Artificial Intelligence Lab Report

Submitted by
Sparsha Srinath Kadaba (1BM22CS287)
Batch: F1

Course: Artificial Intelligence
Course Code: 23CS5PCAIN
Sem & Section: 5F

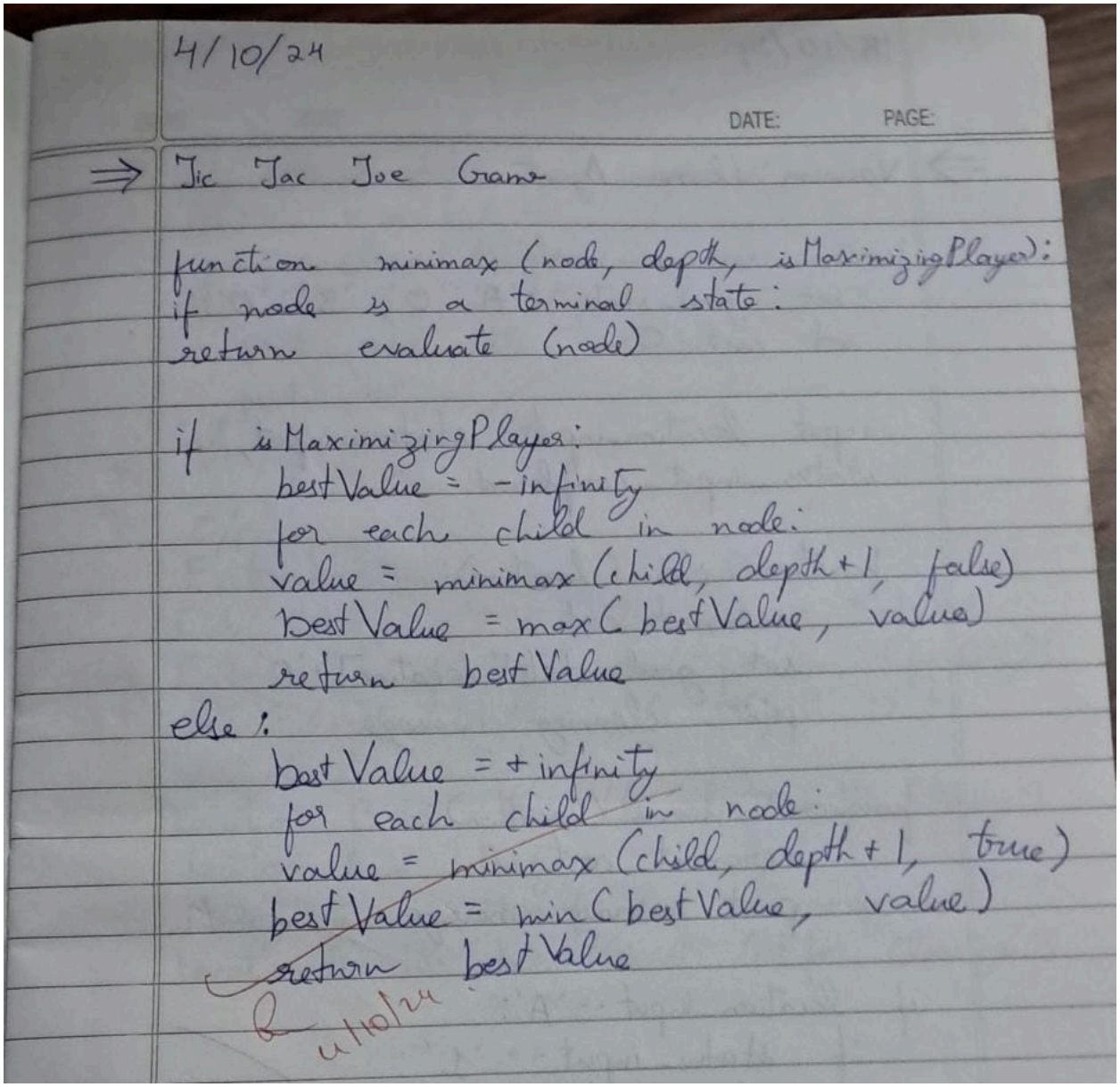
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B. M. S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
2022-2023

Table of Contents

Program 1
Implement Tic –Tac –Toe Game.

Algorithm:


4/10/24

DATE: PAGE:

⇒ Tic Tac Toe Game

function minimax (node, depth, isMaximizingPlayer):
if node is a terminal state:
return evaluate (node)

if isMaximizingPlayer:
bestValue = -infinity
for each child in node:
value = minimax (child, depth+1, false)
bestValue = max (bestValue, value)
return bestValue
else:
bestValue = +infinity
for each child in node:
value = minimax (child, depth+1, true)
bestValue = min (bestValue, value)
return bestValue

4/10/24

Code:

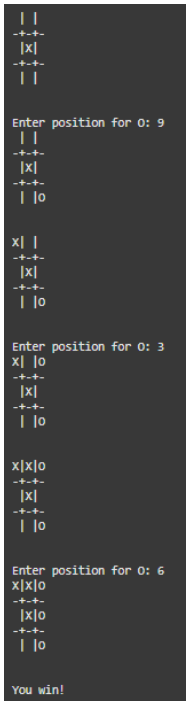
```
#Tic Tac Toe Game
board = {
    1: ' ', 2: ' ', 3: ' ',
    4: ' ', 5: ' ', 6: ' ',
    7: ' ', 8: ' ', 9: ' '
```

```
}

def printBoard(board):
    print(board[1] + '|' + board[2] + '|' + board[3])
    print('-+-+-')
    print(board[4] + '|' + board[5] + '|' + board[6])
    print('-+-+-')
    print(board[7] + '|' + board[8] + '|' + board[9])
    print('\n')

def spaceFree(pos):
    return board[pos] == ' '

def checkWin():
    winning_combinations = [
        (1, 2, 3), (4, 5, 6), (7, 8, 9),  # rows
        (1, 4, 7), (2, 5, 8), (3, 6, 9),  # columns
        (1, 5, 9), (3, 5, 7)              # diagonals
    ]
    for a, b, c in winning_combinations:
        if board[a] == board[b] == board[c] and board[a] != ' ':
            return True
    return False

def checkDraw():
    return all(board[key] != ' ' for key in board.keys())

def insertLetter(letter, position):
    if spaceFree(position):
        board[position] = letter
        printBoard(board)

        if checkDraw():
            print('Draw!')
            return True  # End the game
        elif checkWin():
            if letter == 'X':
                print('Bot wins!')
            else:
                print('You win!')
```

```
            return True  # End the game
        else:
            print('Position taken, please pick a different position.')
            position = int(input('Enter new position: '))
            return insertLetter(letter, position)

        return False  # Game continues

player = 'O'
bot = 'X'

def playerMove():
    position = int(input('Enter position for O: '))
    return insertLetter(player, position)

def compMove():
    bestScore = -1000
    bestMove = 0
    for key in board.keys():
        if board[key] == ' ':
            board[key] = bot
            score = minimax(board, False)
            board[key] = ' '
            if score > bestScore:
                bestScore = score
                bestMove = key

    return insertLetter(bot, bestMove)

def minimax(board, isMaximizing):
    if checkWin():
        return 1 if isMaximizing else -1
    elif checkDraw():
        return 0

    if isMaximizing:
        bestScore = -1000
        for key in board.keys():
            if board[key] == ' ':
                board[key] = bot
```

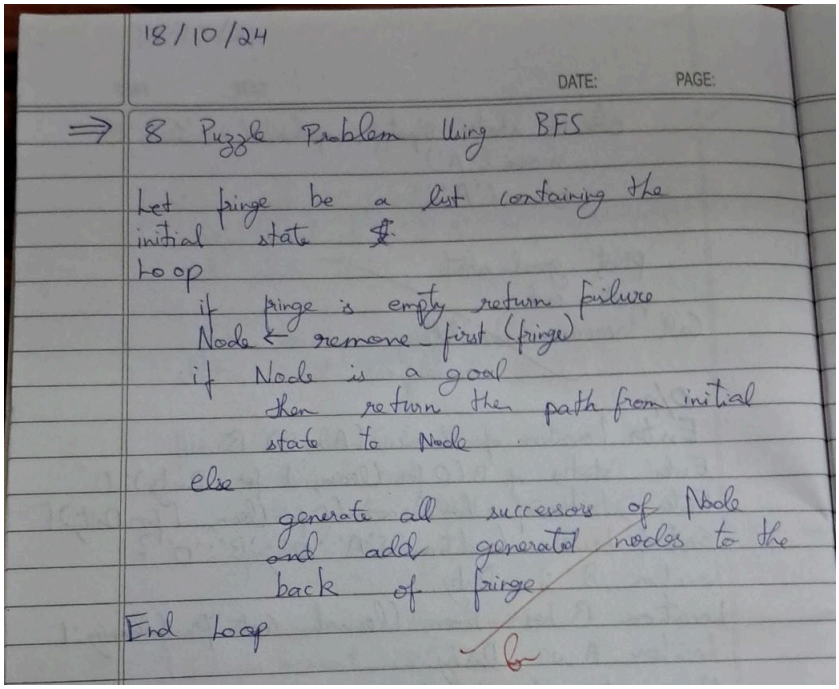
```
            score = minimax(board, False)
            board[key] = ' '
            bestScore = max(score, bestScore)
        return bestScore
    else:
        bestScore = 1000
        for key in board.keys():
            if board[key] == ' ':
                board[key] = player
                score = minimax(board, True)
                board[key] = ' '
                bestScore = min(score, bestScore)
        return bestScore

# Main game loop
game_over = False
while not game_over:
    compMove()
    game_over = playerMove()

print("Sparsha Srinath Kadaba - 1BM22CS287")
```

Output:



Program 2
Solve 8 puzzle problems.

Algorithm:


18/10/24

DATE: PAGE:

⇒ 8 Puzzle Problem Using BFS

Let fringe be a list containing the initial state

Loop

if fringe is empty return failure

Node ← remove-first (fringe)

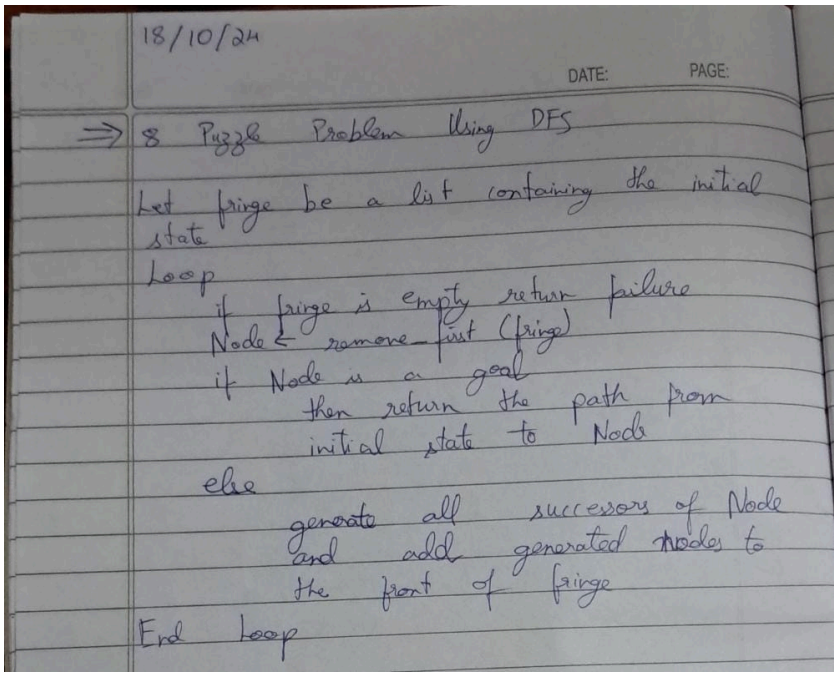
if Node is a goal

then return the path from initial state to Node

else

generate all successors of Node and add generated nodes to the back of fringe

End Loop


18/10/24

DATE: PAGE:

⇒ 8 Puzzle Problem Using DFS

Let fringe be a list containing the initial state

Loop

if fringe is empty return failure

Node ← remove-first (fringe)

if Node is a goal

then return the path from initial state to Node

else

generate all successors of Node and add generated nodes to the front of fringe

End Loop

Code:

```
#8 puzzle problem using bfs
from collections import deque
```

```
class PuzzleState:
    def __init__(self, board, zero_position, path=[]):
        self.board = board
        self.zero_position = zero_position
        self.path = path

    def is_goal(self):
        return self.board == [1, 2, 3, 4, 5, 6, 7, 8, 0]

    def get_possible_moves(self):
        moves = []
        row, col = self.zero_position
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # Right, Down, Left, Up

        for dr, dc in directions:
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                new_board = self.board[:]
                # Swap zero with the adjacent tile
                            new_board[row * 3 + col], new_board[new_row * 3 + new_col] =
new_board[new_row * 3 + new_col], new_board[row * 3 + col]
                moves.append(PuzzleState(new_board, (new_row, new_col), self.path + [new_board]))

        return moves

def bfs(initial_state):
    queue = deque([initial_state])
    visited = set()

    while queue:
        current_state = queue.popleft()

        # Show the current board
        print("Current Board State:")
        print_board(current_state.board)
        print()

        if current_state.is_goal():
            return current_state.path
```

```
        visited.add(tuple(current_state.board))

        for next_state in current_state.get_possible_moves():
            if tuple(next_state.board) not in visited:
                queue.append(next_state)

    return None

def print_board(board):
    for i in range(3):
        print(board[i * 3:i * 3 + 3])

def main():
    print("Enter the initial state of the 8-puzzle (use 0 for the blank tile, e.g., '1 2 3 4 5 6 7 8 0'): ")
    user_input = input()
    initial_board = list(map(int, user_input.split()))

    if len(initial_board) != 9 or set(initial_board) != set(range(9)):
        print("Invalid input! Please enter 9 numbers from 0 to 8.")
        return

    zero_position = initial_board.index(0)
    initial_state = PuzzleState(initial_board, (zero_position // 3, zero_position % 3))

    solution_path = bfs(initial_state)

    if solution_path is None:
        print("No solution found.")
    else:
        print("Solution found in", len(solution_path), "steps.")
        for step in solution_path:
            print_board(step)
            print()

if __name__ == "__main__":
    main()

print("Sparsha Srinath Kadaba - 1BM22CS287")

#8 puzzle problem using dfs
```

```
class PuzzleState:
    def __init__(self, board, zero_position, path=[]):
        self.board = board
        self.zero_position = zero_position
        self.path = path

    def is_goal(self):
        return self.board == [1, 2, 3, 4, 5, 6, 7, 8, 0]

    def get_possible_moves(self):
        moves = []
        row, col = self.zero_position
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # Right, Down, Left, Up

        for dr, dc in directions:
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                new_board = self.board[:]
                # Swap zero with the adjacent tile
                            new_board[row * 3 + col], new_board[new_row * 3 + new_col] =
new_board[new_row * 3 + new_col], new_board[row * 3 + col]
                moves.append(PuzzleState(new_board, (new_row, new_col), self.path + [new_board]))

        return moves

def dfs(initial_state):
    stack = [initial_state]
    visited = set()

    while stack:
        current_state = stack.pop()

        # Show the current board
        print("Current Board State:")
        print_board(current_state.board)
        print()

        if current_state.is_goal():
            return current_state.path
```

```
        visited.add(tuple(current_state.board))

        for next_state in current_state.get_possible_moves():
            if tuple(next_state.board) not in visited:
                stack.append(next_state)

    return None

def print_board(board):
    for i in range(3):
        print(board[i * 3:i * 3 + 3])

def main():
    print("Enter the initial state of the 8-puzzle (use 0 for the blank tile, e.g., '1 2 3 4 5 6 7 8 0'): ")
    user_input = input()
    initial_board = list(map(int, user_input.split()))

    if len(initial_board) != 9 or set(initial_board) != set(range(9)):
        print("Invalid input! Please enter 9 numbers from 0 to 8.")
        return

    zero_position = initial_board.index(0)
    initial_state = PuzzleState(initial_board, (zero_position // 3, zero_position % 3))

    solution_path = dfs(initial_state)

    if solution_path is None:
        print("No solution found.")
    else:
        print("Solution found in", len(solution_path), "steps.")
        for step in solution_path:
            print_board(step)
            print()

if __name__ == "__main__":
    main()

print("Sparsha Srinath Kadaba - 1BM22CS287")
```

Output:

```
Enter the initial state of the 8-puzzle (use 0 for the blank tile, e.g., '1 2 3 4 5 6 7 8 0'):
1 2 3 4 5 6 7 0 8
Current Board State:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

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

Solution found in 1 steps.
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
```

```
Enter the initial state of the 8-puzzle (use 0 for the blank tile, e.g., '1 2 3 4 5 6 7 8 0'):
1 2 3 4 5 6 0 7 8
Current Board State:
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]

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

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

Program 3
Implement iterative deepening search algorithm.

Algorithm:

Code:

```
from copy import deepcopy

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

class PuzzleState:
    def __init__(self, board, parent=None, move=""):
        self.board = board
        self.parent = parent
        self.move = move

    def get_blank_position(self):
        for i in range(3):
            for j in range(3):
                if self.board[i][j] == 0:
                    return i, j

    def generate_successors(self):
        successors = []
        x, y = self.get_blank_position()
        for dx, dy in DIRECTIONS:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_board = deepcopy(self.board)
                        new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y],
new_board[x][y]
                successors.append(PuzzleState(new_board, parent=self))
        return successors

    def is_goal(self, goal_state):
        return self.board == goal_state

    def __str__(self):
        return "\n".join([" ".join(map(str, row)) for row in self.board])
```

```
def depth_limited_search(current_state, goal_state, depth):
    if depth == 0 and current_state.is_goal(goal_state):
        return current_state
    if depth > 0:
        for successor in current_state.generate_successors():
            found = depth_limited_search(successor, goal_state, depth - 1)
            if found:
                return found
    return None

def iterative_deepening_search(start_state, goal_state, max_depth):
    for depth in range(max_depth + 1):
        print(f"\nSearching at depth level: {depth}")
        result = depth_limited_search(start_state, goal_state, depth)
        if result:
            return result
    return None

def get_user_input():
    print("Enter the start state (use 0 for the blank):")
    start_state = []
    for _ in range(3):
        row = list(map(int, input().split()))
        start_state.append(row)

    print("Enter the goal state (use 0 for the blank):")
    goal_state = []
    for _ in range(3):
        row = list(map(int, input().split()))
        goal_state.append(row)

    max_depth = int(input("Enter the maximum depth for search: "))
    return start_state, goal_state, max_depth

def main():
    start_board, goal_board, max_depth = get_user_input()
    start_state = PuzzleState(start_board)
    goal_state = goal_board

    result = iterative_deepening_search(start_state, goal_state, max_depth)
```

```
    if result:
        print("\nGoal reached!")
        path = []
        while result:
            path.append(result)
            result = result.parent
        path.reverse()
        for state in path:
            print(state, "\n")
    else:
        print("Goal state not found within the specified depth.")

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

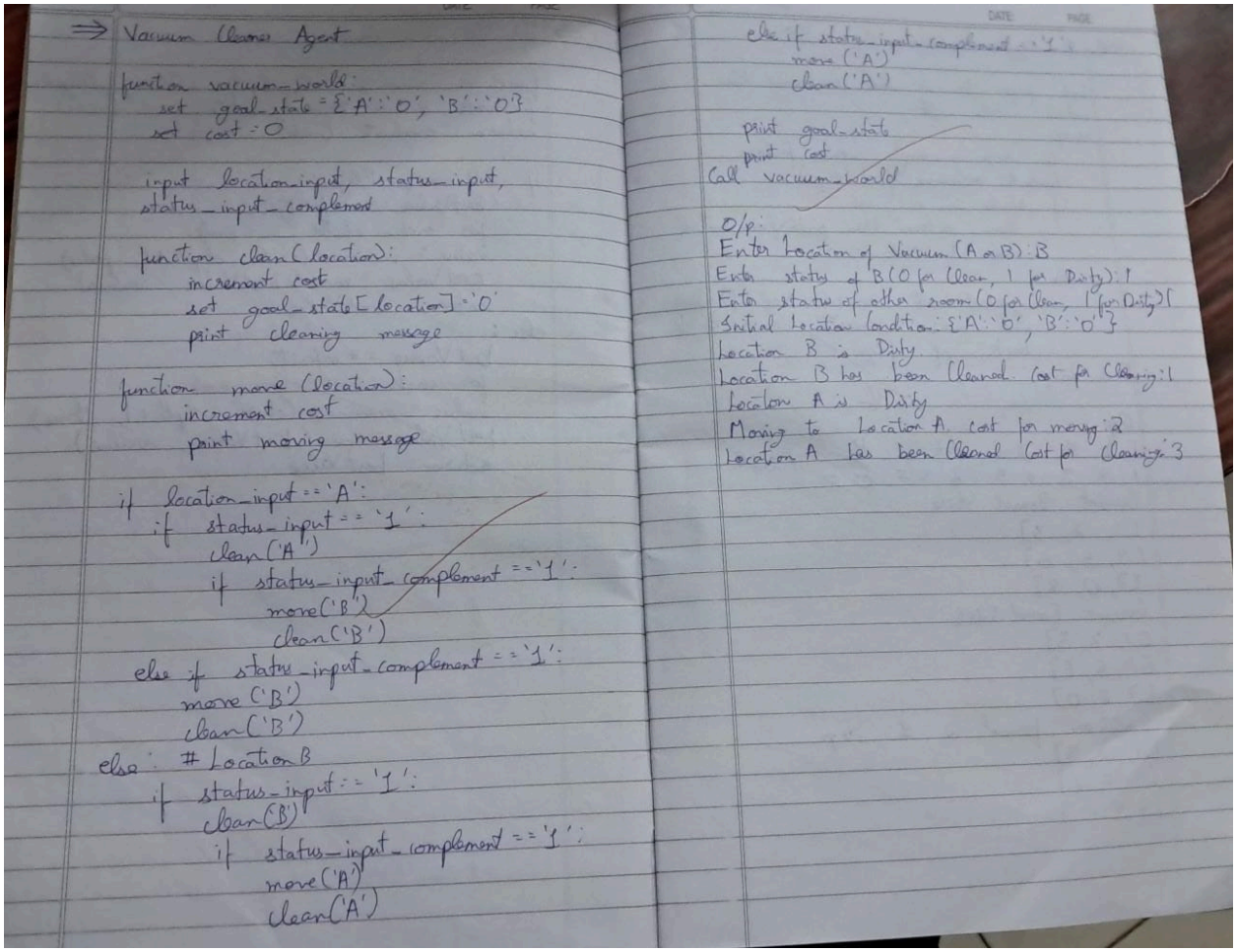
Output:

```
Enter the start state (use 0 for the blank):
2 8 3
1 6 4
7 0 5
Enter the goal state (use 0 for the blank):
1 2 3
8 0 4
7 6 5

Enter the maximum depth for search: 5

Searching at depth level: 0

Searching at depth level: 1
```

Program 4

Implement vacuum cleaner agent.

Algorithm:



Code:

```
#Vacuum Cleaner Agent
def vacuum_world():
    # Initializing goal_state: 0 indicates Clean, 1 indicates Dirty
    goal_state = {'A': '0', 'B': '0'}
    cost = 0

    # User inputs
    location_input = input("Enter Location of Vacuum (A or B): ")
    status_input = input(f"Enter status of {location_input} (0 for Clean, 1 for Dirty): ")
    status_input_complement = input("Enter status of other room (0 for Clean, 1 for Dirty): ")

    print("Initial Location Condition:", goal_state)
```

```
# Function to clean a location
def clean(location):
    nonlocal cost
    goal_state[location] = '0'
    cost += 1
    print(f"Location {location} has been Cleaned. Cost for CLEANING: {cost}")

# Function to move to a location
def move(location):
    nonlocal cost
    cost += 1
    print(f"Moving to Location {location}. Cost for moving: {cost}")

if location_input == 'A':
    if status_input == '1':
        print("Location A is Dirty.")
        clean('A')
        if status_input_complement == '1':
            print("Location B is Dirty.")
            move('B')
            clean('B')
    else:
        print("Location A is already clean.")
        if status_input_complement == '1':
            print("Location B is Dirty.")
            move('B')
            clean('B')

else:  # Vacuum is placed in location B
    if status_input == '1':
        print("Location B is Dirty.")
        clean('B')
        if status_input_complement == '1':
            print("Location A is Dirty.")
            move('A')
            clean('A')
    else:
        print("Location B is already clean.")
        if status_input_complement == '1':
```

```
            print("Location A is Dirty.")
            move('A')
            clean('A')

    # Final output
    print("GOAL STATE:", goal_state)
    print("Performance Measurement:", cost)

# Call the function to run it
vacuum_world()

print("Sparsha Srinath Kadaba - 1BM22CS287")
```

Output:

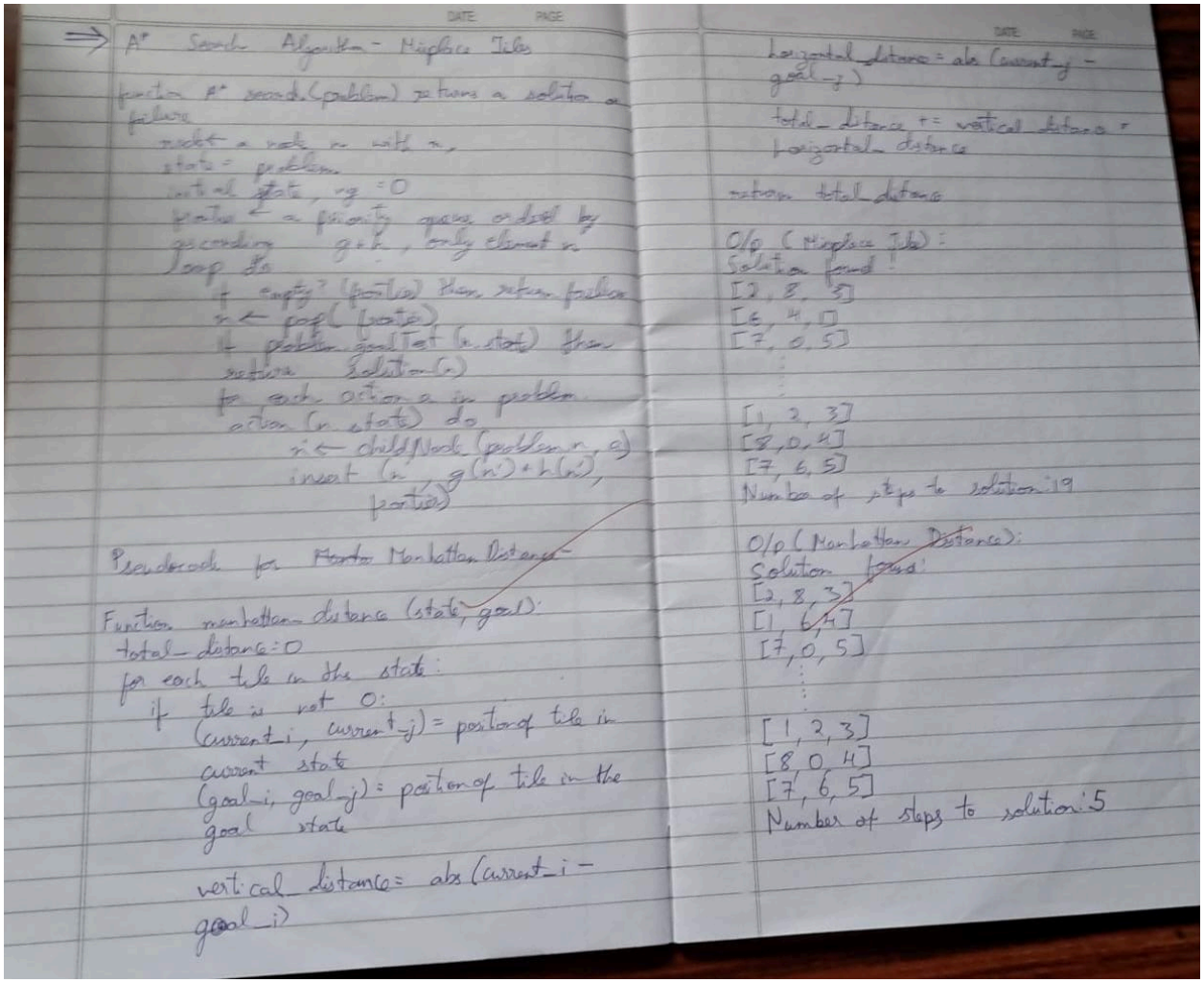
```
Enter Location of Vacuum (A or B): B
Enter status of B (0 for Clean, 1 for Dirty): 1
Enter status of other room (0 for Clean, 1 for Dirty): 1
Initial Location Condition: {'A': '0', 'B': '0'}
Location B is Dirty.
Location B has been Cleaned. Cost for CLEANING: 1
Location A is Dirty.
Moving to Location A. Cost for moving: 2
Location A has been Cleaned. Cost for CLEANING: 3
GOAL STATE: {'A': '0', 'B': '0'}
Performance Measurement: 3
```

Program 5

Implement A* search algorithm.

Implement Hill Climbing Algorithm.

Algorithm:



Code:

#A* Search Algorithm- Misplace Tiles

```
import heapq

# A* search algorithm to solve the 8-puzzle problem
class Puzzle:
    def __init__(self, board, goal):
        self.board = board
        self.goal = goal
        self.n = len(board)
```

```
def find_zero(self, state):
    # Find the index of the empty tile (0)
    for i in range(self.n):
        for j in range(self.n):
            if state[i][j] == 0:
                return i, j

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

def possible_moves(self, state):
    # Generate all possible moves (up, down, left, right)
    moves = []
    i, j = self.find_zero(state)
    if i > 0:  # Up
        new_state = [row[:] for row in state]
        new_state[i][j], new_state[i-1][j] = new_state[i-1][j], new_state[i][j]
        moves.append(new_state)
    if i < self.n - 1:  # Down
        new_state = [row[:] for row in state]
        new_state[i][j], new_state[i+1][j] = new_state[i+1][j], new_state[i][j]
        moves.append(new_state)
    if j > 0:  # Left
        new_state = [row[:] for row in state]
        new_state[i][j], new_state[i][j-1] = new_state[i][j-1], new_state[i][j]
        moves.append(new_state)
    if j < self.n - 1:  # Right
        new_state = [row[:] for row in state]
        new_state[i][j], new_state[i][j+1] = new_state[i][j+1], new_state[i][j]
        moves.append(new_state)
    return moves

def h(self, state):
    # Heuristic: Number of misplaced tiles
    misplaced = 0
    for i in range(self.n):
        for j in range(self.n):
            if state[i][j] != 0 and state[i][j] != self.goal[i][j]:
                misplaced += 1
```

```
        return misplaced

    def astar(self):
        # A* search algorithm
        frontier = []
        heapq.heappush(frontier, (self.h(self.board), 0, self.board, []))  # (f(n), g(n), state, path)
        explored = set()

        while frontier:
            f, g, state, path = heapq.heappop(frontier)

            if self.is_goal(state):
                return path + [state]  # Return the solution path

            explored.add(str(state))

            for move in self.possible_moves(state):
                if str(move) not in explored:
                    heapq.heappush(frontier, (g + 1 + self.h(move), g + 1, move, path + [state]))

        return None

def print_puzzle(path):
    for step in path:
        for row in step:
            print(row)
        print()

# Initial state of the 8-puzzle
initial_state = [
    [2, 8, 3],
    [6, 4, 1],
    [7, 0, 5]
]

# Final state (goal state)
goal_state = [
    [1, 2, 3],
    [8, 0, 4],
    [7, 6, 5]
```

```
]

# Solve the puzzle
puzzle = Puzzle(initial_state, goal_state)
solution = puzzle.astar()

if solution:
    print("Solution found!")
    print_puzzle(solution)
     print(f"Number of steps to solution: {len(solution) - 1}")  # Exclude the initial state from the
count
else:
    print("No solution found.")

print("Sparsha Srinath Kadaba - 1BM22CS287")

#A* Search Algorithm- Manhattan Distance

import heapq

# A* search algorithm to solve the 8-puzzle problem using Manhattan Distance
class Puzzle:
    def __init__(self, board, goal):
        self.board = board
        self.goal = goal
        self.n = len(board)

    def find_zero(self, state):
        # Find the index of the empty tile (0)
        for i in range(self.n):
            for j in range(self.n):
                if state[i][j] == 0:
                    return i, j

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

    def possible_moves(self, state):
        # Generate all possible moves (up, down, left, right)
        moves = []
```

```
        i, j = self.find_zero(state)
        if i > 0:  # Up
            new_state = [row[:] for row in state]
            new_state[i][j], new_state[i-1][j] = new_state[i-1][j], new_state[i][j]
            moves.append(new_state)
        if i < self.n - 1:  # Down
            new_state = [row[:] for row in state]
            new_state[i][j], new_state[i+1][j] = new_state[i+1][j], new_state[i][j]
            moves.append(new_state)
        if j > 0:  # Left
            new_state = [row[:] for row in state]
            new_state[i][j], new_state[i][j-1] = new_state[i][j-1], new_state[i][j]
            moves.append(new_state)
        if j < self.n - 1:  # Right
            new_state = [row[:] for row in state]
            new_state[i][j], new_state[i][j+1] = new_state[i][j+1], new_state[i][j]
            moves.append(new_state)
        return moves

    def manhattan_distance(self, state):
        # Heuristic: Manhattan Distance
        distance = 0
        for i in range(self.n):
            for j in range(self.n):
                if state[i][j] != 0:
                    # Find the goal position of the current tile
                    goal_i, goal_j = divmod(self.goal[i][j], self.n)
                    distance += abs(i - goal_i) + abs(j - goal_j)
        return distance

    def astar(self):
        # A* search algorithm
        frontier = []
            heapq.heappush(frontier, (self.manhattan_distance(self.board), 0, self.board, []))  # (f(n),
g(n), state, path)
        explored = set()

        while frontier:
            f, g, state, path = heapq.heappop(frontier)
```

```
            if self.is_goal(state):
                return path + [state]  # Return the solution path

            explored.add(str(state))

            for move in self.possible_moves(state):
                if str(move) not in explored:
                    heapq.heappush(frontier, (g + 1 + self.manhattan_distance(move), g + 1, move, path
+ [state]))

        return None

def print_puzzle(path):
    for step in path:
        for row in step:
            print(row)
        print()

# Initial state of the 8-puzzle
initial_state = [
    [2, 8, 3],
    [1, 6, 4],
    [7, 0, 5]
]

# Final state (goal state)
goal_state = [
    [1, 2, 3],
    [8, 0, 4],
    [7, 6, 5]
]

# Solve the puzzle
puzzle = Puzzle(initial_state, goal_state)
solution = puzzle.astar()

if solution:
    print("Solution found!")
    print_puzzle(solution)
```

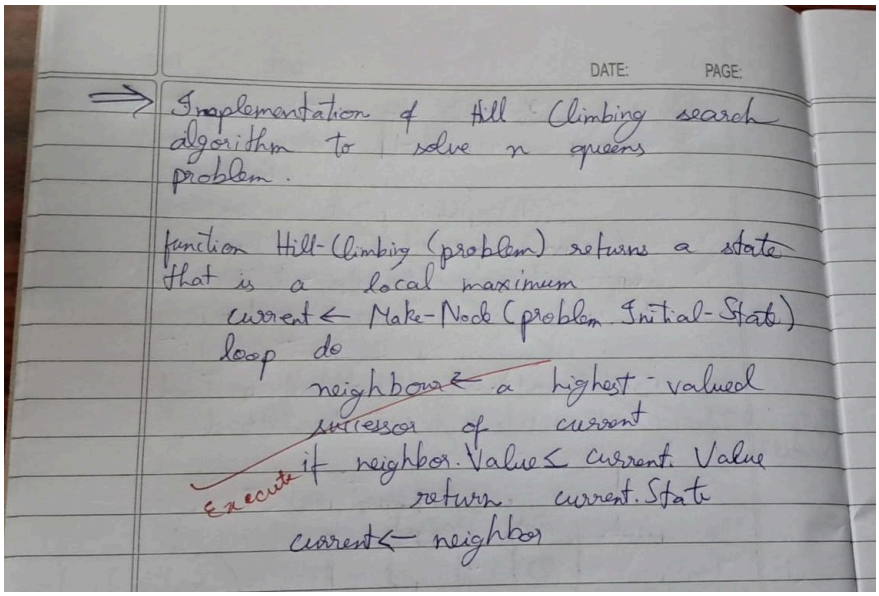
```
    print(f"Number of steps to solution: {len(solution) - 1}")  # Exclude the initial state from the count
else:
    print("No solution found.")

print("Sparsha Srinath Kadaba - 1BM22CS287")
```

Output:

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

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

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

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

Algorithm:



Code:

```
#Hill Climbing Algorithm to solve N Queens problem
import random

# Function to calculate the number of conflicts in the current configuration
```

```
def calculate_conflicts(board):
    n = len(board)
    conflicts = 0

    for i in range(n):
        for j in range(i + 1, n):
            if board[i] == board[j]:
                conflicts += 1  # Same column
            elif abs(board[i] - board[j]) == abs(i - j):
                conflicts += 1  # Same diagonal

    return conflicts

# Function to generate a random configuration of queens (if needed)
def random_board(n):
    return [random.randint(0, n - 1) for _ in range(n)]

# Hill climbing algorithm to solve the N-Queens problem
def hill_climbing(n, current_board):
    current_conflicts = calculate_conflicts(current_board)
    iteration = 0

    # Repeat until we reach a solution or cannot improve
    while current_conflicts != 0:
        iteration += 1
        print(f"Iteration {iteration}, Conflicts: {current_conflicts}")
        print_board(current_board)

        neighbors = []

        # Generate neighbors by moving each queen to a different row in its column
        for i in range(n):
            for row in range(n):
                if row != current_board[i]:
                    new_board = current_board[:]
                    new_board[i] = row
                    neighbors.append(new_board)

        # Find the neighbor with the least conflicts
        best_neighbor = None
```

```
        best_conflicts = current_conflicts

        for neighbor in neighbors:
            neighbor_conflicts = calculate_conflicts(neighbor)
            if neighbor_conflicts < best_conflicts:
                best_conflicts = neighbor_conflicts
                best_neighbor = neighbor

        # If no better neighbor is found, we are stuck, return current board
        if best_conflicts >= current_conflicts:
            print("Stuck in local optimum.")
            return current_board

        # Otherwise, move to the best neighbor
        current_board = best_neighbor
        current_conflicts = best_conflicts

    print(f"Final Iteration {iteration + 1}, Conflicts: {current_conflicts}")
    print_board(current_board)  # Final solution
    return current_board

# Function to print the board in a readable format
def print_board(board):
    n = len(board)
    for i in range(n):
        row = ['Q' if col == board[i] else '.' for col in range(n)]
        print(' '.join(row))
    print()

# Function to get user input for the initial configuration of the queens
def get_user_input(n):
    while True:
        try:
            input_str = input("Enter the list of column positions (space-separated): ")
            board = [int(x) for x in input_str.split()]

            if len(board) != n:
                continue

            if any(x < 0 or x >= n for x in board):
```

```
            continue

        if len(set(board)) != n:
            continue

        return board

    except (ValueError):
        continue

# Main function to handle user input and run the algorithm
def main():
    while True:
        try:
            n = int(input("Enter the size of the board (N): "))
            if n <= 0:
                continue
            else:
                break
        except ValueError:
            continue

    initial_board = get_user_input(n)

    max_restarts = 10
    restart_count = 0

    while restart_count < max_restarts:
        solution = hill_climbing(n, initial_board)

        if calculate_conflicts(solution) == 0:
            return
        else:
            restart_count += 1
            initial_board = random_board(n)  # Restart with a random configuration

    print("\nFailed to find a solution after several attempts.")

# Run the program
if __name__ == "__main__":
```

main()

print("Sparsha Srinath Kadaba - 1BM22CS287")

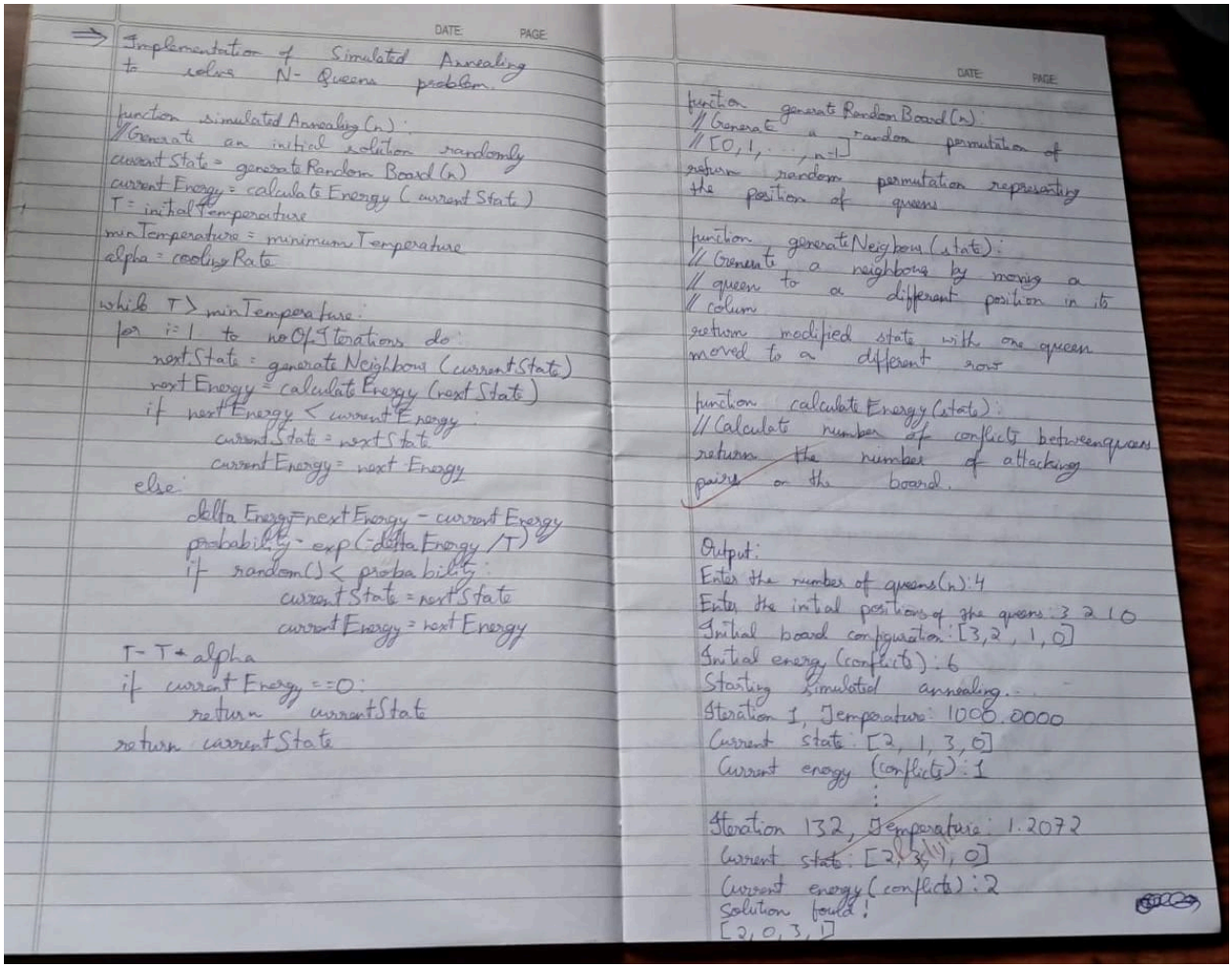
Output:

```
Enter the size of the board (N): 4
Enter the list of column positions (space-separated): 3 1 2 0
Iteration 1, Conflicts: 2
. . . Q
. Q . .
. . Q .
Q . . .

Stuck in local optimum.
Iteration 1, Conflicts: 5
Q . . .
. Q . .
Q . . .
. . . Q
```

Program 6

Write a program to implement Simulated Annealing Algorithm.

Algorithm:



Implementation of Simulated Annealing to solve N-Queens problem

```
function simulatedAnnealing(n):
//Generate an initial solution randomly
currentState = generateRandomBoard(n)
currentEnergy = calculateEnergy(currentState)
T = initialTemperature
minTemperature = minimumTemperature
alpha = coolingRate

while T > minTemperature:
  for i:1 to noOfIterations do:
    nextState = generateNeighbour(currentState)
    nextEnergy = calculateEnergy(nextState)
    if nextEnergy < currentEnergy:
        currentState = nextState
        currentEnergy = nextEnergy
    else:
        deltaEnergy = nextEnergy - currentEnergy
        probability = exp(-deltaEnergy/T)
        if random() < probability:
            currentState = nextState
            currentEnergy = nextEnergy
  T = T * alpha
  if currentEnergy == 0:
      return currentState
return currentState
```

```
function generateRandomBoard(n):
// Generate a random permutation of
// [0, 1, ..., n-1]
return random permutation representing
the position of queens

function generateNeighbour(state):
// Generate a neighbour by moving a
// queen to a different position in its
// colum
return modified state with one queen
moved to a different row

function calculateEnergy(state):
// Calculate number of conflicts between queens
return the number of attacking
pairs on the board.
```

```
Output:
Enter the number of queens(n): 4
Enter the initial positions of the queens: 3 2 1 0
Initial board configuration: [3, 2, 1, 0]
Initial energy (conflicts): 6
Starting simulated annealing...
Iteration 1, Temperature: 1008.0000
Current state: [2, 1, 3, 0]
Current energy (conflicts): 1
:
Iteration 132, Temperature: 1.2072
Current state: [2, 3, 1, 0]
Current energy (conflicts): 2
Solution found!
[2, 0, 3, 1]
```

Code:

```
#Simulated Annealing to solve N Queens problem
import random
import math

def generateRandomBoard(n):
    """Generate a random board represented by a permutation of columns."""
    return random.sample(range(n), n)

def generateNeighbor(state):
    """Generate a neighbor by moving one queen to another row in its column."""
    newState = state[:]
    row = random.randint(0, len(state) - 1)
```

```
    newRow = random.randint(0, len(state) - 1)

    # Ensure the queen moves to a different row (no self-move)
    while newRow == newState[row]:
        newRow = random.randint(0, len(state) - 1)

    newState[row] = newRow
    return newState

def calculateEnergy(state):
    """Calculate the number of attacking pairs of queens."""
    conflicts = 0
    n = len(state)

    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j]:  # Same column
                conflicts += 1
            if abs(state[i] - state[j]) == abs(i - j):  # Same diagonal
                conflicts += 1
    return conflicts

def simulatedAnnealing(n, initialTemperature=1000, minimumTemperature=0.01, alpha=0.95,
numberOfIterations=100):
    """Simulated Annealing algorithm to solve the N-Queens problem."""
    # Generate an initial random board
    currentState = generateRandomBoard(n)
    currentEnergy = calculateEnergy(currentState)
    T = initialTemperature

    print("Starting simulated annealing...")

    # Iterate while the temperature is above the minimum
    iteration = 0
    while T > minimumTemperature:
        iteration += 1
        print(f"\nIteration {iteration}, Temperature: {T:.4f}")
        print(f"Current state: {currentState}")
        print(f"Current energy (conflicts): {currentEnergy}")
```

```
        for _ in range(numberOfIterations):
            # Generate a random neighbor (new board configuration)
            nextState = generateNeighbor(currentState)
            nextEnergy = calculateEnergy(nextState)

            # If the next state is better or accepted by probability
            if nextEnergy < currentEnergy:
                currentState = nextState
                currentEnergy = nextEnergy
            else:
                deltaEnergy = nextEnergy - currentEnergy
                probability = math.exp(-deltaEnergy / T)
                if random.random() < probability:
                    currentState = nextState
                    currentEnergy = nextEnergy

        # Cool the temperature
        T *= alpha

        # If a solution with zero conflicts is found, return the solution
        if currentEnergy == 0:
            return currentState

    # Return the best state found if no perfect solution is found
    return currentState

# Function to handle user input for the board configuration
def getUserInput(n):
    """Get user input for the initial configuration of queens."""
    while True:
        try:
            # Ask the user for the queen positions
            user_input = input(f"Enter the initial positions of the queens (0 to {n-1} for {n} queens,
separated by spaces): ")
            positions = list(map(int, user_input.split()))

            # Validate the input
            if len(positions) != n:
                print(f"Error: You must provide exactly {n} positions.")
                continue
```

```
            # Ensure no duplicates (no two queens in the same column)
            if len(set(positions)) != n:
                print("Error: Two queens cannot be placed in the same column.")
                continue

            # Ensure the positions are within the valid range (0 to n-1)
            if any(pos < 0 or pos >= n for pos in positions):
                print(f"Error: Queen positions must be between 0 and {n-1}.")
                continue

            return positions
        except ValueError:
            print("Error: Invalid input. Please enter integers separated by spaces.")


# Main function to get user input and run the algorithm
if __name__ == "__main__":
    n = int(input("Enter the number of queens (n): "))

    # Get user input for the initial positions of the queens
    initial_state = getUserInput(n)

    # Calculate energy for the initial state
    initial_energy = calculateEnergy(initial_state)
    print(f"Initial board configuration: {initial_state}")
    print(f"Initial energy (conflicts): {initial_energy}")

    # Run the simulated annealing algorithm
    solution = simulatedAnnealing(n)

    # Output the results
    if calculateEnergy(solution) == 0:
        print("\nSolution found!")
        print(solution)
    else:
        print("\nNo perfect solution found. Best found solution:")
        print(solution)

print("Sparsha Srinath Kadaba - 1BM22CS287")
```

Output:

```
Enter the number of queens (n): 4
Enter the initial positions of the queens (0 to 3 for 4 queens, separated by spaces): 3 2 1 0
Initial board configuration: [3, 2, 1, 0]
Initial energy (conflicts): 6
Starting simulated annealing...

Iteration 1, Temperature: 1000.0000
Current state: [2, 1, 3, 0]
Current energy (conflicts): 1

Iteration 2, Temperature: 950.0000
Current state: [3, 0, 0, 1]
Current energy (conflicts): 2
```

Program 7

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

Algorithm:

Code:

```
import itertools

def evaluate_formula(formula, valuation):
    """
    Evaluate the propositional formula under the given truth assignment (valuation).
     The formula is a string of logical operators like 'AND', 'OR', 'NOT', and can contain variables 'A',
    'B', 'C'.
    """
    # Create a local environment (dictionary) for variable assignments
    env = {var: valuation[i] for i, var in enumerate(['A', 'B', 'C'])}

    # Replace logical operators with Python equivalents
    formula = formula.replace('AND', 'and').replace('OR', 'or').replace('NOT', 'not')

    # Replace variables in the formula with their corresponding truth values
    for var in env:
        formula = formula.replace(var, str(env[var]))

    # Evaluate the formula and return the result (True or False)
    try:
        return eval(formula)
    except Exception as e:
        raise ValueError(f"Error in evaluating formula: {e}")

def truth_table(variables):
    """
    Generate all possible truth assignments for the given variables.
    """
    return list(itertools.product([False, True], repeat=len(variables)))

def entails(KB, alpha):
    """
```

```
    Decide if KB entails alpha using a truth-table enumeration algorithm.
    KB is a propositional formula (string), and alpha is another propositional formula (string).
    """
    # Generate all possible truth assignments for A, B, and C
    assignments = truth_table(['A', 'B', 'C'])

     print(f"{'A':<10}{'B':<10}{'C':<10}{'KB':<15}{'alpha':<15}{'KB entails alpha?'}")  # Header
for the truth table
    print("-" * 70)  # Separator for readability

    for assignment in assignments:
        # Evaluate KB and alpha under the current assignment
        KB_value = evaluate_formula(KB, assignment)
        alpha_value = evaluate_formula(alpha, assignment)

        # Print the current truth assignment and the results for KB and alpha

print(f"{str(assignment[0]):<10}{str(assignment[1]):<10}{str(assignment[2]):<10}{str(KB_valu
e):<15}{str(alpha_value):<15}{'Yes' if KB_value and alpha_value else 'No'}")

        # If KB is true and alpha is false, then KB does not entail alpha
        if KB_value and not alpha_value:
            return False

    # If no counterexample was found, then KB entails alpha
    return True

# Define the formulas for KB and alpha
alpha = 'A OR B'
KB = '(A OR C) AND (B OR NOT C)'

# Check if KB entails alpha
result = entails(KB, alpha)

# Print the final result of entailment
print(f"\nDoes KB entail alpha? {result}")
```

Output:

Output:

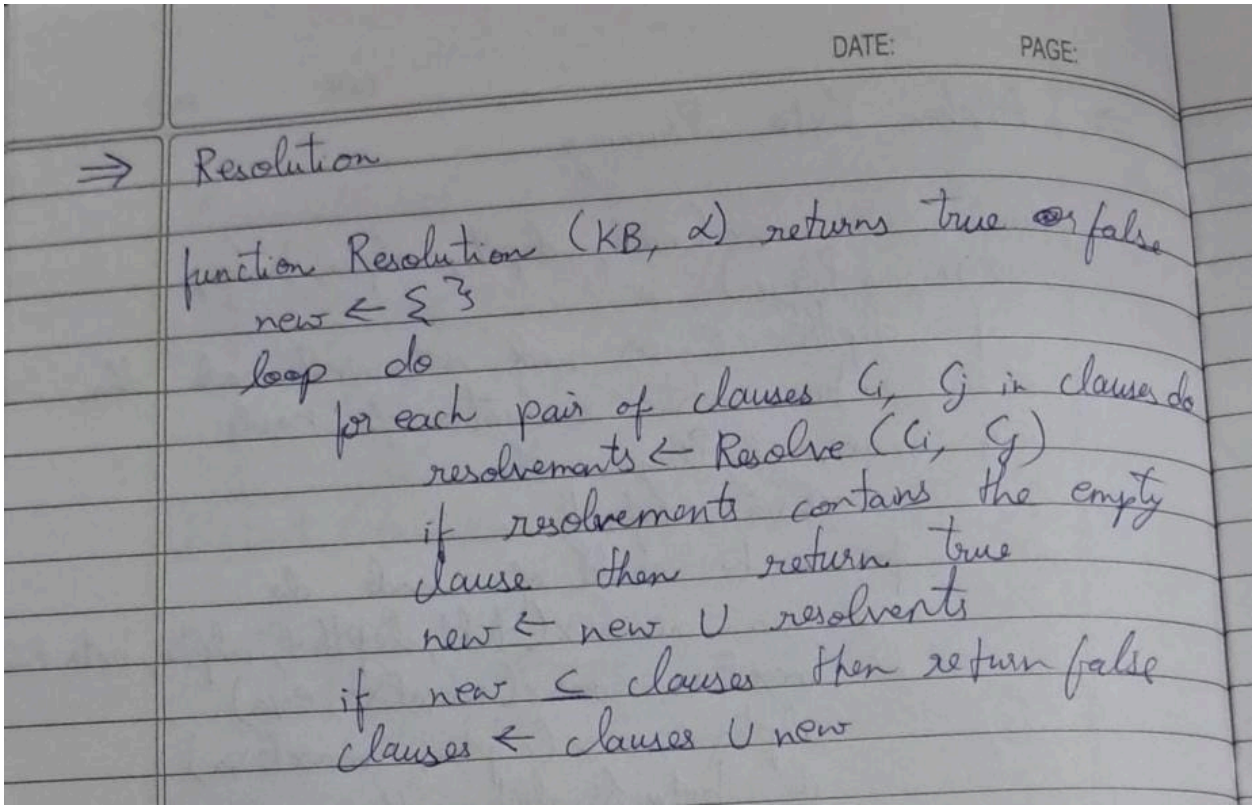
A	B	C	KB	alpha	KB entails alpha?
False	False	False	False	False	No
False	False	True	False	False	No
False	True	False	False	True	No
False	True	True	True	True	Yes
True	False	False	True	True	Yes
True	False	True	False	True	No
True	True	False	True	True	Yes
True	True	True	True	True	Yes

Does KB entail alpha? True

Program 8

Create a knowledge base using prepositional logic and prove the given query using resolution.

Algorithm:



Code:

```
# Step 1: Helper function to parse user inputs into clauses (with '!' for negation)
def parse_clause(clause_input):
    """
    Parses a user input string into a tuple of literals for the clause.
    Replaces '!' with '\u00ac' for negation handling.
    Example: "!Food(x), Likes(John, x)" -> ("\u00acFood(x)", "Likes(John, x)")
    """
    return tuple(literal.strip().replace("!", "\u00ac") for literal in clause_input.split(","))

# Step 2: Collect knowledge base (KB) from user
def get_knowledge_base():
    print("Enter the premises of the knowledge base, one by one.")
    print("Use ',' to separate literals in a clause. Use '!' for negation.")
    print("Example: !Food(x), Likes(John, x)")
    print("Type 'done' when finished.")
```

```
    kb = []
    while True:
        clause_input = input("Enter a clause (or 'done' to finish): ").strip()
        if clause_input.lower() == "done":
            break
        kb.append(parse_clause(clause_input))
    return kb

# Step 3: Add negated conclusion
def get_negated_conclusion():
    print("\nEnter the conclusion to be proved.")
    print("It will be negated automatically for proof by contradiction.")
    conclusion = input("Enter the conclusion (e.g., Likes(John, Peanuts)): ").strip()
    negated = f"!{conclusion}" if not conclusion.startswith("!") else conclusion[1:]
    return (negated.replace("!", "\u00ac"),)  # Convert '!' to '\u00ac' for consistency

# Step 4: Resolution algorithm
def resolve(clause1, clause2):
    """
    Resolves two clauses and returns the resolvent (new clause).
    If no resolvable literals exist, returns an empty set.
    """
    resolved = set()
    for literal in clause1:
        if f"\u00ac{literal}" in clause2:
            temp1 = set(clause1)
            temp2 = set(clause2)
            temp1.remove(literal)
            temp2.remove(f"\u00ac{literal}")
            resolved = temp1.union(temp2)
        elif literal.startswith("\u00ac") and literal[1:] in clause2:
            temp1 = set(clause1)
            temp2 = set(clause2)
            temp1.remove(literal)
            temp2.remove(literal[1:])
            resolved = temp1.union(temp2)
    return tuple(resolved)

def resolution(kb):
    """
```

```
    Runs the resolution algorithm on the knowledge base (KB).
    Returns True if an empty clause is derived (proving the conclusion), or
    False if resolution fails.
    """
    clauses = set(kb)
    new = set()
    while True:
        # Generate all pairs of clauses
        pairs = [(ci, cj) for ci in clauses for cj in clauses if ci != cj]
        for (ci, cj) in pairs:
            resolvent = resolve(ci, cj)
            if not resolvent:  # Empty clause found
                return True
            new.add(resolvent)
        if new.issubset(clauses):  # No new clauses
            return False
        clauses = clauses.union(new)

# Step 5: Main execution
if __name__ == "__main__":
    print("=== Resolution Proof System ===")
    print("Provide the knowledge base and conclusion to prove.")

    # Collect user inputs
    kb = get_knowledge_base()
    negated_conclusion = get_negated_conclusion()
    kb.append(negated_conclusion)

    # Show the knowledge base
    print("\nKnowledge Base (KB):")
    for clause in kb:
        print("  ", " \u2228 ".join(clause))  # Join literals with OR for readability

    # Perform resolution
    print("\nStarting resolution process...")
    result = resolution(kb)
    if result:
        print("\nProof complete: The conclusion is TRUE.")
    else:
        print("\nResolution failed: The conclusion could not be proved.")
```

Output:

Knowledge Base (KB):

¬Food(x) ∨ Likes(John ∨ x)

Food(Apple)

Food(Vegetables)

¬Eats(x ∨ y) ∨ ¬Killed(x) ∨ Food(y)

Eats(Anil ∨ Peanuts)

¬Killed(Anil)

¬Likes(John,Peanuts)

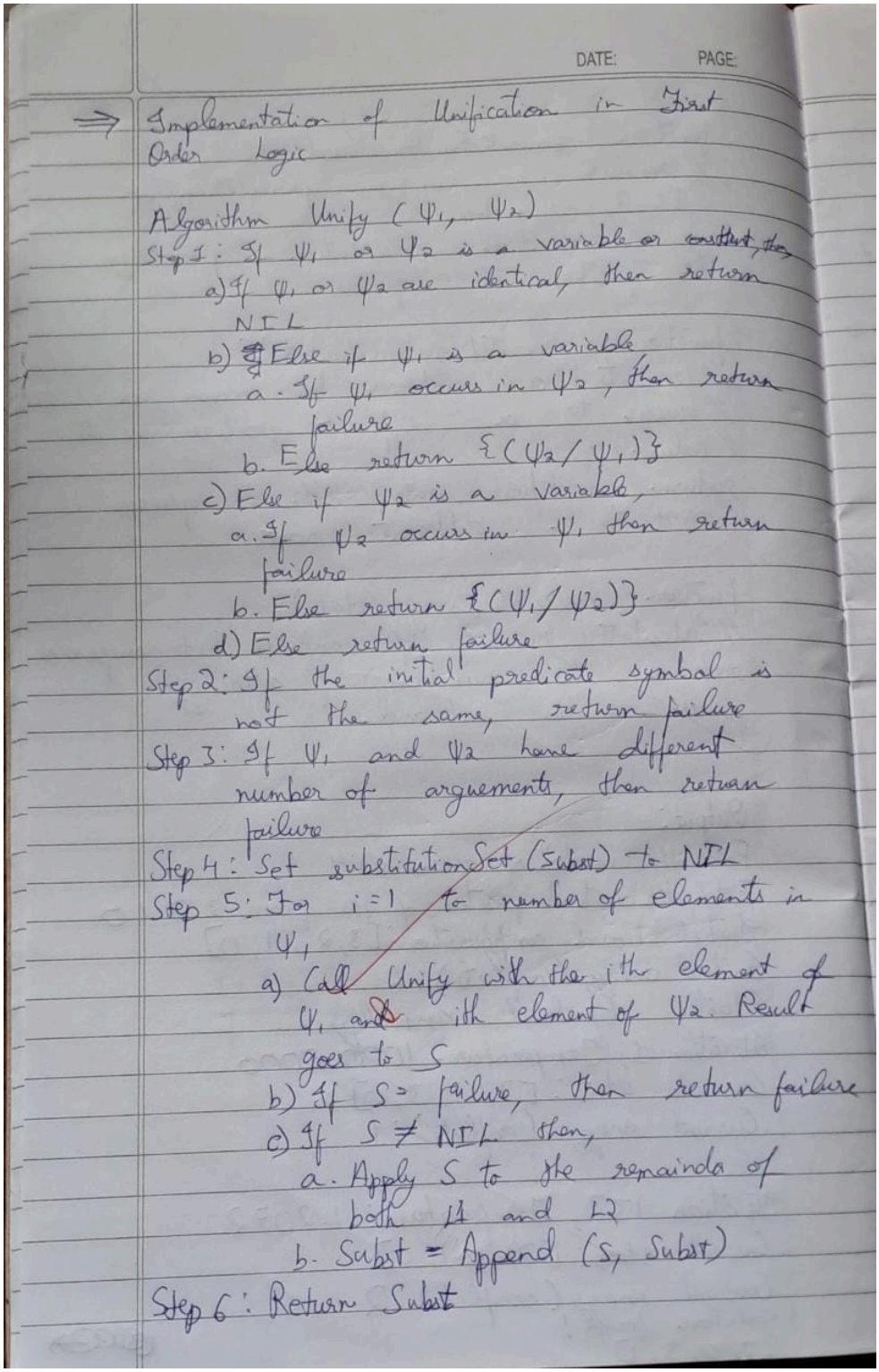
Starting resolution process...

Proof complete: The conclusion is TRUE.

Program 9

Implement unification in first order logic.

Algorithm:



⇒ Implementation of Unification in First Order Logic

Algorithm Unify (ψ_1, ψ_2)

Step 1: If ψ_1 or ψ_2 is a variable or constant, then

a) If ψ_1 or ψ_2 are identical, then return NIL

b) Else if ψ_1 is a variable

a. If ψ_1 occurs in ψ_2, then return failure

b. Else return $\{(\psi_2 / \psi_1)\}$

c) Else if ψ_2 is a variable,

a. If ψ_2 occurs in ψ_1 then return failure

b. Else return $\{(\psi_1 / \psi_2)\}$

d) Else return failure

Step 2: If the initial predicate symbol is not the same, return failure

Step 3: If ψ_1 and ψ_2 have different number of arguements, then return failure

Step 4: Set substitution Set (Subst) to NIL

Step 5: For i=1 to number of elements in ψ_1

a) Call Unify with the ith element of ψ_1 and ith element of ψ_2. Result goes to S

b) If S= failure, then return failure

c) If S ≠ NIL then,

a. Apply S to the remainda of both L1 and L2

b. Subst = Append (S, Subst)

Step 6: Return Subst

Code:

```
#Unification in First Order Logic
def unify(x1, x2):
    """
    Unify two expressions (x1 and x2) based on the given unification algorithm.
    Returns a substitution set (SUBST) or FAILURE if unification is not possible.
    """
    if is_variable_or_constant(x1) or is_variable_or_constant(x2):
        if x1 == x2:
            return []  # Return NIL (empty substitution set)
        elif is_variable(x1):
            if occurs_check(x1, x2):
                return "FAILURE"
            else:
                return [(x2, x1)]  # Return {x2/x1}
        elif is_variable(x2):
            if occurs_check(x2, x1):
                return "FAILURE"
            else:
                return [(x1, x2)]  # Return {x1/x2}
        else:
            return "FAILURE"

    if not is_same_predicate(x1, x2):
        return "FAILURE"

    if len(x1) != len(x2):
        return "FAILURE"

    subst = []  # Substitution set

    for i in range(len(x1)):
        s = unify(x1[i], x2[i])
        if s == "FAILURE":
            return "FAILURE"
        elif s:
            subst.extend(s)
            apply_substitution(s, x1[i+1:])
            apply_substitution(s, x2[i+1:])
```

```
    return subst

def is_variable_or_constant(expr):
    """Check if the expression is a variable or a constant."""
    return isinstance(expr, str) and expr.isalnum()

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

def occurs_check(var, expr):
    """Check if the variable occurs in the expression."""
    if var == expr:
        return True
    elif isinstance(expr, (list, tuple)):
        return any(occurs_check(var, sub_expr) for sub_expr in expr)
    return False

def is_same_predicate(x1, x2):
    """Check if the initial predicate symbols of x1 and x2 are the same."""
    if isinstance(x1, (list, tuple)) and isinstance(x2, (list, tuple)):
        return x1[0] == x2[0]
    return False

def apply_substitution(subst, expr):
    """Apply the substitution set to the given expression."""
    for old, new in subst:
        if expr == old:
            return new
        elif isinstance(expr, (list, tuple)):
            return [apply_substitution(subst, sub_expr) for sub_expr in expr]
    return expr

def parse_input(expr):
    """Parse user input into a list or tuple representing the predicate."""
    try:
        return eval(expr)
    except Exception as e:
        print(f"Error in input format: {e}")
        return None
```

```
# User Input
print("Enter two expressions to unify. Use list/tuple format.")
print("Example: ['P', 'x', 'a'] represents P(x, a)")

expr1_input = input("Enter the first expression: ")
expr2_input = input("Enter the second expression: ")

expr1 = parse_input(expr1_input)
expr2 = parse_input(expr2_input)

if expr1 is not None and expr2 is not None:
    result = unify(expr1, expr2)
    print("Unification Result:", result)
else:
    print("Invalid input format. Please try again.")

print("Sparsha Srinath Kadaba - 1BM22CS287")
```

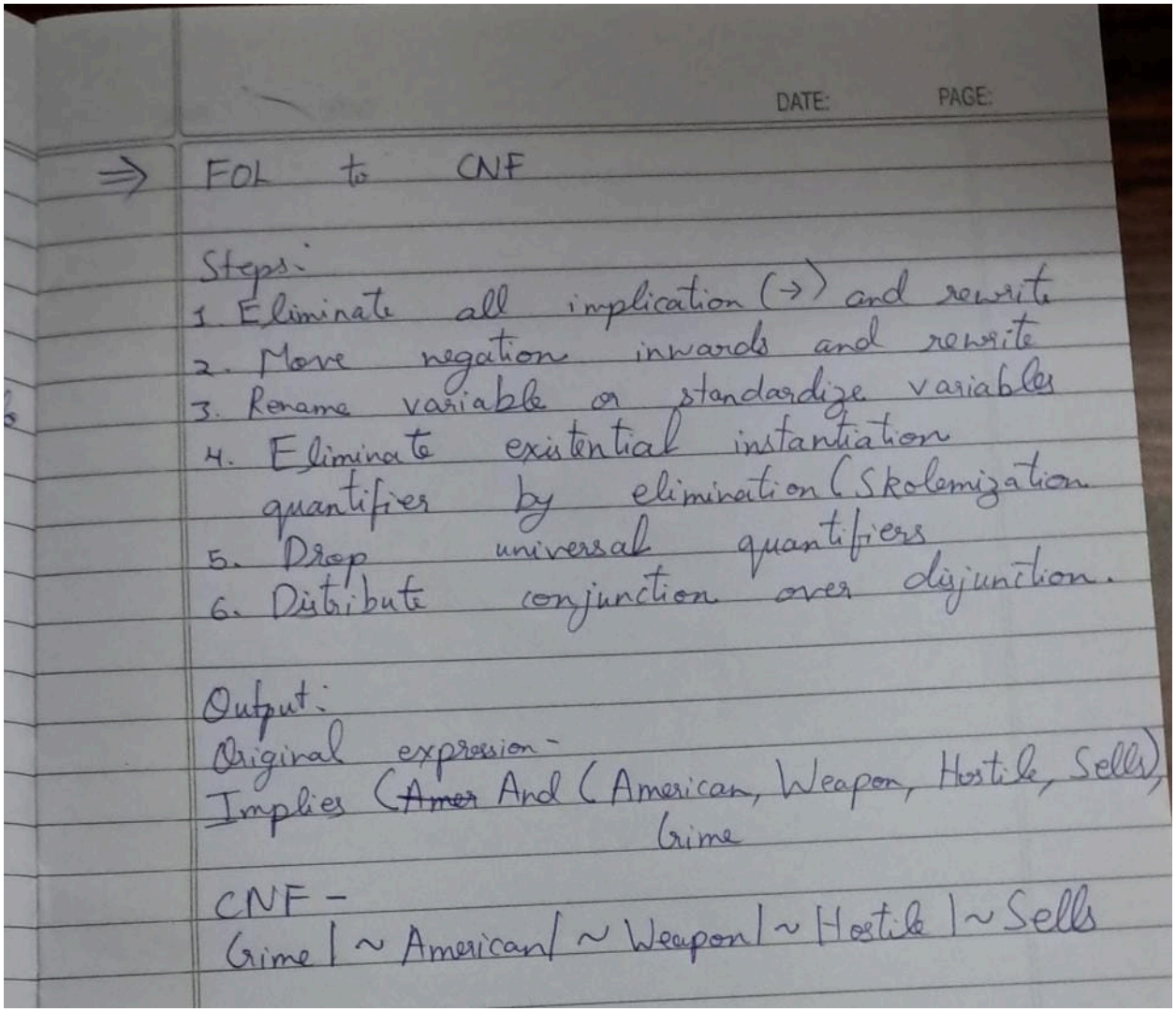
Output:

```
Enter two expressions to unify. Use list/tuple format.
Example: ['P', 'x', 'a'] represents P(x, a)
Enter the first expression: ['P', 'a', 'b']
Enter the second expression: ['P', 'c', 'd']
Unification Result: [('c', 'a'), ('d', 'b')]
Sparsha Srinath Kadaba - 1BM22CS287
```

Program 10

Convert a given first order logic statement into Conjunctive Normal Form (CNF).

Algorithm:



Code:

```
# Conversion of FOL to CNF
from sympy import symbols
from sympy.logic.boolalg import Not, Or, And, Implies, Equivalent, to_cnf

def convert_to_cnf(expression):
    return to_cnf(expression)

def main():
    # Define symbols
    P, Q, R = symbols('P Q R')
```

```
    # Example FOL statements
    expression1 = Implies(P, Q)  # P -> Q
    expression2 = Equivalent(P, Q)  # P <-> Q
    expression3 = Or(And(P, Q), R)  # (P & Q) | R

    # Convert to CNF
    cnf1 = convert_to_cnf(expression1)
    cnf2 = convert_to_cnf(expression2)
    cnf3 = convert_to_cnf(expression3)

    # Display the results
    print("Original Expression 1:", expression1)
    print("CNF 1:", cnf1)
    print()

    print("Original Expression 2:", expression2)
    print("CNF 2:", cnf2)
    print()

    print("Original Expression 3:", expression3)
    print("CNF 3:", cnf3)

if __name__ == "__main__":
    main()

print("\nSparsha Srinath Kadaba - 1BM22CS287")
```

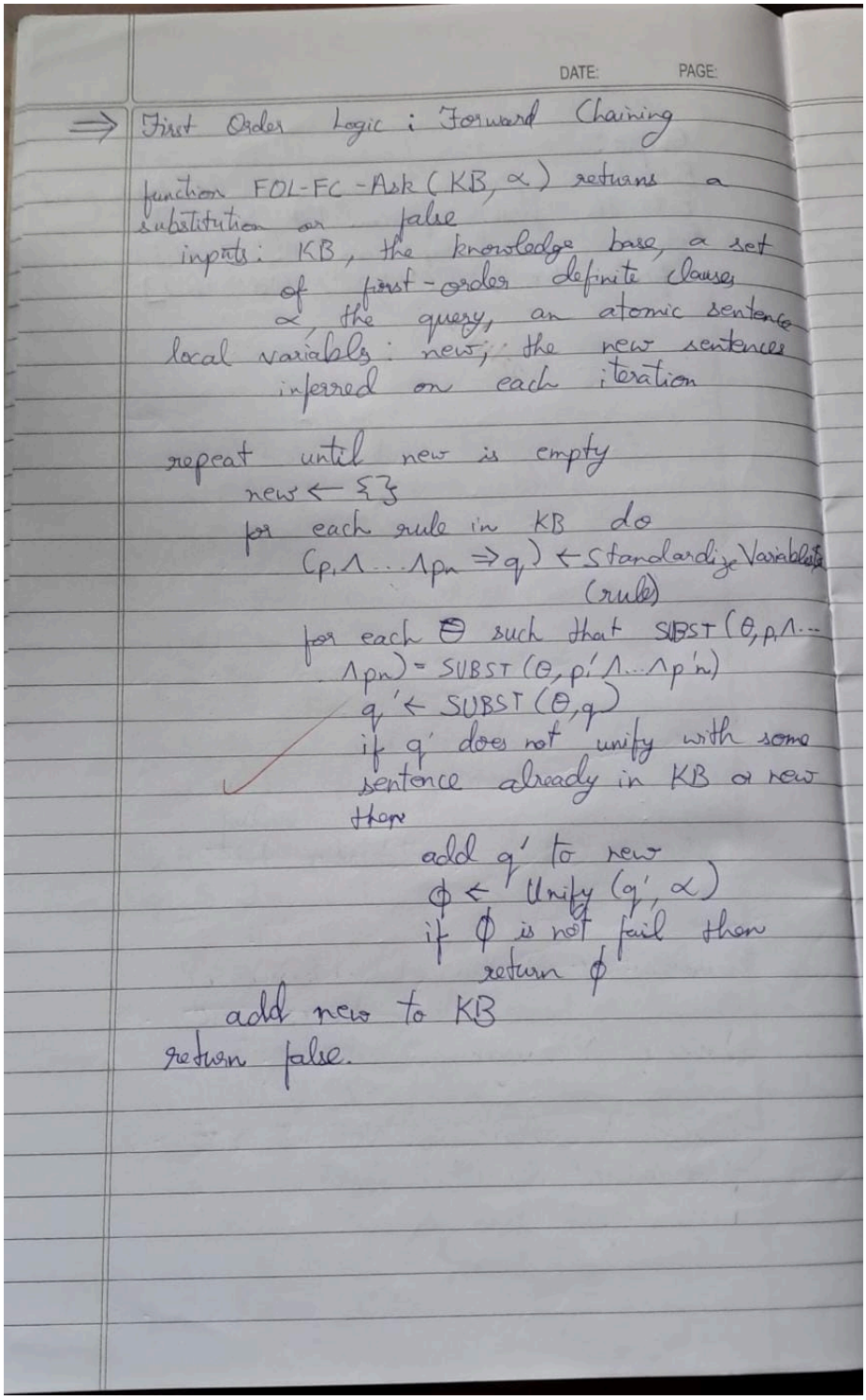
Output:

```
Original Expression 1: Implies(P, Q)
CNF 1: Q | ~P

Original Expression 2: Equivalent(P, Q)
CNF 2: (P | ~Q) & (Q | ~P)

Original Expression 3: R | (P & Q)
CNF 3: (P | R) & (Q | R)

Sparsha Srinath Kadaba - 1BM22CS287
```

Program 11

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

Algorithm:



DATE: PAGE:

⇒ First Order Logic : Forward Chaining

function FOL-FC-Ask(KB, α) returns a substitution or false
inputs: KB, the knowledge base, a set of first-order definite clauses
α, the query, an atomic sentence
local variables: new, the new sentences inferred on each iteration

repeat until new is empty
new ← {}
for each rule in KB do
(p1 ∧ ... ∧ pn ⇒ q) ← Standardize-Variables (rule)
for each θ such that SUBST(θ, p1 ∧ ... ∧ pn) = SUBST(θ, p'1 ∧ ... ∧ p'n)
q' ← SUBST(θ, q)
if q' does not unify with some sentence already in KB or new then
add q' to new
φ ← Unify(q', α)
if φ is not fail then return φ
add new to KB
return false.

Code:

```
#First Order Logic: Forward Chaining

def forward_reasoning_algorithm():
    print("Enter the knowledge base (rules and facts), one per line. Enter 'END' to finish:")

    # Initialize the knowledge base and facts
    knowledge_base = []
    facts = set()

    # Input: Knowledge base (rules and facts)
    while True:
        line = input().strip()
        if line == "END":
            break
        if "=>" in line:  # Rule with premises and conclusion
            premises, conclusion = line.split(" => ")
            knowledge_base.append((premises.split(), conclusion.strip()))
        else:  # Fact
            facts.add(line.strip())

    # Input: Query
    query = input("Enter the query (atomic sentence): ").strip()

    # Forward-chaining algorithm
    inferred = set()  # Store all inferred facts
    new_inferences = True

    while new_inferences:
        new_inferences = False
        for premises, conclusion in knowledge_base:
            # Check if all premises are satisfied in the current set of facts
            if all(p in facts for p in premises) and conclusion not in facts:
                # Infer the conclusion
                facts.add(conclusion)
                inferred.add(conclusion)
                new_inferences = True

        # Break if no new facts are inferred in this iteration
        if not new_inferences:
```

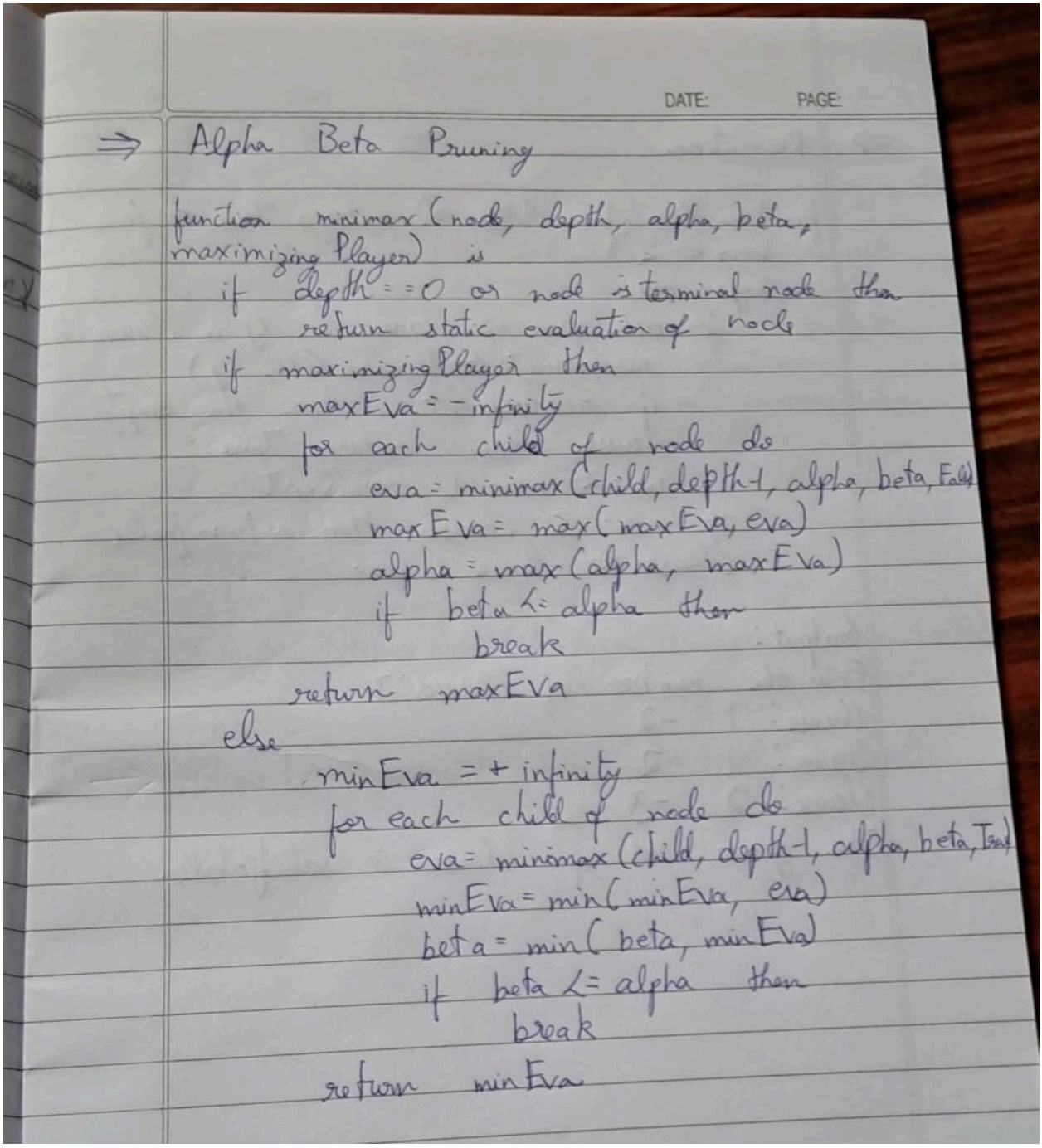
```
            break

    # Check if the query can be inferred
    if query in facts:
        print("Query satisfied.")
    else:
        print("Query cannot be inferred from the knowledge base.")


# Run the algorithm
forward_reasoning_algorithm()

print("\nSparsha Srinath Kadaba - 1BM22CS287")
```

Output:

```
Enter the knowledge base (rules and facts), one per line. Enter 'END' to finish:
fever
cough
soreThroat
flu
cold
fever cough => flu
fever soreThroat => cold
fever headAche => flu
headAche
END
Enter the query (atomic sentence): cold
Query satisfied.

Sparsha Srinath Kadaba - 1BM22CS287
```

Program 12
Implement Alpha-Beta Pruning.

Algorithm:



⇒ Alpha Beta Pruning

```
function minimax(node, depth, alpha, beta,
maximizingPlayer) is
    if depth == 0 or node is terminal node then
        return static evaluation of node
    if maximizingPlayer then
        maxEva = -infinity
        for each child of node do
            eva = minimax(child, depth-1, alpha, beta, False)
            maxEva = max(maxEva, eva)
            alpha = max(alpha, maxEva)
            if beta <= alpha then
                break
        return maxEva
    else
        minEva = +infinity
        for each child of node do
            eva = minimax(child, depth-1, alpha, beta, True)
            minEva = min(minEva, eva)
            beta = min(beta, minEva)
            if beta <= alpha then
                break
        return minEva
```

Code:

```
#Alpha Beta Pruning
def alpha_beta_pruning(node, depth, alpha, beta, maximizing_player):
    """
```

```
    Alpha-Beta Pruning Algorithm.
    Args:
        node: Current node value (can be a game state or a value in the tree).
        depth: Depth of the node in the tree (0 for leaves).
        alpha: Best value the maximizer can guarantee.
        beta: Best value the minimizer can guarantee.
        maximizing_player: Boolean, True if the current player is maximizing.
    Returns:
        Best value for the current player.
    """
    # Base case: if at a leaf node or maximum depth
    if depth == 0 or not isinstance(node, list):
        return node

    if maximizing_player:
        max_eval = float('-inf')
        for child in node:
            eval = alpha_beta_pruning(child, depth - 1, alpha, beta, False)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, max_eval)
            if beta <= alpha:
                break  # Beta cutoff
        return max_eval
    else:
        min_eval = float('inf')
        for child in node:
            eval = alpha_beta_pruning(child, depth - 1, alpha, beta, True)
            min_eval = min(min_eval, eval)
            beta = min(beta, min_eval)
            if beta <= alpha:
                break  # Alpha cutoff
        return min_eval


# User input for tree structure and depth
def main():
    import ast

    print("Enter the tree structure (as a nested list):")
    tree = ast.literal_eval(input("Tree: "))  # Converts input string to Python list
```

```
    print("Enter the depth of the tree (levels of decision-making):")
    depth = int(input("Depth: "))

    print("Enter 1 if the first player is maximizing, otherwise enter 0:")
    maximizing_player = bool(int(input("Maximizing Player (1/0): ")))

    # Call the alpha-beta pruning algorithm
    optimal_value = alpha_beta_pruning(tree, depth, float('-inf'), float('inf'), maximizing_player)
    print("Optimal Value:", optimal_value)


# Example usage
if __name__ == "__main__":
    main()
print("\nSparsha Srinath Kadaba - 1BM22CS287")
```

Output:

```
Enter the tree structure (as a nested list):
Tree: [[3, 6, 1], [2, 4]]
Enter the depth of the tree (levels of decision-making):
Depth: 2
Enter 1 if the first player is maximizing, otherwise enter 0:
Maximizing Player (1/0): 1
Optimal Value: 2

Sparsha Srinath Kadaba - 1BM22CS287
```