VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Praneeta M Reddy (1BM22CS205)

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 **Praneeta M Reddy(1BM22CS205),** who is 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.

Radhika A D Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

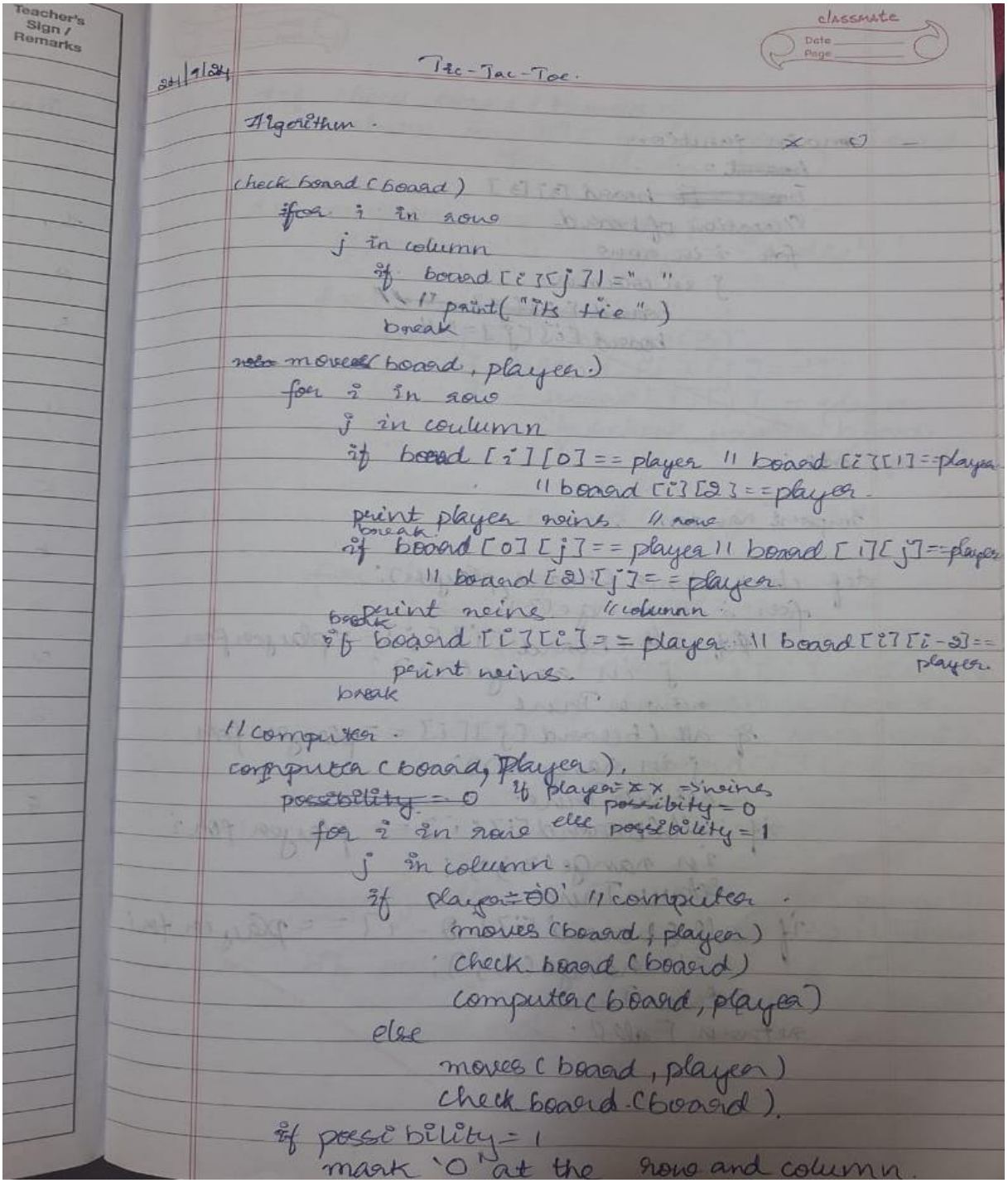
Index

Github Link:
https://github.com/Praneeta205/AI-5-SEM

Program-1

Implement Tic –Tac –Toe Game

Algorithm:



Code:

```
import random

def check_winner(board, player):
    for i in range(3):
        if all(board[i][j] == player for j in range(3)):  # Check row
            return True
        if all(board[j][i] == player for j in range(3)):  # Check column
            return True
    if all(board[i][i] == player for i in range(3)):  # Check diagonal
        return True
    if all(board[i][2 - i] == player for i in range(3)):  # Check anti-diagonal
        return True
    return False

def check_board(board):
    return any(cell == "" for row in board for cell in row)

def computer_move(board, player):
    opponent = 'O' if player == 'X' else 'X'

    # Check for a winning move
    for i in range(3):
        for j in range(3):
            if board[i][j] == "":
                board[i][j] = player
                if check_winner(board, player):
                    return
                board[i][j] = ""

    # Block opponent's winning move
    for i in range(3):
        for j in range(3):
            if board[i][j] == "":
                board[i][j] = opponent
                if check_winner(board, opponent):
                    board[i][j] = player  # Block
                    return
                board[i][j] = ""

    # Make a random move
    empty_cells = [(i, j) for i in range(3) for j in range(3) if board[i][j] == ""]
    if empty_cells:
        i, j = random.choice(empty_cells)
        board[i][j] = player
```

```
def print_board(board):
    for row in board:
        print(" | ".join(cell if cell else " " for cell in row))
        print("-" * 9)

def tic_tac_toe():
    board = [["" for _ in range(3)] for _ in range(3)]
    first_player = random.choice(['X', 'O'])  # Randomly choose who goes first
    current_player = first_player

    print(f"Player {first_player} goes first!")

    while check_board(board):
        print_board(board)
        if current_player == 'X':
            row, col = map(int, input("Enter your move (row and column): ").split())
            if board[row][col] == "":
                board[row][col] = current_player
                if check_winner(board, current_player):
                    print_board(board)
                    print("Player X wins!")
                    return
                current_player = 'O'
            else:
                print("Invalid move. Try again.")
        else:
            computer_move(board, current_player)
            if check_winner(board, current_player):
                print_board(board)
                print("Computer O wins!")
                return
            current_player = 'X'

    print_board(board)
    print("It's a draw!")

# Run the games
tic_tac_toe()
```

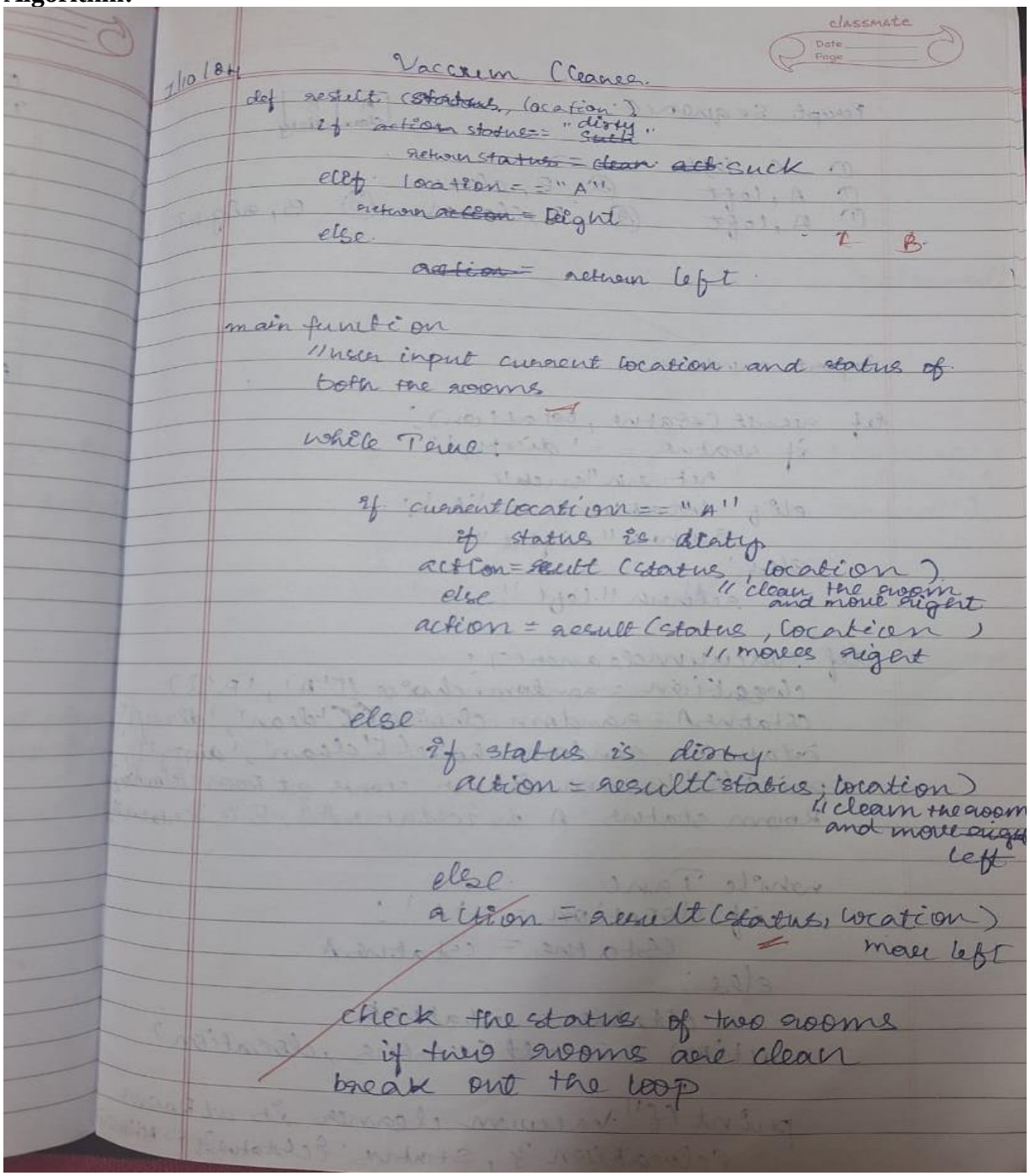
Output:

```
Enter your move (row and column): 0 0
X |   |
---------
  |   |
---------
  |   |
---------
X |   |
---------
  |   |
---------
  | O |
---------
Enter your move (row and column): 1 1
X |   |
---------
  | X |
---------
  | O |
---------
X |   |
---------
  | X |
---------
  | O | O
---------
Enter your move (row and column): 0 2
X |   | X
---------
  | X |
---------
  | O | O
---------
X |   | X
---------
  | X |
---------
O | O | O
---------
Computer O wins!
```

Program-2

Implement Vacuum Cleaner Agent:

Algorithm:



For 2 rooms:
Code:

```
import random

def result(status, location):
    if status == 'dirty':
        return "suck"
    directions = {'A': "right", 'B': "up"}
    return directions.get(location, "up")

def vacuumcleaner():
    locations = ['A', 'B']
    statuses = {location: random.choice(['clean', 'dirty']) for location in locations}
    clocation = random.choice(locations)

    print(f"Vacuum cleaner is at Room {clocation}, "
          f"Room statuses: {', '.join([f'{loc}: {status}' for loc, status in statuses.items()])}")

    while any(status == 'dirty' for status in statuses.values()):
        cstatus = statuses[clocation]
        action = result(cstatus, clocation)
        print(f"Vacuum cleaner is at Room {clocation}, Status: {cstatus} -> Action: {action}")

        if action == "suck":
            statuses[clocation] = "clean"
        else:
            clocation = {'right': 'B', 'up': 'A'}.get(action, clocation)

    print("All rooms are now clean. Task finished.")

vacuumcleaner()
```

Output:

```
Vacuum cleaner is at Room A, Room statuses: A: dirty, B: dirty
Vacuum cleaner is at Room A, Status: dirty -> Action: suck
Vacuum cleaner is at Room A, Status: clean -> Action: right
Vacuum cleaner is at Room B, Status: dirty -> Action: suck
All rooms are now clean. Task finished.
```

For 4 rooms:
Code:

```
import random

def result(status, location):
    if status == 'dirty':
        return "suck"
    directions = {'A': "right", 'B': "down", 'C': "left", 'D': "up"}
    return directions.get(location, "up")

def vacuumcleaner():
    locations = ['A', 'B', 'C', 'D']
    statuses = {location: random.choice(['clean', 'dirty']) for location in locations}
    clocation = random.choice(locations)

    print(f"Vacuum cleaner is at Room {clocation}, "
          f"Room statuses: {', '.join([f'{loc}: {status}' for loc, status in statuses.items()])}")

    while any(status == 'dirty' for status in statuses.values()):
        cstatus = statuses[clocation]
        action = result(cstatus, clocation)
        print(f"Vacuum cleaner is at Room {clocation}, Status: {cstatus} -> Action: {action}")

        if action == "suck":
            statuses[clocation] = "clean"
        else:
            clocation = {'right': 'B', 'down': 'C', 'left': 'D', 'up': 'A'}.get(action, clocation)

    print("All rooms are now clean. Task finished.")

vacuumcleaner()
```

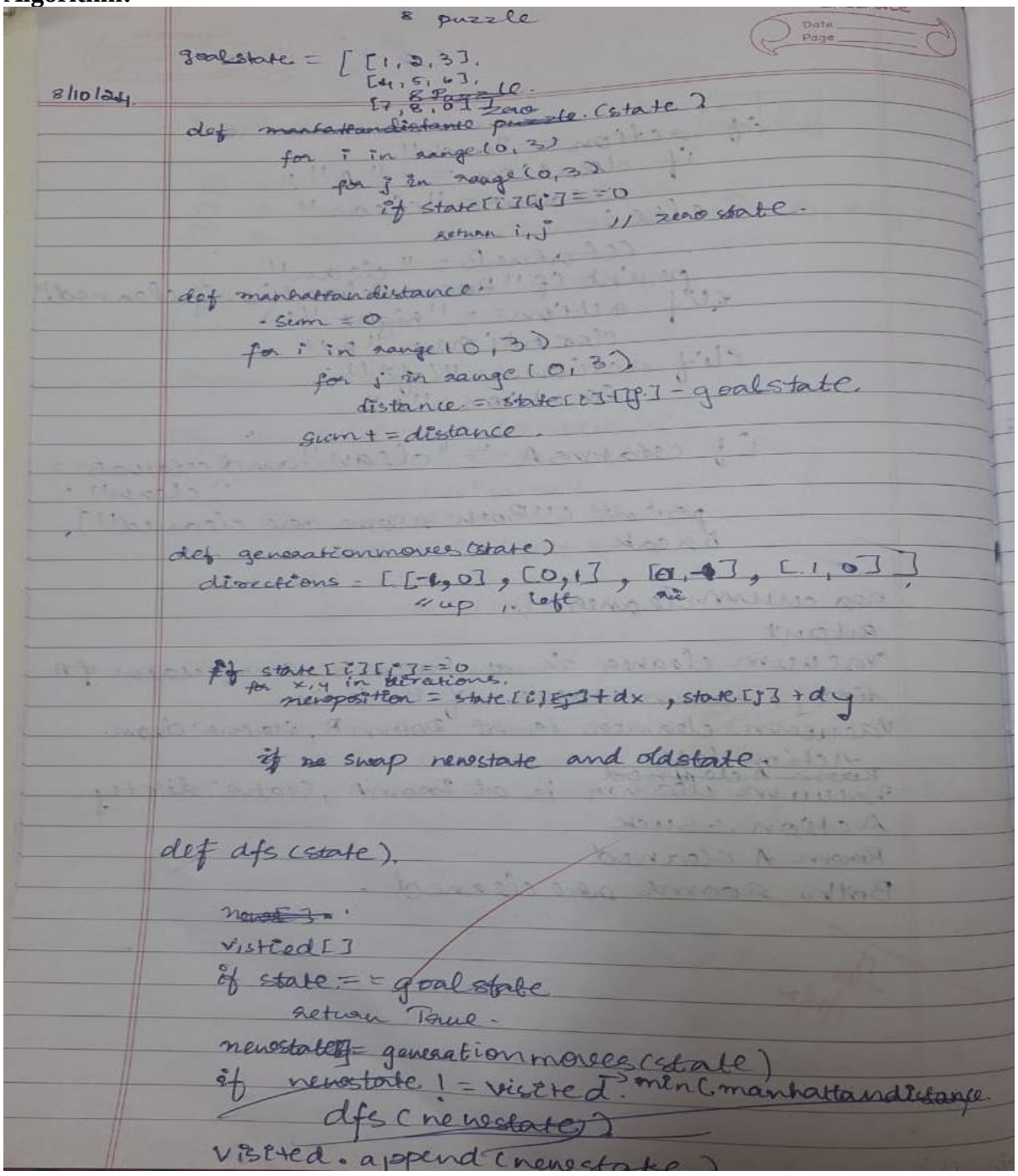
Output:

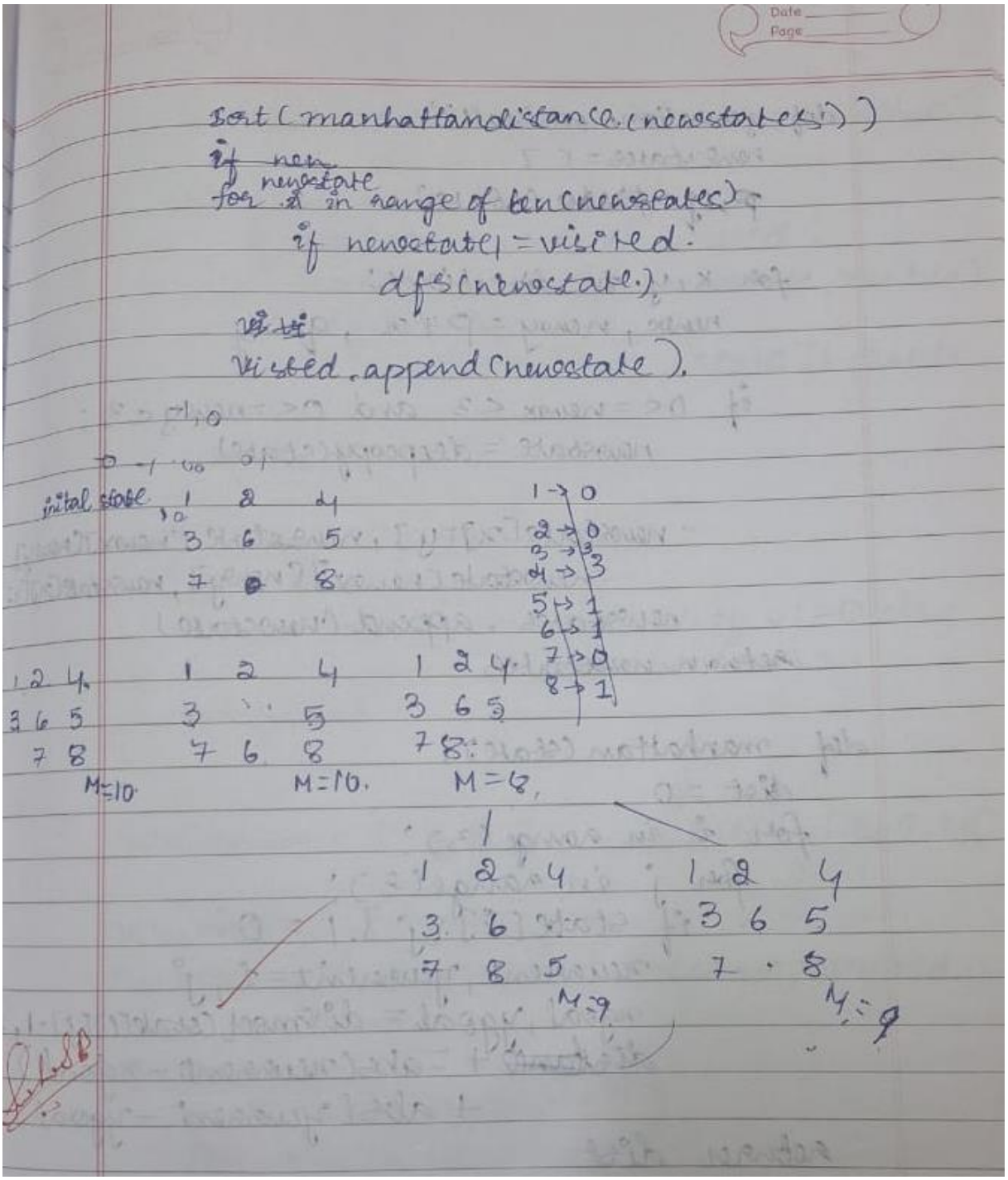
```
Vacuum cleaner is at Room A, Room statuses: A: clean, B: clean, C: dirty, D: clean
Vacuum cleaner is at Room A, Status: clean -> Action: right
Vacuum cleaner is at Room B, Status: clean -> Action: down
Vacuum cleaner is at Room C, Status: dirty -> Action: suck
All rooms are now clean. Task finished.
```

Program-3

Implement 8puzzle using DFS:

Algorithm:





Code:

```
from copy import deepcopy

DIRECTIONS = [(0, 1), (1, 0), (0, -1), (-1, 0)]

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

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

def generate_new_states(state):
    new_states = []
    x, y = find_zero(state)

    for dx, dy in DIRECTIONS:
        new_x, new_y = x + dx, y + dy
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            new_state = deepcopy(state)
            # Swap empty space with the neighboring tile
            new_state[x][y], new_state[new_x][new_y] = new_state[new_x][new_y], new_state[x][y]
            new_states.append(new_state)

    return new_states

def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0: # Skip the empty tile
                # Current position
                x_current, y_current = i, j
                # Goal position
                x_goal, y_goal = divmod(state[i][j] - 1, 3)
                distance += abs(x_current - x_goal) + abs(y_current - y_goal)
    return distance

# Hybrid DFS with Manhattan distance
def dfs_manhattan(state, visited):
    if state == GOAL_STATE:
        return True, [state]

    visited.append(state)

    new_states = generate_new_states(state)

    # Sort new states by Manhattan distance (ascending)
    new_states.sort(key=lambda s: manhattan_distance(s))

    for new_state in new_states:
        if new_state not in visited:
            success, path = dfs_manhattan(new_state, visited)
```

```
            if success:
                return True, [state] + path

    return False, []

def print_puzzle(puzzle):
    for row in puzzle:
        print(' '.join(str(x) if x != 0 else ' ' for x in row))
    print()

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

visited = []
success, solution_path = dfs_manhattan(initial_state, visited)

if success:
    print("Solution found!")
    for step in solution_path:
        print_puzzle(step)
else:
    print("No solution found.")
```

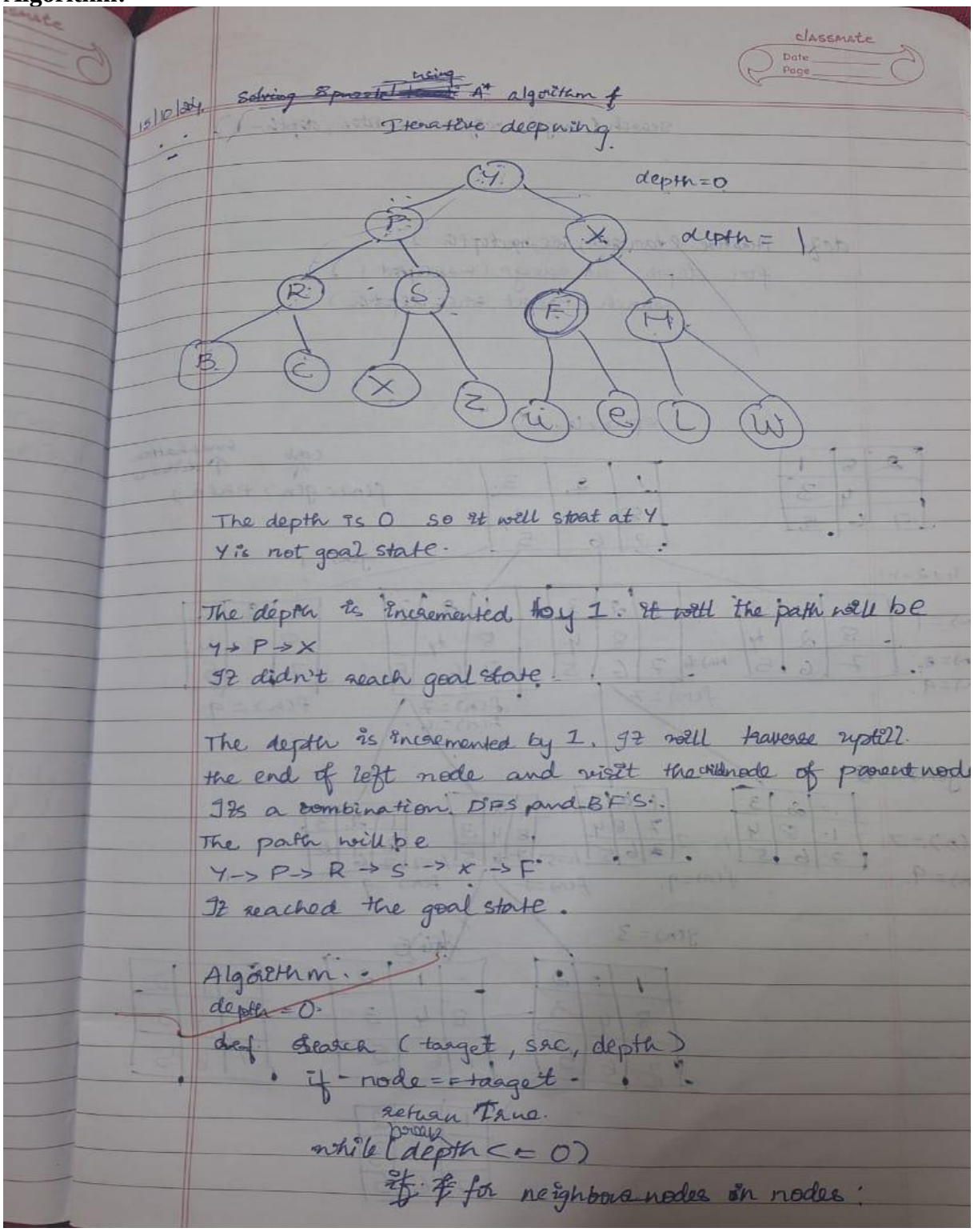
Output:

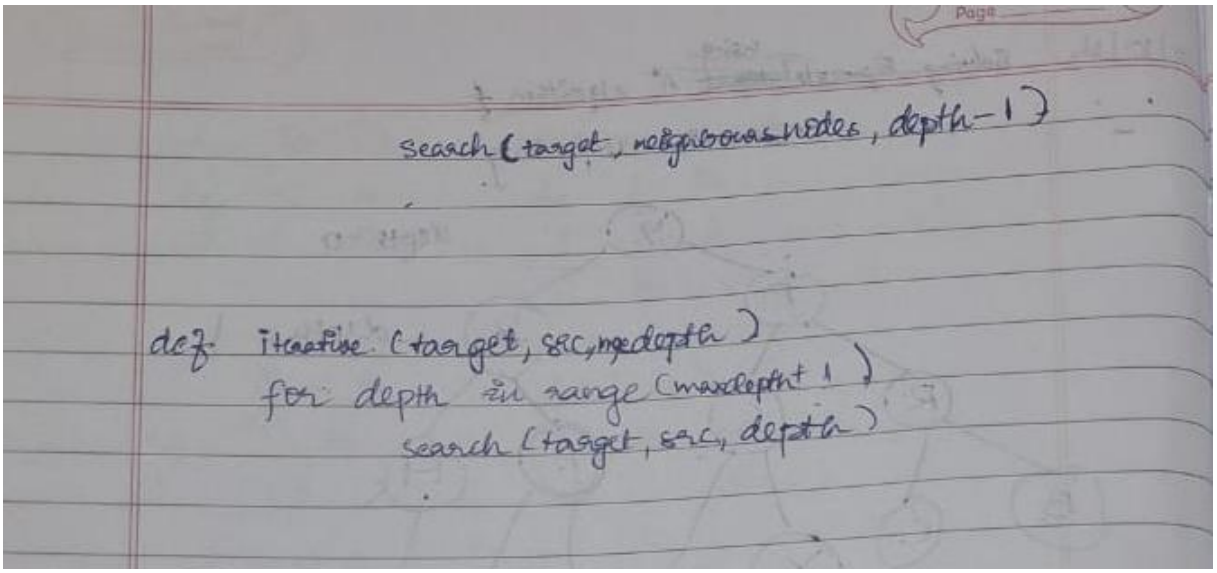
```
Solution found!
4 1 3
7 2 6
5 8

4 1 3
7 2 6
5   8

4 1 3
7 2 6
  5 8

4 1 3
  2 6
7 5 8

  1 3
4 2 6
7 5 8

1   3
4 2 6
7 5 8

1 2 3
4   6
7 5 8

1 2 3
4 5 6
7   8

1 2 3
4 5 6
7 8
```

Program-4

Implement 8 puzzle using Iterative Deepening

Algorithm:





Code:

```
import copy

class Node:
    def __init__(self, state, parent=None, depth=0):
        self.state = state
        self.parent = parent
        self.depth = depth

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

        # Define possible moves
        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')

        # Generate children based on possible actions
        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, 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.state)
        node = node.parent
    path.reverse()
    for state in path:
        for row in state:
            print(row)
        print()

# Test the implementation
initial_state = [[1, 2, 3], [0, 4, 6], [7, 5, 8]]
```

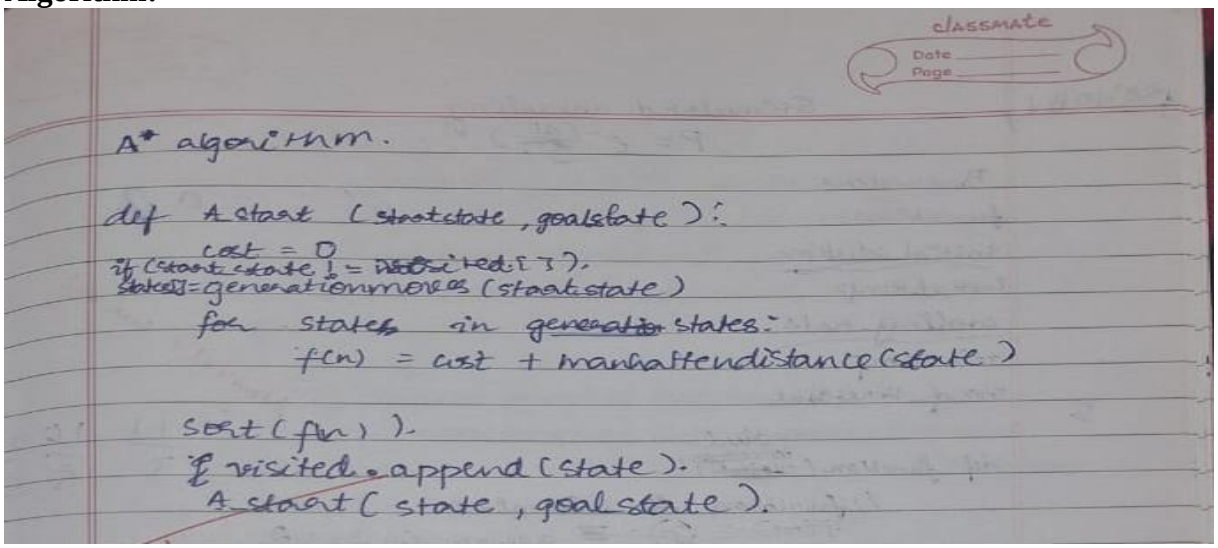
```
goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
max_depth = 5

solution = iterative_deepening_search(initial_state, goal_state, max_depth)
if solution:
    print("Solution found:")
    print_solution(solution)
else:
    print("Solution not found.")
```

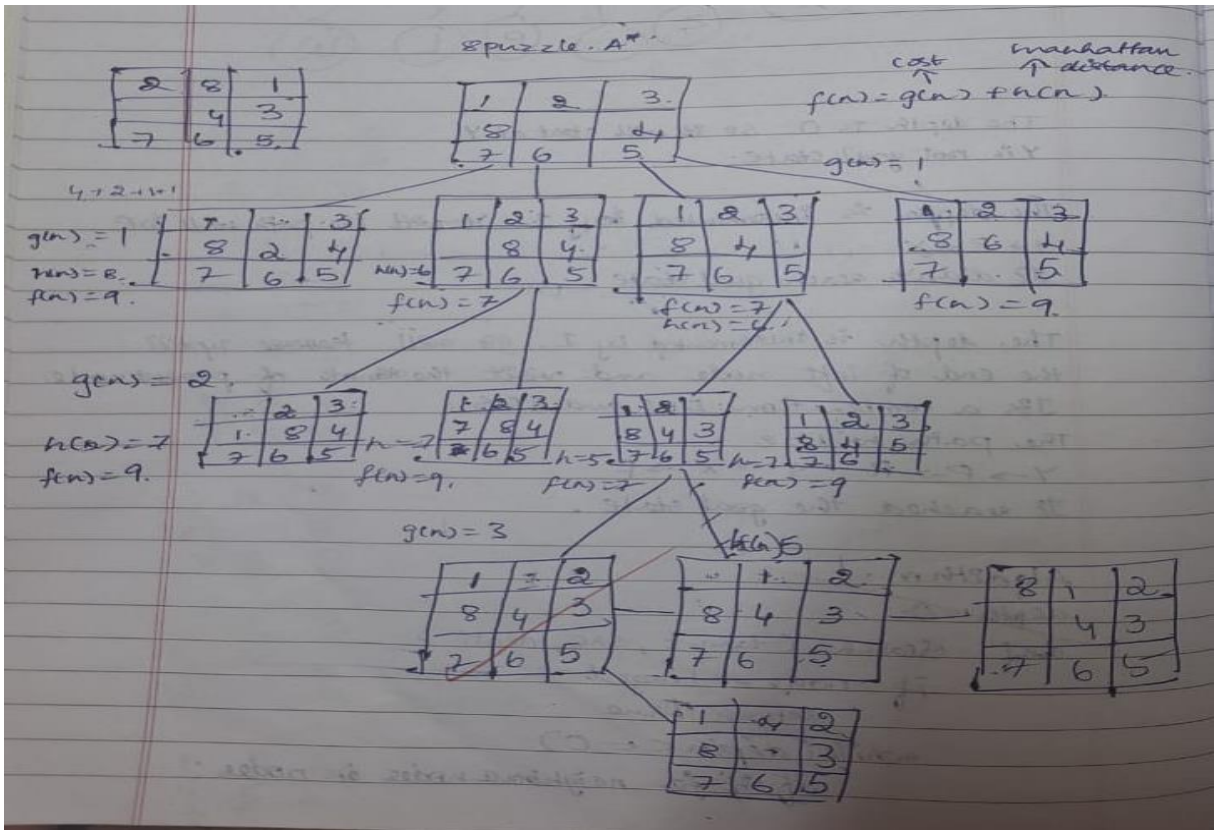
Output:

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

[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]
```

Implement 8 puzzle using A* algorithm

Algorithm:





Code:

```
import heapq

# The goal state of the puzzle
```

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

# Heuristic function: Count the number of misplaced tiles
def misplaced_tiles_heuristic(state):
    misplaced_count = 0
    for row in range(3):
        for col in range(3):
            # Ignore the blank tile (0) and count if the tile is in the wrong position
            if state[row][col] != 0 and state[row][col] != GOAL_STATE[row][col]:
                misplaced_count += 1
    return misplaced_count

# Find the position of the blank tile (represented by 0)
def find_blank_tile(state):
    for row in range(3):
        for col in range(3):
            if state[row][col] == 0:
                return row, col

# Generate all possible valid neighbor states by sliding adjacent tiles
def generate_neighbors(state):
    neighbors = []
    blank_row, blank_col = find_blank_tile(state)  # Get the position of the blank tile
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]  # Directions: right, left, down, up
    for delta_row, delta_col in directions:
        new_row, new_col = blank_row + delta_row, blank_col + delta_col
        # Ensure the new position is within bounds
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            # Create a new state by swapping the blank tile with the adjacent tile
            new_state = [list(row) for row in state]  # Copy the current state
            new_state[blank_row][blank_col], new_state[new_row][new_col] =
new_state[new_row][new_col], new_state[blank_row][blank_col]
            # Append the new state as a tuple of tuples (immutable)
            neighbors.append(tuple(tuple(row) for row in new_state))
    return neighbors

# Reconstruct the path from start state to goal state by backtracking from the goal
def reconstruct_path(came_from, current_state):
    path = [current_state]
    while current_state in came_from:
        current_state = came_from[current_state]
        path.append(current_state)
    path.reverse()  # Reverse the path to show it from start to goal
    return path
```

```
# A* search algorithm to find the solution path
def a_star_search(start_state):
    open_list = []  # List of states to be evaluated
    heapq.heappush(open_list, (0 + misplaced_tiles_heuristic(start_state), 0, start_state))

    g_scores = {start_state: 0}  # g(n): Cost to reach the current state
    came_from = {}  # To track the path
    visited_states = set()  # To avoid revisiting the same state

    while open_list:
        # Pop the state with the lowest f(n) = g(n) + h(n)
        _, g, current_state = heapq.heappop(open_list)

        # If we reach the goal state, reconstruct the path
        if current_state == GOAL_STATE:
            path = reconstruct_path(came_from, current_state)
            return path, g

        visited_states.add(current_state)

        # Evaluate neighbors (possible next states)
        for neighbor in generate_neighbors(current_state):
            if neighbor in visited_states:
                continue  # Skip already visited states

            tentative_g = g_scores[current_state] + 1  # The cost of moving to the neighbor

            # If this neighbor is found with a lower g(n) value, update the path
            if tentative_g < g_scores.get(neighbor, float('inf')):
                came_from[neighbor] = current_state
                g_scores[neighbor] = tentative_g
                f_score = tentative_g + misplaced_tiles_heuristic(neighbor)  # f(n) = g(n) + h(n)
                heapq.heappush(open_list, (f_score, tentative_g, neighbor))

    return None, None  # If no solution is found

# Function to print the state of the puzzle in a readable format
def print_state(state):
    for row in state:
        print(row)
    print()

# Main function
if __name__ == "__main__":
    # Starting state of the puzzle
    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)

    # Perform A* search to find the solution path
    solution_path, solution_cost = a_star_search(start_state)

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

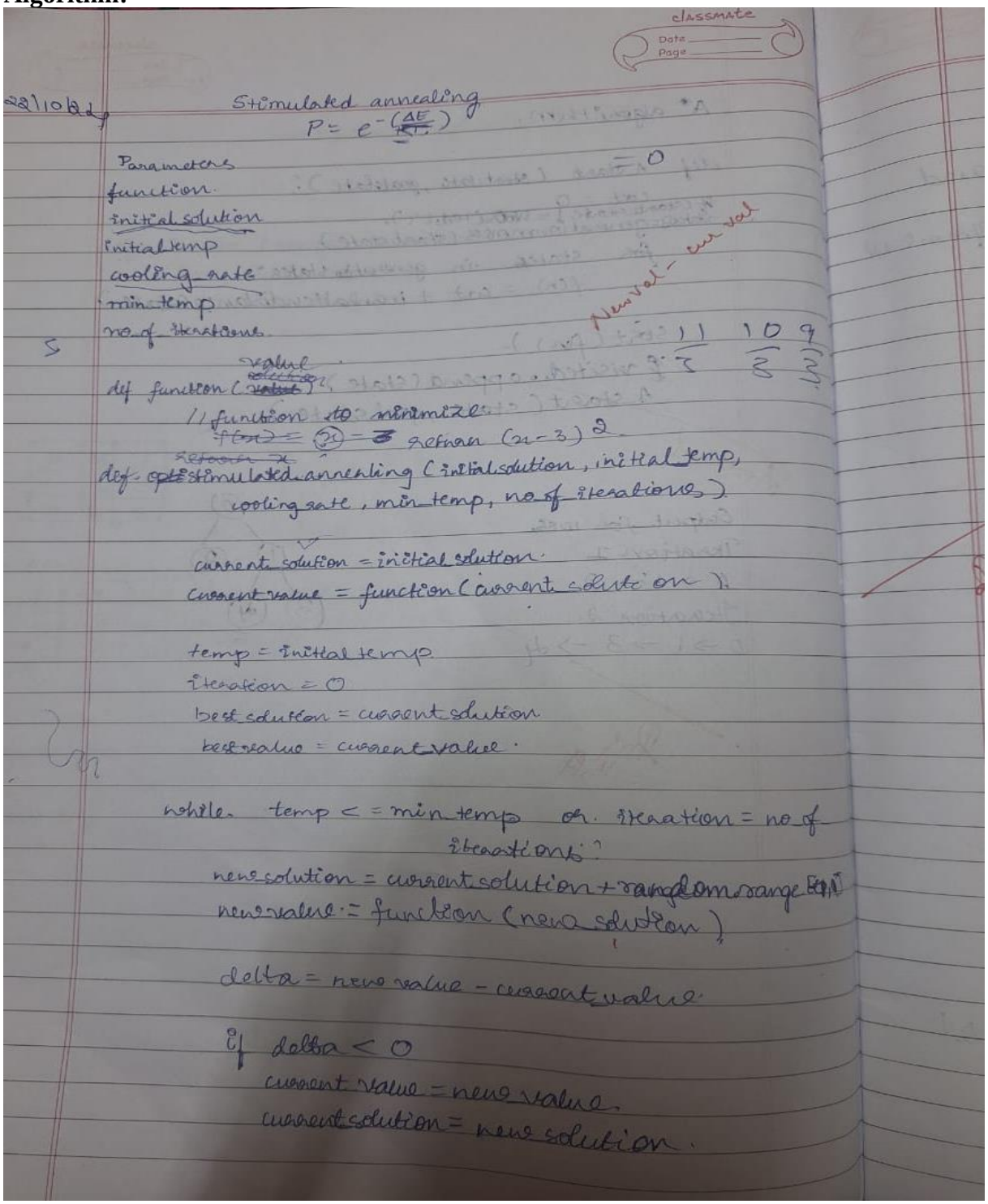
Output:

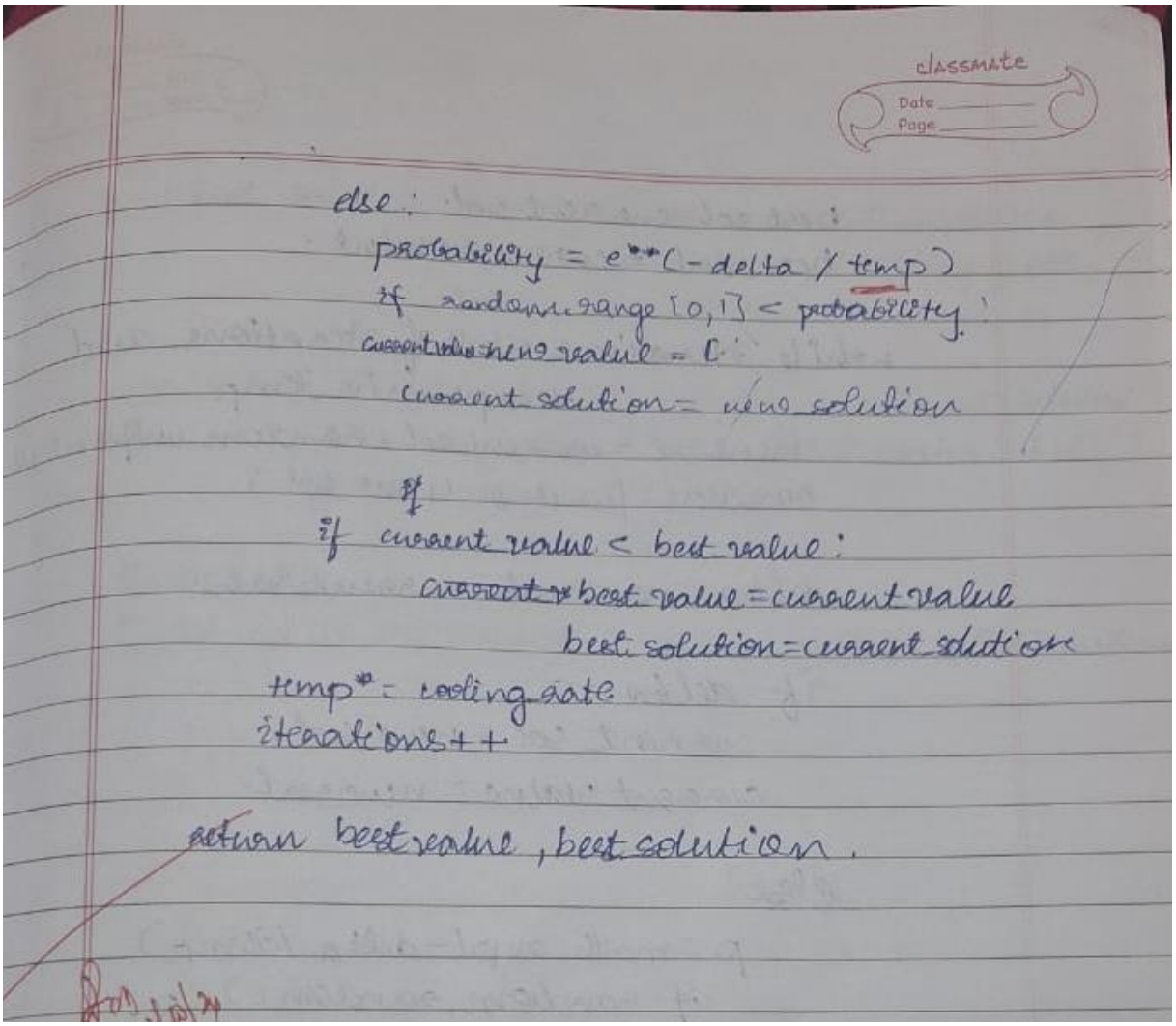
```
Initial State:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

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

Goal state achieved with cost: 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-5

Implement Simulated Annealing

Algorithm:





Code:

```
import random
import math

# Function to calculate the number of conflicts on the board
def calculate_conflicts(board):
    n = len(board)
    conflicts = 0
    for i in range(n):
        for j in range(i + 1, n):  # Compare queens at positions i and j
            # Check if queens are in the same column (board[i] == board[j])
            # or if they are in the same diagonal (abs(board[i] - board[j]) == abs(i - j))
            if board[i] == board[j] or abs(board[i] - board[j]) == abs(i - j):
                conflicts += 1
    return conflicts

# Function to perform simulated annealing
def simulated_annealing(board, max_steps=5000, initial_temp=200, cooling_rate=0.995):
```

```
    current_conflicts = calculate_conflicts(board)
    temperature = initial_temp
    n = len(board)

    for step in range(max_steps):
        if current_conflicts == 0:
            # Solution found (no conflicts)
            return board

        # Select a random position to modify (choose a column to move the queen)
        col = random.randint(0, n-1)
        new_row = random.randint(0, n-1)

        # Generate a new state (new board configuration)
        new_board = board[:]
        new_board[col] = new_row

        # Calculate the number of conflicts in the new configuration
        new_conflicts = calculate_conflicts(new_board)

        # Calculate change in energy (number of conflicts)
        delta = new_conflicts - current_conflicts

        # Accept the new state based on probability
        if delta < 0 or random.uniform(0, 1) < math.exp(-delta / temperature):
            board = new_board
            current_conflicts = new_conflicts

        # Cool down the temperature
        temperature *= cooling_rate

    return None  # No solution found within max_steps

# Main function
def main():
    n = int(input("Enter the number of queens (default is 8): ") or 8)
    print(f"Enter the positions of the queens as an array of size {n}:")
    print(f"(Example: 0,4,7,5,2,6,1,3 or space-separated values)")
    input_str = input().strip()

    # Preprocess the input to handle both comma-separated and space-separated values
    if ',' in input_str:
        board = list(map(int, input_str.split(',')))
    else:
        board = list(map(int, input_str.split()))

    if len(board) != n:
```

```
        print("Error: The number of positions must match the number of queens.")
        return

    # Attempt multiple restarts
    for attempt in range(5):  # Up to 5 attempts
        solution = simulated_annealing(board)
        if solution:
            print(f"\nSolution found on attempt {attempt + 1}:")
            for row in range(n):
                line = ['.'] * n
                line[solution[row]] = 'Q'
                print(" ".join(line))
            return

    print("No solution found after multiple attempts.")

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

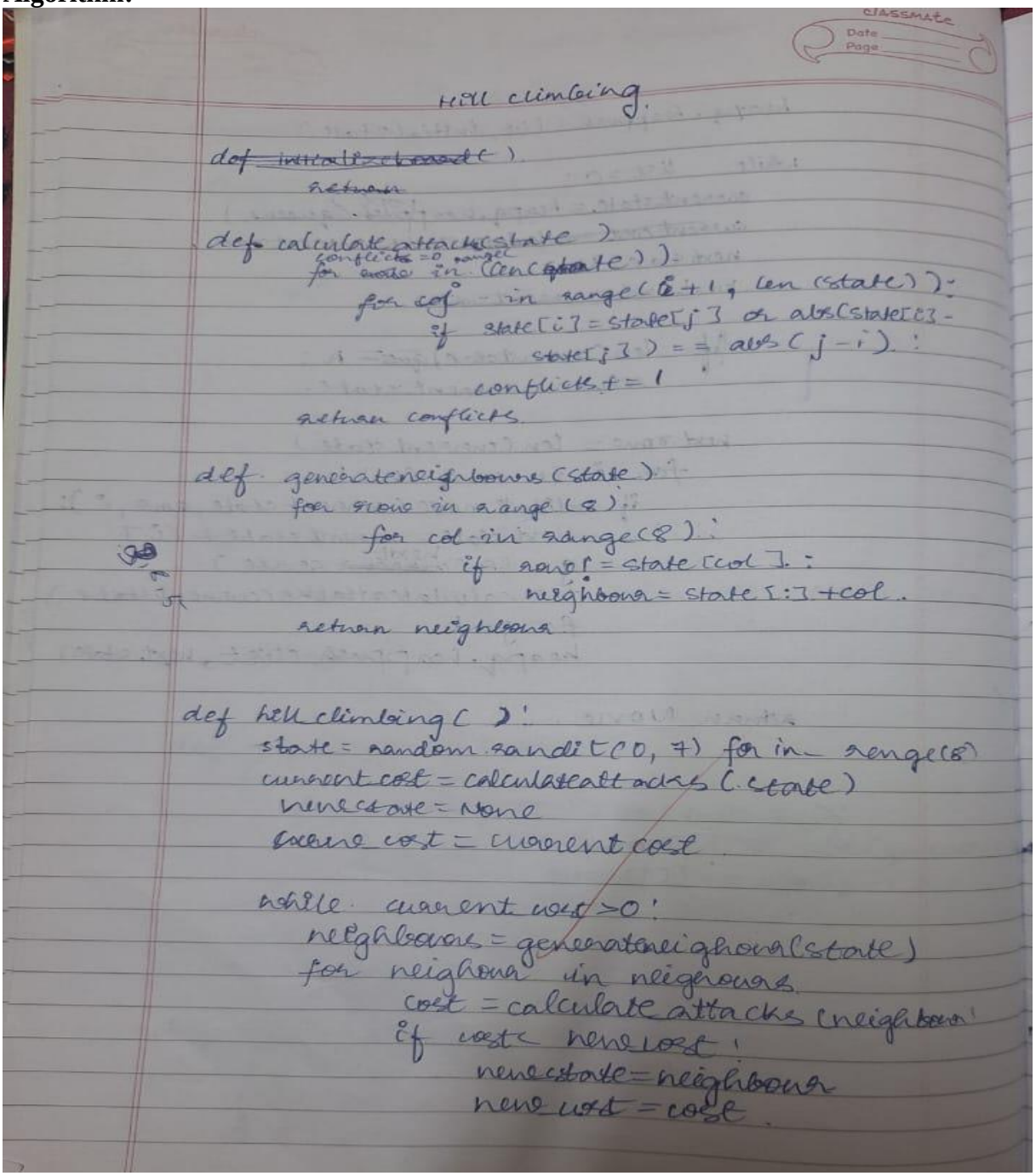
Output:

```
Enter the number of queens (default is 8): 8
Enter the positions of the queens as an array of size 8:
(Example: 0,4,7,5,2,6,1,3 or space-separated values)
0,3,2,1,6,7,4,5

Solution found on attempt 1:
. . . Q . . . .
. . . . . . Q .
Q . . . . . . .
. . . . . . . Q
. . . . Q . . .
. Q . . . . . .
. . . . . Q . .
. . Q . . . . .
```

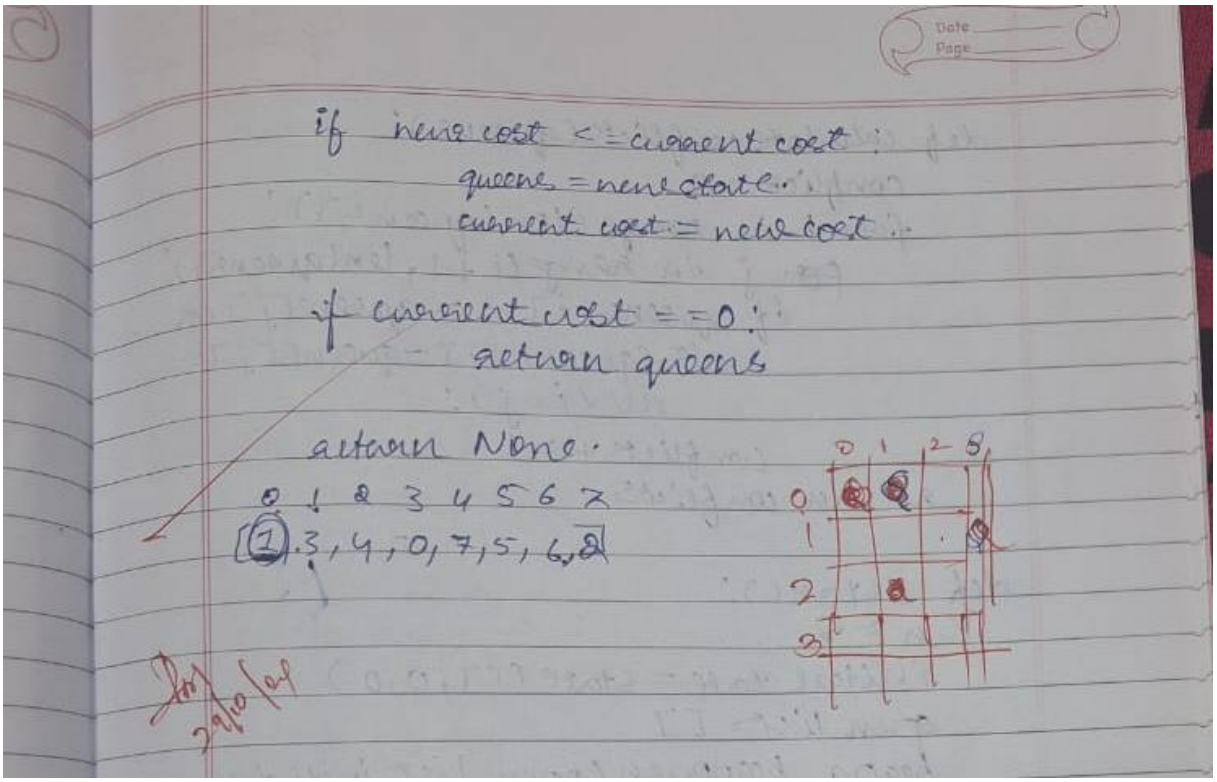
Program-6

Implement N Queens Using Hill Climbing

Algorithm:



```
Hill climbing.

def ~~initialize_board~~():
    return

def calculate_attacks(state):
    conflicts = 0
    for state in (enumerate(state)):
        for col in range(i+1, len(state)):
            if state[i] = state[j] or abs(state[i] -
                state[j]) == abs(j - i):
                conflicts += 1
    return conflicts

def generateneighbours(state):
    for row in range(8):
        for col in range(8):
            if row != state[col]:
                neighbour = state[:] + col.
    return neighbours

def hill_climbing():
    state = random.sandi[(0, 7) for in range(8)
    current_cost = calculateattacks(state)
    neweste = None
    newcost = current cost

    while current cost > 0:
        neighbours = generateneighbour(state)
        for neighbour in neighbours:
            cost = calculate attacks(neighbour)
            if cost < newcost:
                neweste = neighbour
                new cost = cost.
```



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
```

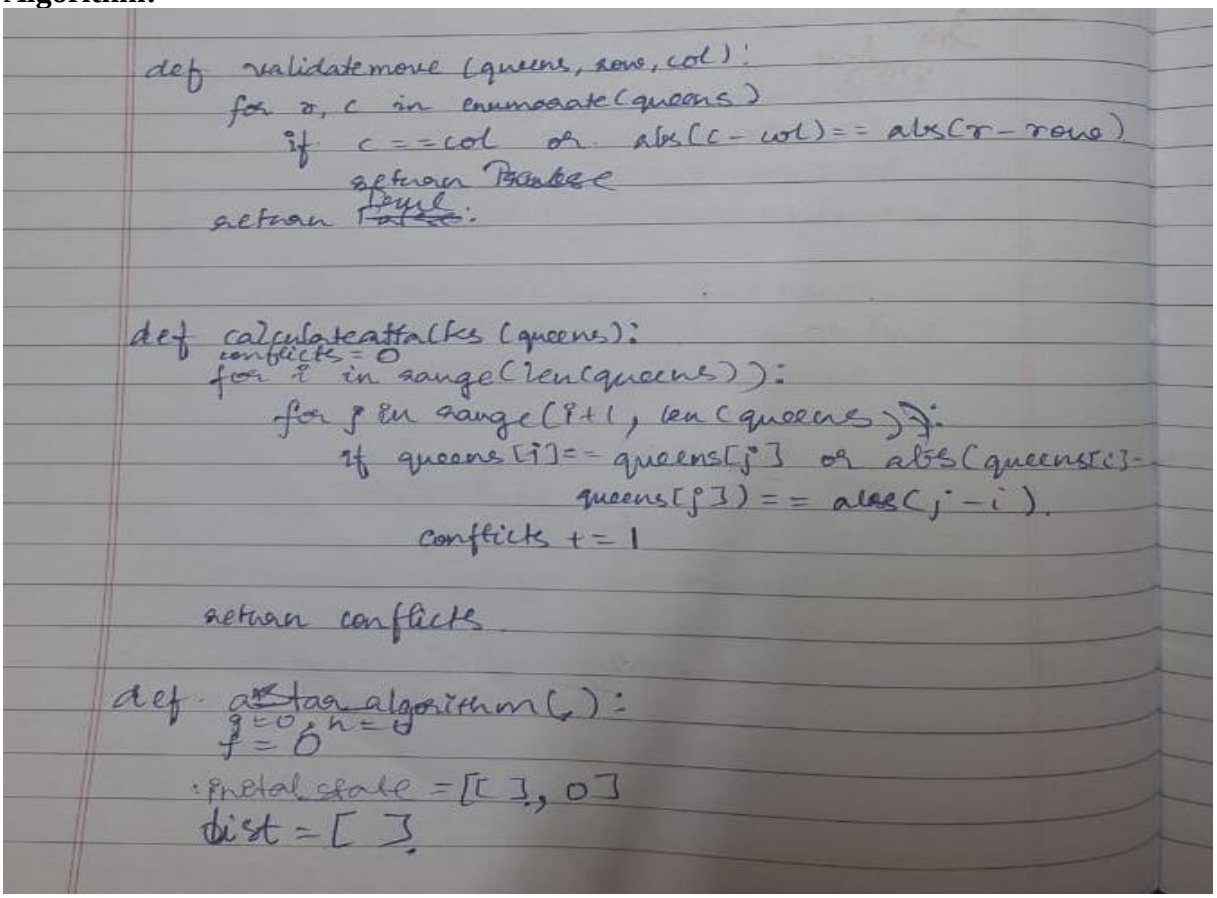
```
def hill_climbing(n, max_iterations=1000):
    """Perform hill climbing search to solve the N-Queens problem."""
    current_state = [random.randint(0, n-1) for _ in range(n)]
    current_cost = calculate_cost(current_state)

    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}")
```

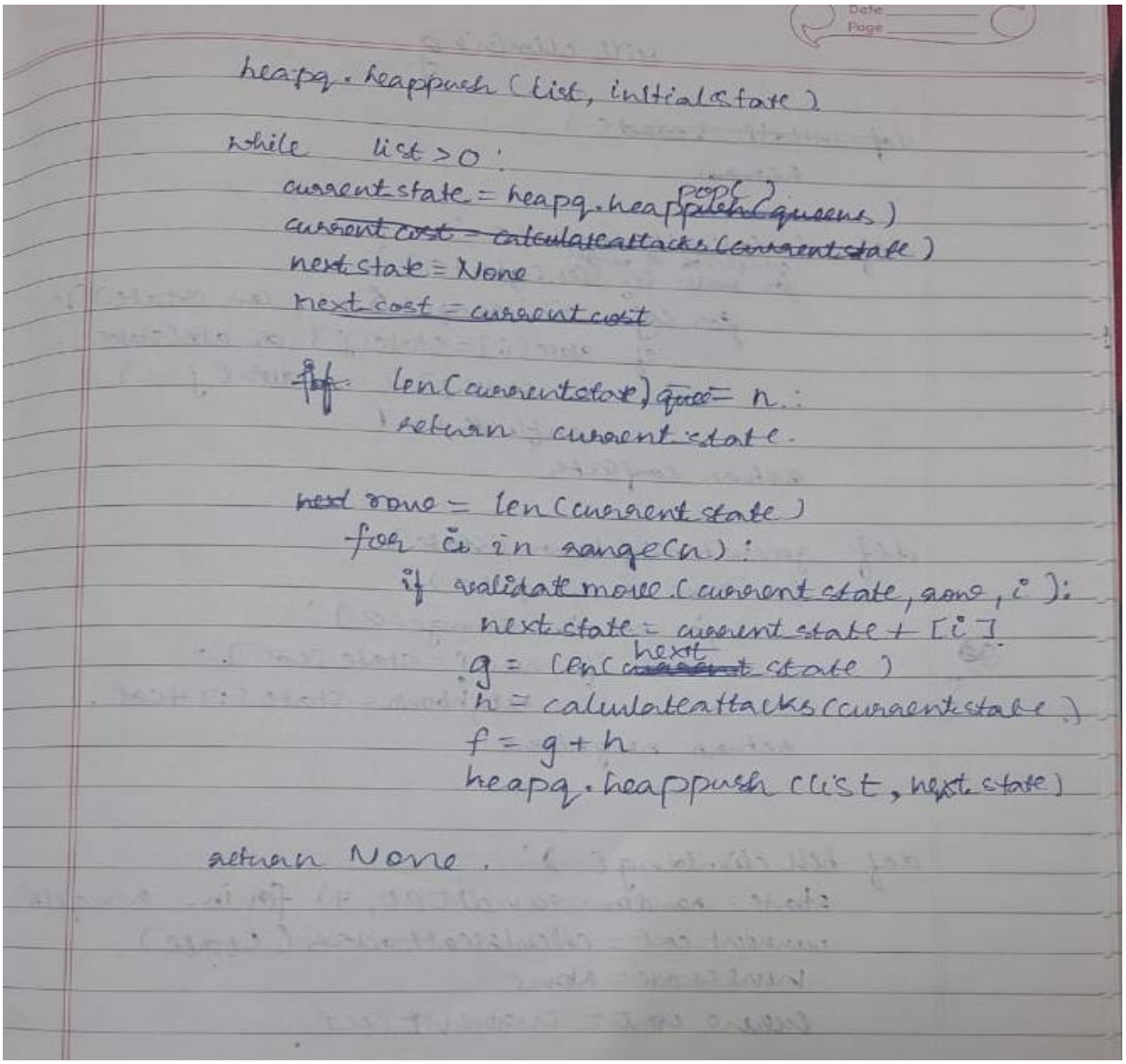
Output:

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

Implement N queen using A* algorithm

Algorithm:





Code:

```
import heapq

# Helper function to calculate the heuristic (number of conflicts)
def heuristic(board):
    conflicts = 0
    for i in range(len(board)):
        for j in range(i + 1, len(board)):
            if board[i] == board[j] or abs(board[i] - board[j]) == j - i:
                conflicts += 1
    return conflicts

# A* Search for 8-queens
def a_star_8_queens():
    n = 8
```

```
    open_set = []
    # Initial state: empty board
    heapq.heappush(open_set, (0, []))  # (f, board)

    while open_set:
        f, board = heapq.heappop(open_set)

        # Goal check
        if len(board) == n and heuristic(board) == 0:
            return board

        # Generate successors
        row = len(board)
        for col in range(n):
            new_board = board + [col]
            if heuristic(new_board) == 0:  # No conflicts so far
                g = row + 1
                h = heuristic(new_board)
                heapq.heappush(open_set, (g + h, new_board))

    return None  # No solution found

# Run A* search
solution = a_star_8_queens()
print("Solution board (column positions for each row):", solution)
```

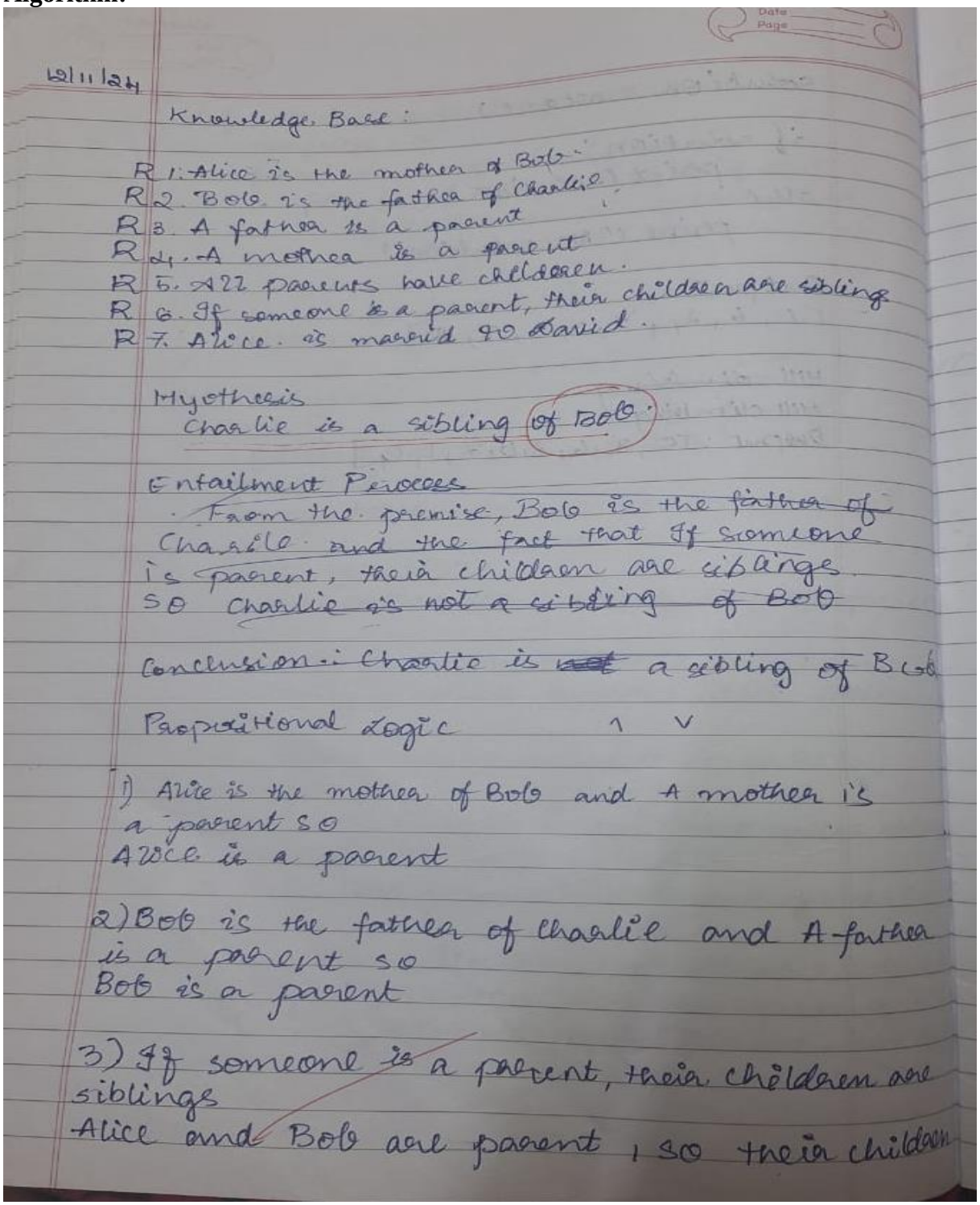
Output:

```
Solution board (column positions for each row): [0, 4, 7, 5, 2, 6, 1, 3]
```

Program-7

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

Algorithm:



12/11/24

Knowledge Base:

R1: Alice is the mother of Bob.
R2. Bob is the father of Charlie.
R3. A father is a parent
R4. A mother is a parent
R5. All parents have childeren.
R6. If someone is a parent, their children are sibling
R7. Alice is maried to David.

Hypothesis
Charlie is a sibling of Bob.

Entailment Process
From the premise, Bob is the father of Charile and the fact that If someone is parent, their children are siblings.
So Charlie is not a sibling of Bob

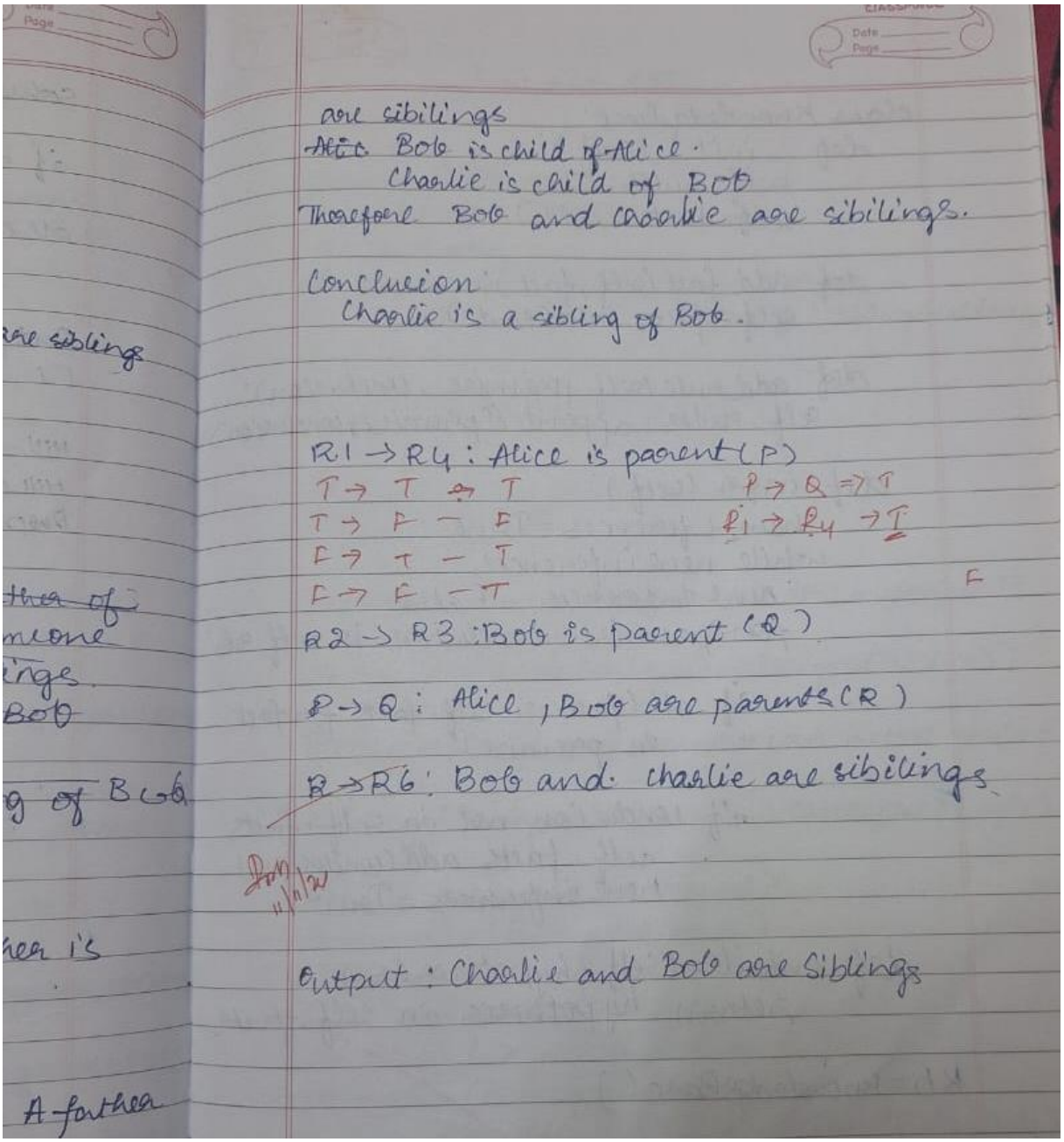
Conclusion: Charlie is ~~not~~ a sibling of Bob

Propositional Logic ∧ ∨

1) Alice is the mother of Bob and A mother is a parent so
Alice is a parent

2) Bob is the father of Charlie and A father is a parent so
Bob is a parent

3) If someone is a parent, their children are siblings
Alice and Bob are parent, so their children



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)
    return False

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 = list(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.")
```

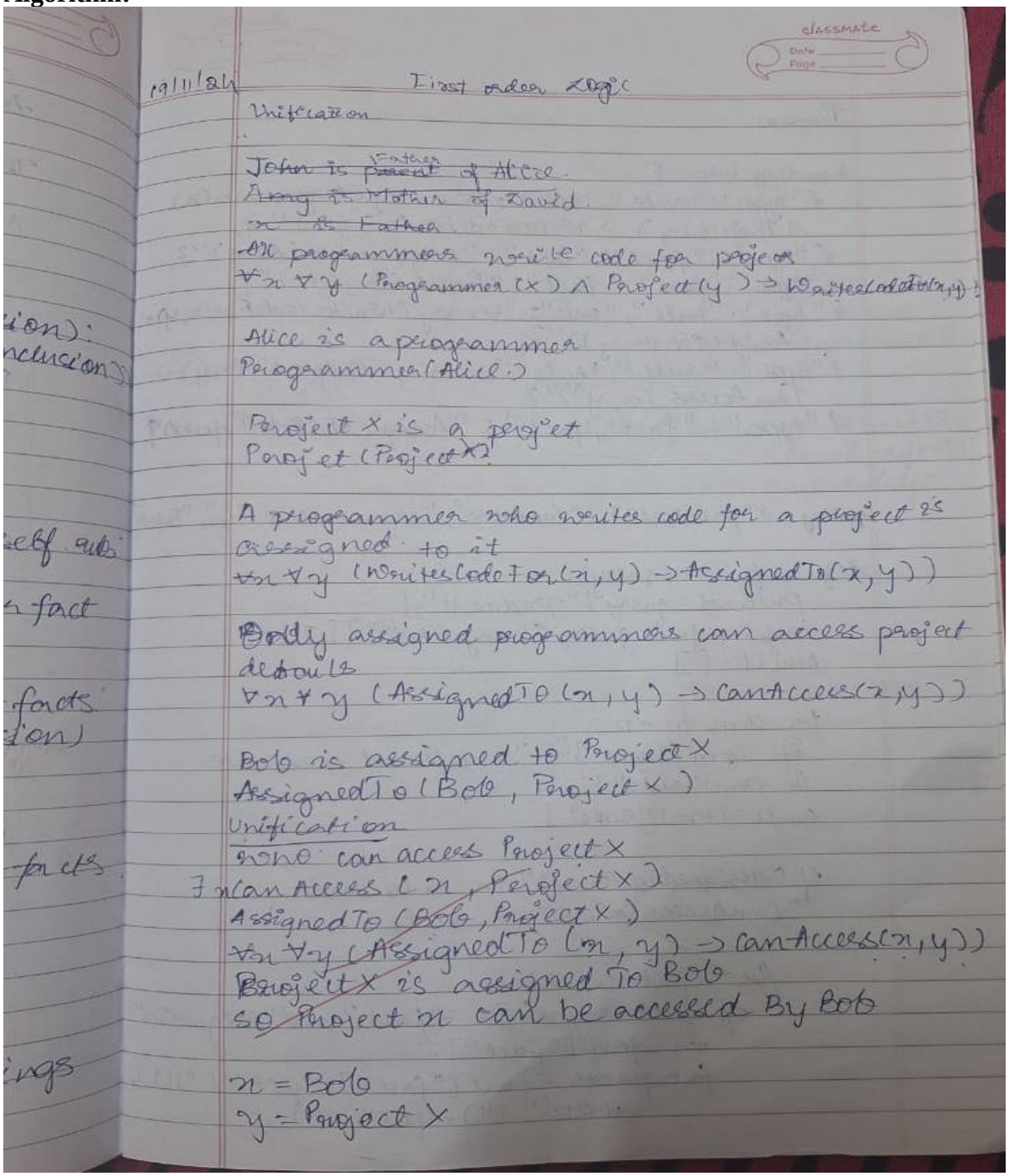
Output:

```
A | B | C | AVC | BV¬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-8

Implement unification in first order logic.

Algorithm:



Code:

```
def is_variable(x):
    """Check if x is a variable."""
    return isinstance(x, str) and x[0].islower()

def unify(x, y, subst):
    """Unify two terms x and y under a given substitution subst."""
    print(f"Comparing: {x} with {y}")
    if subst is None:
        return None
    elif x == y:
        print(f"Both are equal: {x} == {y}")
        return subst
    elif is_variable(x):
        return unify_variable(x, y, subst)
    elif is_variable(y):
        return unify_variable(y, x, subst)
    elif isinstance(x, tuple) and isinstance(y, tuple) and len(x) == len(y):
        for xi, yi in zip(x, y):
            subst = unify(xi, yi, subst)
            if subst is None:
                return None
        return subst
    else:
        print(f"Cannot unify {x} and {y}")
        return None

def unify_variable(var, x, subst):
    """Handle variable unification."""
    if var in subst:
        print(f"Variable {var} is already in substitution. Resolving with {subst[var]}.")
        return unify(subst[var], x, subst)
    elif occurs_check(var, x, subst):
        print(f"Occurs check failed: {var} occurs in {x}.")
        return None  # Avoid infinite loops in recursive substitutions
    else:
        print(f"Adding substitution: {var} -> {x}")
        new_subst = subst.copy()
        new_subst[var] = x
        return new_subst

def occurs_check(var, x, subst):
    """Check if var occurs in x to avoid infinite substitution."""
    if var == x:
        return True
    elif isinstance(x, tuple):
        return any(occurs_check(var, xi, subst) for xi in x)
```

```
    elif is_variable(x) and x in subst:
        return occurs_check(var, subst[x], subst)
    else:
        return False

def parse_sentence_to_expression(sentence):
    """Convert an English sentence to a logical expression."""
    sentence = sentence.strip().replace("(", " ( ").replace(")", " ) ").replace(",", " , ")
    tokens = sentence.split()
    stack = []
    current = []
    for token in tokens:
        if token == "(":
            stack.append(current)
            current = []
        elif token == ")":
            if stack:
                last = stack.pop()
                last.append(tuple(current))
                current = last
        elif token == ",":
            continue
        else:
            current.append(token)
    return tuple(current) if len(current) == 1 else tuple(current)

def unification_with_explanation(expr1, expr2):
    """Perform unification on two expressions with step-by-step explanation."""
    print("\nStarting Unification Process...\n")
    subst = unify(expr1, expr2, {})
    if subst is not None:
        print("\nUnification Successful!")
        print("Substitution:", subst)
    else:
        print("\nUnification Failed!")

# Input from the user
print("Enter the logical expressions in English-like format.")
print("Example: Eats(x, Apple)")
sentence1 = input("Enter the first expression: ")
sentence2 = input("Enter the second expression: ")

# Parse sentences
expr1 = parse_sentence_to_expression(sentence1)
expr2 = parse_sentence_to_expression(sentence2)

# Perform unification with explanation
```

unification_with_explanation(expr1, expr2)

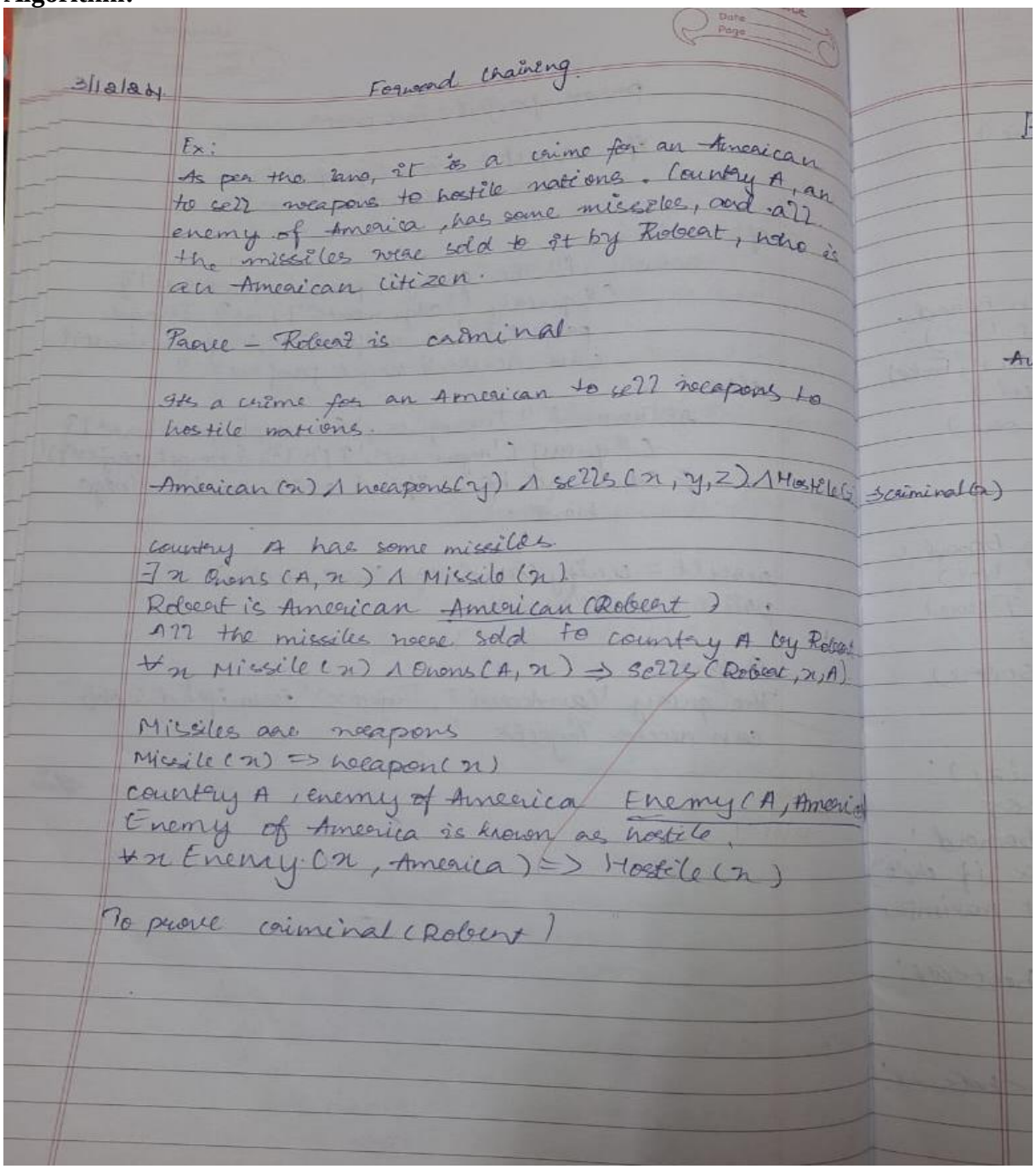
Output:

```
Enter the logical expressions in English-like format.
Example: Eats(x, Apple)
Enter the first expression: Works(x,Google)
Enter the second expression: Works(John,y)

Starting Unification Process...

Comparing: ('Works', ('x', 'Google')) with ('Works', ('John', 'y'))
Comparing: Works with Works
Both are equal: Works == Works
Comparing: ('x', 'Google') with ('John', 'y')
Comparing: x with John
Adding substitution: x -> John
Comparing: Google with y
Adding substitution: y -> Google

Unification Successful!
Substitution: {'x': 'John', 'y': 'Google'}
```

Program-9

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

Algorithm:



Code

```
class ForwardChainingFOL:
    def __init__(self):
        self.facts = set()  # Set of known facts
        self.rules = []  # List of rules in the form (premises, conclusion)

    def add_fact(self, fact):
        self.facts.add(fact)

    def add_rule(self, premises, conclusion):
        self.rules.append((premises, conclusion))

    def unify(self, fact1, fact2):
        """
        Unifies two facts if possible. Returns a substitution dictionary or None if unification fails.
        """
        if fact1 == fact2:
            return {}  # No substitution needed
        if "(" in fact1 and "(" in fact2:
            # Split into predicate and arguments
            pred1, args1 = fact1.split("(", 1)
            pred2, args2 = fact2.split("(", 1)
            args1 = args1[:-1].split(",")
            args2 = args2[:-1].split(",")
            if pred1 != pred2 or len(args1) != len(args2):
                return None
            # Unify arguments
            substitution = {}
            for a1, a2 in zip(args1, args2):
                if a1 != a2:
                    if a1.islower():  # a1 is a variable
                        substitution[a1] = a2
                    elif a2.islower():  # a2 is a variable
                        substitution[a2] = a1
                    else:  # Both are constants and different
                        return None
            return substitution
        return None

    def apply_substitution(self, fact, substitution):
        """
        Applies a substitution to a fact and returns the substituted fact.
        """
        if "(" in fact:
            pred, args = fact.split("(", 1)
            args = args[:-1].split(",")
            substituted_args = [substitution.get(arg, arg) for arg in args]
```

```
        return f"{pred}({','.join(substituted_args)})"
    return fact

def forward_chain(self, goal):
    iteration = 1
    while True:
        new_facts = set()
        print(f"\n=== Iteration {iteration} ===")
        print("Known Facts:")
        for fact in self.facts:
            print(f"- {fact}")

        print("\nApplying rules...")
        rule_triggered = False
        for premises, conclusion in self.rules:
            substitutions = [{}]
            for premise in premises:
                new_substitutions = []
                for fact in self.facts:
                    for sub in substitutions:
                        unified = self.unify(self.apply_substitution(premise, sub), fact)
                        if unified is not None:
                            new_substitutions.append({**sub, **unified})
                substitutions = new_substitutions

            for sub in substitutions:
                inferred_fact = self.apply_substitution(conclusion, sub)
                if inferred_fact not in self.facts:
                    rule_triggered = True
                    print(f"Rule triggered: {premises} → {conclusion}")
                    print(f" New fact inferred: {inferred_fact}")
                    new_facts.add(inferred_fact)

        if not new_facts:
            if not rule_triggered:
                print("No rules triggered in this iteration.")
                print("No new facts inferred in this iteration.")
                break

        self.facts.update(new_facts)
        if goal in self.facts:
            print(f"\nGoal {goal} reached!")
            return True
        iteration += 1

    print("\nGoal not reached.")
    return False
```

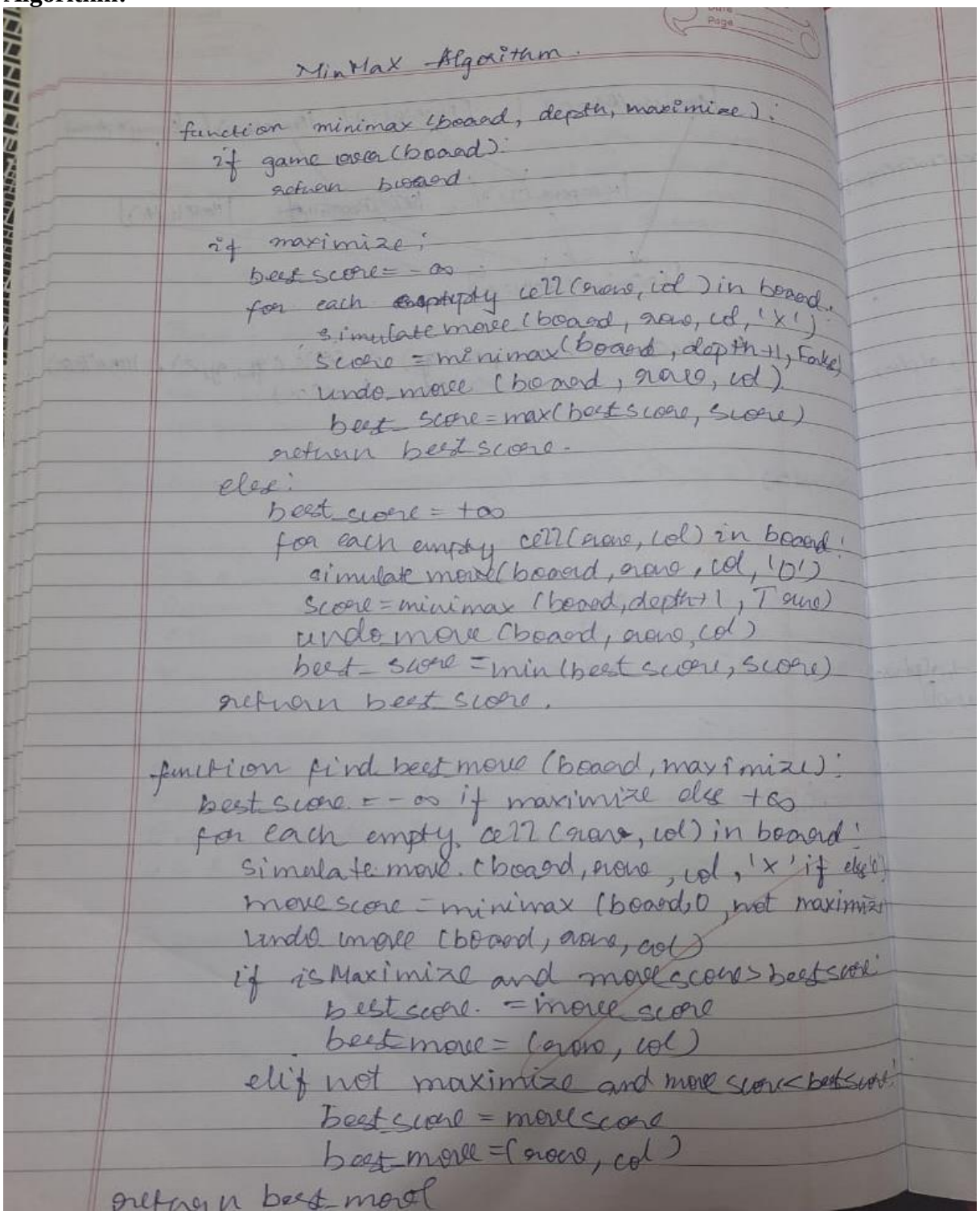
```
# Problem setup
fc = ForwardChainingFOL()

# Facts
fc.add_fact("American(Robert)")
fc.add_fact("Enemy(A,America)")
fc.add_fact("Owns(A,T1)")
fc.add_fact("Missile(T1)")

# Rules
fc.add_rule(["Missile(T1)"], "Weapon(T1)")
fc.add_rule(["Enemy(A,America)"], "Hostile(A)")
fc.add_rule(["Missile(p)", "Owns(A,p)"], "Sells(Robert,p,A)")
fc.add_rule(["American(p)", "Weapon(q)", "Sells(p,q,r)", "Hostile(r)"], "Criminal(p)")

# Goal
goal = "Criminal(Robert)"

# Perform forward chaining
if fc.forward_chain(goal):
    print(f"\nFinal result: Goal achieved: {goal}")
else:
    print("\nFinal result: Goal not achieved.")
```

Output:

```
=== Iteration 1 ===
Known Facts:
- Enemy(A,America)
- Owns(A,T1)
- American(Robert)
- Missile(T1)

Applying rules...
Rule triggered: ['Missile(T1)'] → Weapon(T1)
 New fact inferred: Weapon(T1)
Rule triggered: ['Enemy(A,America)'] → Hostile(A)
 New fact inferred: Hostile(A)
Rule triggered: ['Missile(p)', 'Owns(A,p)'] → Sells(Robert,p,A)
 New fact inferred: Sells(Robert,T1,A)

=== Iteration 2 ===
Known Facts:
- Hostile(A)
- Enemy(A,America)
- Sells(Robert,T1,A)
- Weapon(T1)
- Owns(A,T1)
- American(Robert)
- Missile(T1)

Applying rules...
Rule triggered: ['American(p)', 'Weapon(q)', 'Sells(p,q,r)', 'Hostile(r)'] → Criminal(p)
 New fact inferred: Criminal(Robert)

Goal Criminal(Robert) reached!
```

Implement Minmax Algorithm

Algorithm:



MinMax Algorithm.

```
function minimax(board, depth, maximize):
    if game_over(board):
        return board

    if maximize:
        best_score = -∞
        for each empty cell(row, col) in board:
            simulate_move(board, row, col, 'X')
            score = minimax(board, depth+1, False)
            undo_move(board, row, col)
            best_score = max(best_score, score)
        return best_score
    else:
        best_score = +∞
        for each empty cell(row, col) in board:
            simulate_move(board, row, col, 'O')
            score = minimax(board, depth+1, True)
            undo_move(board, row, col)
            best_score = min(best_score, score)
        return best_score

function find_best_move(board, maximize):
    best_score = -∞ if maximize else +∞
    for each empty cell(row, col) in board:
        simulate_move(board, row, col, 'X' if else 'O')
        move_score = minimax(board, 0, not maximize)
        undo_move(board, row, col)
        if isMaximize and move_score > best_score:
            best_score = move_score
            best_move = (row, col)
        elif not maximize and move_score < best_score:
            best_score = move_score
            best_move = (row, col)
    return best_move
```

Code:

```
import math

# Utility function to check if a player has won (using the 2D board)
def check_winner(board, player):
    # Check rows and columns
    for i in range(3):
        if all([board[i][j] == player for j in range(3)]) or all([board[j][i] == player for j in range(3)]):
            return True

    # Check diagonals
    if (board[0][0] == player and board[1][1] == player and board[2][2] == player) or \
       (board[0][2] == player and board[1][1] == player and board[2][0] == player):
        return True

    return False

# Function to check if the board is full
def is_board_full(board):
    return not any(spot == ' ' for row in board for spot in row)

# Evaluate the board
def evaluate(board):
    if check_winner(board, 'X'):
        return 10
    elif check_winner(board, 'O'):
        return -10
    else:
        return 0

# Minimax algorithm to find the best move
def minimax(board, depth, is_maximizing_player):
    score = evaluate(board)

    # If maximizer wins or minimizer wins
    if score == 10:
        return score - depth
    if score == -10:
        return score + depth

    # If board is full (draw)
    if is_board_full(board):
        return 0

    if is_maximizing_player:
        best = -math.inf
        for i in range(3):
```

```
            for j in range(3):
                if board[i][j] == ' ':
                    board[i][j] = 'X'  # Maximizing player (AI)
                    best = max(best, minimax(board, depth + 1, False))
                    board[i][j] = ' '  # Undo move
        return best
    else:
        best = math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == ' ':
                    board[i][j] = 'O'  # Minimizing player (human)
                    best = min(best, minimax(board, depth + 1, True))
                    board[i][j] = ' '  # Undo move
        return best

# Function to 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] = 'X'  # AI is 'X'
                move_val = minimax(board, 0, False)
                board[i][j] = ' '  # Undo move
                if move_val > best_val:
                    best_move = (i, j)
                    best_val = move_val

    return best_move

# Function to print the current board
def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("--+---+--")

# Main function to run the game
def play_game():
    board = [[' ' for _ in range(3)] for _ in range(3)]  # Initialize a 3x3 board
    current_player = 'O'  # 'O' goes first

    while True:
        print_board(board)
```

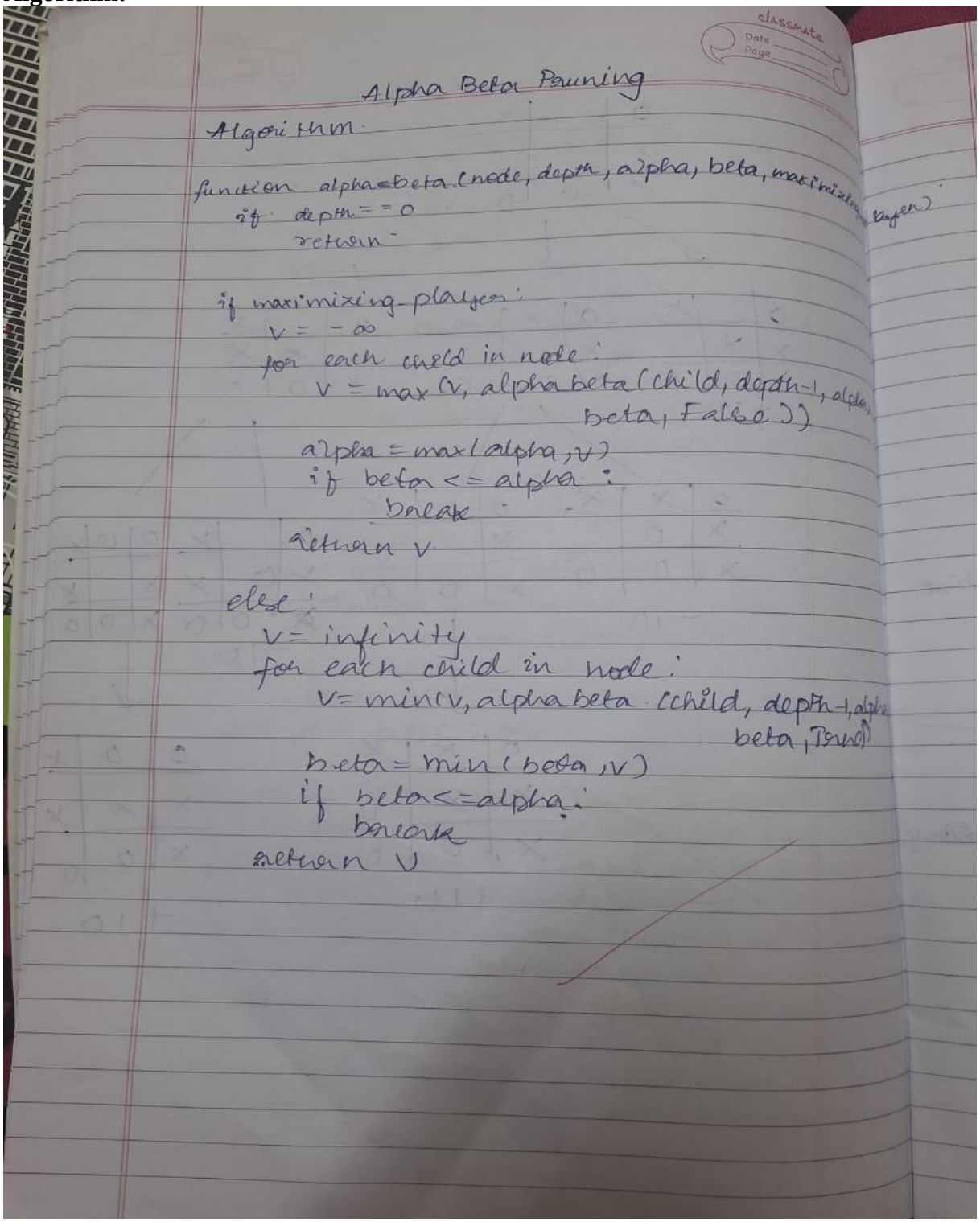
```
        if current_player == 'O':
            # Human move (player 'O')
            try:
                move = int(input("Enter your move (1-9): ")) - 1
                row, col = divmod(move, 3)
                if board[row][col] != ' ':
                    print("Invalid move. Try again.")
                    continue
                board[row][col] = 'O'
                current_player = 'X'
            except (ValueError, IndexError):
                print("Invalid input. Please enter a number between 1 and 9.")
        else:
            # AI move (player 'X')
            print("AI is making its move...")
            row, col = find_best_move(board)
            board[row][col] = 'X'
            current_player = 'O'

        # Check if someone has won
        if check_winner(board, 'X'):
            print_board(board)
            print("AI wins!")
            break
        elif check_winner(board, 'O'):
            print_board(board)
            print("You win!")
            break

        # Check if board is full (draw)
        if is_board_full(board):
            print_board(board)
            print("It's a draw!")
            break

# Start the game
play_game()
```

Output:

```
Enter your move (1-9): 1
O |   |
--+---+--
  |   |
--+---+--
  |   |
--+---+--
AI is making its move...
O |   |
--+---+--
  | X |
--+---+--
  |   |
--+---+--
Enter your move (1-9): 2
O | O |
--+---+--
  | X |
--+---+--
  |   |
--+---+--
AI is making its move...
O | O | X
--+---+--
  | X |
--+---+--
  |   |
--+---+--
Enter your move (1-9): 4
O | O | X
--+---+--
O | X |
--+---+--
  |   |
--+---+--
AI is making its move...
O | O | X
--+---+--
O | X |
--+---+--
X |   |
--+---+--
AI wins!
```

Implement Alpha-beta Pruning

Algorithm:



Code:

```
import math

# Alpha-Beta Pruning Algorithm
def alpha_beta_search(depth, index, is_max, values, alpha, beta, target_depth, branch_factor):
    """Recursive function for Alpha-Beta Pruning."""
    # Base case: If the target depth is reached, return the leaf node value
    if depth == target_depth:
        return values[index]

    if is_max:
        # Maximizer's turn
        best = -math.inf
        for i in range(branch_factor):  # Iterate over all children
            val = alpha_beta_search(depth + 1, index * branch_factor + i, False, values, alpha, beta,
target_depth, branch_factor)
            best = max(best, val)
            alpha = max(alpha, best)
            if beta <= alpha:
                break  # Prune remaining branches
        return best
    else:
        # Minimizer's turn
        best = math.inf
        for i in range(branch_factor):  # Iterate over all children
            val = alpha_beta_search(depth + 1, index * branch_factor + i, True, values, alpha, beta,
target_depth, branch_factor)
            best = min(best, val)
            beta = min(beta, best)
            if beta <= alpha:
                break  # Prune remaining branches
        return best

def main():
    # User Input: Values of leaf nodes
    print("Enter the values of leaf nodes separated by spaces:")
    values = list(map(int, input().split()))

    # User Input: Number of branches per node
    print("Enter the branching factor (number of children per node):")
    branch_factor = int(input())

    # Calculate depth of the game tree
    target_depth = math.log(len(values), branch_factor)
    if target_depth != int(target_depth):
        print(f"Error: The number of leaf nodes must be a power of {branch_factor}.")
        return
```

```
    target_depth = int(target_depth)

    # Run Alpha-Beta Pruning
    result = alpha_beta_search(0, 0, True, values, -math.inf, math.inf, target_depth, branch_factor)

    # Display the result
    print(f"The optimal value determined by Alpha-Beta Pruning is: {result}")

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

Output:

```
Enter the values of leaf nodes separated by spaces:
25 8 6 4 5 10 9 11 13
Enter the branching factor (number of children per node):
3
The optimal value determined by Alpha-Beta Pruning is: 9
```