VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT

on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Pooja M (1BM22CS195)

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 **Pooja M (1BM22CS195) ,**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.

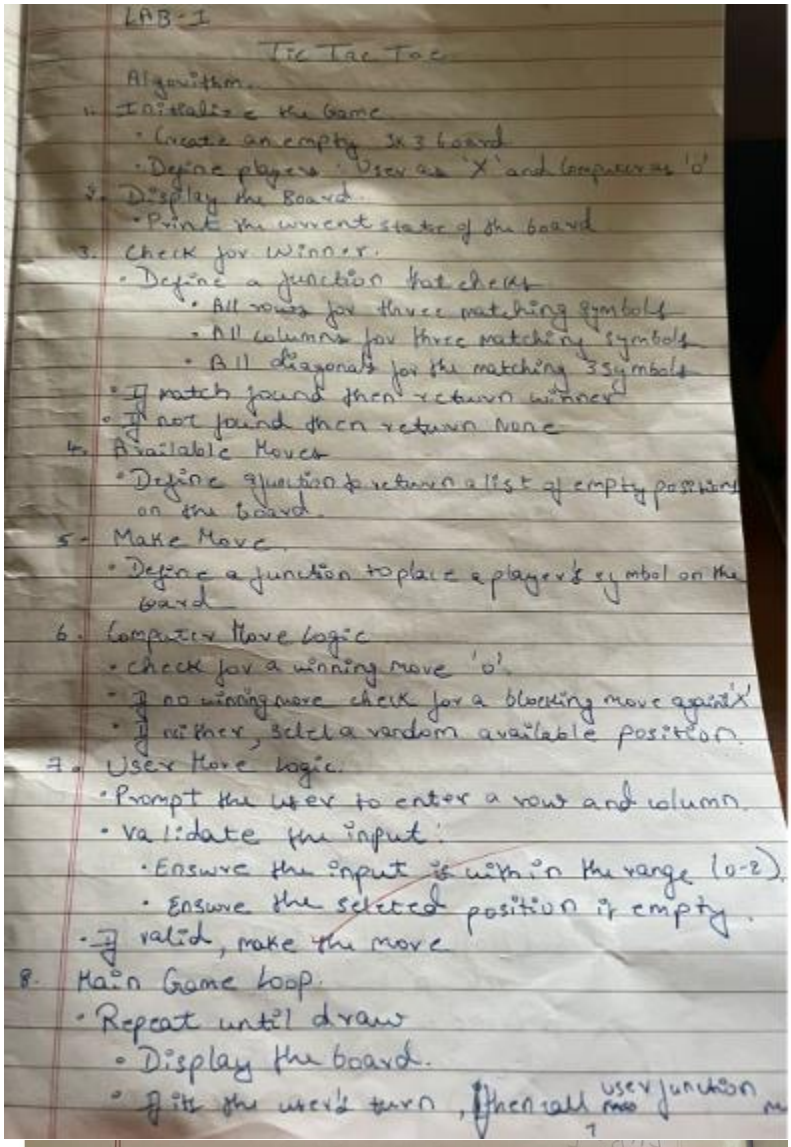
Sneha S Bagalkot Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

GITHUB LINK: **https://github.com/pooja195manjunath/AI**

Program 1-Tic Tac Toe

Algorithm

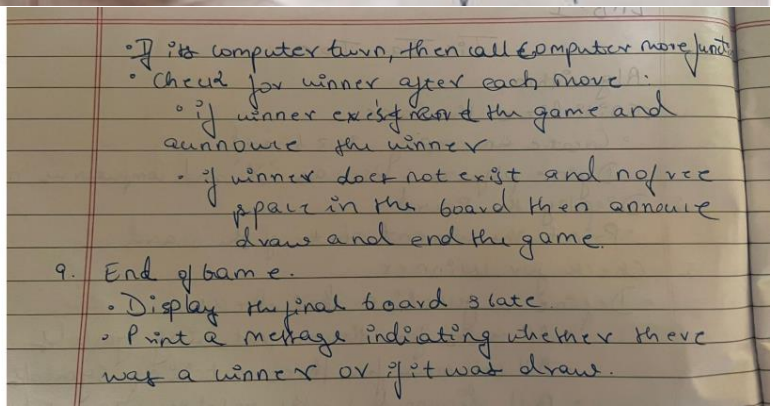


LAB-1

Tic Tac Toe

Algorithm.

1. Initialize the Game
   - Create an empty 3x3 board
   - Define players: User as 'X' and computer as 'O'
2. Display the Board
   - Print the current state of the board
3. Check for Winner.
   - Define a function that checks
     - All rows for three matching symbols
     - All columns for three matching symbols
     - All diagonals for the matching 3 symbols
   - If match found then return winner
   - If not found then return None
4. Available Moves
   - Define function to return a list of empty positions on the board.
5. Make Move.
   - Define a function to place a player's symbol on the board
6. Computer Move Logic
   - check for a winning move 'O'.
   - If no winning move check for a blocking move against 'X'
   - If neither, select a random available position.
7. User Move Logic.
   - Prompt the user to enter a row and column.
   - Validate the input:
     - Ensure the input is within the range (0-2).
     - Ensure the selected position is empty.
   - If valid, make the move
8. Main Game loop.
   - Repeat until draw
     - Display the board.
     - If its the user's turn, then call user function make move



     - If its computer turn, then call computer move function
     - Check for winner after each move:
       - if winner exist end the game and announce the winner
       - if winner does not exist and no free space in the board then announce draw and end the game.
9. End of game.
   - Display the final board state.
   - Print a message indicating whether there was a winner or if it was draw.

Code:

```
import random

def initialize_board():
    return [[' ' for _ in range(3)] for _ in range(3)]

def display_board(board):
    for row in board:
        print('|'.join(row))
        print('-' * 5)

def check_winner(board):
    for row in board:
        if row[0] == row[1] == row[2] != ' ':
            return row[0]
    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col] != ' ':
            return board[0][col]
    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]
    return None

def available_moves(board):
    return [(i, j) for i in range(3) for j in range(3) if board[i][j] == ' ']

def check_two_in_a_row(board, player):
    for row in range(3):
        if board[row].count(player) == 2 and board[row].count(' ') == 1:
            return row, board[row].index(' ')
    for col in range(3):
```

```
        if [board[row][col] for row in range(3)].count(player) == 2:
            empty_index = [row for row in range(3) if board[row][col] == ' ']
            if empty_index:
                return empty_index[0], col
    if [board[i][i] for i in range(3)].count(player) == 2:
        empty_index = [i for i in range(3) if board[i][i] == ' ']
        if empty_index:
            return empty_index[0], empty_index[0]
    if [board[i][2 - i] for i in range(3)].count(player) == 2:
        empty_index = [i for i in range(3) if board[i][2 - i] == ' ']
        if empty_index:
            return empty_index[0], 2 - empty_index[0]
    return None

def make_move(board, player, move):
    board[move[0]][move[1]] = player

def computer_move(board):
    move = check_two_in_a_row(board, 'O')
    if move:
        make_move(board, 'O', move)
        return
    move = check_two_in_a_row(board, 'X')
    if move:
        make_move(board, 'O', move)
        return
    moves = available_moves(board)
    if moves:
        move = random.choice(moves)
        make_move(board, 'O', move)

def user_move(board):
```

```
    while True:
        try:
            row = int(input("Enter row (0-2): "))
            col = int(input("Enter column (0-2): "))
            if board[row][col] == ' ':
                make_move(board, 'X', (row, col))
                return
            else:
                print("That spot is already taken. Try again.")
        except (ValueError, IndexError):
            print("Invalid input. Please enter numbers between 0 and 2.")

def play_game():
    board = initialize_board()
    players = ['X', 'O']
    current_player = 0
    for _ in range(9):
        display_board(board)
        if current_player == 0:
            user_move(board)
        else:
            computer_move(board)
        winner = check_winner(board)
        if winner:
            display_board(board)
            print(f"Player {winner} wins!")
            return
        current_player = 1 - current_player
    display_board(board)
    print("It's a draw!")

play_game()
```

print(“Pooja M”)

print(“1BM22CS195”)

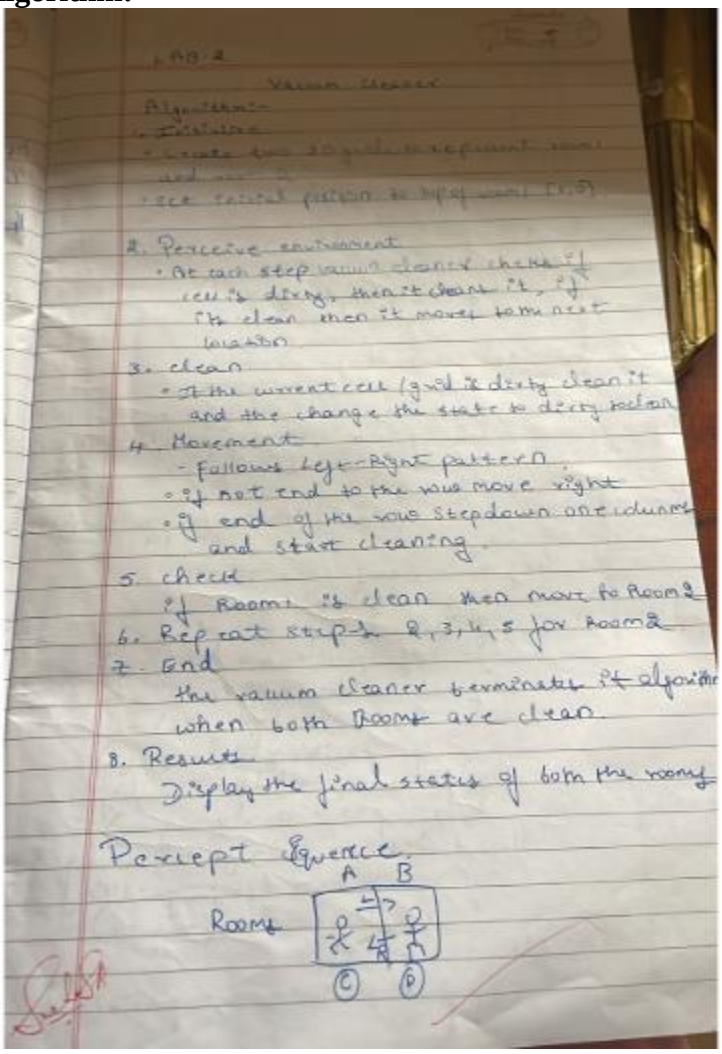
Output:

Output:-

```
>>>
=== RESTART: C:/Users/User/AppData/Local/Programs/Python/Python311/AI lab1.py =
 | |
-----
 | |
-----
 | |
-----
Enter row (0-2): 1
Enter column (0-2): 1
 | |
-----
 |X|
-----
 | |
-----
 | |
-----
O|X|
-----
 | |
-----
Enter row (0-2): 1
Enter column (0-2): 2
 | |
-----
O|X|X
-----
 | |
-----
 | |
-----
O|X|X
-----
O| |
-----
Enter row (0-2): 1
Enter column (0-2): 0
That spot is already taken. Try again.
Enter column (0-2): 1
That spot is already taken. Try again.
Enter row (0-2): 2
Enter column (0-2): 1
 | |
-----
O|X|X
-----
O|X|
-----
O| |
-----
O|X|X
-----
O|X|
-----
Player O wins!
```

Program 2 - Vacuum Cleaner

Algorithm:



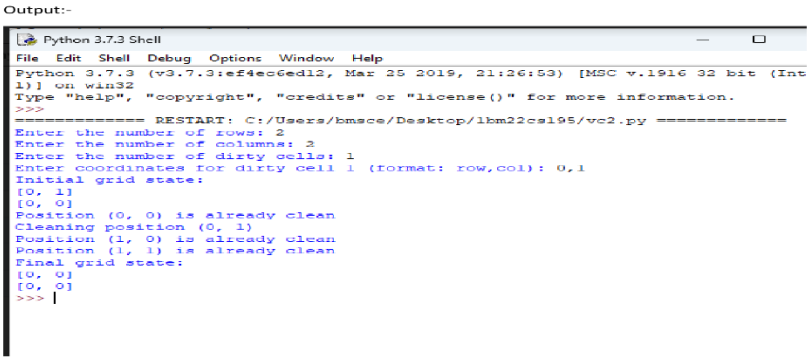
Code:

```
class VacuumCleaner:
    def __init__(self, grid):
        self.grid = grid
        self.position = (0, 0)

    def clean(self):
        x, y = self.position
        if self.grid[x][y] == 1:
            print(f"Cleaning position {self.position}")
            self.grid[x][y] = 0
        else:
            print(f"Position {self.position} is already clean")

    def move(self, direction):
        x, y = self.position
        if direction == 'up' and x > 0:
            self.position = (x - 1, y)
        elif direction == 'down' and x < len(self.grid) - 1:
            self.position = (x + 1, y)
        elif direction == 'left' and y > 0:
            self.position = (x, y - 1)
        elif direction == 'right' and y < len(self.grid[0]) - 1:
            self.position = (x, y + 1)
        else:
            print("Move not possible")

    def run(self):
        rows = len(self.grid)
        cols = len(self.grid[0])
        for i in range(rows):
            for j in range(cols):
```

```
                self.position = (i, j)
                self.clean()

        print("Final grid state:")
        for row in self.grid:
            print(row)

def get_dirty_coordinates(rows, cols, num_dirty_cells):
    dirty_cells = set()
    while len(dirty_cells) < num_dirty_cells:
        try:
            coords = input(f"Enter coordinates for dirty cell {len(dirty_cells) + 1} (format: row,col): ")
            x, y = map(int, coords.split(','))
            if 0 <= x < rows and 0 <= y < cols:
                dirty_cells.add((x, y))
            else:
                print("Coordinates are out of bounds. Try again.")
        except ValueError:
            print("Invalid input. Please enter coordinates in the format: row,col")
    return dirty_cells

rows = int(input("Enter the number of rows: "))
cols = int(input("Enter the number of columns: "))
num_dirty_cells = int(input("Enter the number of dirty cells: "))

if num_dirty_cells > rows * cols:
    print("Number of dirty cells exceeds total cells in the grid. Adjusting to maximum.")
    num_dirty_cells = rows * cols

initial_grid = [[0 for _ in range(cols)] for _ in range(rows)]
dirty_coordinates = get_dirty_coordinates(rows, cols, num_dirty_cells)
```

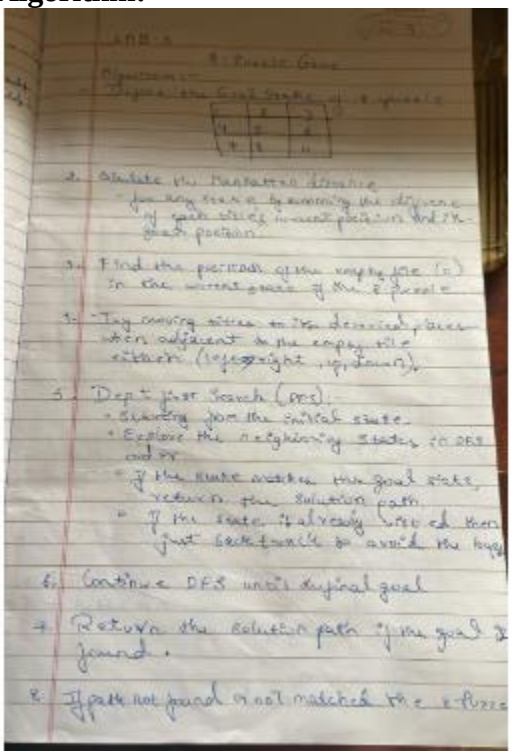
```
for x, y in dirty_coordinates:
    initial_grid[x][y] = 1

vacuum = VacuumCleaner(initial_grid)

print("Initial grid state:")
for row in initial_grid:
    print(row)

vacuum.run()
print(“Pooja M”)
print(“1BM22CS195”)
```

Output:



Program 3 - 8 Puzzle Game Using DFS

Algorithm:



Code:

```
class PuzzleState:
    def __init__(self, board, empty_pos, moves=[]):
        self.board = board
        self.empty_pos = empty_pos
        self.moves = moves
    def is_goal(self):
        return self.board == [1, 2, 3, 4, 5, 6, 7, 8, 0]
    def get_possible_moves(self):
        x, y = self.empty_pos
        moves = []
        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            nx, ny = x + dx, y + dy
            if 0 <= nx < 3 and 0 <= ny < 3:
                new_board = self.board[:]
                 new_board[x * 3 + y], new_board[nx * 3 + ny] = new_board[nx * 3 + ny], new_board[x * 3
+ y]
                moves.append((new_board, (nx, ny)))
        return moves
def dfs(initial_state):
    stack, visited = [initial_state], set()
    while stack:
        current_state = stack.pop()
        if current_state.is_goal():
            return current_state.moves
        visited.add(tuple(current_state.board))
        for new_board, new_empty_pos in current_state.get_possible_moves():
            new_state = PuzzleState(new_board, new_empty_pos, current_state.moves + [new_board])
            if tuple(new_board) not in visited:
                stack.append(new_state)
    return None
def print_matrix(board):
```

```
    for i in range(0, 9, 3):
        print(board[i:i + 3])
    print()
def main():
    initial_board = [1, 2, 3, 4, 0, 5, 7, 8, 6]
    empty_pos = initial_board.index(0)
    initial_state = PuzzleState(initial_board, (empty_pos // 3, empty_pos % 3))
    print("Initial state:")
    print_matrix(initial_board)
    solution = dfs(initial_state)
    if solution:
        print("Solution found:")
        for step in solution:
            print_matrix(step)
    else:
        print("No solution found.")
if __name___== "__main__":
    main()
print(“Pooja M”)
print(“1BM22CS195”)
```

Output:

main()

Output:-

```
Type "help", "copyright", "credits" or "license()" for mor
>>>
===== RESTART: C:/Users/User/AppData/Local/Programs/Python
Initial state:
[1, 2, 3]
[4, 0, 5]
[7, 8, 6]

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

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

>>>
```

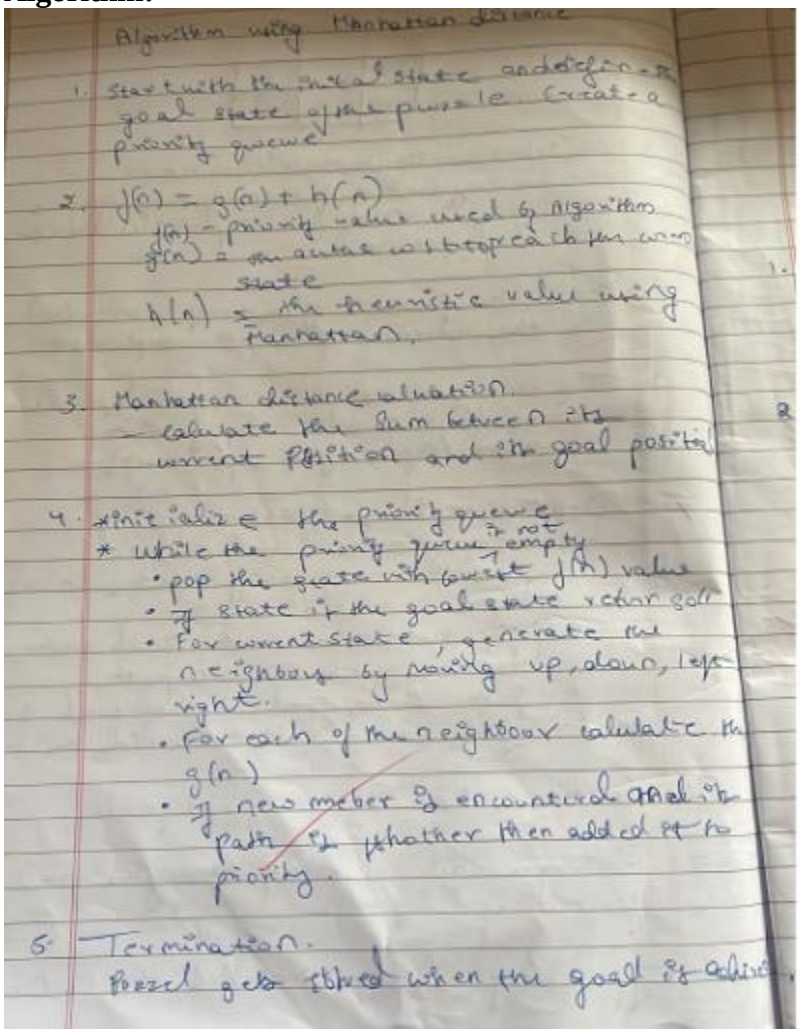
Code:-

Using Manhattan Distance

```
class SlidingPuzzleSolver:
    def __init__(self, initial_state):
        self.initial_state = initial_state
        self.goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]


    def manhattan_distance(self, state):
        distance = 0
        for i in range(3):
            for j in range(3):
                if state[i][j] != 0:
                    goal_i = (state[i][j] - 1) // 3
                    goal_j = (state[i][j] - 1) % 3
                    distance += abs(i - goal_i) + abs(j - goal_j)
        return distance
```

8 Puzzle Game Using Manhattan Distance:

Algorithm:


Algorithm using Manhattan distance

1. Start with the initial state and define the goal state of the puzzle. Create a priority queue

2. f(n) = g(n) + h(n)
f(n) - priority value used by algorithm
g(n) = the actual cost to reach the current state
h(n) = the heuristic value using Manhattan.

3. Manhattan distance calculation.
- calculate the sum between its current position and its goal position

4. Initialize the priority queue
* While the priority queue is not empty
- pop the state with lowest f(n) value
- If state is the goal state return solution
- For current state, generate the neighbours by moving up, down, left, right.
- For each of the neighbour calculate the g(n)
- If new meber is encountered add its path to whether then added it to priority.

5. Termination.
Puzzel gets solved when the goal is achieved.

Code:

```
def manhattan_distance(state, goal_state):
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                goal_i = (state[i][j] - 1) // 3
                goal_j = (state[i][j] - 1) % 3
                distance += abs(i - goal_i) + abs(j - goal_j)
    return distance

def get_neighbors(state):
    i, j = next((i, j) for i in range(3) for j in range(3) if state[i][j] == 0)
    moves = [(i-1, j), (i+1, j), (i, j-1), (i, j+1)]
    return [swap(state, i, j, x, y) for x, y in moves if 0 <= x < 3 and 0 <= y < 3]

def swap(state, i1, j1, i2, j2):
    new_state = [row[:] for row in state]
    new_state[i1][j1], new_state[i2][j2] = new_state[i2][j2], new_state[i1][j1]
    return new_state

def dfs_with_manhattan(state, goal, visited=set()):
    if state == goal:
        return [state]
    visited.add(str(state))
    neighbors = sorted(get_neighbors(state), key=lambda x: manhattan_distance(x, goal))
    for neighbor in neighbors:
        if str(neighbor) not in visited:
            path = dfs_with_manhattan(neighbor, goal, visited)
            if path:
                return [state] + path
    return None
```

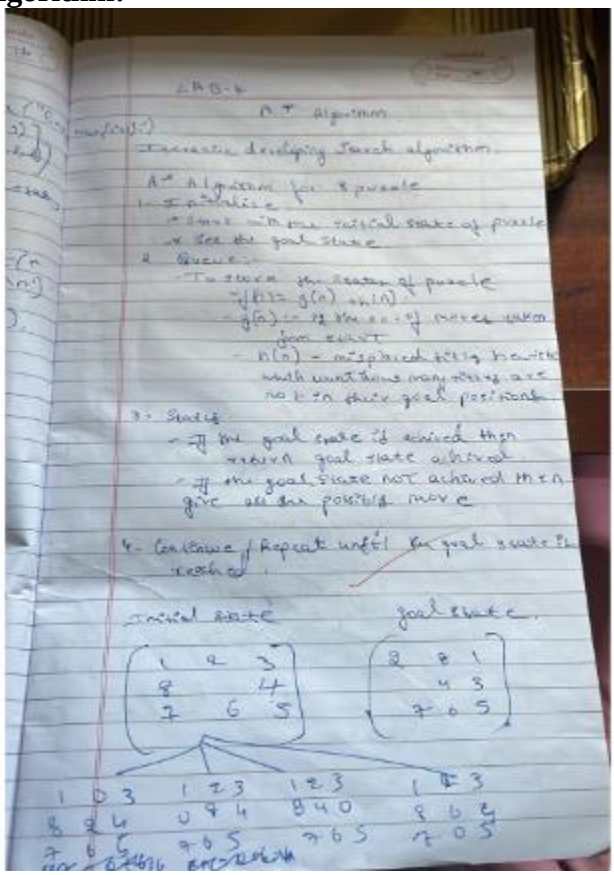
```
# Take user input for initial state
initial_state = [[int(x) for x in input(f"Enter row {i+1}: ").split()] for i in range(3)]
goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

solution = dfs_with_manhattan(initial_state, goal_state)
if solution:
    print("Solution found:")
    for state in solution:
        print(*state, sep='\n', end='\n\n')
else:
    print("No solution found.")
print(“Pooja M”)
print(“1BM22CS195”)
```

Output:

```
if solution:
    print("Solution found:")
    for state in solution:
        print(*state, sep='\n', end='\n\n')
else:
    print("No solution found.")
Output:-
>>
===== RESTART: C:/Users/User/AppData/Local/Programs/
Enter row 1: 1 0 3
Enter row 2: 4 2 6
Enter row 3: 7 5 8
Solution found:
[1, 0, 3]
[4, 2, 6]
[7, 5, 8]

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

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

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

Program 4 - 8 Puzzle Game Using A*

Algorithm:



Code:

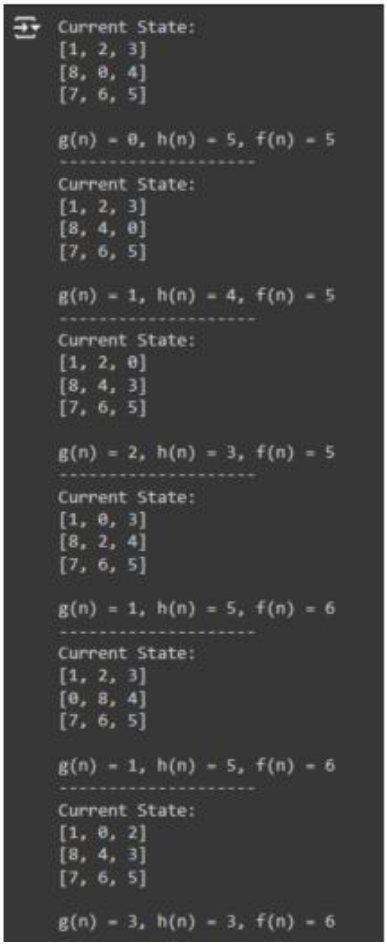
```
import heapq

# Goal state where blank (0) is the first tile
goal_state = [
    [0, 1, 2],
    [3, 4, 5],
    [6, 7, 8]
]
```

```
# Helper functions
def flatten(puzzle):
    return [item for row in puzzle for item in row]

def find_blank(puzzle):
    for i in range(3):
        for j in range(3):
            if puzzle[i][j] == 0:
                return i, j

def misplaced_tiles(puzzle):
    flat_puzzle = flatten(puzzle)
    flat_goal = flatten(goal_state)
    return sum([1 for i in range(9) if flat_puzzle[i] != flat_goal[i] and flat_puzzle[i] != 0])

def generate_neighbors(puzzle):
    x, y = find_blank(puzzle)
    neighbors = []
    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_puzzle = [row[:] for row in puzzle]
            new_puzzle[x][y], new_puzzle[nx][ny] = new_puzzle[nx][ny], new_puzzle[x][y]
            neighbors.append(new_puzzle)
    return neighbors

def is_goal(puzzle):
    return puzzle == goal_state

def print_puzzle(puzzle):
```

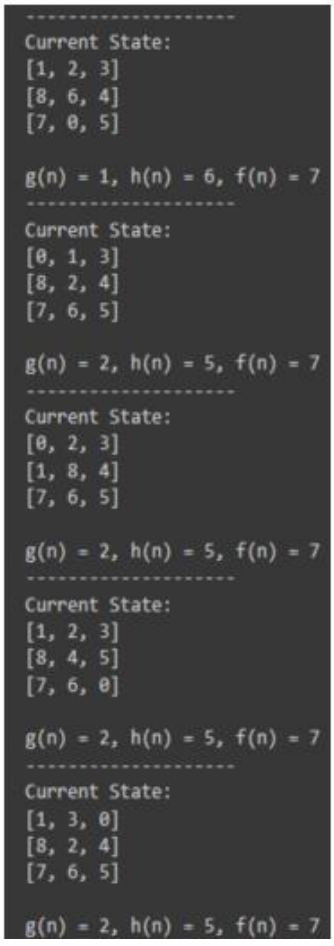
```
    for row in puzzle:
        print(row)
    print()

def a_star_misplaced_tiles(initial_state):
    # Priority queue (min-heap) and visited states
    frontier = []
    heapq.heappush(frontier, (misplaced_tiles(initial_state), 0, initial_state, []))
    visited = set()

    while frontier:
        f, g, current_state, path = heapq.heappop(frontier)
        # Print the current state
        print("Current State:")
        print_puzzle(current_state)
        h = misplaced_tiles(current_state)
        print(f"g(n) = {g}, h(n) = {h}, f(n) = {g + h}")
        print("-" * 20)

        if  is_goal(current_state):
            print("Goal reached!")
            return path

        visited.add(tuple(flatten(current_state)))
        for neighbor in generate_neighbors(current_state):
            if tuple(flatten(neighbor)) not in visited:
                h = misplaced_tiles(neighbor)
                heapq.heappush(frontier, (g + 1 + h, g + 1, neighbor, path + [neighbor]))

    return None  # No solution found

# Initial puzzle state
```

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

solution = a_star_misplaced_tiles(initial_state)
if solution:
    print("Solution found!")
else:
    print("No solution found.")
print(“Pooja M”)
print(“1BM22CS195”)
```

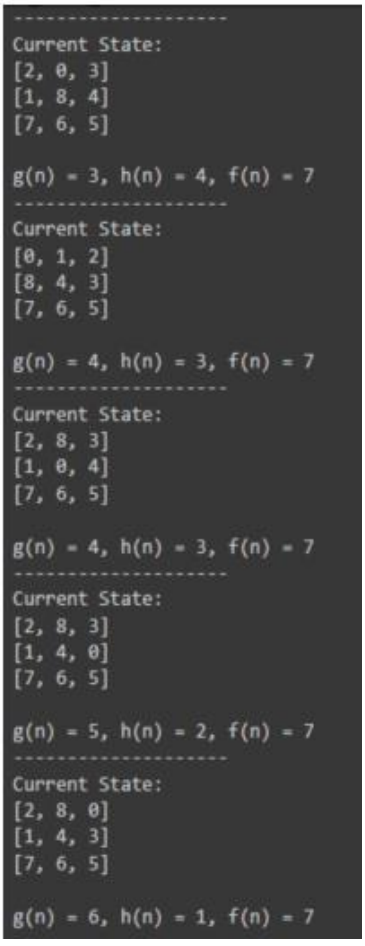
Output:



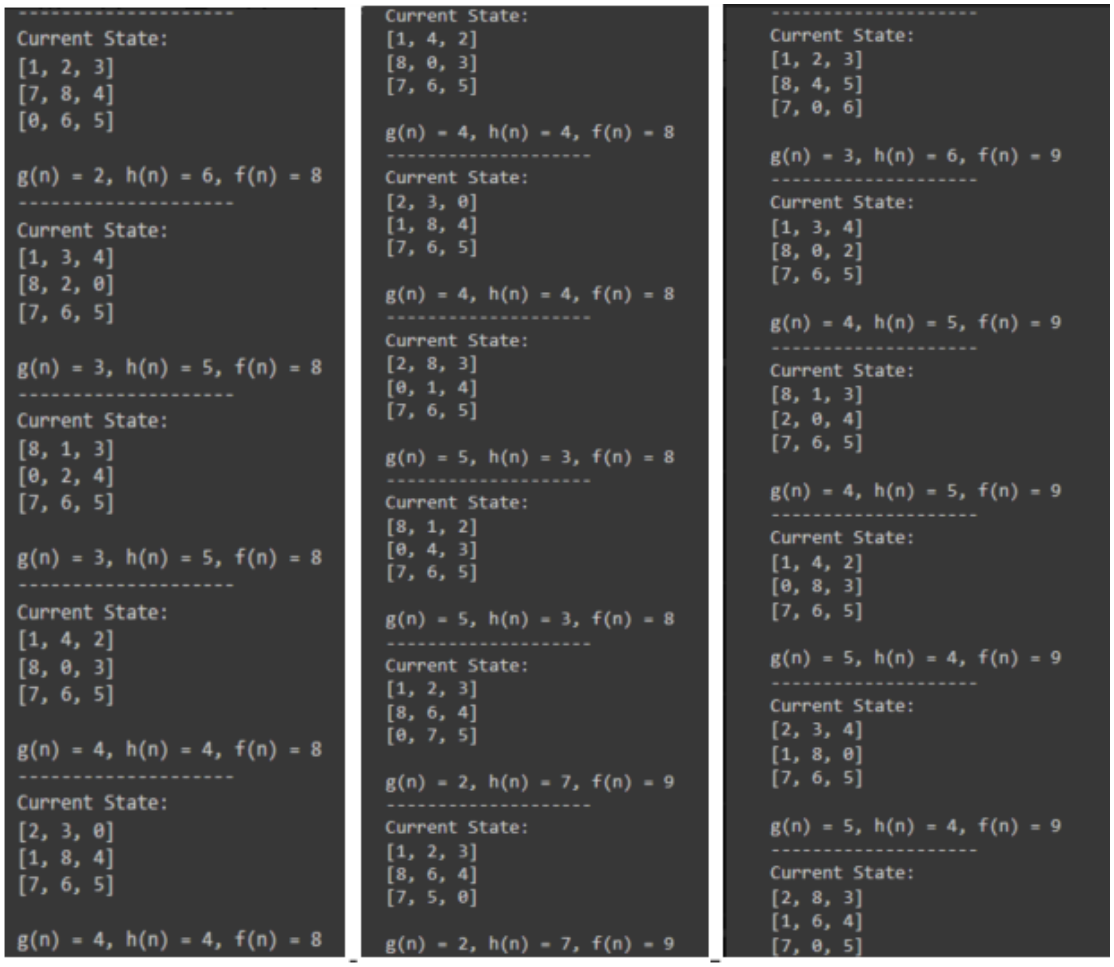
```
Current State:
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]

g(n) = 0, h(n) = 5, f(n) = 5
--------------------
Current State:
[1, 2, 3]
[8, 4, 0]
[7, 6, 5]

g(n) = 1, h(n) = 4, f(n) = 5
--------------------
Current State:
[1, 2, 0]
[8, 4, 3]
[7, 6, 5]

g(n) = 2, h(n) = 3, f(n) = 5
--------------------
Current State:
[1, 0, 3]
[8, 2, 4]
[7, 6, 5]

g(n) = 1, h(n) = 5, f(n) = 6
--------------------
Current State:
[1, 2, 3]
[0, 8, 4]
[7, 6, 5]

g(n) = 1, h(n) = 5, f(n) = 6
--------------------
Current State:
[1, 0, 2]
[8, 4, 3]
[7, 6, 5]

g(n) = 3, h(n) = 3, f(n) = 6
```



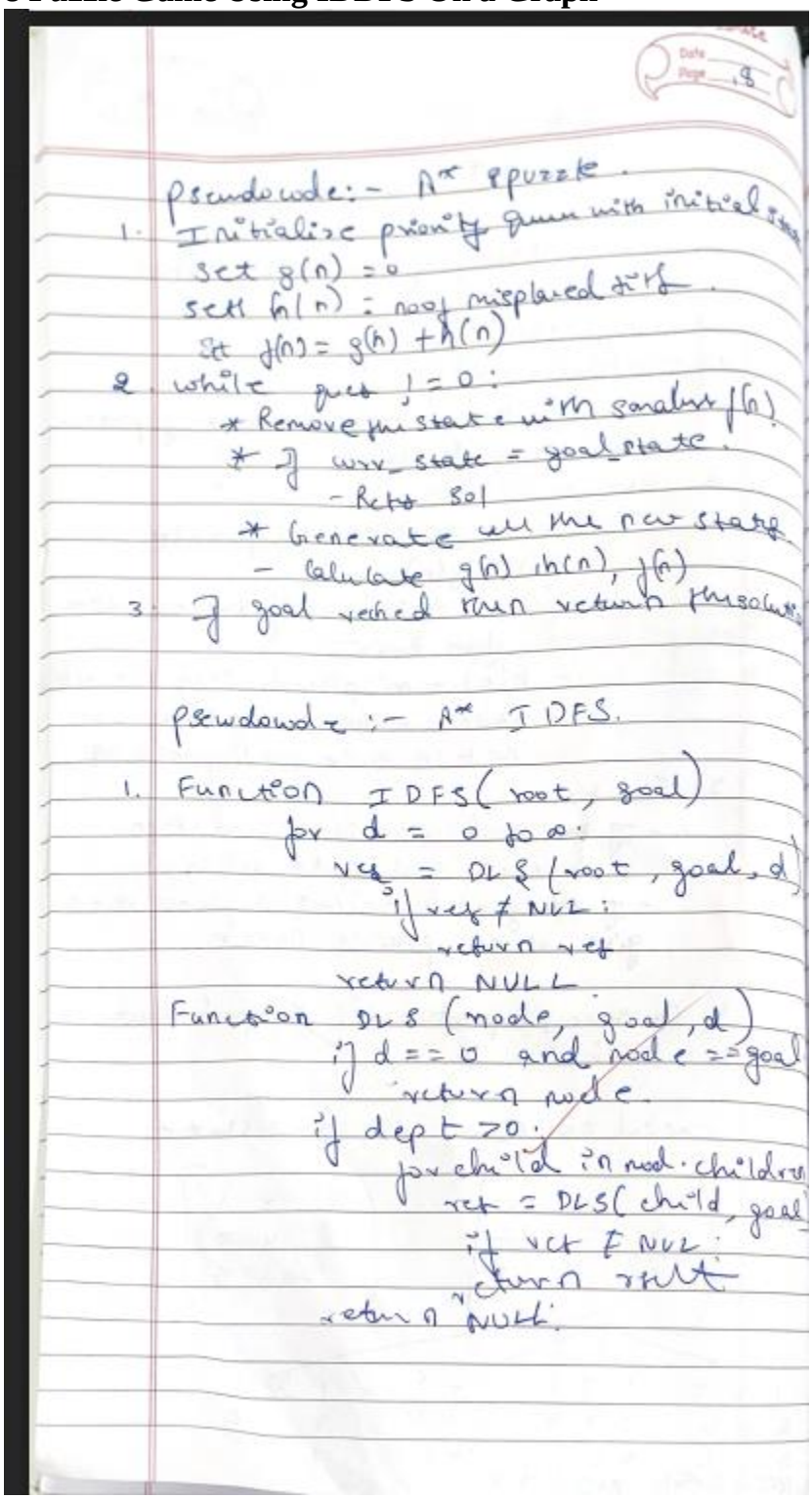
```
--------------------
Current State:
[1, 2, 3]
[8, 6, 4]
[7, 0, 5]

g(n) = 1, h(n) = 6, f(n) = 7
--------------------
Current State:
[0, 1, 3]
[8, 2, 4]
[7, 6, 5]

g(n) = 2, h(n) = 5, f(n) = 7
--------------------
Current State:
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]

g(n) = 2, h(n) = 5, f(n) = 7
--------------------
Current State:
[1, 2, 3]
[8, 4, 5]
[7, 6, 0]

g(n) = 2, h(n) = 5, f(n) = 7
--------------------
Current State:
[1, 3, 0]
[8, 2, 4]
[7, 6, 5]

g(n) = 2, h(n) = 5, f(n) = 7
```



```
--------------------
Current State:
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]

g(n) = 3, h(n) = 4, f(n) = 7
--------------------
Current State:
[0, 1, 2]
[8, 4, 3]
[7, 6, 5]

g(n) = 4, h(n) = 3, f(n) = 7
--------------------
Current State:
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]

g(n) = 4, h(n) = 3, f(n) = 7
--------------------
Current State:
[2, 8, 3]
[1, 4, 0]
[7, 6, 5]

g(n) = 5, h(n) = 2, f(n) = 7
--------------------
Current State:
[2, 8, 0]
[1, 4, 3]
[7, 6, 5]

g(n) = 6, h(n) = 1, f(n) = 7
```


Current State:
[1, 2, 3]
[7, 8, 4]
[0, 6, 5]
g(n) = 2, h(n) = 6, f(n) = 8
Current State:
[1, 3, 4]
[8, 2, 0]
[7, 6, 5]
g(n) = 3, h(n) = 5, f(n) = 8
Current State:
[8, 1, 3]
[0, 2, 4]
[7, 6, 5]
g(n) = 3, h(n) = 5, f(n) = 8
Current State:
[1, 4, 2]
[8, 0, 3]
[7, 6, 5]
g(n) = 4, h(n) = 4, f(n) = 8
Current State:
[2, 3, 0]
[1, 8, 4]
[7, 6, 5]
g(n) = 4, h(n) = 4, f(n) = 8
Current State:
[1, 4, 2]
[8, 0, 3]
[7, 6, 5]
g(n) = 4, h(n) = 4, f(n) = 8
Current State:
[2, 3, 0]
[1, 8, 4]
[7, 6, 5]
g(n) = 5, h(n) = 3, f(n) = 8
Current State:
[8, 1, 2]
[0, 4, 3]
[7, 6, 5]
g(n) = 5, h(n) = 3, f(n) = 8
Current State:
[1, 2, 3]
[8, 6, 4]
[0, 7, 5]
g(n) = 2, h(n) = 7, f(n) = 9
Current State:
[1, 2, 3]
[8, 6, 4]
[7, 5, 0]
g(n) = 2, h(n) = 7, f(n) = 9
Current State:
[1, 2, 3]
[8, 4, 5]
[7, 0, 6]
g(n) = 3, h(n) = 6, f(n) = 9
Current State:
[1, 3, 4]
[8, 0, 2]
[7, 6, 5]
g(n) = 4, h(n) = 5, f(n) = 9
Current State:
[8, 1, 3]
[2, 0, 4]
[7, 6, 5]
g(n) = 4, h(n) = 5, f(n) = 9
Current State:
[1, 4, 2]
[0, 8, 3]
[7, 6, 5]
g(n) = 5, h(n) = 4, f(n) = 9
Current State:
[2, 3, 4]
[1, 8, 0]
[7, 6, 5]
g(n) = 5, h(n) = 4, f(n) = 9
Current State:
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]
g(n) = 5, h(n) = 4, f(n) = 9
Current State:
[8, 1, 3]
[2, 4, 0]
[7, 6, 5]
g(n) = 5, h(n) = 4, f(n) = 9
Current State:
[2, 8, 3]
[1, 4, 5]
[7, 6, 0]
g(n) = 6, h(n) = 3, f(n) = 9
Current State:
[8, 1, 0]
[2, 4, 3]
[7, 6, 5]
g(n) = 6, h(n) = 3, f(n) = 9
Current State:
[2, 0, 8]
[1, 4, 3]
[7, 6, 5]
g(n) = 7, h(n) = 2, f(n) = 9
Current State:
[8, 0, 1]
[2, 4, 3]
[7, 6, 5]
g(n) = 7, h(n) = 2, f(n) = 9
Current State:
[0, 8, 1]
[2, 4, 3]
[7, 6, 5]
g(n) = 8, h(n) = 1, f(n) = 9
[1, 4, 3]
[7, 6, 5]
g(n) = 7, h(n) = 2, f(n) = 9
Current State:
[8, 0, 1]
[2, 4, 3]
[7, 6, 5]
g(n) = 7, h(n) = 2, f(n) = 9
Current State:
[0, 8, 1]
[2, 4, 3]
[7, 6, 5]
g(n) = 8, h(n) = 1, f(n) = 9
Current State:
[2, 8, 1]
[0, 4, 3]
[7, 6, 5]
g(n) = 9, h(n) = 0, f(n) = 9
Goal reached!
Solution found!

8 Puzzle Game Using IDDFS On a Graph

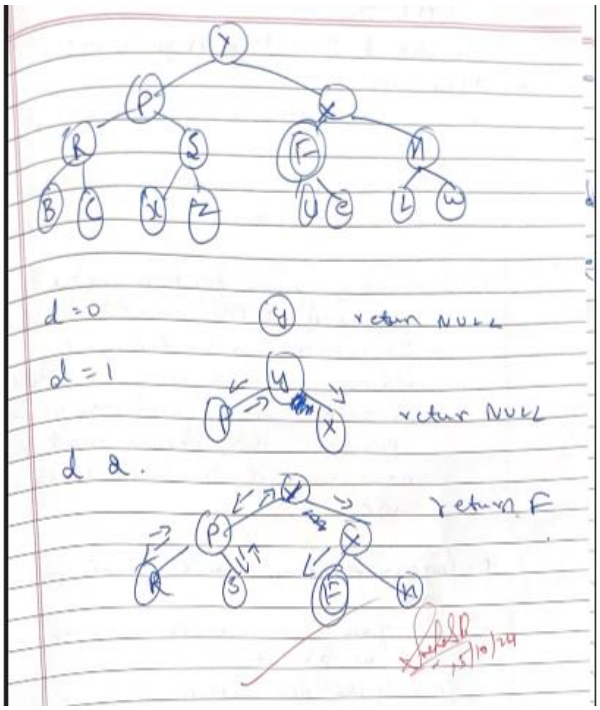


Pseudocode:- A* 8puzzle.

1. Initialize priority queue with initial state
   set g(n) = 0
   set h(n) = no of misplaced tiles
   set f(n) = g(n) + h(n)
2. while queue != 0:
   * Remove min state with smallest f(n)
   * if curr_state = goal_state.
     - Return Sol
   * Generate all the new states
     - Calculate g(n) h(n), f(n)
3. if goal reached then return the solution

Pseudocode :- A* IDFS.

1. Function IDFS(root, goal)
   for d = 0 to ∞:
   res = DLS(root, goal, d)
   if res ≠ NULL:
   return res
   return NULL
   Function DLS (node, goal, d)
   if d == 0 and node == goal
   return node.
   if depth > 0:
   for child in node.children
   res = DLS(child, goal
   if res ≠ NULL:
   return result
   return NULL.



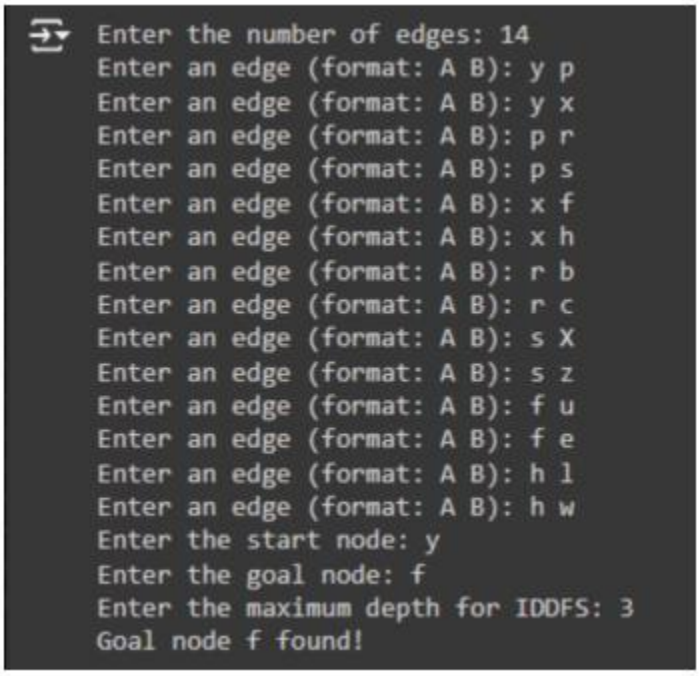
Code:

```
class Graph:
    def __init__(self):
        self.adjacency_list = {}

    def add_edge(self, u, v):
        if u not in self.adjacency_list:
            self.adjacency_list[u] = []
        self.adjacency_list[u].append(v)

    def depth_limited_dfs(self, node, goal, limit, visited):
        if limit < 0:
            return False
        if node == goal:
```

```
            return True
        visited.add(node)
        for neighbor in self.adjacency_list.get(node, []):
            if neighbor not in visited:
                if self.depth_limited_dfs(neighbor, goal, limit - 1, visited):
                    return True
        visited.remove(node)  # Allow revisiting for the next iteration
        return False

    def iddfs(self, start, goal, max_depth):
        for depth in range(max_depth + 1):
            visited = set()
            if self.depth_limited_dfs(start, goal, depth, visited):
                return True
        return False

def main():
    graph = Graph()
    # Input number of edges
    num_edges = int(input("Enter the number of edges: "))
    # Input edges
    for _ in range(num_edges):
        edge = input("Enter an edge (format: A B): ").split()
        graph.add_edge(edge[0], edge[1])

    start_node = input("Enter the start node: ")
    goal_node = input("Enter the goal node: ")
    max_depth = int(input("Enter the maximum depth for IDDFS: "))

    if graph.iddfs(start_node, goal_node, max_depth):
        print(f"Goal node {goal_node} found!")
    else:
```

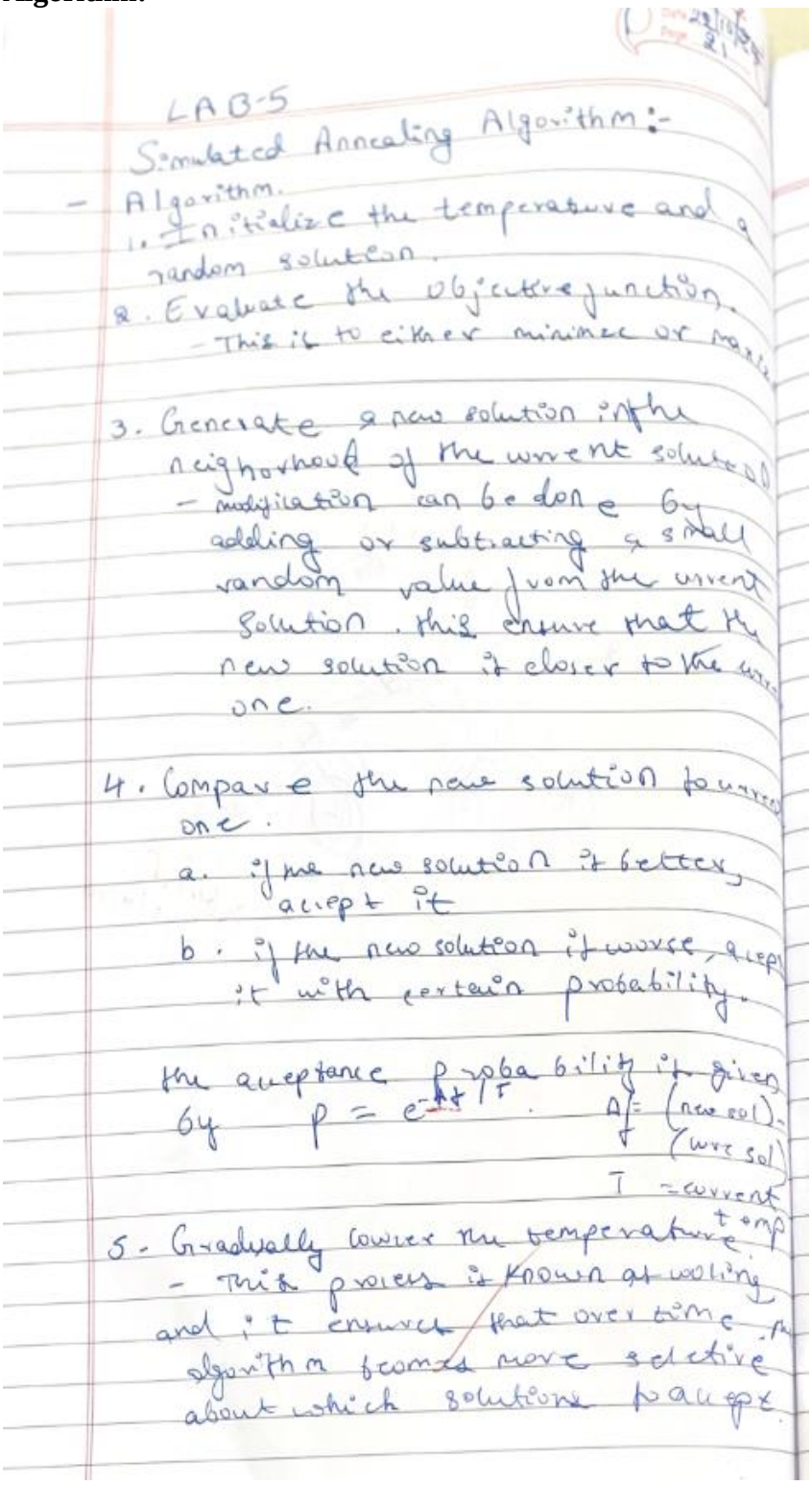
```
        print(f"Goal node {goal_node} not found within depth {max_depth}.")

if __name___== "__main__":
    main()
print("Pooja M")
print("1BM22CS195")
```

Output:



Program 5 - Simulated Annealing Algorithm

Algorithm:



LAB-5

Simulated Annealing Algorithm:-

- Algorithm.

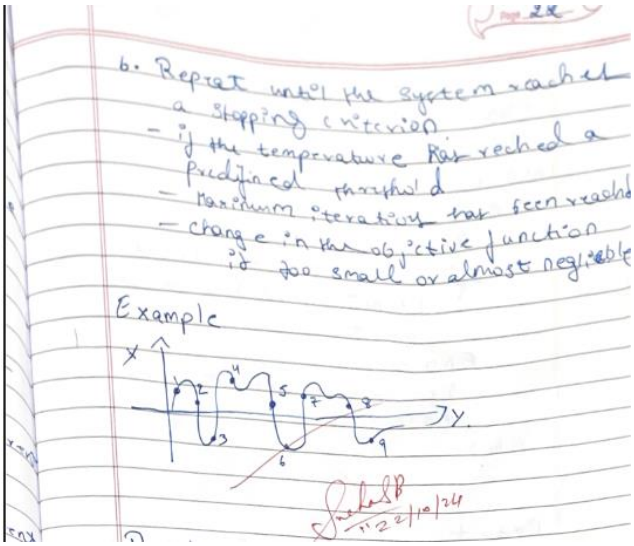
1. Initialize the temperature and a random solution.
2. Evaluate the objective function.
   - This is to either minimize or maximize
3. Generate a new solution in the neighborhood of the current solution.
   - modification can be done by adding or subtracting a small random value from the current solution. this ensure that the new solution is closer to the current one.
4. Compare the new solution to current one.
   a. if the new solution is better, accept it
   b. if the new solution is worse, accept it with certain probability.

the acceptance probability is given by $p = e^{-\Delta t / T}$.

Δ = (new sol) - (curr sol)

T = current temp

5. Gradually lower the temperature.
   - This process is known as cooling and it ensures that over time the algorithm becomes more selective about which solutions to accept.


6. Repeat until the system reached a stopping criterion.
- if the temperature has reached a predifined threshold
- maximum iteration has been reached
- change in the objective function is too small or almost negligible
Example
X
Y
1
2
3
4
5
6
7
8
9

Code:

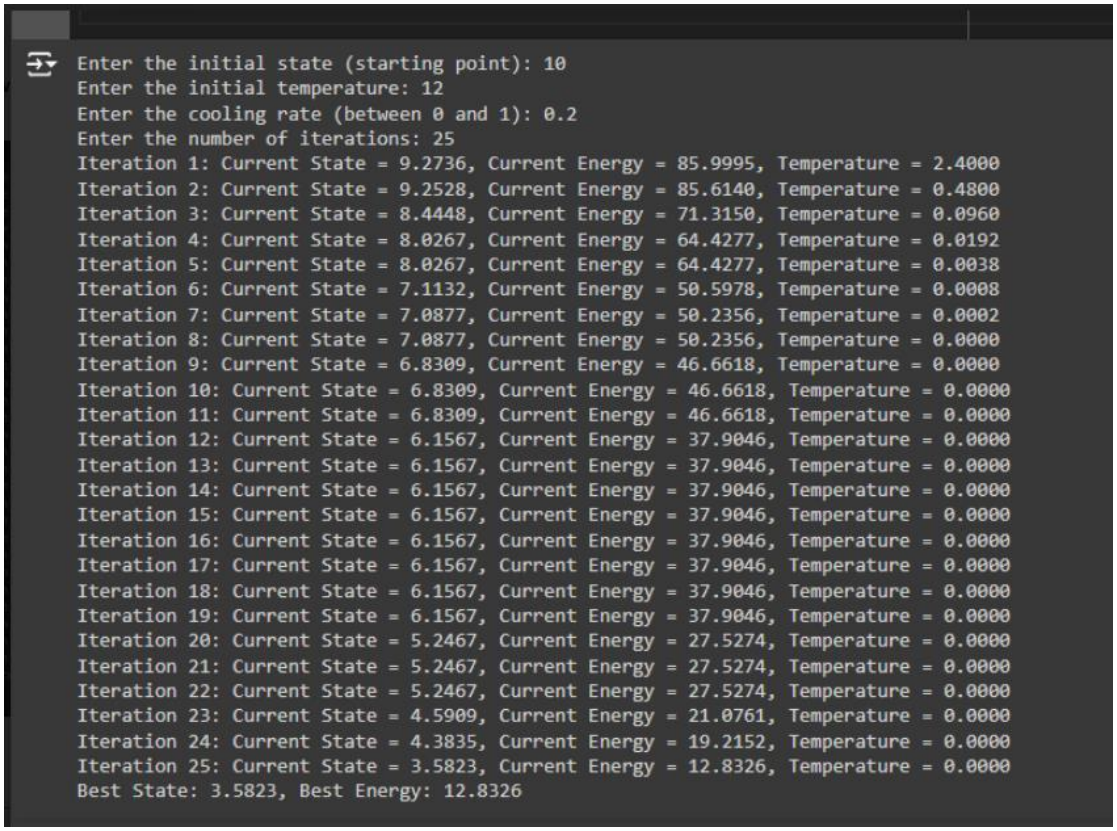
```
import numpy as np
import math
import random

def objective_function(x):
    """Objective function to minimize: f(x) = x^2"""
    return x ** 2

def simulated_annealing(initial_state, initial_temp, cooling_rate, max_iterations):
    """Simulated Annealing algorithm to find the minimum of the objective function."""
    current_state = initial_state
    current_energy = objective_function(current_state)
    best_state = current_state
    best_energy = current_energy
    temp = initial_temp

    for iteration in range(max_iterations):
        # Generate a new candidate state by perturbing the current state
        candidate_state = current_state + random.uniform(-1, 1)
        candidate_energy = objective_function(candidate_state)

        # Calculate energy difference
        energy_diff = candidate_energy - current_energy

        # If the candidate state is better, or accepted with a certain probability
        if energy_diff < 0 or random.uniform(0, 1) < math.exp(-energy_diff / temp):
            current_state = candidate_state
            current_energy = candidate_energy

        # Update best state found
        if current_energy < best_energy:
```

```
            best_state = current_state
            best_energy = current_energy

        # Cool down the temperature
        temp *= cooling_rate

        # Print the current state and temperature for debugging
            print(f"Iteration {iteration + 1}: Current State = {current_state:.4f}, Current Energy = {current_energy:.4f}, Temperature = {temp:.4f}")

    return best_state, best_energy

# Get user input for parameters
try:
    initial_state = float(input("Enter the initial state (starting point): "))
    initial_temp = float(input("Enter the initial temperature: "))
    cooling_rate = float(input("Enter the cooling rate (between 0 and 1): "))
    max_iterations = int(input("Enter the number of iterations: "))

    # Validate cooling rate
    if cooling_rate <= 0 or cooling_rate >= 1:
        raise ValueError("Cooling rate must be between 0 and 1.")

    # Execute the simulated annealing algorithm
        best_state, best_energy = simulated_annealing(initial_state, initial_temp, cooling_rate, max_iterations)

    # Output the best state and energy found
    print(f"Best State: {best_state:.4f}, Best Energy: {best_energy:.4f}")

except ValueError as e:
    print(f"Invalid input: {e}")
```

print("Pooja M")

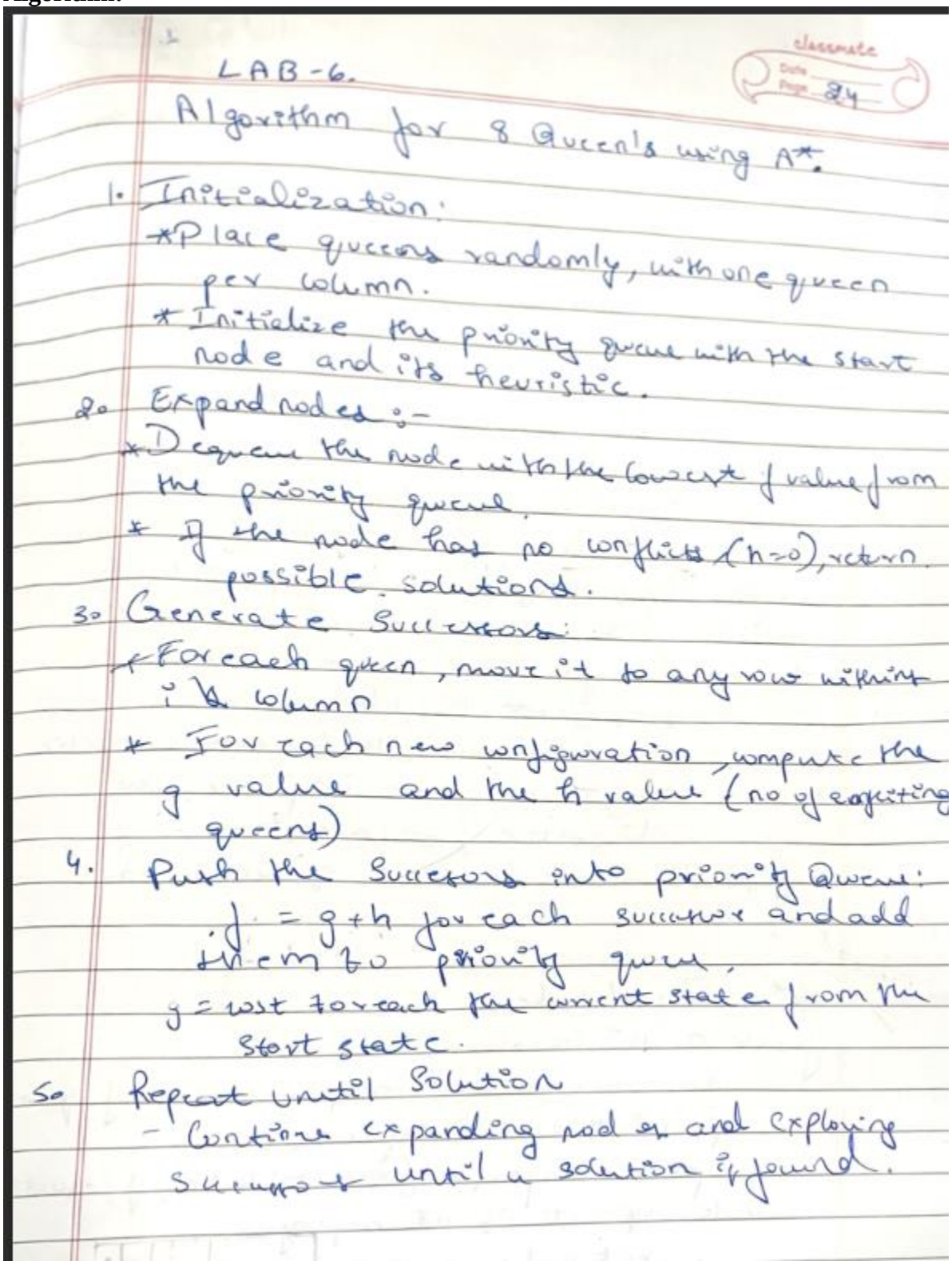
print("1BM22CS195")

Output:



Program 6 - Implementing A* on 8 Queens

Algorithm:



LAB-6.

Algorithm for 8 Queen's using A*.

1. Initialization:
   * Place queens randomly, with one queen per column.
   * Initialize the priority queue with the start node and its heuristic.
2. Expand nodes:-
   * Dequeue the node with the lowest f value from the priority queue.
   * If the node has no conflicts (h=0), return possible solutions.
3. Generate Successors:
   * For each queen, move it to any row within its column
   * For each new configuration, compute the g value and the h value (no of conflicting queens)
4. Push the Successors into priority Queue:
   f = g+h for each successor and add them to priority queue.
   g = cost to reach the current state from the start state.
5. Repeat until Solution
   - Continue expanding nodes and exploring successors until a solution is found.

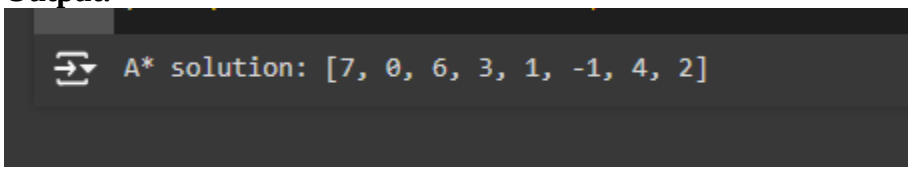
Code:

```
import numpy as np
import heapq

class Node:
    def __init__(self, state, g, h):
        self.state = state  # Current state of the board
        self.g = g          # Cost to reach this state
        self.h = h          # Heuristic cost to reach goal
        self.f = g + h      # Total cost

    def __lt__(self, other):
```

```
        return self.f < other.f

def heuristic(state):
    # Count pairs of queens that can attack each other
    attacks = 0
    for i in range(len(state)):
        for j in range(i + 1, len(state)):
            if state[i] == state[j] or abs(state[i] - state[j]) == j - i:
                attacks += 1
    return attacks

def a_star_8_queens():
    initial_state = [-1] * 8  # -1 means no queen placed
    open_list = []
    closed_set = set()
    initial_h = heuristic(initial_state)
    heapq.heappush(open_list, Node(initial_state, 0, initial_h))

    while open_list:
        current_node = heapq.heappop(open_list)
        current_state = current_node.state
        closed_set.add(tuple(current_state))

        # Check if we reached the goal
        if current_node.h == 0:
            return current_state

        for col in range(8):
            if current_state[col] == -1:  # Only place a queen if none is present in this column
                for row in range(8):
                    new_state = current_state.copy()
                    new_state[col] = row
```

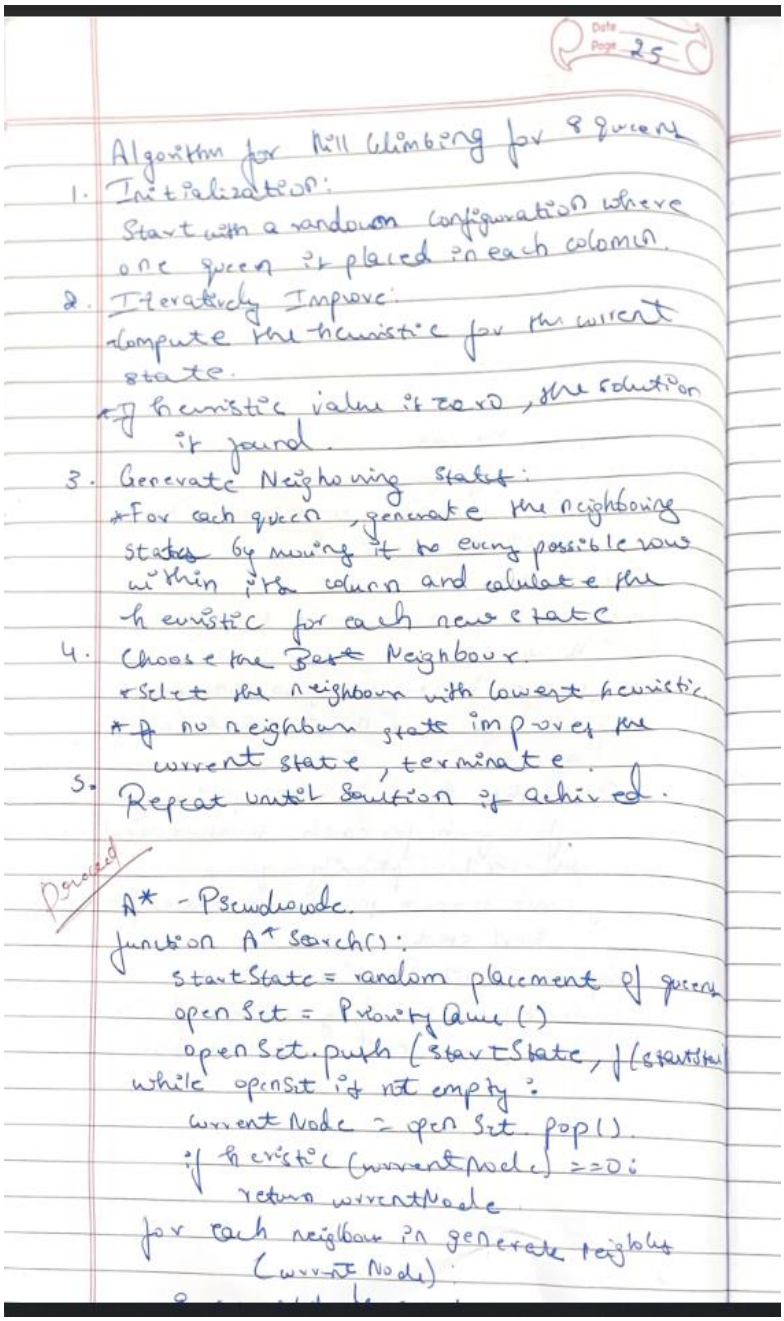
```
            if tuple(new_state) not in closed_set:
                g_cost = current_node.g + 1
                h_cost = heuristic(new_state)
                heapq.heappush(open_list, Node(new_state, g_cost, h_cost))
    return None

solution = a_star_8_queens()
print("A* solution:", solution)
print("Pooja M")
print("1BM22CS195")
```

Output:



Implementing Hill Climbing on 8 Queens

Algorithm:



Algorithm for Hill Climbing for 8 queens

1. Initialization:
   Start with a random configuration where one queen is placed in each column.
2. Iteratively Improve:
   Compute the heuristic for the current state.
   * If heuristic value is zero, the solution is found.
3. Generate Neighboring states:
   * For each queen, generate the neighboring states by moving it to every possible row within its column and calculate the heuristic for each new state.
4. Choose the Best Neighbour.
   * Select the neighbour with lowest heuristic.
   * If no neighbour state improves the current state, terminate.
5. Repeat until Solution is achieved.

Corrected

A* - Pseudocode.

```
function A* Search():
    startState = random placement of queens
    openSet = PriorityQueue()
    openSet.push(startState, f(startState))
    while openSet is not empty:
        currentNode = openSet.pop()
        if heuristic(currentNode) == 0:
            return currentNode
        for each neighbour in generateNeighbours
                (currentNode):
```

Code:

```
import random

def heuristic(state):
    # Count pairs of queens that can attack each other
    attacks = 0
    for i in range(len(state)):
        for j in range(i + 1, len(state)):
            if state[i] == state[j] or abs(state[i] - state[j]) == j - i:
                attacks += 1
    return attacks

def hill_climbing_8_queens():
    # Random initial state
    state = [random.randint(0, 7) for _ in range(8)]
    while True:
        current_h = heuristic(state)
        if current_h == 0:  # Found a solution
            return state

        next_state = None
        next_h = float('inf')

        for col in range(8):
            for row in range(8):
                if state[col] != row:  # Only consider moving the queen
                    new_state = state.copy()
                    new_state[col] = row
                    h = heuristic(new_state)
                    if h < next_h:
                        next_h = h
                        next_state = new_state
```

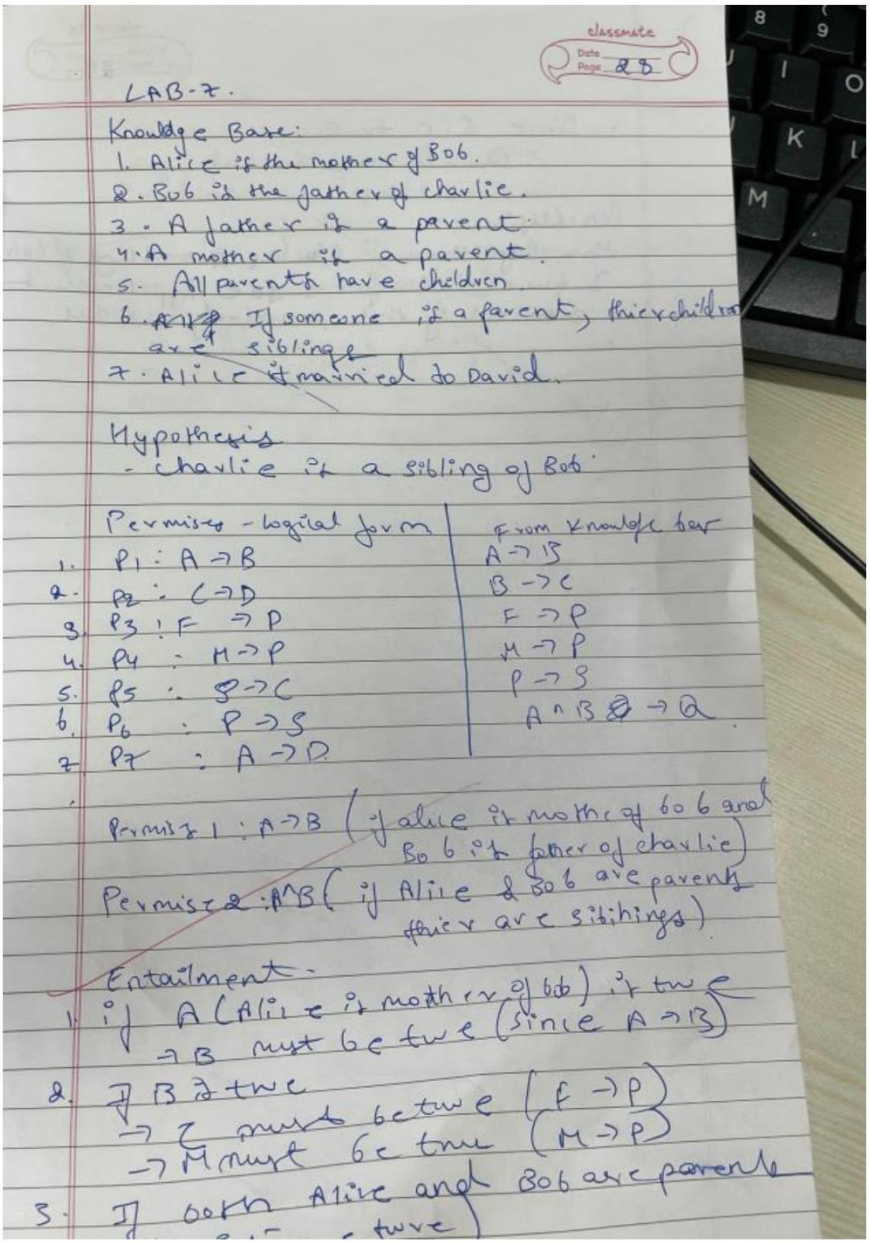
```
        if next_h >= current_h:  # No better neighbor found
            return None  # Stuck at local maximum

        state = next_state

def hill_climbing_with_random_restarts(max_restarts=100):
    for _ in range(max_restarts):
        solution = hill_climbing_8_queens()
        if solution:
            return solution
    return None  # No solution found after max_restarts

solution = hill_climbing_with_random_restarts()
if solution:
    print("Hill Climbing solution:", solution)
else:
    print("No solution found after maximum restarts.")

print("Pooja M")
print("1BM22CS195")
```

Output:

```
Hill Climbing solution: [4, 2, 7, 3, 6, 0, 5, 1]
```

Program 7 - Entailment Using Literals

Algorithm:



LAB-7.

Knowledge Base:
1. Alice is the mother of Bob.
2. Bob is the father of charlie.
3. A father is a parent.
4. A mother is a parent.
5. All parents have children.
6. If someone is a parent, their children are siblings.
7. Alice is married to David.

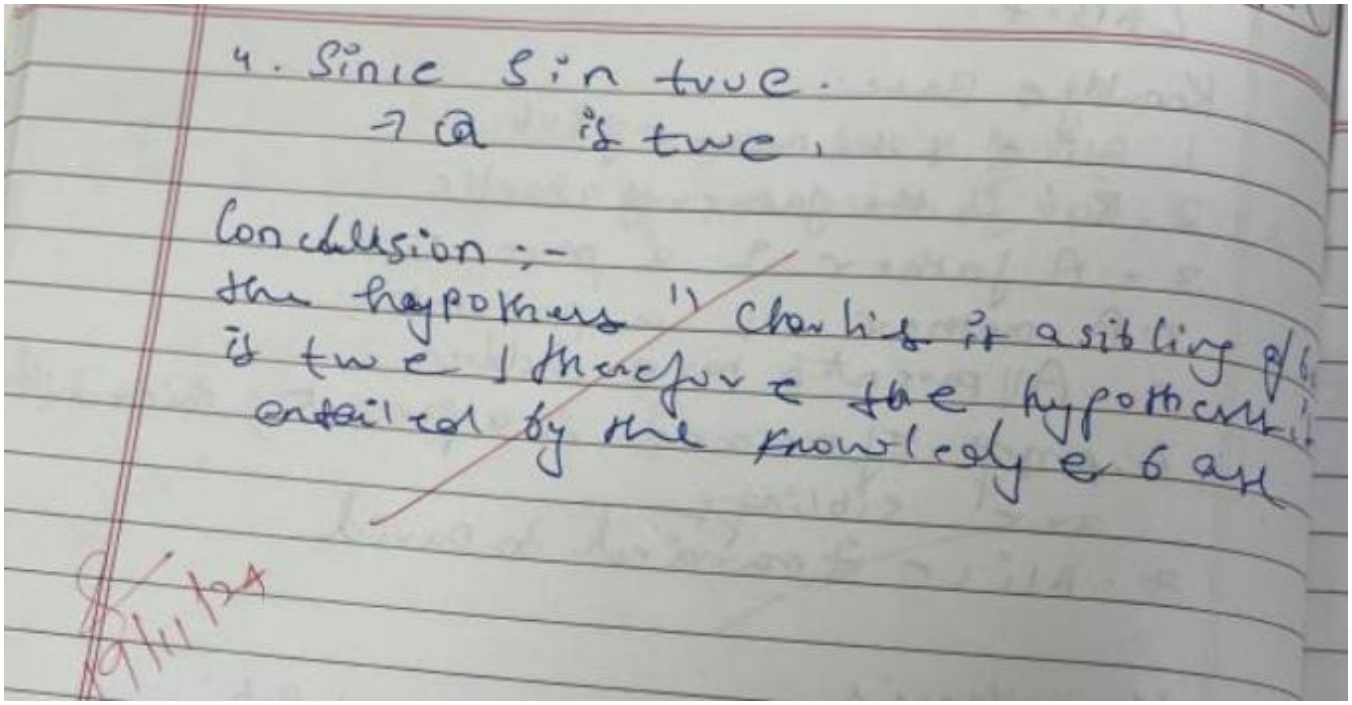
Hypothesis
- charlie is a sibling of Bob.

Permises - logical form	From Knowledge Base
1. $P_1 : A \rightarrow B$	$A \rightarrow B$
2. $P_2 : C \rightarrow D$	$B \rightarrow C$
3. $P_3 : F \rightarrow P$	$F \rightarrow P$
4. $P_4 : M \rightarrow P$	$M \rightarrow P$
5. $P_5 : S \rightarrow C$	$P \rightarrow S$
6. $P_6 : P \rightarrow S$	$A \wedge B \rightarrow Q$
7. $P_7 : A \rightarrow D$	

Permise 1 : A → B (if alice is mother of bob and Bob is father of charlie)

Permise 2 : A^B (if Alice & Bob are parents their are sibilings)

Entailment:
1. if A (Alice is mother of bob) is true → B must be true (since A → B)
2. If B is true → C must be true (F → P) → M must be true (M → P)
3. If both Alice and Bob are parents



Code:

```
import re

# Helper function to parse user input into logical predicates
def parse_input(input_sentence, knowledge_base):
    # Convert the sentence to lowercase for consistency
    input_sentence = input_sentence.lower()

    # Match patterns for predicates and facts (e.g., 'X is the mother of Y' or 'X is married to Y')
    # Fact or Rule: "X is the mother of Y"
    mother_match = re.match(r"(\w+) is the mother of (\w+)", input_sentence)
    # Fact or Rule: "X is the father of Y"
    father_match = re.match(r"(\w+) is the father of (\w+)", input_sentence)
    # General rule: "All X have children"
    parent_match = re.match(r"all (\w+) have children", input_sentence)
    # Rule for parent-child relation and siblings
    parent_rule_match = re.match(r"if someone is a parent, their children are siblings", input_sentence)
    # General fact: "X is married to Y"
```

```
married_match = re.match(r"(\w+) is married to (\w+)", input_sentence)

# Parsing rules and facts
if mother_match:
    mother, child = mother_match.groups()
    # Add the mother-child relationship to knowledge base
    knowledge_base["Mother"].append((mother.capitalize(), child.capitalize()))
elif father_match:
    father, child = father_match.groups()
    # Add the father-child relationship to knowledge base
    knowledge_base["Father"].append((father.capitalize(), child.capitalize()))
elif parent_match:
    parent = parent_match.group(1)
    # Rule: All X are parents with children
```

```
knowledge_base["ParentRule"].append((parent.capitalize(), "HasChildren")) elif
parent_rule_match:
        # General rule: If someone is a parent, their children are siblings
        knowledge_base["ParentSiblingRule"].append(("Parent", "Siblings"))
    elif married_match:
        spouse1, spouse2 = married_match.groups()
        # Add the married relationship to knowledge base
        knowledge_base["Married"].append((spouse1.capitalize(), spouse2.capitalize()))

# Function to check if two children are siblings
def are_siblings(child1, child2, knowledge_base):
    # Check if both children share the same parent
    parents = set()
    for mother, child in knowledge_base["Mother"]:
        if child == child1:
            parents.add(mother)
        if child == child2:
            parents.add(mother)
    for father, child in knowledge_base["Father"]:
        if child == child1:
            parents.add(father)
        if child == child2:
            parents.add(father)
    return len(parents) > 1  # If both children share a parent, they are siblings

# Function to check the hypothesis "Charlie is a sibling of Bob"
def check_hypothesis(hypothesis, knowledge_base):
    # Parse the hypothesis
    hyp_match = re.match(r"(\w+) is a sibling of (\w+)", hypothesis.lower())
    if hyp_match:
        child1, child2 = hyp_match.groups()
        # Check if the children are siblings
```

```
        if are_siblings(child1.capitalize(), child2.capitalize(), knowledge_base):
            return True
    return False

# Main function for user input and entailment reasoning
def main():
    # Create an empty knowledge base
    knowledge_base = {
        "Mother": [],
        "Father": [],
        "ParentRule": [],
        "ParentSiblingRule": [],
        "Married": []
    }

    print("Enter knowledge base rules. Type 'done' when finished.")
    # Allow the user to input knowledge base facts, rules, or actions
    while True:
        user_input = input("Enter rule: ").strip()
        if user_input.lower() == "done":
            break
        parse_input(user_input, knowledge_base)

    # Print the current knowledge base
    print("\nCurrent Knowledge Base:")
    for category, items in knowledge_base.items():
        print(f"{category}: {items}")

    # Ask for the hypothesis (the statement to check)
    hypothesis = input("\nEnter hypothesis to check: ").strip()

    # Check if the hypothesis is entailed
```

```
    if check_hypothesis(hypothesis, knowledge_base):
        print(f"\nConclusion: The hypothesis '{hypothesis}' is entailed by the knowledge base.")
    else:
        print(f"\nConclusion: The hypothesis '{hypothesis}' is NOT entailed by the knowledge base.")

# Run the program
main()

print("Pooja M")
print("1BM22CS195")
```

Output:

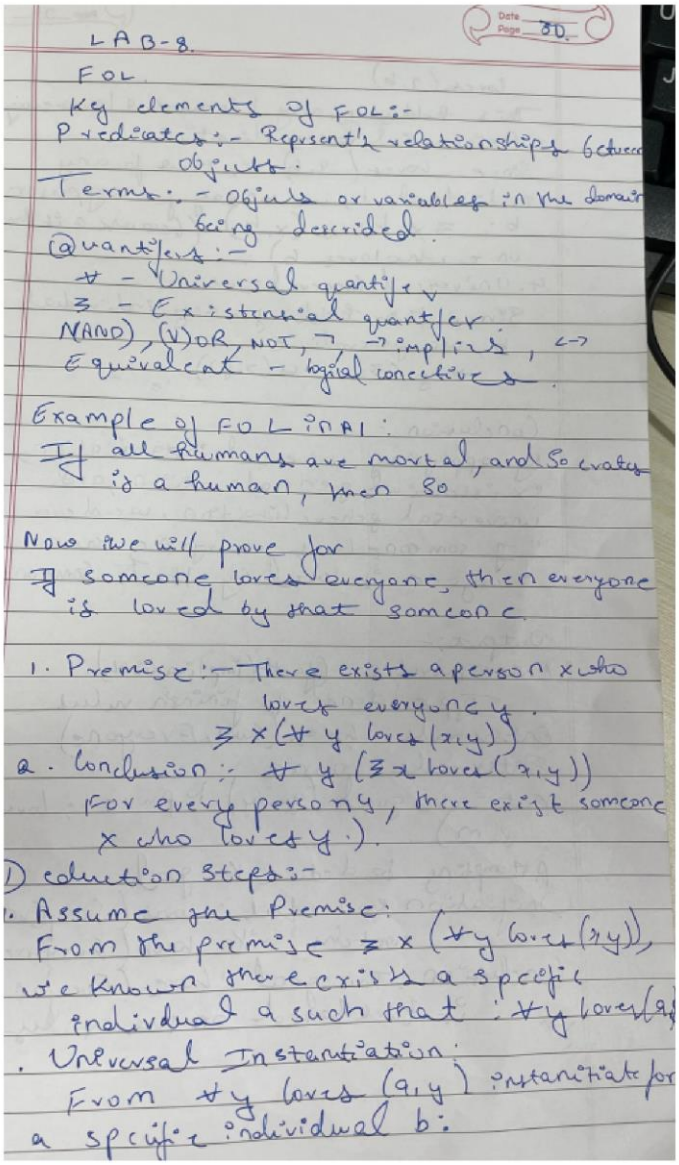
output:

```
Welcome to the Entailment Checker!
Enter the fact: Alice is the mother of Bob. (e.g., 'Alice is the mother of Bob')
 Alice is the mother of Bob
Enter the fact: Bob is the father of Charlie. (e.g., 'Bob is the father of Charlie')
Bob is the father of Charlie
Enter the fact: A father is a parent. (e.g., 'A father is a parent')
A father is a parent
Enter the fact: A mother is a parent. (e.g., 'A mother is a parent')
 A mother is a parent
Enter the fact: All parents have children. (e.g., 'All parents have children')
All parents have children
Enter the fact: Parents' children are siblings. (e.g., 'Parents' children are siblings')
Parents' children are siblings
Enter the fact: Alice is married to David. (e.g., 'Alice is married to David')
Alice is married to David

Since Alice is Bob's mother and Bob is Charlie's father, Charlie and Bob are siblings.
Conclusion: Charlie is a sibling of Bob. The hypothesis is entailed by the knowledge base.
```

Observation book

Program 8 - FOL using Unification

Algorithm:



LAB-8.

FOL

Key elements of FOL:-

Predicates:- Represents relationships between objects.

Terms:- Objects or variables in the domain being described.

Quantifiers:-

∀ - Universal quantifier

∃ - Existential quantifier

NAND), (V)OR, NOT, ¬ → implies, <-> Equivalent - logical connectives

Example of FOL in AI:

If all humans are mortal, and Socrates is a human, then so

Now we will prove for

If someone loves everyone, then everyone is loved by that someone

1. Premise:- There exists a person x who loves everyone y.

∃ x (∀ y loves(x,y))

2. Conclusion:- ∀ y (∃ x loves(x,y))

For every person y, there exist someone x who loves y).

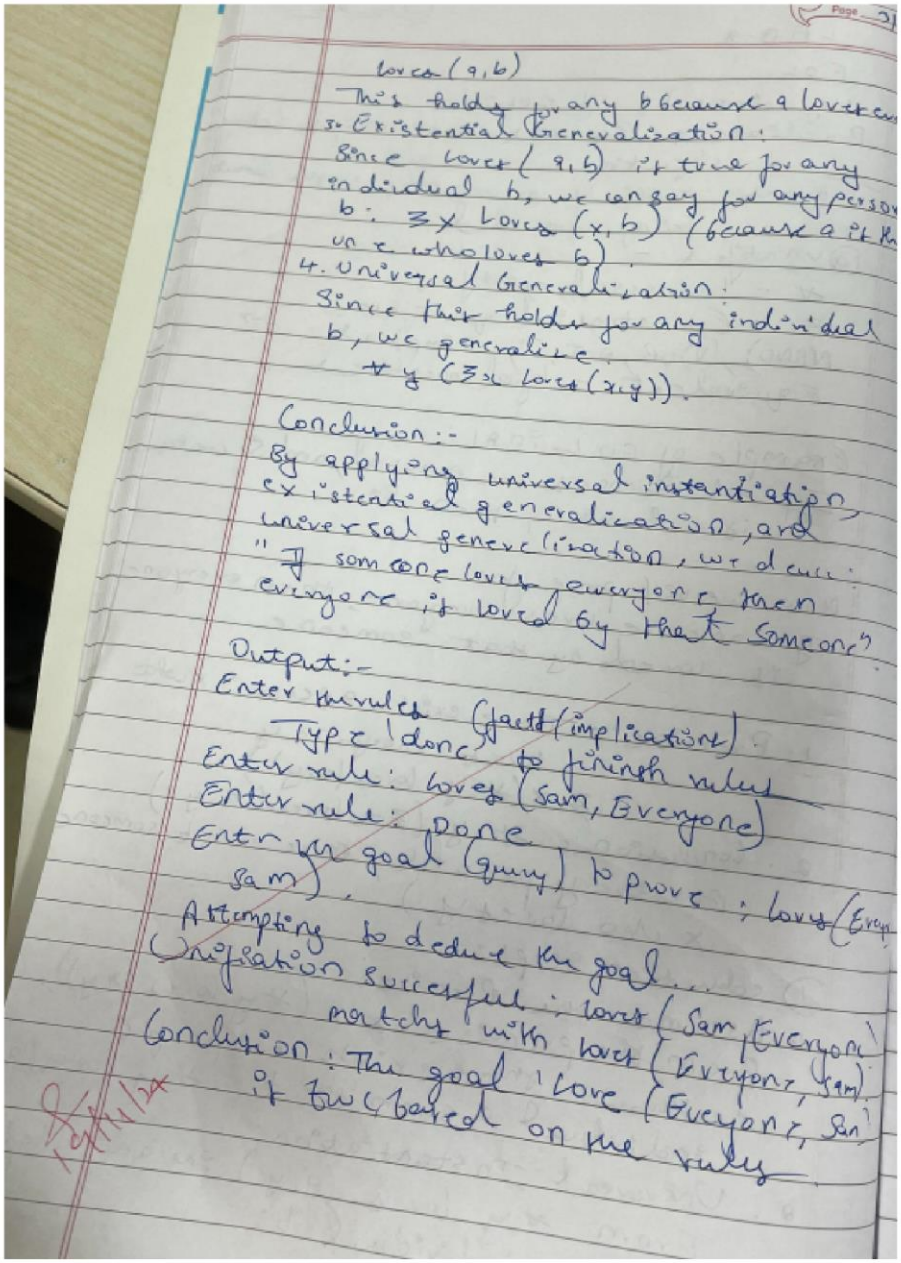
1) eduction steps:-

1. Assume the Premise:

From the premise ∃ x (∀ y loves(x,y)), we know there exists a specific individual a such that: ∀ y loves(a,

2. Universal Instantiation:

From ∀ y loves (a,y) ; instantiate for a specific individual b:


Loves(a,b)
This holds for any b because a loves ev
3. Existential Generalization:
Since Loves(a,b) is true for any
individual b, we can say for any person
b: ∃x Loves(x,b) (because a is th
or x who loves b)
4. Universal Generalization:
Since this holds for any individual
b, we generalize.
∀y (∃x Loves(x,y)).

Conclusion:-
By applying universal instantiation,
existential generalization, and
universal generalization, we deduce:
"If someone loves everyone, then
everyone is loved by that someone"

Output:-
Enter the rules (facts/implications)
Type 'done' to finish rules
Enter rule: loves(Sam, Everyone)
Enter rule: done
Enter the goal (query) to prove: loves(Every
Sam)
Attempting to deduce the goal...
Unification successful: loves(Sam, Everyone)
matches with loves(Everyone, Sam)
Conclusion: The goal 'Love(Everyone, Sa
is true based on the rules.

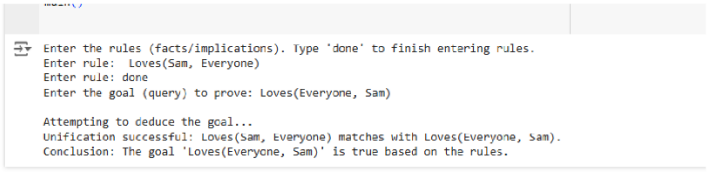
Code:

```
import re

# Define a simple function for extracting predicates from sentences
def extract_predicate(sentence):
    # Regular expression to find patterns like Predicate(Argument)
    pattern = r"([A-Za-z]+)\((\w+)\)"
    match = re.search(pattern, sentence)
    if match:
        predicate = match.group(1)
        subject = match.group(2)
        return predicate, subject
    return None, None

# Function for unification
def unify(fact, query):
    # Check if the fact and query are the same
    if fact == query:
        return True
    # Extract predicate and subject from fact and query
    fact_predicate, fact_subject = extract_predicate(fact)
    query_predicate, query_subject = extract_predicate(query)
    # If predicates match, unify the subjects
    if fact_predicate == query_predicate:
        if fact_subject == query_subject:
            return True
        else:
            # Here, we could handle variable substitution (unification)
            return False
    return False
```

```
# Function to deduce the goal using given rules
def deduct(rules, goal):
    # Try to find unification for the goal from the rules
    for rule in rules:
        if unify(rule, goal):
            print(f"Unification successful: {rule} matches with {goal}.")
            return True
    return False

# Main function to handle user input
def main():
    # Step 1: Get the rules (facts/implications) from the user
    print("Enter the rules (facts/implications). Type 'done' to finish entering rules.")
    rules = []
    while True:
        rule_input = input("Enter rule: ")
        if rule_input.lower() == 'done':
            break
        else:
            rules.append(rule_input.strip())

    # Step 2: Get the goal (query) from the user
    goal_input = input("Enter the goal (query) to prove: ").strip()

    # Step 3: Try to deduce the goal using the given rules
    print("\nAttempting to deduce the goal...")
    if deduct(rules, goal_input):
        print(f"Conclusion: The goal '{goal_input}' is true based on the rules.")
    else:
        print(f"Conclusion: The goal '{goal_input}' cannot be proven with the provided rules.")
```

```
# Run the program
main()
print("Pooja M")
print("1BM22CS195")
```

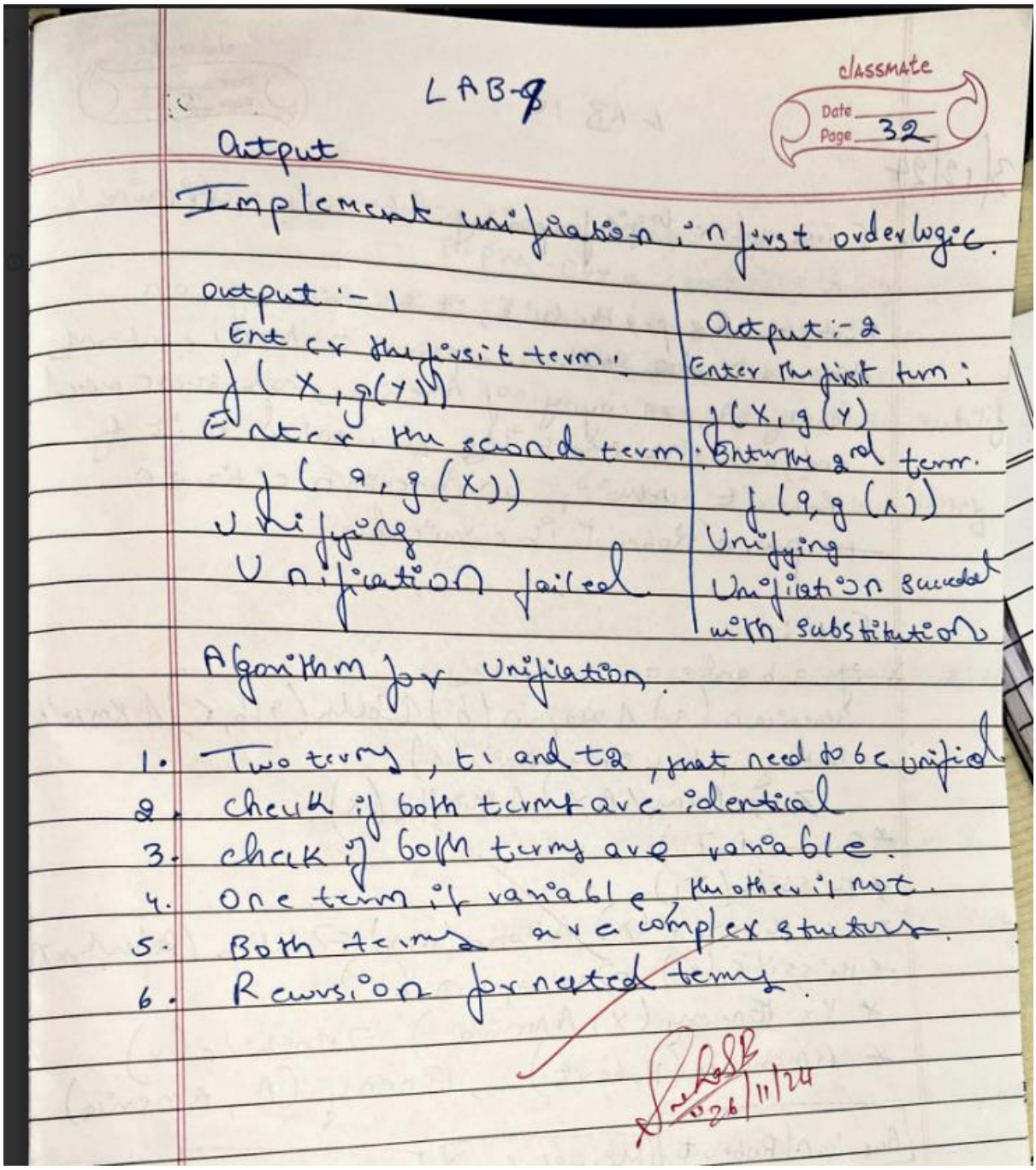
Output:



observation book

Program 9 - Unification

Observation book:



Code:

```
def is_variable(term):
    """
    Check if a term is a variable.
```

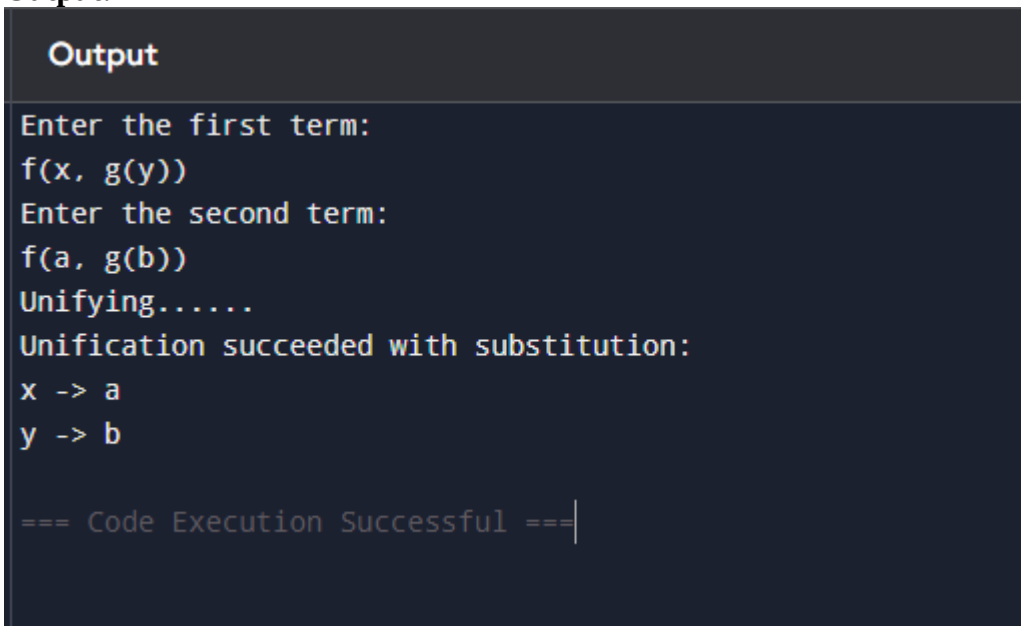
```
    Variables are typically single lowercase letters.
    """
    return isinstance(term, str) and term.islower()


def unify(expr1, expr2, subst={}):
    """
    Unify two expressions expr1 and expr2 under the given substitution subst.
    """
    if subst is None:
        return None  # Failure case
    if expr1 == expr2:
        return subst  # Expressions are identical
```

```
    if is_variable(expr1):
        return unify_variable(expr1, expr2, subst)
    if is_variable(expr2):
        return unify_variable(expr2, expr1, subst)
    if isinstance(expr1, tuple) and isinstance(expr2, tuple):
        if len(expr1) != len(expr2):
            return None  # Different arity
        # Recursively unify each component
        for arg1, arg2 in zip(expr1, expr2):
            subst = unify(arg1, arg2, subst)
            if subst is None:
                return None  # Failure
        return subst
    return None  # No unification possible

def unify_variable(var, term, subst):
    """
    Unify a variable with a term, updating the substitution.
    """
    if var in subst:
        return unify(subst[var], term, subst)  # Apply substitution to var
    if term in subst:
        return unify(var, subst[term], subst)  # Apply substitution to term
    if occurs_check(var, term, subst):
        return None  # Circular substitution detected
    # Add var -> term to the substitution
    subst = subst.copy()
    subst[var] = term
    return subst

def occurs_check(var, term, subst):
    """
```

```
    Check if var occurs in term (directly or indirectly) to prevent circular substitutions.
    """
    if var == term:
        return True
    if isinstance(term, tuple):
        return any(occurs_check(var, t, subst) for t in term)
    if term in subst:
        return occurs_check(var, subst[term], subst)
    return False

def parse_input(expr):
    """
    Parse user input into a structured format (nested tuples for functions and terms).
    Example: "f(X, g(y))" -> ('f', 'X', ('g', 'y'))
    """
    expr = expr.strip()
    if '(' not in expr:
        return expr  # Simple variable or constant
    func_name = expr[:expr.index('(')].strip()
    args = expr[expr.index('(') + 1:expr.rindex(')')].split(',')
    args = [parse_input(arg.strip()) for arg in args]
    return (func_name, *args)

def format_output(expr):
    """
    Convert the nested tuple representation back into a string for output.
    Example: ('f', 'X', ('g', 'y')) -> "f(X, g(y))"
    """
    if isinstance(expr, str):
        return expr
    return f"{expr[0]}({', '.join(format_output(arg) for arg in expr[1:])})"
```

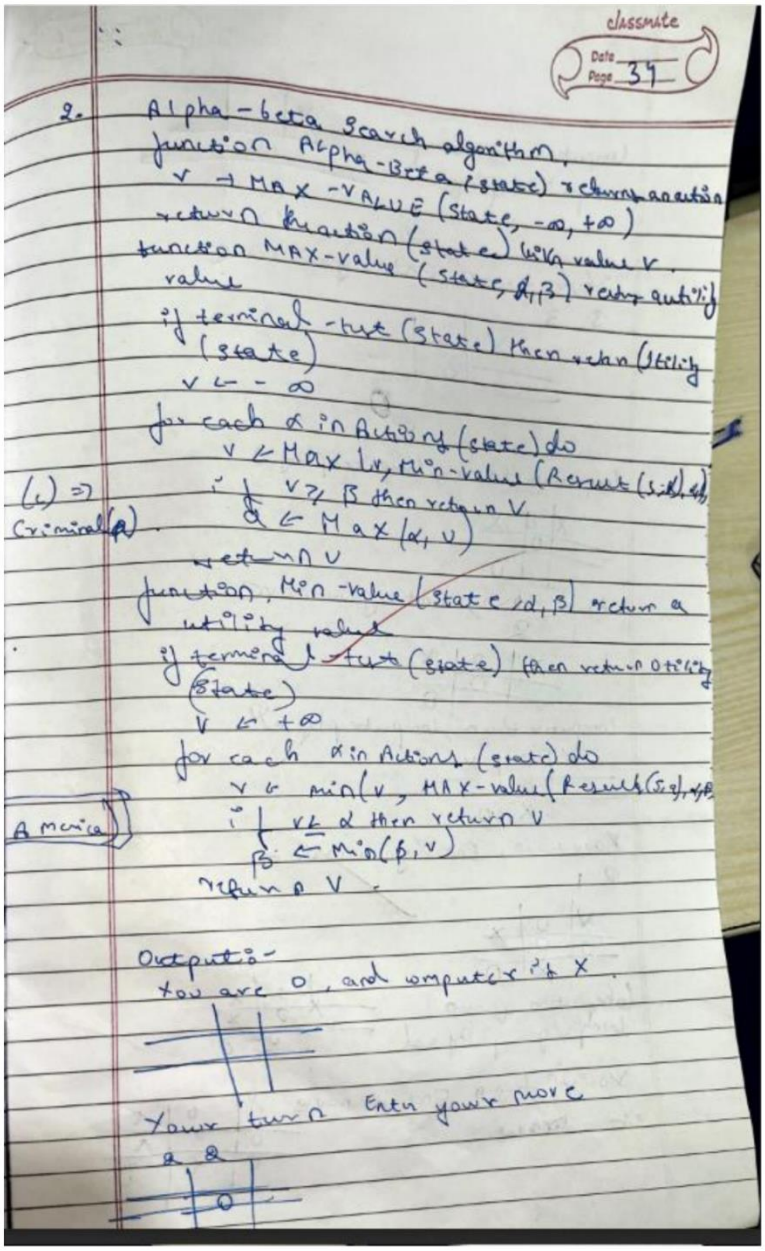
```
# Main Program
if __name___== "__main__":
    print("Enter the first term:")
    expr1 = parse_input(input().strip())
    print("Enter the second term:")
    expr2 = parse_input(input().strip())
    print("Unifying. .... ")
    result = unify(expr1, expr2)
    if result is None:
        print("Unification failed")
    else:
        print("Unification succeeded with substitution:")
        for var, term in result.items():
            print(f"{var} -> {format_output(term)}")

print("Pooja M")
print("1BM22CS195")
```

Output:



Program 10 - Tic Tac Toe using Min-Max.

Algorithm:



Code:

```
import math
# Constants for players
HUMAN = 'O'  # Minimizer
AI = 'X'      # Maximizer
# Initialize empty board
def create_board():
    return [[' ' for _ in range(3)] for _ in range(3)]
# Check if there are any moves left on the board
def is_moves_left(board):
    for row in board:
        if ' ' in row:
            return True
    return False
# Check for a win condition
def evaluate(board):
    # Rows, columns, diagonals check
    for row in board:
        if row[0] == row[1] == row[2] and row[0] != ' ':
            return 1 if row[0] == AI else -1
    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col] and board[0][col] != ' ':
            return 1 if board[0][col] == AI else -1
    if board[0][0] == board[1][1] == board[2][2] and board[0][0] != ' ':
        return 1 if board[0][0] == AI else -1
    if board[0][2] == board[1][1] == board[2][0] and board[0][2] != ' ':
        return 1 if board[0][2] == AI else -1
    return 0  # No winner
# Minimax algorithm with Alpha-Beta Pruning
def minimax(board, depth, is_maximizing, alpha, beta):
    score = evaluate(board)
    # Terminal condition
```

```
    if score == 1:  # AI wins
        return score - depth  # Prefer quicker wins
    if score == -1:  # Human wins
        return score + depth  # Prefer slower losses
    if not is_moves_left(board):  # Draw
        return 0
    if   is_maximizing:
        best  =  -math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == ' ':
                    board[i][j] = AI
                    best = max(best, minimax(board, depth + 1, False, alpha, beta))
                    board[i][j] = ' '
                    alpha = max(alpha, best)
                    if beta <= alpha:
                        break
        return best
    else:
        best = math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == ' ':
                    board[i][j] = HUMAN
                    best = min(best, minimax(board, depth + 1, True, alpha, beta))
                    board[i][j] = ' '
                    beta = min(beta, best)
                    if beta <= alpha:
                        break
        return best
# Find the best move for the AI
def find_best_move(board):
```

```
    best_val = -math.inf
    best_move = (-1, -1)
    for i in range(3):
        for j in range(3):
            if board[i][j] == ' ':
                board[i][j] = AI
                move_val = minimax(board, 0, False, -math.inf, math.inf)
                board[i][j] = ' '
                if move_val > best_val:
                    best_val = move_val
                    best_move = (i, j)
    return best_move
# Print the board
def print_board(board):
    for row in board:
        print('|'.join(row))
    print('-' * 5)
# Example usage
if __name___== '__main__':
    board = create_board()
    while is_moves_left(board):
        print_board(board)
        # Human makes a move
        row, col = map(int, input("Enter row and column (0, 1, 2): ").split())
        if board[row][col] == ' ':
            board[row][col] = HUMAN
        else:
            print("Invalid move! Try again.")
            continue
        if evaluate(board) != 0 or not is_moves_left(board):
            break
        # AI makes a move
```

```
        print("AI is making a move...")
        ai_move = find_best_move(board)
        board[ai_move[0]][ai_move[1]] = AI
        if evaluate(board) != 0 or not is_moves_left(board):
            break

    # Final result
    print_board(board)
    result = evaluate(board)
    if result == 1:
        print("AI wins!")
    elif result == -1:
        print("Human wins!")
    else:
        print("It's a draw!")
print("Pooja M")
print("1BM22CS195")
```

Output:

```
        print("Computer wins!")
    elif winner == PLAYER_O:
        print("Congratulations! You win!")
    else:
        print("It's a draw!")

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

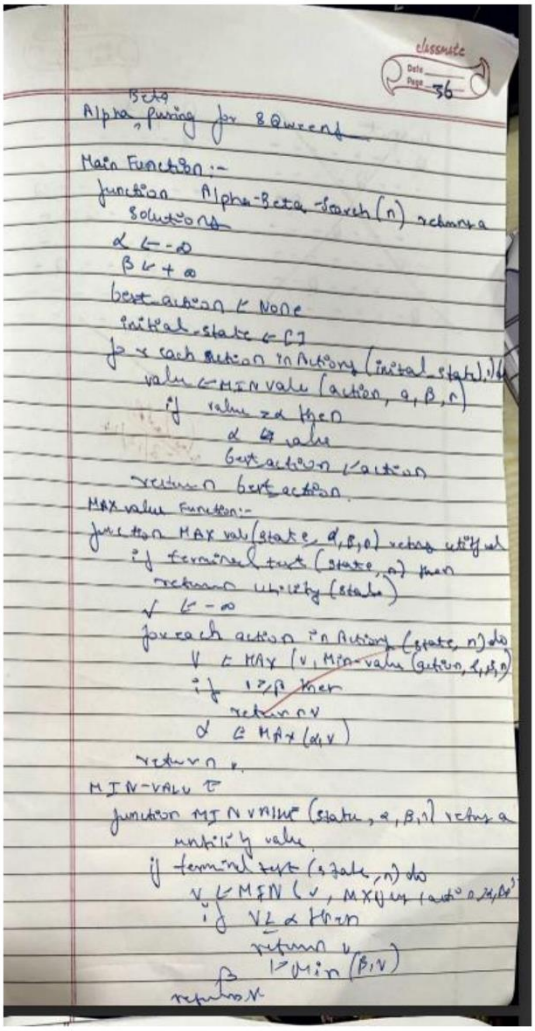
ouput:-

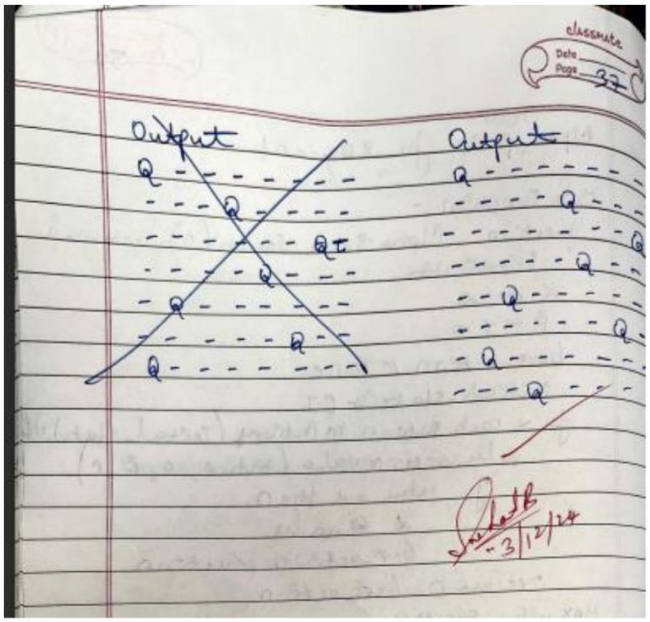
```
Welcome to Tic-Tac-Toe!
You are 'O', and the computer is 'X'.
- | - | -
- | - | -
- | - | -

Your turn! Enter your move as 'row col' (e.g., '1 2'):
2 2
You played:
- | - | -
- | O | -
- | - | -

Computer's turn...
Computer played:
X | - | -
- | O | -
- | - | -

Your turn! Enter your move as 'row col' (e.g., '1 2'):
33
Invalid input! Please enter two numbers separated by a space.
Your turn! Enter your move as 'row col' (e.g., '1 2'):
3 3
You played:
X | - | -
- | O | -
- | - | O

Computer's turn...
Computer played:
X |   | X
- | O | -
- | - | O
```

```
1 2
You played:
X | O | X
- | O | -
- | - | O

Computer's turn...
Computer played:
X | O | X
- | O | -
- | X | O

Your turn! Enter your move as 'row col' (e.g., '1 2'):
2 1
You played:
X | O | X
O | O | -
- | X | O

Computer's turn...
Computer played:
X | O | X
O | O | X
- | X | O

Your turn! Enter your move as 'row col' (e.g., '1 2'):
3 1
You played:
X | O | X
O | O | X
O | X | O

It's a draw!
```

Alpha-Beta pruning For 8 Queens.

Algorithm:




classmate
Date
Page 27
Output
Output
3/12/24

Code:

```
def is_safe(board, row, col):
    """
    Check if it's safe to place a queen at board[row][col]
```

```
    """
    # Check for queen in the same column
    for i in range(row):
        if board[i][col] == 1:
            return False
    # Check for queen in the left diagonal
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    # Check for queen in the right diagonal
    for i, j in zip(range(row, -1, -1), range(col, len(board))):
        if board[i][j] == 1:
            return False
    return True
def solve_with_alpha_beta(board, row, alpha, beta):
    """
    Solve the 8-Queens problem using Alpha-Beta Pruning.
    """
    if row >= len(board):  # All queens placed successfully
        return True
```

```
    for col in range(len(board)):
        if is_safe(board, row, col):
            # Place the queen
            board[row][col] = 1
            # Recursive call to place the next queen
            if solve_with_alpha_beta(board, row + 1, alpha, beta):
                return True
            # Backtrack if placing the queen here leads to failure
            board[row][col] = 0
        # Update alpha and beta for pruning (though not strictly necessary for 8-Queens)
        alpha = max(alpha, col)
        if beta <= alpha:
            break  # Prune
    return False
def solve_8_queens():
    """
    Solves the 8-Queens problem and prints the solution.
    """
    n = 8
    board = [[0 for _ in range(n)] for _ in range(n)]
```

```
    # Start solving with Alpha-Beta Pruning
    if solve_with_alpha_beta(board, 0, -float('inf'), float('inf')):
        print("Solution:")
        for row in board:
            print(' '.join('Q' if cell == 1 else '.' for cell in row))
    else:
        print("No solution found.")
# Execute the solver
if __name__ == "__main__":
    solve_8_queens()
print("Pooja M")
print("1BM22CS195")
```

Output:

```
output:-
Solution: [0, 4, 7, 5, 2, 6, 1, 3]
Observation book :-
```