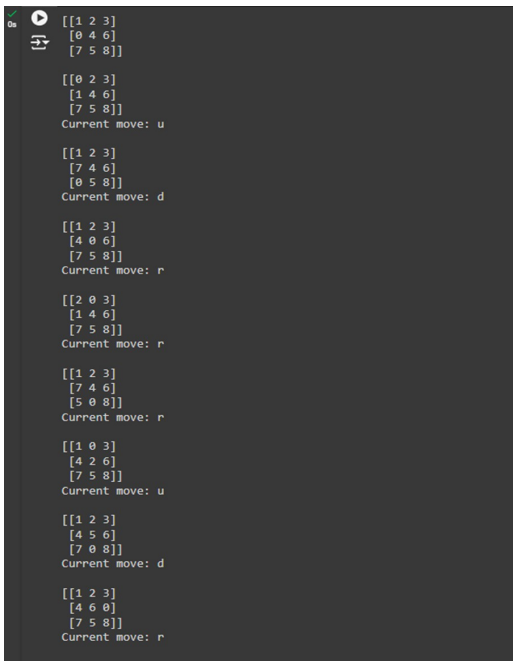
```
import numpy as np


def bfs(src, target):
    queue = [(src, None)]  # State and last move
    visited = set()
```

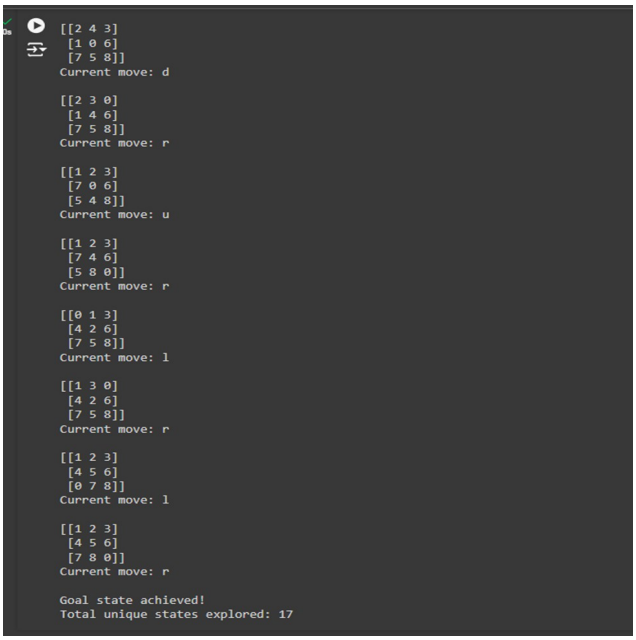
```
    state_count = 0  # Initialize state count


    while queue:
        state, last_move = queue.pop(0)
        state_tuple = tuple(state)  # Convert state to tuple for set
operations


    if state_tuple not in visited:
            visited.add(state_tuple)
            state_count += 1  # Increment the state count


            print_board(state)
            if last_move:
                print(f"Current move: {last_move}\n")


            if state == target:
                print("Goal state achieved!")
                break


            for move, direction in possible_moves(state):
                if tuple(move) not in visited:
                    queue.append((move, direction))


    print(f"Total unique states explored: {state_count}")
```

```
def possible_moves(state):
    b = state.index(0)
    directions = []

    if b not in [0, 1, 2]: directions.append('u')
    if b not in [6, 7, 8]: directions.append('d')
    if b not in [0, 3, 6]: directions.append('l')
    if b not in [2, 5, 8]: directions.append('r')

    return [(gen(state, d, b), d) for d in directions]

def gen(state, direction, b):
    temp = state.copy()

    if direction == 'u': temp[b], temp[b-3] = temp[b-3], temp[b]
    if direction == 'd': temp[b], temp[b+3] = temp[b+3], temp[b]
    if direction == 'l': temp[b], temp[b-1] = temp[b-1], temp[b]
    if direction == 'r': temp[b], temp[b+1] = temp[b+1], temp[b]

    return temp

def print_board(state):
    board = np.array(state).reshape(3, 3)
    print(board)
```

```
# Initial configuration and target configuration
 src = [1,     3,     4, 6,    5, 8]
     2,         0,       7,

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

# Run BFS to solve the

puzzle bfs(src, target)
```

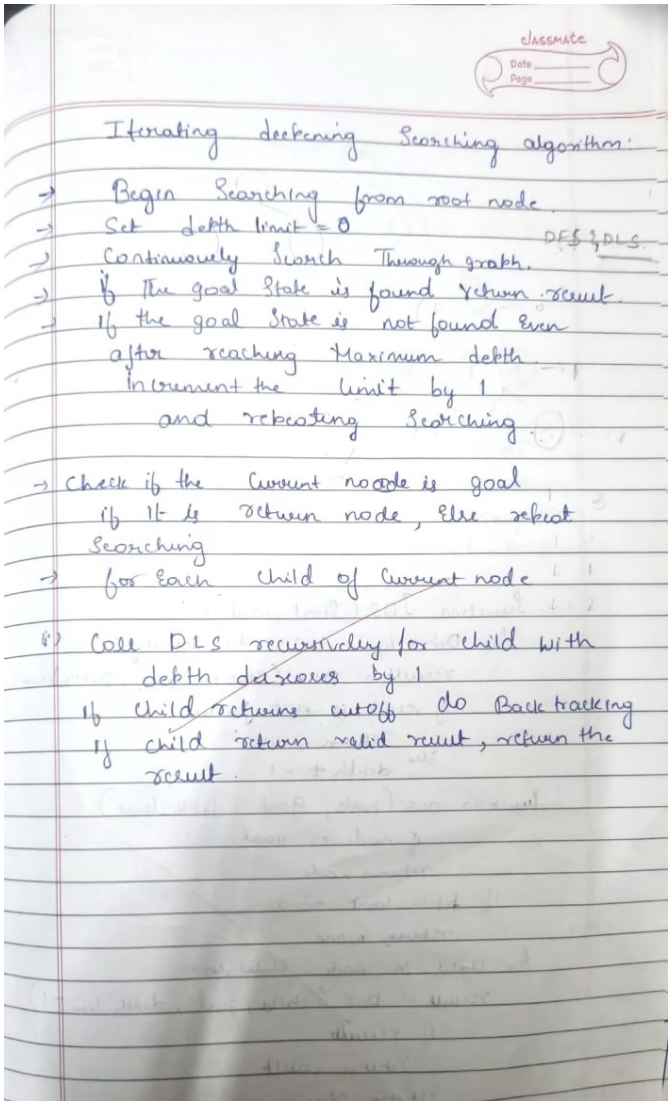
Output:

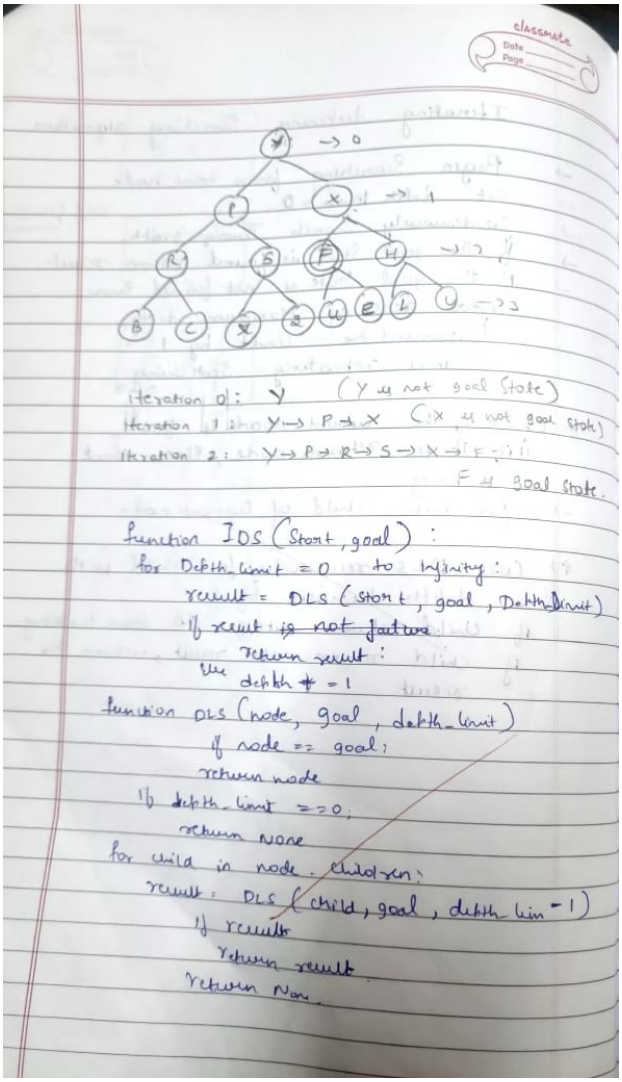




Implement Iterative deepening search algorithm

Algorithm:





Code:

```
import copy



class Node:

    def _init_(self, state, parent=None, action=None, depth=0):

        self.state = state

        self.parent = parent

        self.action = action
```

```
        self.depth = depth

    def __lt__(self, other):
        return self.depth < other.depth

    def expand(self):
        children = []
        row, col = self.find_blank()
        possible_actions = []

        if row > 0:  # Can move the blank tile up
            possible_actions.append('Up')
        if row < 2:  # Can move the blank tile down
            possible_actions.append('Down')
        if col > 0:  # Can move the blank tile left
            possible_actions.append('Left')
        if col < 2:  # Can move the blank tile right
            possible_actions.append('Right')

        for action in possible_actions:
            new_state = copy.deepcopy(self.state)
            if action == 'Up':
                new_state[row][col], new_state[row - 1][col] = new_state[row
- 1][col], new_state[row][col]
            elif action == 'Down':
```

```
                new_state[row][col], new_state[row + 1][col] = new_state[row
+ 1][col], new_state[row][col]

            elif action == 'Left':

                new_state[row][col], new_state[row][col - 1] =
new_state[row][col - 1], new_state[row][col]

            elif action == 'Right':

                new_state[row][col], new_state[row][col + 1] =
new_state[row][col + 1], new_state[row][col]



            children.append(Node(new_state, self, action, self.depth + 1))

        return children



    def find_blank(self):

        for row in range(3):

            for col in range(3):

                if self.state[row][col] == 0:

                    return row, col

        raise ValueError("No blank tile found")



def depth_limited_search(node, goal_state, limit):

    if node.state == goal_state:

        return node

    if node.depth >= limit:

        return None

    for child in node.expand():
```

```
        result = depth_limited_search(child, goal_state, limit)
        if result is not None:
            return result
    return None


def iterative_deepening_search(initial_state, goal_state, max_depth):
    for depth in range(max_depth):
        result = depth_limited_search(Node(initial_state), goal_state, depth)
        if result is not None:
            return result
    return None


def print_solution(node):
    path = []
    while node is not None:
        path.append((node.action, node.state))
        node = node.parent
    path.reverse()


    for action, state in path:
        if action:
            print(f"Action: {action}")
        for row in state:
            print(row)
```

```
        print()


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

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


max_depth = 5

solution = iterative_deepening_search(initial_state, goal_state, max_depth)


if solution:

    print("Solution found:")

    print_solution(solution)

else:

    print("Solution not found.")
```

Output:

```
Solution found:
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]

Action: Right
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

Action: Down
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

Action: Right
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
```

```
initial_state = [[1, 2, 3], [0, 4, 6], [7, 5, 8]]
goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

max_depth = 3
solution = iterative_deepening_search(initial_state, goal_state, max_depth)

if solution:
    print("Solution found:")
    print_solution(solution)
else:
    print("Solution not found.")
```

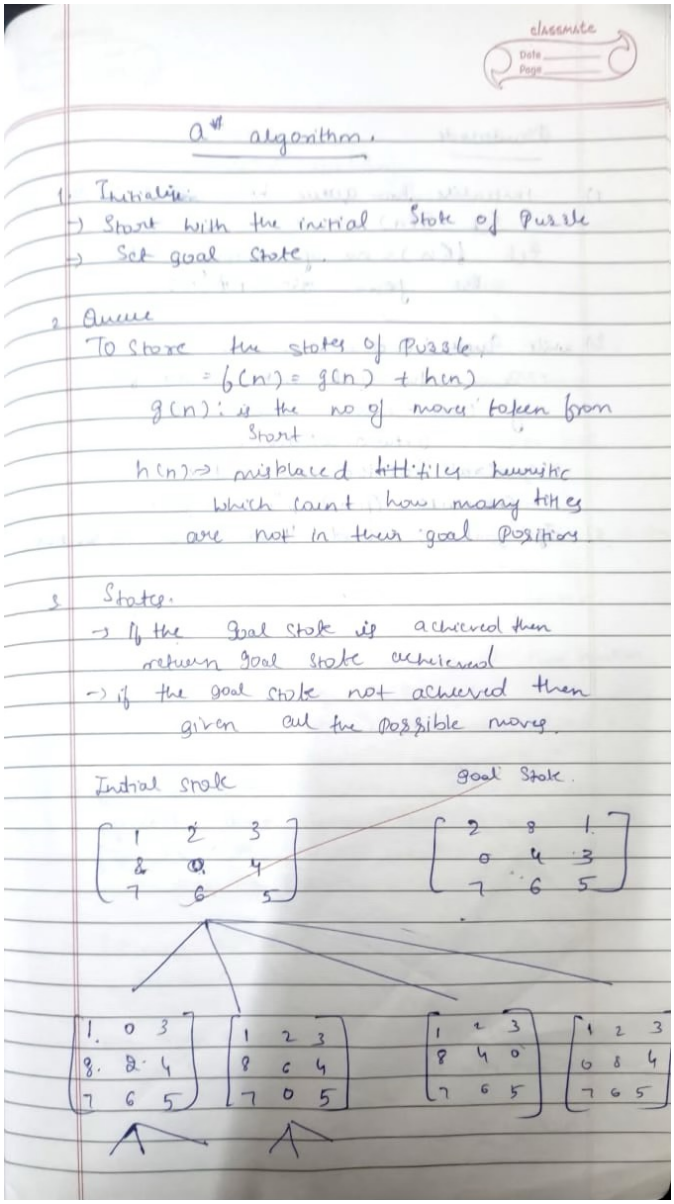
```
Solution not found.
```

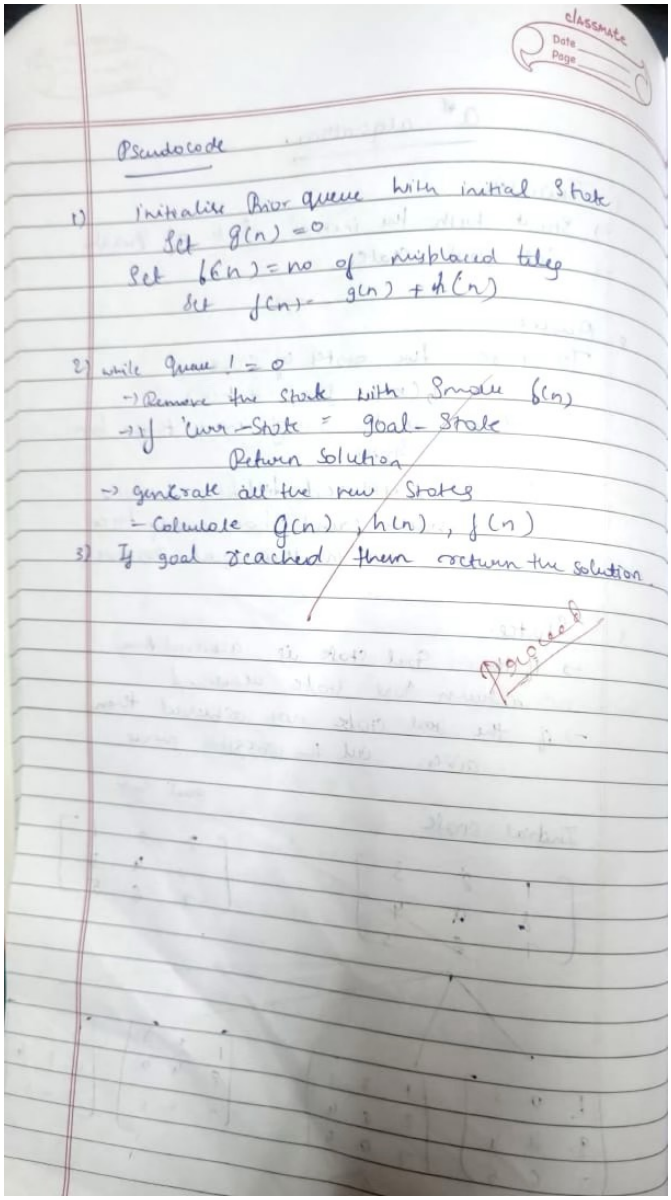
Program 3

Implement A* search algorithm using

a) Number of misplaced tiles

Algorithm:





Code:

```
import heapq

GOAL_STATE =
((1,              2  3)
                  ,  ,

             (8,  0  4)
                  ,  ,

             (7,  6  5)
                  ,  )
```

```
def misplaced_tile(state):
    misplaced = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0 and state[i][j] != GOAL_STATE[i][j]:
                misplaced += 1
    return misplaced
```

```
def find_blank(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
def generate_neighbors(state):
    neighbors = []
    x, y = find_blank(state)
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = [list(row) for row in state] new_state[x]
            [y], new_state[nx][ny] = new_state[nx][ny],
new_state[x][y]
            neighbors.append(tuple(tuple(row) for row in new_state))
    return neighbors
def reconstruct_path(came_from, current):
    path = [current]
    while current in came_from:
        current = came_from[current]
        path.append(current)
    path.reverse()
    return path
```

```
def a_star(start):
    open_list = []
    heapq.heappush(open_list, (0 + misplaced_tile(start), 0, start))
    g_score = {start: 0}
    came_from = {}
    visited = set()
    while open_list:
        _, g, current = heapq.heappop(open_list)

        if current == GOAL_STATE:
            path = reconstruct_path(came_from, current)
            return path, g
        visited.add(current)
        for neighbor in generate_neighbors(current):
            if neighbor in visited:
                continue
            tentative_g = g_score[current] + 1

            if tentative_g < g_score.get(neighbor, float('inf')):
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g
                f_score = tentative_g + misplaced_tile(neighbor)   # f(n) =
g(n) + h(n)

                heapq.heappush(open_list, (f_score, tentative_g, neighbor))
```

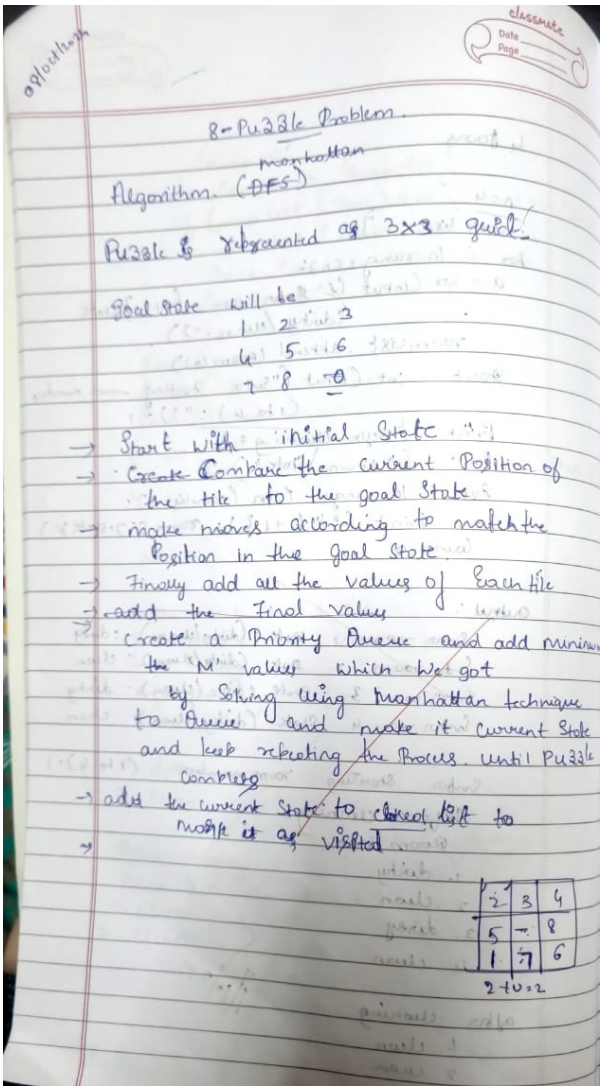
```
    return None, None

def print_state(state):
    for row in state:
        print(row)
    print()


if __name__ == "__main__":
    start_state = ((2, 8, 3),
                   (1, 6, 4),
                   (7, 0, 5))
    print("Initial State:")
    print_state(start_state)
    print("Goal State:")
    print_state(GOAL_STATE)
    solution, cost = a_star(start_state)
    if solution:
        print(f"Goal state achieved with cost: {cost}")
        print("Steps:")
        for step in solution:
            print_state(step)
    else:
        print("No solution found.")
```

Output:

```
Initial State:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

Goal State:
(1, 2, 3)
(8, 0, 4)
(7, 6, 5)

Solution found with cost: 5
Steps:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

(2, 8, 3)
(1, 0, 4)
(7, 6, 5)

(2, 0, 3)
(1, 8, 4)
(7, 6, 5)

(0, 2, 3)
(1, 8, 4)
(7, 6, 5)

(1, 2, 3)
(0, 8, 4)
(7, 6, 5)

(1, 2, 3)
(8, 0, 4)
(7, 6, 5)
```

b)Manhattan Distance

Algorithm:



Code:

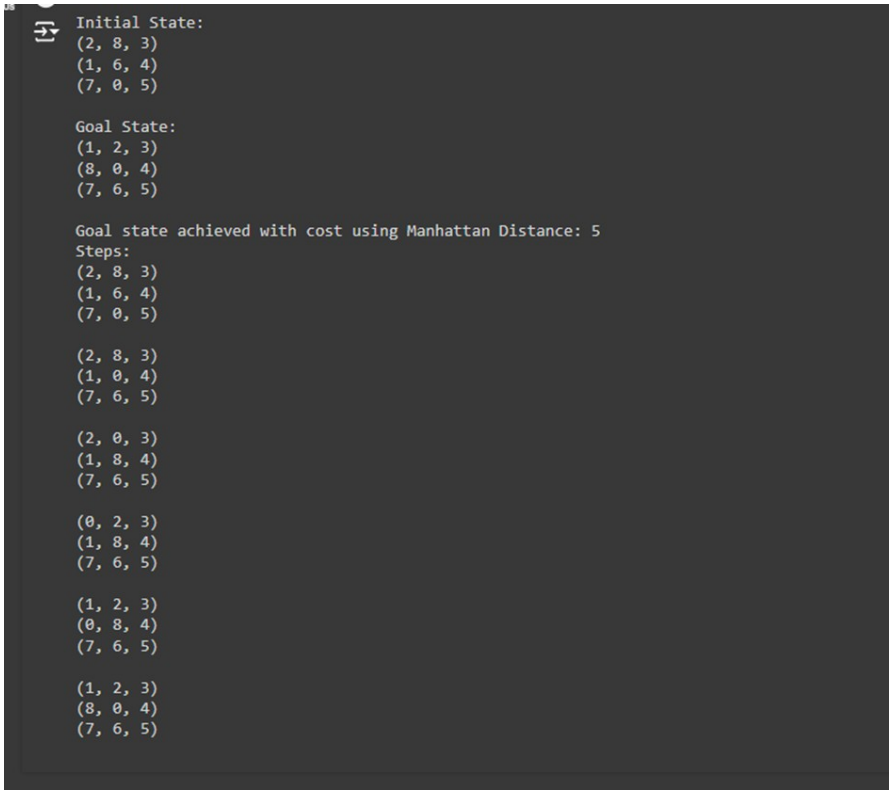
```
import heapq

GOAL_STATE = ((1, 2, 3),
              (8, 0, 4),
              (7, 6, 5))

def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            value = state[i][j]
            if value != 0:

                goal_x, goal_y = divmod(value - 1, 3)
                distance += abs(goal_x - i) + abs(goal_y - j)
    return distance
def find_blank(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
def generate_neighbors(state):
    neighbors = []
    x, y = find_blank(state)
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
```

```
    for dx, dy in directions:

        nx, ny = x + dx, y + dy

        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = [list(row) for row in state] new_state[x]

            [y], new_state[nx][ny] = new_state[nx][ny],

new_state[x][y]

            neighbors.append(tuple(tuple(row) for row in new_state))

    return neighbors



def reconstruct_path(came_from, current):

    path = [current]

    while current in came_from:

        current = came_from[current]

        path.append(current)

    path.reverse()

    return path



def a_star(start):

    open_list = []

    heapq.heappush(open_list, (manhattan_distance(start), 0, start))



    g_score = {start: 0}

    came_from = {}
```

```
    visited = set()

    while open_list:
        f, g, current = heapq.heappop(open_list)

        if current == GOAL_STATE:
            path = reconstruct_path(came_from, current)
            return path, g

        visited.add(current)
        for neighbor in generate_neighbors(current):
            if neighbor in visited:
                continue

            tentative_g = g_score[current] + 1

            if tentative_g < g_score.get(neighbor, float('inf')):
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g
                f_score = tentative_g + manhattan_distance(neighbor)

                heapq.heappush(open_list, (f_score, tentative_g, neighbor))

    return None, None
```

```
def print_state(state):
    for row in state:
        print(row)
    print()


if __name___ == "__main__":
    start_state = ((2, 8, 3),
                   (1, 6, 4),
                   (7, 0, 5))
    print("Initial State:")
    print_state(start_state)
    print("Goal State:")
    print_state(GOAL_STATE)


    solution, cost = a_star(start_state)
    if solution:
        print(f"Goal state achieved with cost: {cost}")
        print("Steps:")
        for step in solution:
            print_state(step)
    else: print("No solution found.")
```

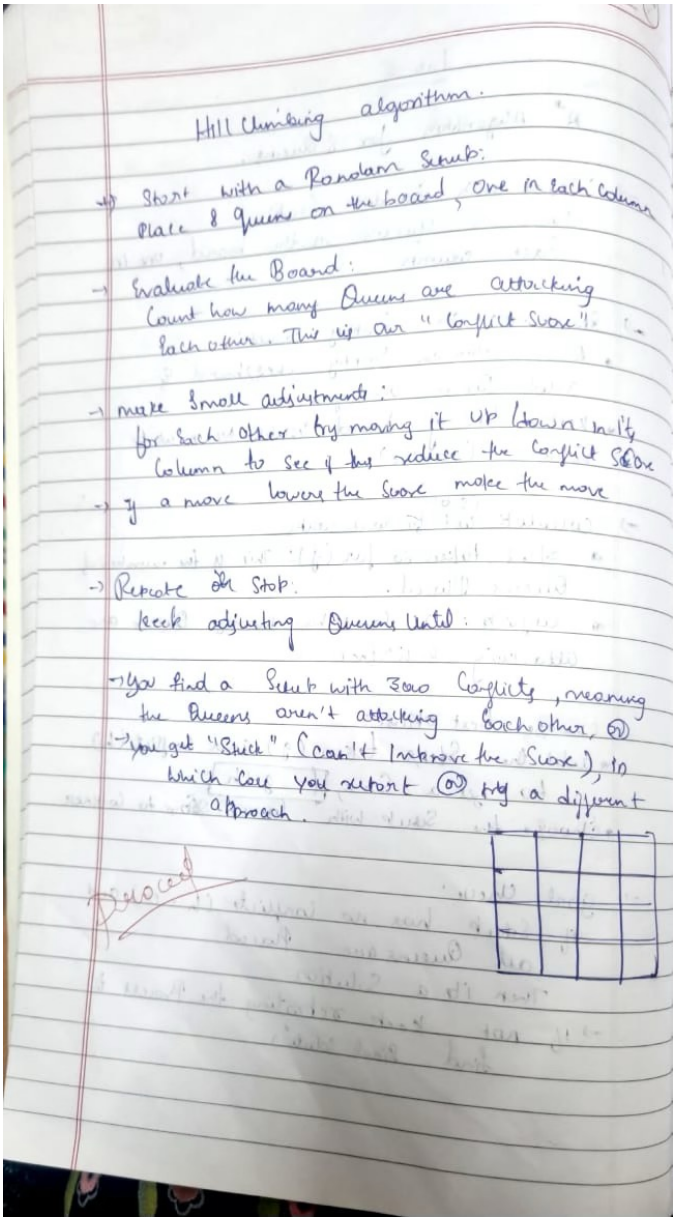
Output:


Initial State:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)
Goal State:
(1, 2, 3)
(8, 0, 4)
(7, 6, 5)
Goal state achieved with cost using Manhattan Distance: 5
Steps:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)
(2, 8, 3)
(1, 0, 4)
(7, 6, 5)
(2, 0, 3)
(1, 8, 4)
(7, 6, 5)
(0, 2, 3)
(1, 8, 4)
(7, 6, 5)
(1, 2, 3)
(0, 8, 4)
(7, 6, 5)
(1, 2, 3)
(8, 0, 4)
(7, 6, 5)

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:



Code:

```
import random

def calculate_cost(state):

    """Calculate the number of conflicts in the current state."""

    cost = 0

    n = len(state)

    for i in range(n):

        for j in range(i + 1, n):

            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i -
j):

                cost += 1

    return cost
def get_neighbors(state):

    """Generate all possible neighbors by moving each queen in its column."""

    neighbors = []

    n = len(state)

    for col in range(n):

        for row in range(n):

            if state[col] != row:  # Move the queen in column `col` to a
different row

                new_state = list(state)

                new_state[col] = row

                neighbors.append(new_state)

    return neighbors
```

```
def hill_climbing(n, max_iterations=1000):
    """Perform hill climbing search to solve the N-Queens problem."""
    current_state = [random.randint(0, n - 1) for _ in range(n)]
    current_cost = calculate_cost(current_state)

    for iteration in range(max_iterations):
        if current_cost == 0:  # Found a solution
            return current_state

        neighbors = get_neighbors(current_state)
        neighbor_costs = [(neighbor, calculate_cost(neighbor)) for neighbor
in neighbors]
        next_state, next_cost = min(neighbor_costs, key=lambda x: x[1])

        if next_cost >= current_cost:  # No improvement found
            print(f"Local maximum reached at iteration {iteration}.
Restarting...")
            return None  # Restart with a new random state
     current_state, current_cost = next_state, next_cost
        print(f"Iteration {iteration}: Current state: {current_state}, Cost:
{current_cost}")

    print(f"Max iterations reached without finding a solution.")
    return None
# Get user-defined input for the number of queens
```

```
try:
    n = int(input("Enter the number of queens (N): "))
    if n <= 0:
        raise ValueError("N must be a positive integer.")
except ValueError as e:
    print(e)
    n = 4  # Default to 4 if input is invalid
solution = None
# Keep trying until a solution is found
while solution is None:
    solution = hill_climbing(n)


print(f"Solution found: {solution}")
```

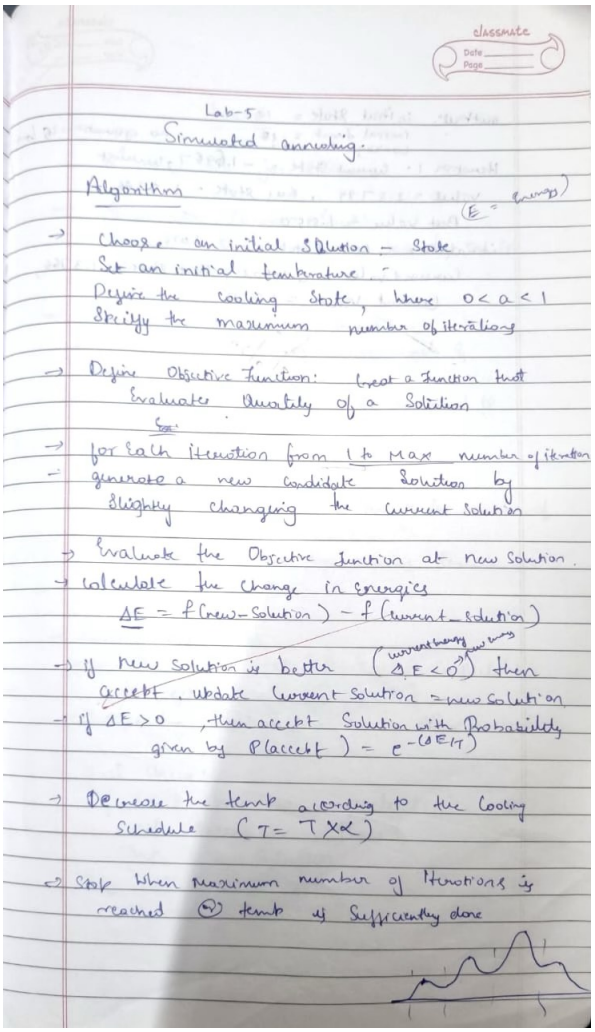
Output:

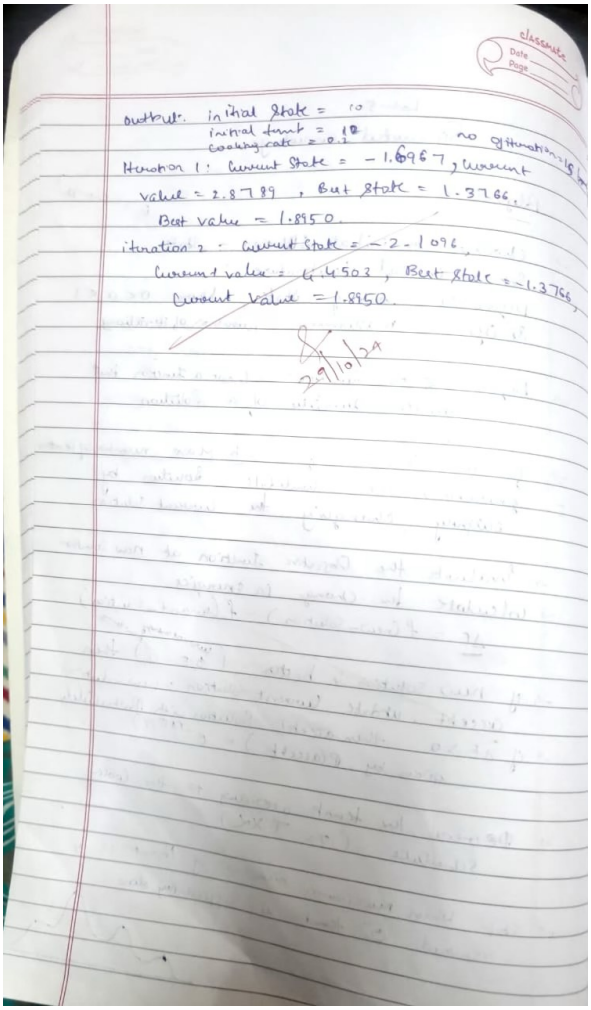
```
Enter the number of queens (N): 4
Iteration 0: Current state: [1, 3, 2, 0], Cost: 1
Local maximum reached at iteration 1. Restarting...
Iteration 0: Current state: [1, 1, 2, 0], Cost: 2
Iteration 1: Current state: [1, 3, 2, 0], Cost: 1
Local maximum reached at iteration 2. Restarting...
Iteration 0: Current state: [1, 3, 0, 3], Cost: 1
Iteration 1: Current state: [1, 3, 0, 2], Cost: 0
Solution found: [1, 3, 0, 2]
```

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:





Code:

```
import mlrose_hiive as mlrose
import numpy as np


def queens_max(position):
    n = len(position)
    attacks = 0


    for i in range(n):
        for j in range(i + 1, n):
            # Check if queens attack each other
            if position[i] == position[j] or abs(position[i] - position[j])
== j - i:
                attacks += 1


    # The fitness is the total number of pairs of queens minus the number of
attacks
    return (n * (n - 1) // 2) - attacks


# Define the custom fitness function
objective = mlrose.CustomFitness(queens_max)


# Set up the optimization problem
problem = mlrose.DiscreteOpt(length=8, fitness_fn=objective, maximize=True,
max_val=8)
```

```
T = mlrose.ExpDecay()



# Define the initial position

initial_position = np.array([4, 6, 1, 5, 2, 0, 3, 7])



# Run the simulated annealing algorithm

best_state, best_fitness, _ = mlrose.simulated_annealing(problem, schedule=T,
max_attempts=500, max_iters=5000, init_state=initial_position)



print('The best position found is:', best_state)

print('The number of queens that are not attacking each other is:',
best_fitness)
```

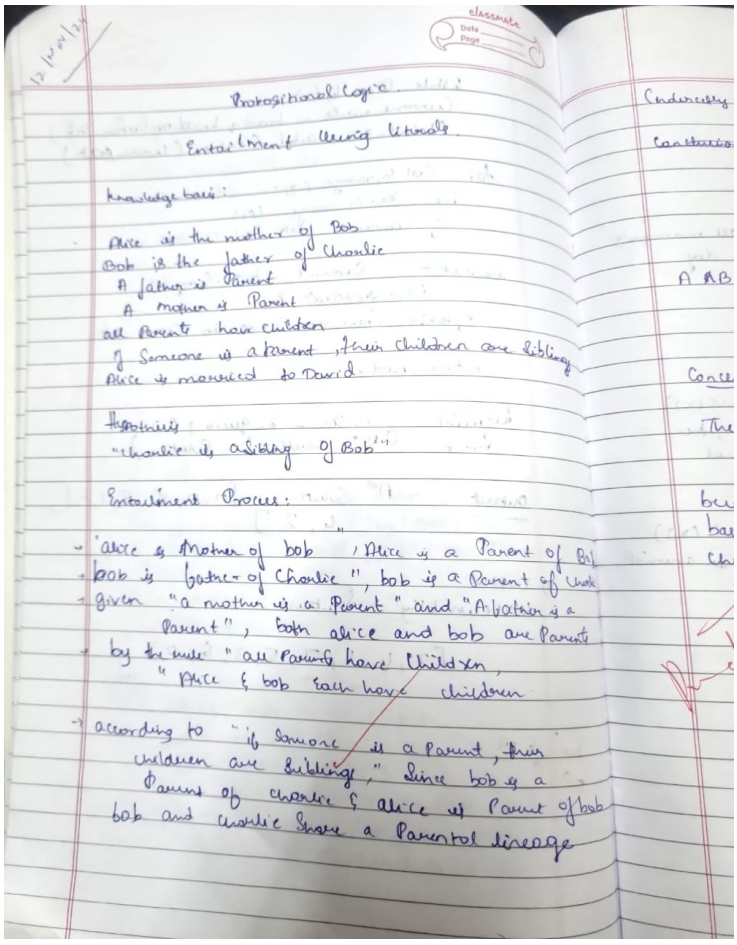
Output:

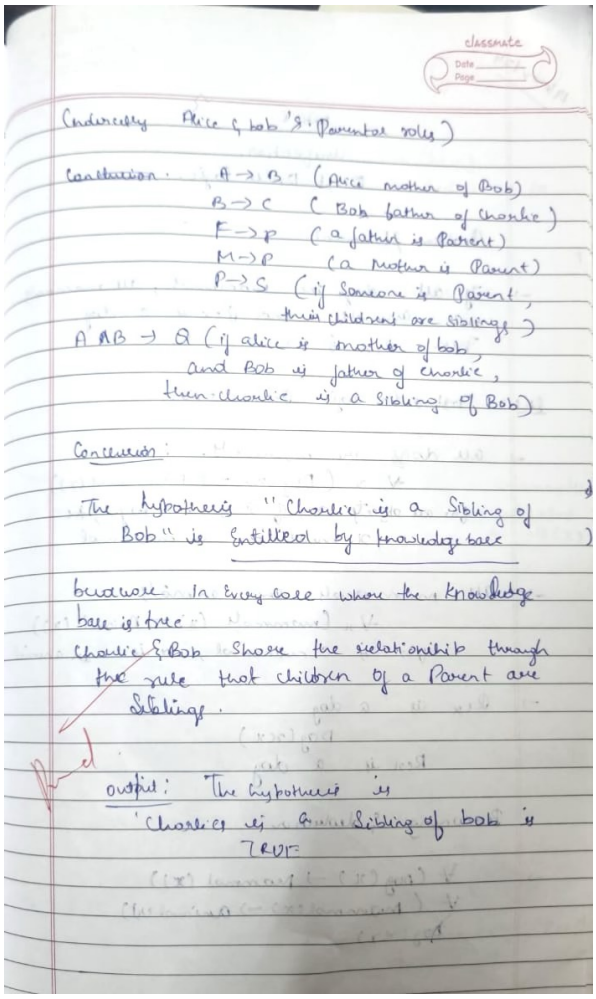
```
The best position found is: [2 5 7 1 3 0 6 4]
The number of queens that are not attacking each other is: 28.0
```

Program 6

Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.

Algorithm:





Code:

```
from itertools import product


def pl_true(sentence, model):
    """Evaluates if a sentence is true in a given model."""
    if isinstance(sentence, str):
        return model.get(sentence, False)
    elif isinstance(sentence, tuple) and len(sentence) == 2:  # NOT operation
        operator, operand = sentence
        if operator == "NOT":
            return not pl_true(operand, model)
    elif isinstance(sentence, tuple) and len(sentence) == 3:
        operator, left, right = sentence
        if operator == "AND":
            return pl_true(left, model) and pl_true(right, model)
        elif operator == "OR":
            return pl_true(left, model) or pl_true(right, model)
        elif operator == "IMPLIES":
            return not pl_true(left, model) or pl_true(right, model)
        elif operator == "IFF":
            return pl_true(left, model) == pl_true(right, model)


def tt_entails(kb, alpha, symbols):
    """Checks if KB entails alpha using truth-table enumeration."""
```

```
    all_models = product([False, True], repeat=len(symbols))
    valid_models = []

    for values in all_models:
        model = dict(zip(symbols, values))
        kb_value = pl_true(kb, model)
        alpha_value = pl_true(alpha, model)

        if kb_value:  # If KB is true in this model
            if not alpha_value:  # If KB is true but α is not, entailment fails
                return False, None
            else:
                valid_models.append(model)

    return True, valid_models

def print_truth_table(kb, alpha, symbols):
    """Generates and prints the truth table for KB and α."""
    headers = ["A      ", "B      ", "C      ", "A ∨ C    ", "B ∨ ¬C ", "KB ", "α     "]
    print(" | ".join(headers))
    print("-" * (len(headers) * 9))  # Separator line

    # Generate all combinations of truth values
```

```
    for values in product([False, True], repeat=len(symbols)):
        model = dict(zip(symbols, values))

        # Evaluate sub-expressions and main expressions
        a_or_c = pl_true(("OR", "A", "C"), model)
        b_or_not_c = pl_true(("OR", "B", ("NOT", "C")), model)
        kb_value = pl_true(kb, model)
        alpha_value = pl_true(alpha, model)

        # Print the truth table row
        row = values + (a_or_c, b_or_not_c, kb_value, alpha_value)
        row_str = " | ".join(str(v).ljust(7) for v in row)

        # Highlight rows where both KB and α are true
        if kb_value and alpha_value:
            print(f"\033[92m{row_str}\033[0m")   # Green color for rows where
KB and α are true
        else:
            print(row_str)

# Define the knowledge base and query
symbols = ["A", "B", "C"]
kb = ("AND", ("OR", "A", "C"), ("OR", "B", ("NOT", "C")))
alpha = ("OR", "A", "B")
```

```
# Print the truth table

print_truth_table(kb, alpha, symbols)



# Run the truth-table entailment check

entailment, models = tt_entails(kb, alpha, symbols)



# Print the result

print("\nResult:")

if entailment:

    print("KB entails α.")

    print("The values of A, B, C for which KB and α are true:")

    for model in models:

        print(model)

else:

    print("KB does not entail α.")
```

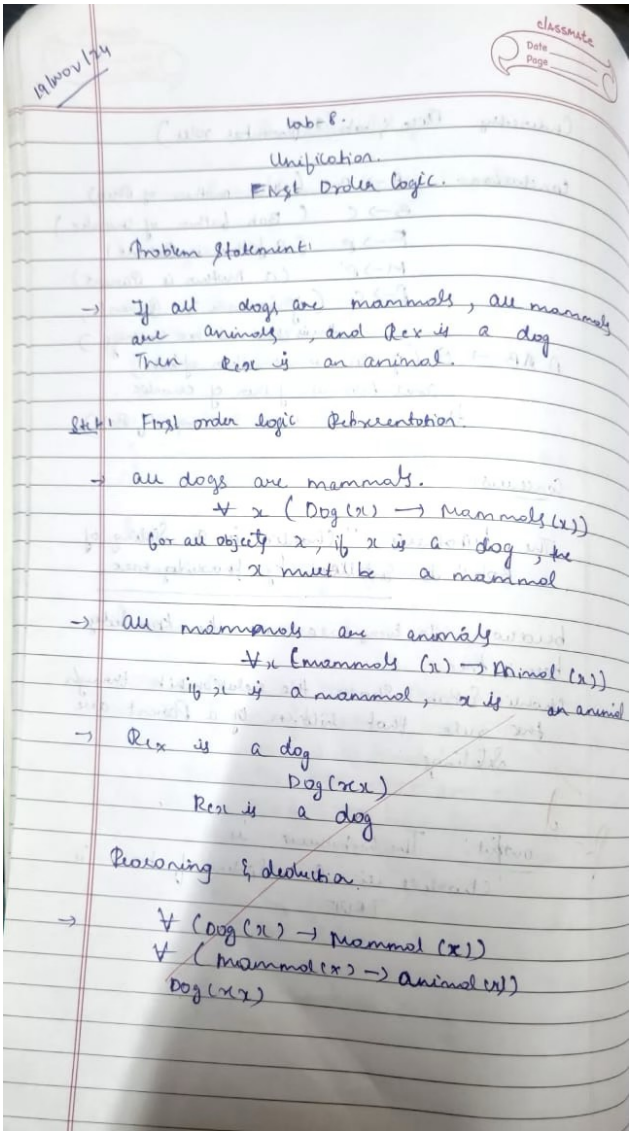
Output:

```
A     | B     | C     | A ∨ C  | B ∨ ¬C | KB     | α
-------------------------------------------------------------
False | False | False | False  | True   | False  | False
False | False | True  | True   | False  | False  | False
False | True  | False | False  | True   | False  | True
False | True  | True  | True   | True   | True   | True
True  | False | False | True   | True   | True   | True
True  | False | True  | True   | False  | False  | True
True  | True  | False | True   | True   | True   | True
True  | True  | True  | True   | True   | True   | True

Result:
KB entails α.
The values of A, B, C for which KB and α are true:
{'A': False, 'B': True, 'C': True}
{'A': True, 'B': False, 'C': False}
{'A': True, 'B': True, 'C': False}
{'A': True, 'B': True, 'C': True}
```

Program 7

Implement unification in first order logic.

Algorithm:


Lab-8.

Unification.

First Order Logic.

Problem Statement:

→ If all dogs are mammals, all mammals are animals, and Rex is a dog Then Rex is an animal.

Step1: First order logic Representation.

→ all dogs are mammals.

∀ x (Dog(x) → Mammals(x))

for all object x, if x is a dog, the x must be a mammal

→ All mammals are animals

∀x (mammals(x) → Animal(x))

if x is a mammal, x is an animal

→ Rex is a dog

Dog(Rex)

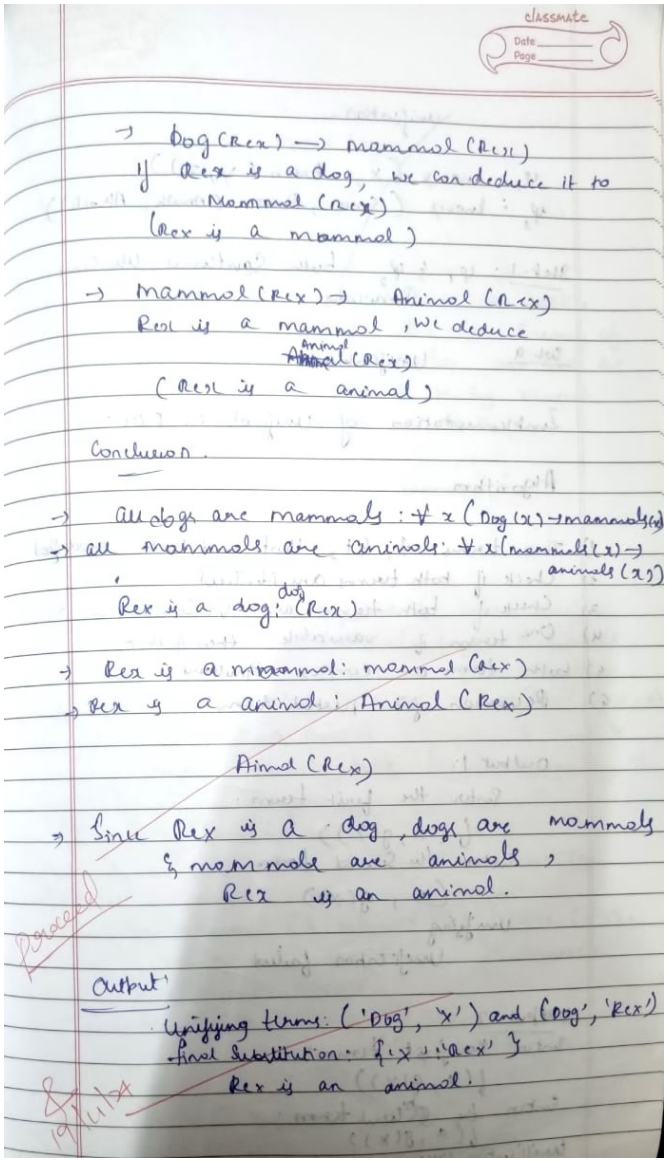
Rex is a dog

Reasoning & deduction

→ ∀ (Dog(x) → Mammal(x))

∀ (mammal(x) → animal(x))

Dog(Rex)


→ Dog(Rex) → mammal(Rex)

If Rex is a dog, we can deduce it to Mammal(Rex)

(Rex is a mammal)

→ Mammal(Rex) → Animal(Rex)

Rex is a mammal, we deduce Animal(Rex)

(Rex is a animal)

Conclusion.

→ All dogs are mammals: ∀ x (Dog(x) → mammals(x))

→ all mammals are animals: ∀ x (mammals(x) → animals(x))

Rex is a dog: dog(Rex)

→ Rex is a mammal: mammal(Rex)

→ Rex is a animal: Animal(Rex)

Animal(Rex)

→ Since Rex is a dog, dogs are mammals & mammals are animals, Rex is an animal.

Output:

Unifying terms: ('Dog', 'x') and ('Dog', 'Rex')

Final Substitution: {'x': 'Rex'}

Rex is an animal.

Code:

```
def unify(expr1, expr2, subst=None):
    if subst is None:
        subst = {}

    # Apply substitutions to both expressions
    expr1 = apply_substitution(expr1, subst)
    expr2 = apply_substitution(expr2, subst)

    # Base case: Identical expressions
    if expr1 == expr2:
        return subst

    # If expr1 is a variable
    if is_variable(expr1):
        return unify_variable(expr1, expr2, subst)

    # If expr2 is a variable
    if is_variable(expr2):
        return unify_variable(expr2, expr1, subst)

    # If both are compound expressions (e.g., f(a), P(x, y))
    if is_compound(expr1) and is_compound(expr2):
        if expr1[0] != expr2[0] or len(expr1[1]) != len(expr2[1]):
```

```
            return None  # Predicate/function symbols or arity mismatch
        for arg1, arg2 in zip(expr1[1], expr2[1]):
            subst = unify(arg1, arg2, subst)
            if subst is None:
                return None
        return subst

    # If they don't unify
    return None

def unify_variable(var, expr, subst):
    """Handle variable unification."""
    if var in subst:  # Variable already substituted
        return unify(subst[var], expr, subst)
    if occurs_check(var, expr, subst):  # Occurs-check
        return None
    subst[var] = expr
    return subst

def apply_substitution(expr, subst):
    """Apply the current substitution set to an expression."""
    if is_variable(expr) and expr in subst:
        return apply_substitution(subst[expr], subst)
    if is_compound(expr):
```

```
        return (expr[0], [apply_substitution(arg, subst) for arg in expr[1]])
    return expr

def occurs_check(var, expr, subst):
    """Check for circular references."""
    if var == expr:
        return True
    if is_compound(expr):
        return any(occurs_check(var, arg, subst) for arg in expr[1])
    if is_variable(expr) and expr in subst:
        return occurs_check(var, subst[expr], subst)
    return False

def is_variable(expr):
    """Check if the expression is a variable."""
    return isinstance(expr, str) and expr.islower()

def is_compound(expr):
    """Check if the expression is a compound expression."""
    return isinstance(expr, tuple) and len(expr) == 2 and isinstance(expr[1],
list)

# Testing the algorithm with the given cases
if __name__ == "__main__":
    # Case 1: p(f(a), f(b)) and p(x, x)
```

```
    expr1 = ("p", [("f", ["a"]), ("g", ["b"])])

    expr2 = ("p", ["x", "x"])

    result = unify(expr1, expr2)

    print("Case 1 Result:", result)


    # Case 2: p(b, x, f(g(z))) and p(z, f(y), f(y))

    expr1 = ("p", ["b", "x", ("f", [("g", ["z"])])])

    expr2 = ("p", ["z", ("f", ["y"]), ("f", ["y"])])

    result = unify(expr1, expr2)

    print("Case 2 Result:", result)
```

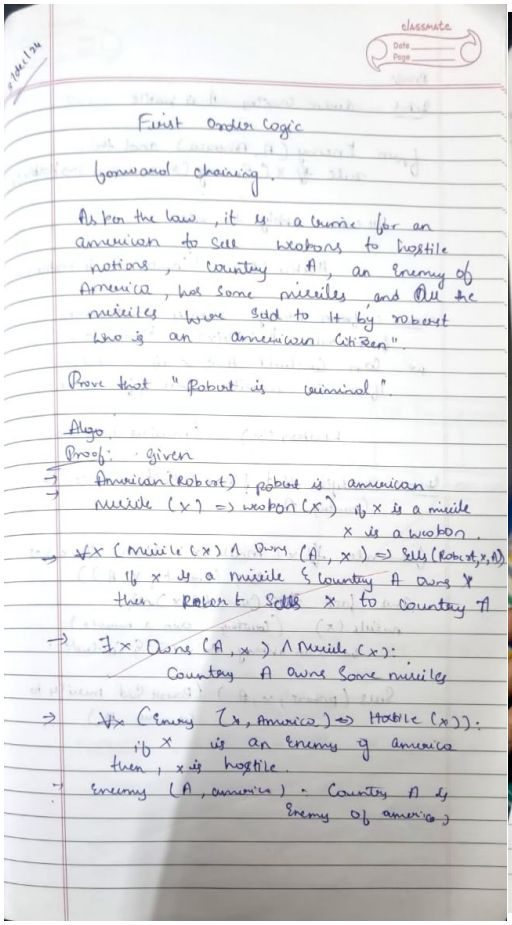
Output:

```
Case 1 Result: None
Case 2 Result: {'b': 'z', 'x': ('f', ['y']), 'y': ('g', ['z'])}
```

Program 8

Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.

Algorithm:


First Order Logic

forward chaining.

As per the law, it is a crime for an American to sell weapons to hostile nations, Country A, an Enemy of America, has some missiles, and all the missiles were sold to it by Robert who is an American citizen".

Prove that "Robert is criminal".

Algo

Proof: Given

→ American(Robert): Robert is american

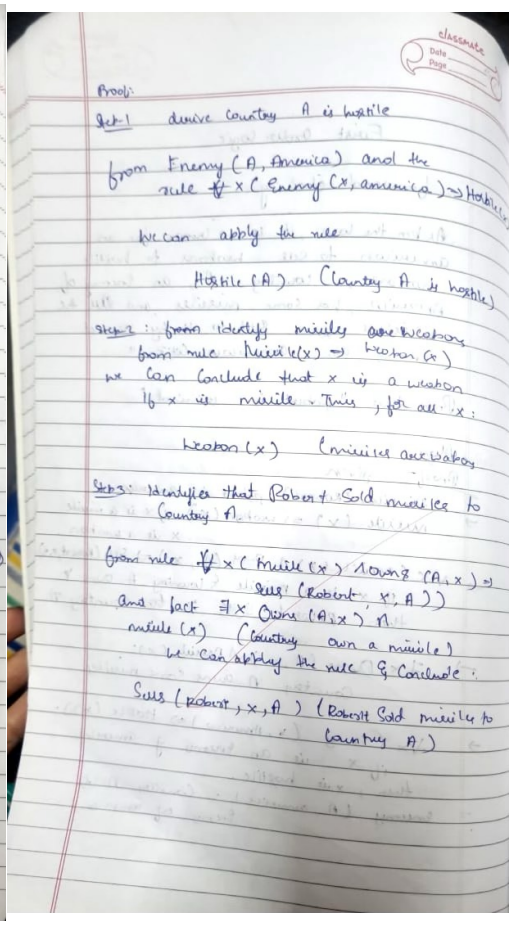
→ missile(x) => weapon(x): If x is a missile x is a weapon.

→ ∀x (missile(x) ∧ Owns(A, x) => Sells(Robert, x, A)): If x is a missile & Country A owns x then Robert Sells x to Country A

→ ∃x Owns(A, x) ∧ missile(x): Country A owns some missiles

→ ∀x (Enemy(x, America) => Hostile(x)): if x is an Enemy of America then, x is hostile.

→ Enemy(A, America): Country A is Enemy of America


Proof:

Fact 1 derive Country A is hostile

from Enemy(A, America) and the rule ∀x (Enemy(x, America) => Hostile)

we can apply the rule

Hostile(A) (Country A is hostile)

Step 2: from identify missiles are weapons from rule missile(x) => weapon(x)

we can conclude that x is a weapon if x is missile. This, for all x:

weapon(x) (missiles are weapons)

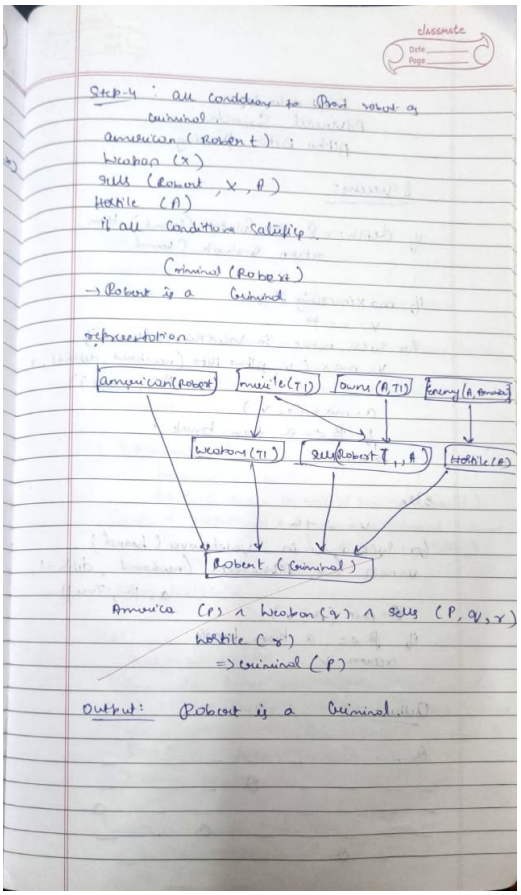
Step 3: identifies that Robert sold missiles to Country A.

from rule ∀x (missile(x) ∧ owns(A, x) => Sells(Robert, x, A))

and fact ∃x Owns(A, x) ∧ missile(x) (Country own a missile)

we can apply the rule & Conclude:

Sells(Robert, x, A) (Robert sold missile to Country A)


Step-4: all conditions to Prove robot is criminal

American(Robert)

Weapon(x)

Sells(Robert, x, A)

Hostile(A)

if all conditions satisfies

Criminal(Robert)

→ Robert is a Criminal

representation

America(p) ∧ Weapon(q) ∧ Sells(P, q, r) ∧ Hostile(r) => Criminal(P)

Output: Robert is a Criminal.

Code:

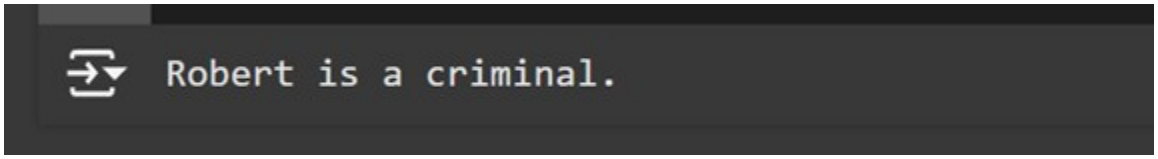
```
# Define the knowledge base with facts and rules
knowledge_base = [
    # Rule: Selling weapons to a hostile nation makes one a criminal
    {
        "type": "rule",
        "if": [
            {"type": "sells", "seller": "?X", "item": "?Z", "buyer": "?Y"},
            {"type": "hostile_nation", "nation": "?Y"},
            {"type": "citizen", "person": "?X", "country": "america"}
        ],
        "then": {"type": "criminal", "person": "?X"}
    },
    # Facts
    {"type": "hostile_nation", "nation": "CountryA"},
    {"type": "sells", "seller": "Robert", "item": "missiles", "buyer": "CountryA"},
    {"type": "citizen", "person": "Robert", "country": "america"}
]

# Forward chaining function
def forward_reasoning(kb, query):
    inferred = []  # Track inferred facts
    while True:
        new_inferences = []
        for rule in [r for r in kb if r["type"] == "rule"]:
```

```
            conditions = rule["if"]
            conclusion = rule["then"]
            substitutions = {}
            if match_conditions(conditions, kb, substitutions):
                inferred_fact = substitute(conclusion, substitutions)
                if inferred_fact not in kb and inferred_fact not in new_inferences:
                    new_inferences.append(inferred_fact)
        if not new_inferences:
            break
        kb.extend(new_inferences)
        inferred.extend(new_inferences)
    return query in kb

# Helper to match conditions
def match_conditions(conditions, kb, substitutions):
    for condition in conditions:
        if not any(match_fact(condition, fact, substitutions) for fact in kb):
            return False
    return True

# Helper to match a single fact
def match_fact(condition, fact, substitutions):
    if condition["type"] != fact["type"]:
        return False
    for key, value in condition.items():
        if key == "type":
```

```
            continue
        if isinstance(value, str) and value.startswith("?"):  # Variable
            variable = value
            if variable in substitutions:
                if substitutions[variable] != fact[key]:
                    return False
            else:
                substitutions[variable] = fact[key]
        elif fact[key] != value:  # Constant
            return False
    return True

# Substitute variables with their values
def substitute(conclusion, substitutions):
    result = conclusion.copy()
    for key, value in conclusion.items():
        if isinstance(value, str) and value.startswith("?"):
            result[key] = substitutions[value]
    return result

# Query: Is Robert a criminal?
query = {"type": "criminal", "person": "Robert"}

# Run the reasoning algorithm
if forward_reasoning(knowledge_base, query):
    print("Robert is a criminal.")
```

```
else:
    print("Could not prove that Robert is a criminal.")
```

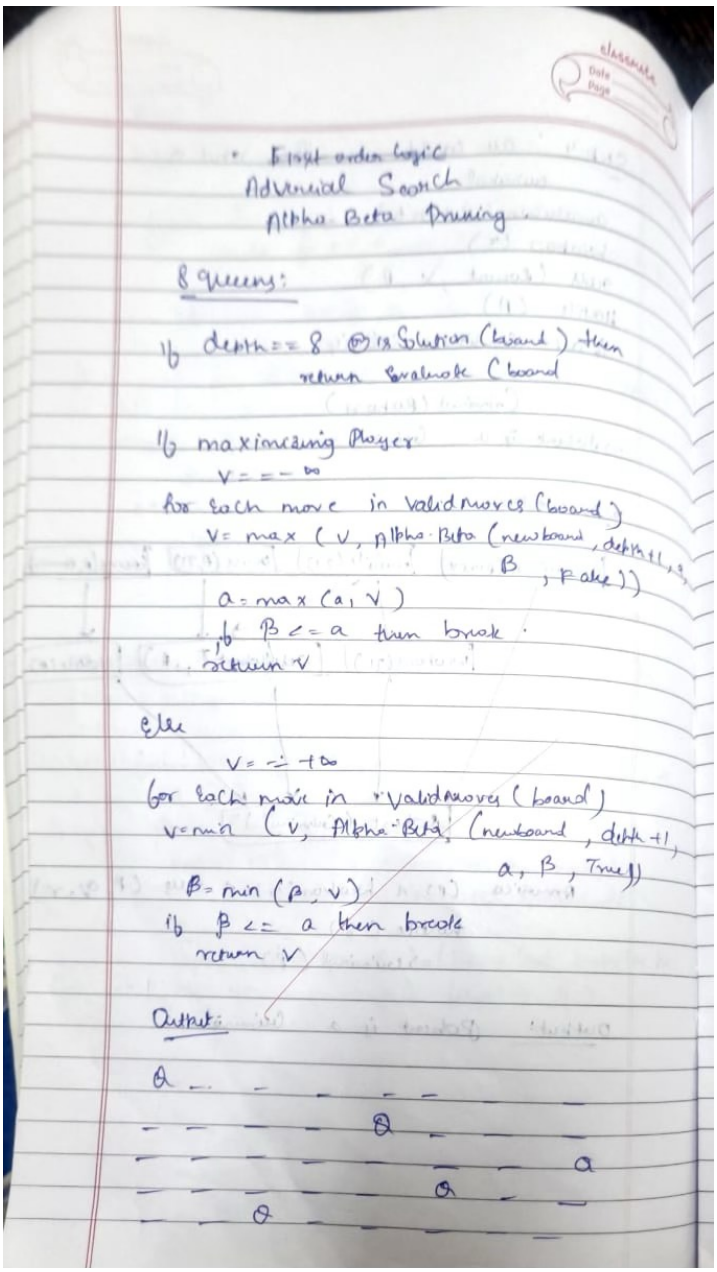
Output:



Program 9

Implement Alpha-Beta Pruning.

Algorithm:



Code:

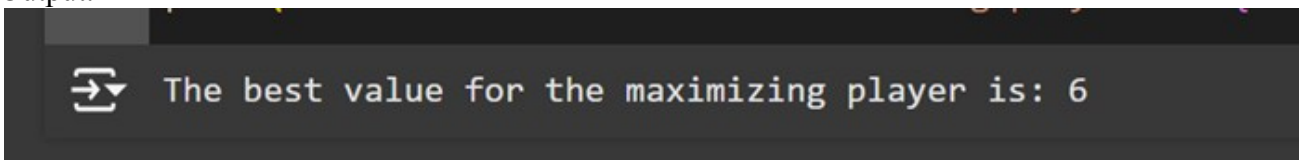
```
import math
def minimax(node, depth, is_maximizing):
    """
    Implement the Minimax algorithm to solve the decision tree.

    Parameters:
    node (dict): The current node in the decision tree, with the following structure:
        {
            'value': int,
            'left': dict or None,
            'right': dict or None
        }
    depth (int): The current depth in the decision tree.
    is_maximizing (bool): Flag to indicate whether the current player is the maximizing player.
    Returns:
    int: The utility value of the current node.
    """
    # Base case: Leaf node
    if node['left'] is None and node['right'] is None:
        return node['value']
    # Recursive case
    if is_maximizing:
        best_value = -math.inf
        if node['left']:
```

```
            best_value = max(best_value, minimax(node['left'], depth + 1, False))
        if node['right']:
            best_value = max(best_value, minimax(node['right'], depth + 1, False))
        return best_value
    else:
        best_value = math.inf
        if node['left']:
            best_value = min(best_value, minimax(node['left'], depth + 1, True))
        if node['right']:
            best_value = min(best_value, minimax(node['right'], depth + 1, True))
        return best_value
# Example usage
decision_tree = {
    'value': 5,
    'left': {
        'value': 6,
        'left': {
            'value': 7,
            'left': {
                'value': 4,
                'left': None,
                'right': None
            },
            'right': {
                'value': 5,
                'left': None,
```

```
            'right': None
        }
    },
    'right': {
        'value': 3,
        'left': {
            'value': 6,
            'left': None,
            'right': None
        },
        'right': {
            'value': 9,
            'left': None,
            'right': None
        }
    }
},
'right': {
    'value': 8,
    'left': {
        'value': 7,
        'left': {
            'value': 6,
            'left': None,
            'right': None
        },
```

```
            'right': {
                'value': 9,
                'left': None,
                'right': None
            }
        },
        'right': {
            'value': 8,
            'left': {
                'value': 6,
                'left': None,
                'right': None
            },
            'right': None
        }
    }
}
# Find the best move for the maximizing player
best_value = minimax(decision_tree, 0, True)
print(f"The best value for the maximizing player is: {best_value}")
```

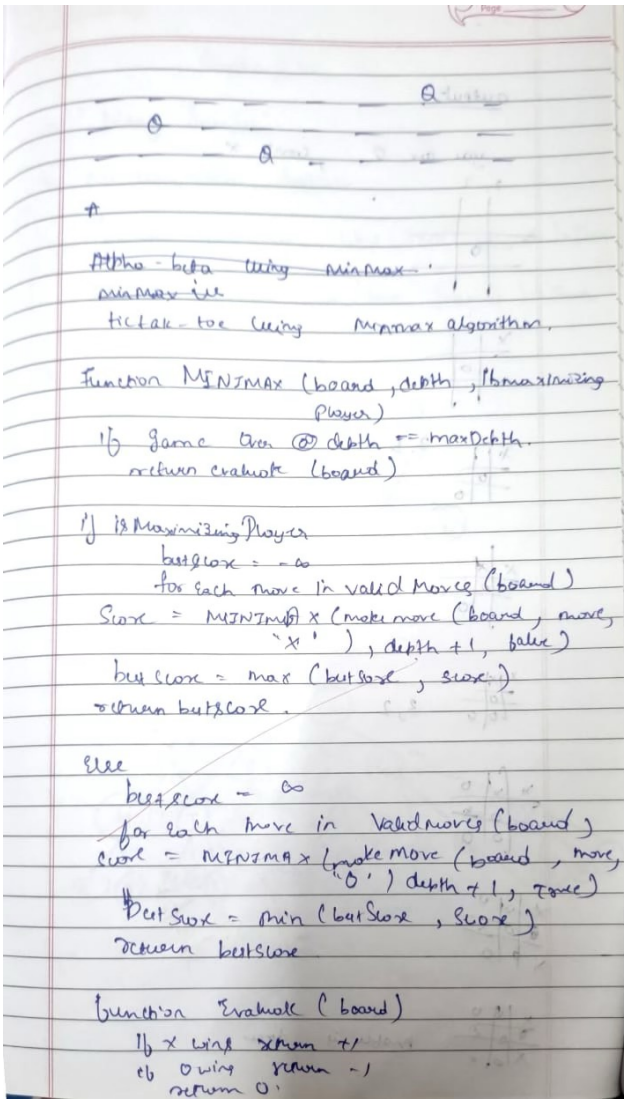
Output:

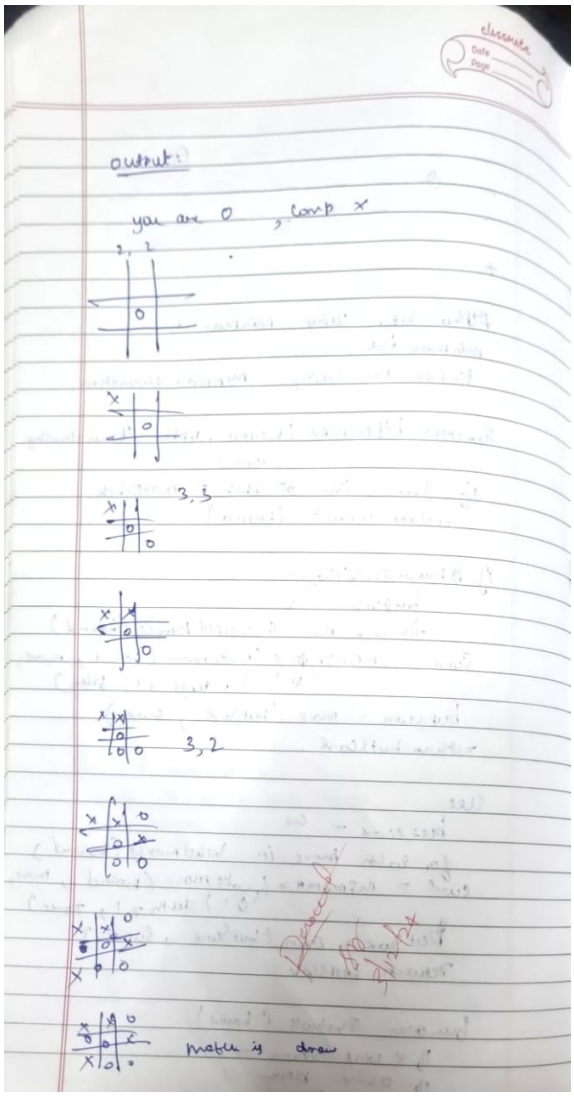


PROGRAM 10:

Tic-Tac-Toe using the Minimax algorithm:

ALGORTIHM:





CODE:

```
import math

# Initialize the board
def create_board():
    return [[' ' for _ in range(3)] for _ in range(3)]

# Display the board
def print_board(board):
    for row in board:
        print('|'.join(row))
        print('-' * 5)

# Check for winner
def check_winner(board):
    # Check rows and columns
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] != ' ':
            return board[i][0]
        if board[0][i] == board[1][i] == board[2][i] != ' ':
            return board[0][i]
    # Check diagonals
    if board[0][0] == board[1][1] == board[2][2] != ' ':
        return board[0][0]
    if board[0][2] == board[1][1] == board[2][0] != ' ':
        return board[0][2]
    # Check for draw
    if all(cell != ' ' for row in board for cell in row):
        return 'Draw'
    return None

# Minimax algorithm
def minimax(board, depth, is_maximizing):
    winner = check_winner(board)
    if winner == 'X':  # AI wins
        return 10 - depth
    if winner == 'O':  # Human wins
        return depth - 10
    if winner == 'Draw':  # Draw
        return 0

    if is_maximizing:
        max_eval = -math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == ' ':
                    board[i][j] = 'X'
```

```
                eval = minimax(board, depth + 1, False)
                board[i][j] = ' '
                max_eval = max(max_eval, eval)
        return max_eval
    else:
        min_eval = math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == ' ':
                    board[i][j] = 'O'
                    eval = minimax(board, depth + 1, True)
                    board[i][j] = ' '
                    min_eval = min(min_eval, eval)
        return min_eval

# Find the best move for AI
def find_best_move(board):
    best_value = -math.inf
    best_move = None
    for i in range(3):
        for j in range(3):
            if board[i][j] == ' ':
                board[i][j] = 'X'
                move_value = minimax(board, 0, False)
                board[i][j] = ' '
                if move_value > best_value:
                    best_value = move_value
                    best_move = (i, j)
    return best_move

# Main game loop
def tic_tac_toe():
    board = create_board()
    print("Tic Tac Toe! You are 'O', AI is 'X'.")
    while True:
        print_board(board)
        if check_winner(board) is not None:
            result = check_winner(board)
            if result == 'Draw':
                print("It's a draw!")
            else:
                print(f"The winner is: {result}")
            break

        # Human move
        print("Your turn!")
        while True:
```

```
            try:
                row, col = map(int, input("Enter your move (row and column, 0-2): ").split())
                if board[row][col] == ' ':
                    board[row][col] = 'O'
                    break
                else:
                    print("Cell already taken. Try again.")
            except (ValueError, IndexError):
                print("Invalid input. Enter row and column numbers between 0 and 2.")

        if check_winner(board) is not None:
            continue

        # AI move
        print("AI's turn!")
        best_move = find_best_move(board)
        if best_move:
            board[best_move[0]][best_move[1]] = 'X'

# Run the game
if __name__ == "__main__":
    tic_tac_toe()
```

OUTPUT:

Output

```
Tic Tac Toe! You are 'O', AI is 'X'.
 | |
-----
 | |
-----
 | |
-----
Your turn!
Enter your move (row and column, 0-2): 1 2
AI's turn!
 | |X
-----
 | |O
-----
 | |
-----
Your turn!
Enter your move (row and column, 0-2): 2 1
AI's turn!
X| |X
-----
 | |O
-----
 |O|
-----
Your turn!
Enter your move (row and column, 0-2): 1 1
AI's turn!
```

```
AI's turn!
 | |X
-----
 | |O
-----
 | |
-----
Your turn!
Enter your move (row and column, 0-2): 2 1
AI's turn!
X| |X
-----
 | |O
-----
 |O|
-----
Your turn!
Enter your move (row and column, 0-2): 1 1
AI's turn!
X|X|X
-----
 |O|O
-----
 |O|
-----
The winner is: X

=== Code Execution Successful ===
```