VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Shashank C (1BM22CS254)

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 **Shashank C (1BM22CS254),** 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.

Sonika Sharma D Assistant Professor Department of CSE, BMSCE	Dr. Jyothi S Nayak Professor & HOD Department of CSE, BMSCE

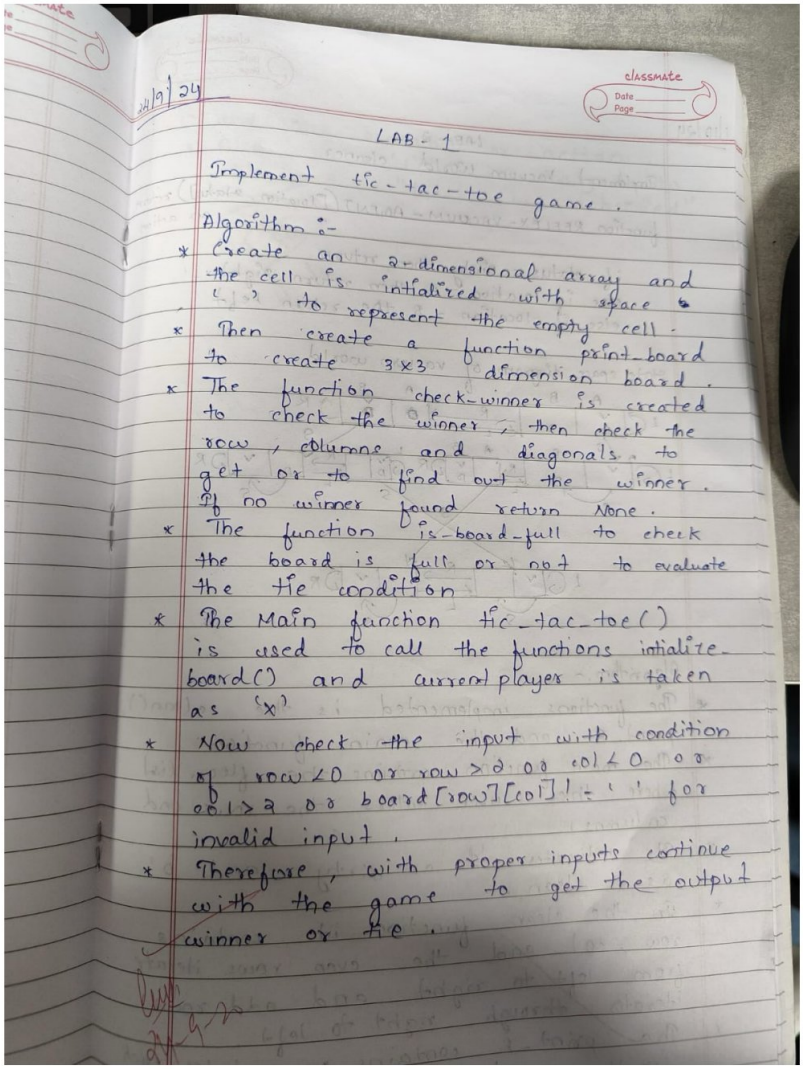
Index

Github Link:
https://github.com/ShashankCS254/AI-LAB.git

Program 1

Implement Tic –Tac –Toe Game
Implement vacuum cleaner agent

Algorithm:

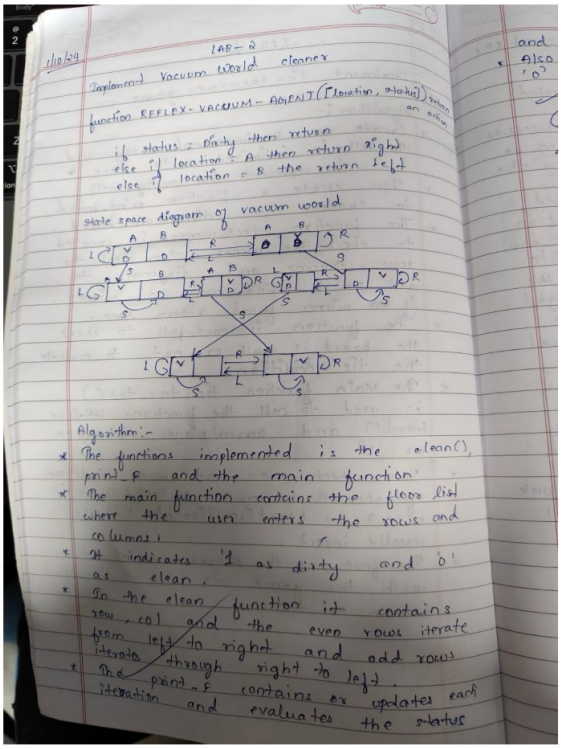


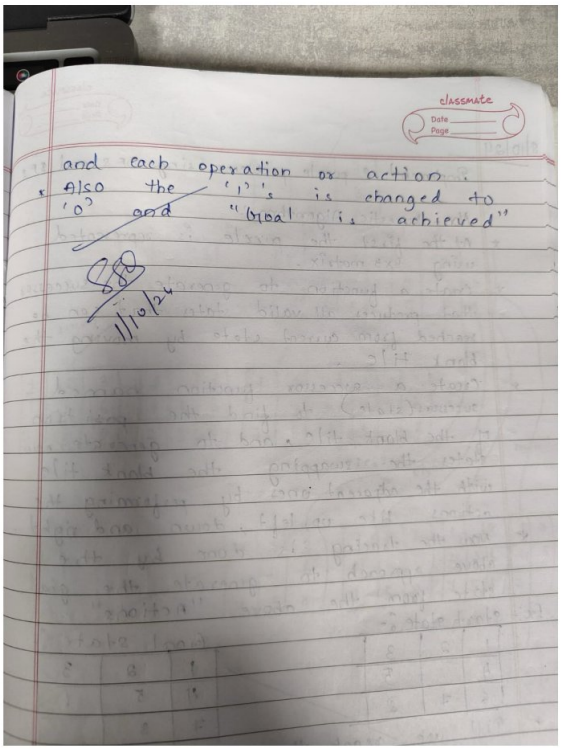
LAB - 1

Implement tic-tac-toe game.

Algorithm :-

* Create an 2-dimensional array and the cell is intialized with space ' ' to represent the empty cell.
* Then create a function print_board to create 3x3 dimension board.
* The function check_winner is created to check the winner, then check the row, columns and diagonals to get or to find out the winner. If no winner found return None.
* The function is_board_full to check the board is full or not to evaluate the tie condition
* The Main function tic_tac_toe() is used to call the functions intialize_board() and current player is taken as 'x'
* Now check the input with condition of row < 0 or row > 2 or col < 0 or col > 2 or board[row][col] != ' ' for invalid input.
* Therefore, with proper inputs continue with the game to get the output winner or tie.





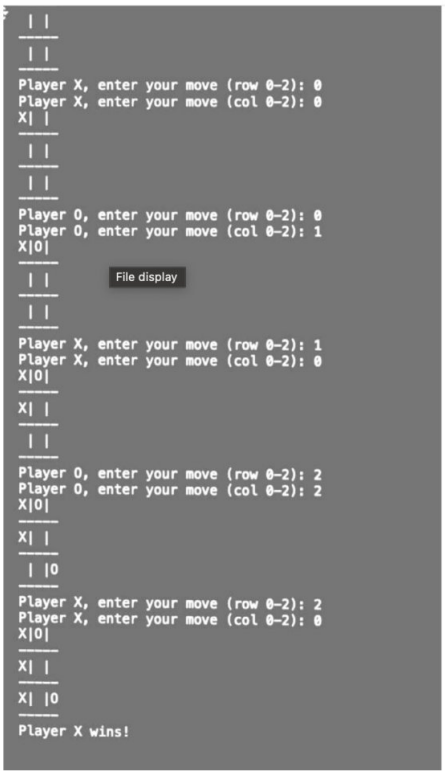
Code: 1: Tic - Tac - Toe

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

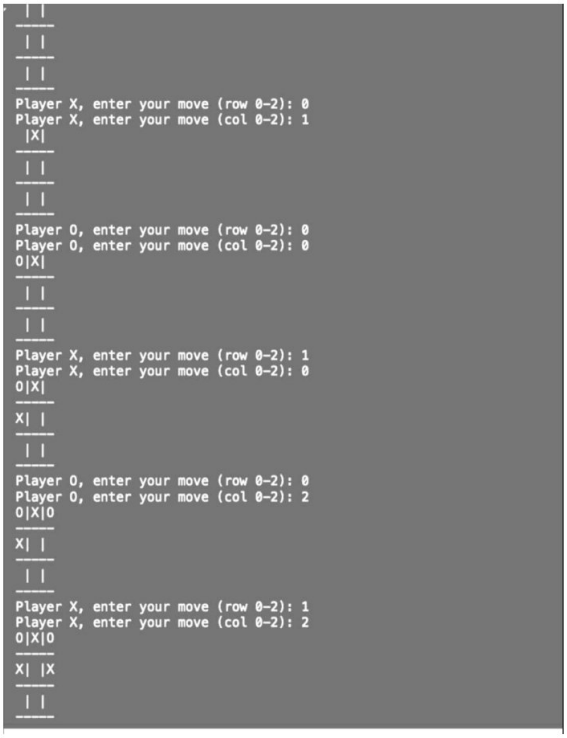
```
def print_board(board):
    for row in board:
        print('|'.join(row))
        print('-' * 5)

def check_winner(board):
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] != ' ':
            return board[i][0]
        if board[0][i] == board[1][i] == board[2][i] != ' ':
            return board[0][i]
    if board[0][0] == board[1][1] == board[2][2] != ' ':
        return board[0][0]
    if board[0][2] == board[1][1] == board[2][0] != ' ':
        return board[0][2]

    return None

def is_board_full(board):
    return all(cell != ' ' for row in board for cell in row)

def tic_tac_toe():
    board = initialize_board()
    current_player = 'X'

    while True:
        print_board(board)
        row = int(input(f"Player {current_player}, enter your move (row 0-2): "))
        col = int(input(f"Player {current_player}, enter your move (col 0-2): "))

        if row < 0 or row > 2 or col < 0 or col > 2 or board[row][col] != ' ':
            print("Invalid move. Try again.")
            continue

        board[row][col] = current_player

        winner = check_winner(board)
        if winner:
            print_board(board)
            print(f"Player {winner} wins!")
            break

        if is_board_full(board):
            print_board(board)
            print("It's a tie!")
            break

        current_player = 'O' if current_player == 'X' else 'X'

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

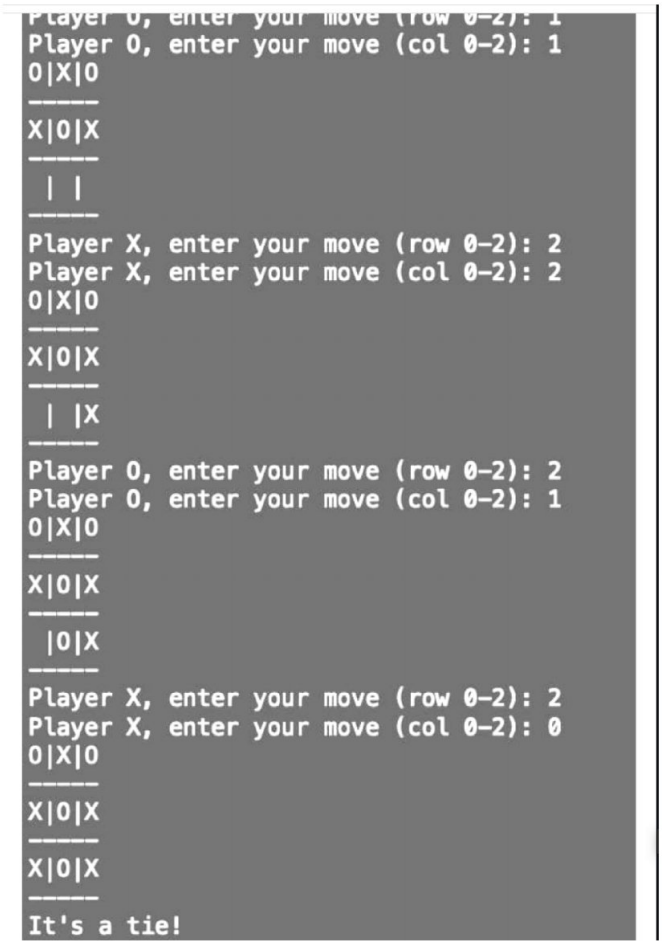
Output:

Win :



Draw :





2. Vacuum Cleaner :

```
def clean(floor):
    row, col = len(floor), len(floor[0])
    total_cost = 0

    for i in range(row):
        if i % 2 == 0:
            for j in range(col):
                if floor[i][j] == 1:
                    print_F(floor, i, j)
                    floor[i][j] = 0
                    total_cost += 1
                print_F(floor, i, j)
        else:
            for j in range(col - 1, -1, -1):
                if floor[i][j] == 1:
                    print_F(floor, i, j)
                    floor[i][j] = 0
                    total_cost += 1
                print_F(floor, i, j)
```

```
    return total_cost

def print_F(floor, row, col):
    print("The Floor matrix is as below:")
    for r in range(len(floor)):
        for c in range(len(floor[r])):
            if r == row and c == col:
                print(f" >{floor[r][c]}< ", end='')
            else:
                print(f"  {floor[r][c]}  ", end='')
        print()
    print()

def main():
    floor = []
    m = int(input("Enter the No. of Rows: "))
    n = int(input("Enter the No. of Columns: "))

    print("Enter clean status for each cell (1 - dirty, 0 - clean):")
    for i in range(m):
        row = []
        for j in range(n):
            value = int(input(f"Enter value for cell ({i+1}, {j+1}): "))
            row.append(value)
        floor.append(row)

    print()
    total_cost = clean(floor)
    print(f"Total cost is: {total_cost}")

main()
print("Goal achieved")
```

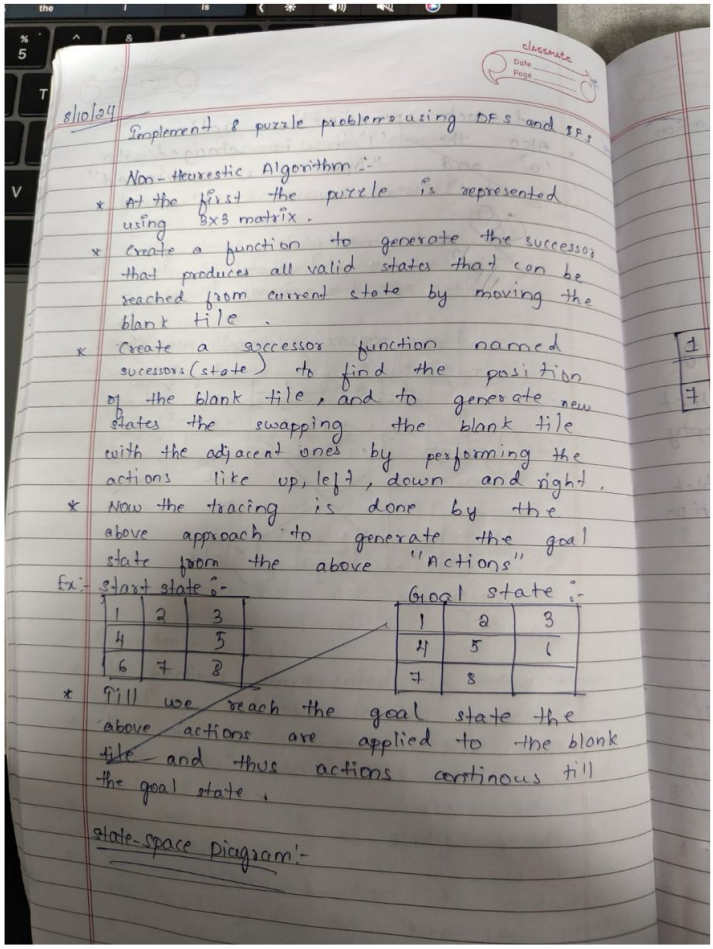
Output :

```
Enter the No. of Rows: 1
Enter the No. of Columns: 2
Enter clean status for each cell (1 - dirty, 0 - clean):
Enter value for cell (1, 1): 1
Enter value for cell (1, 2): 0

The Floor matrix is as below:
 >1<   0

The Floor matrix is as below:
 >0<   0

The Floor matrix is as below:
  0   >0<

Total cost is: 1
Goal achieved
```

Program 2

Implement 8 puzzle problems using Depth First Search (DFS)
Implement Iterative deepening search algorithm

Algorithm:



8/10/24

Implement 8 puzzle problems using DFS and IDFS

Non-heuristic Algorithm:-

* At the first the puzzle is represented using 3x3 matrix.
* Create a function to generate the successors that produces all valid states that can be reached from current state by moving the blank tile.
* Create a successor function named successors(state) to find the position of the blank tile, and to generate new states the swapping the blank tile with the adjacent ones by performing the actions like up, left, down and right.
* Now the tracing is done by the above approach to generate the goal state from the above "Actions"

Ex: Start state:-

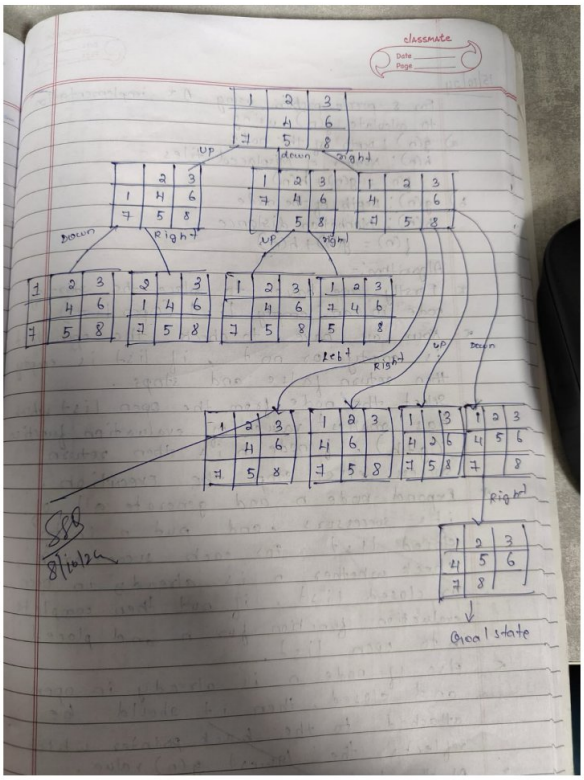
1	2	3
4		5
6	7	8

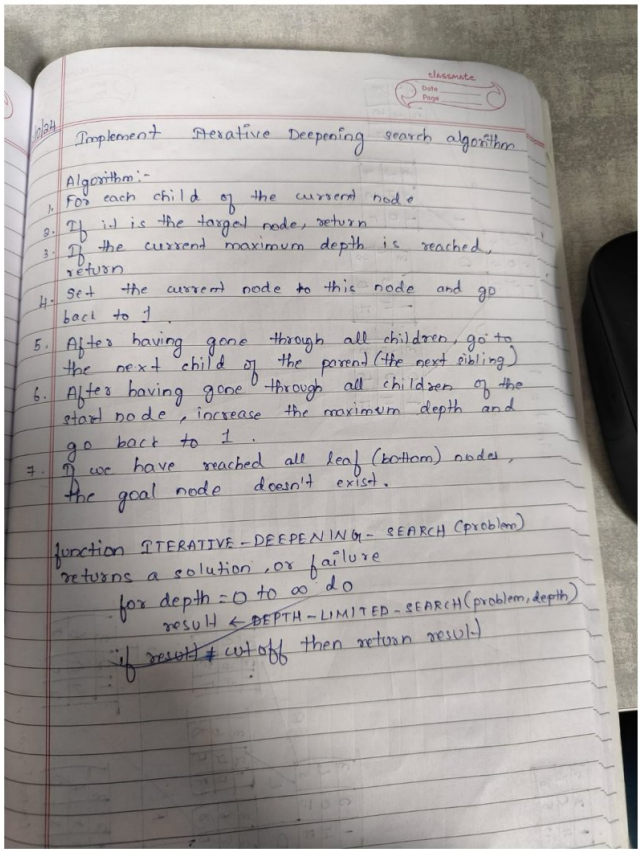
Goal state:-

1	2	3
4	5	6
7	8	

* Till we reach the goal state the above actions are applied to the blank tile and thus actions continous till the goal state.

State-Space Diagram:-





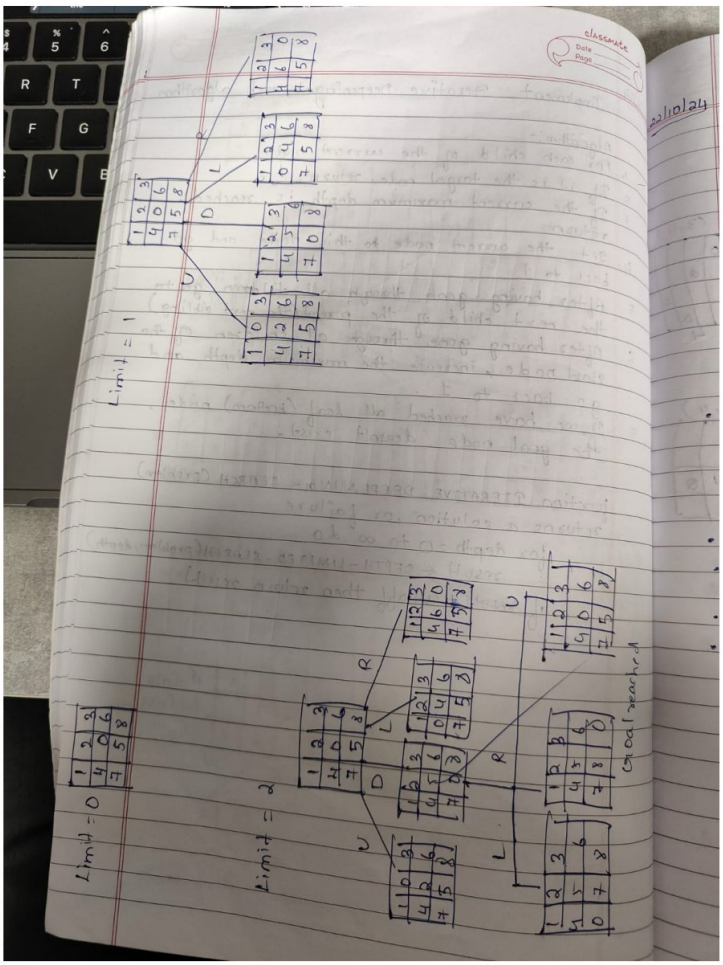
Implement Iterative Deepening search algorithm

Algorithm:-

1. For each child of the current node
2. If it is the target node, return
3. If the current maximum depth is reached, return
4. Set the current node to this node and go back to 1.
5. After having gone through all children, go to the next child of the parent (the next sibling)
6. After having gone through all children of the start node, increase the maximum depth and go back to 1.
7. If we have reached all leaf (bottom) nodes, the goal node doesn't exist.

function ITERATIVE-DEEPENING-SEARCH (problem)
returns a solution, or failure

for depth = 0 to ∞ do
result ← DEPTH-LIMITED-SEARCH (problem, depth)
if result ≠ cut-off then return result

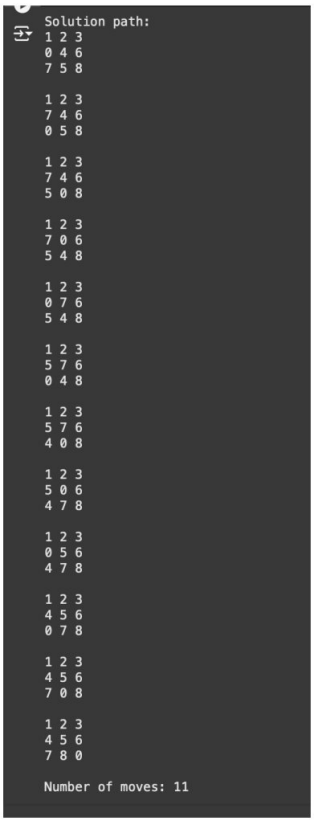


Code:
1: DFS

```
import copy

n = 3  # Size of the puzzle (3x3 for the 8-puzzle)

# Directions for moving the empty tile: Down, Left, Up, Right
directions = [(1, 0), (0, -1), (-1, 0), (0, 1)]

def print_matrix(matrix):
    """Print the current state of the matrix."""
    for row in matrix:
        print(" ".join(map(str, row)))
    print()

def dfs(matrix, empty_pos, goal, visited, path, depth, max_depth):
    """Depth-First Search to solve the puzzle."""
    if matrix == goal:
        print("Solution path:")
        for state in path + [matrix]:  # Include the final state
            print_matrix(state)
        print(f"Number of moves: {len(path)}")  # Display the number of moves
        return True

    if depth >= max_depth:
        return False  # Stop if we reach the maximum depth
```

```
    visited.add(tuple(map(tuple, matrix)))  # Mark the current state as visited

    for dx, dy in directions:
        new_x, new_y = empty_pos[0] + dx, empty_pos[1] + dy
        if 0 <= new_x < n and 0 <= new_y < n:  # Check boundaries
            new_matrix = copy.deepcopy(matrix)
            # Swap the empty tile with the adjacent tile
            new_matrix[empty_pos[0]][empty_pos[1]], new_matrix[new_x][new_y] = new_matrix[new_x]
[new_y], new_matrix[empty_pos[0]][empty_pos[1]]

            if tuple(map(tuple, new_matrix)) not in visited:
                if dfs(new_matrix, (new_x, new_y), goal, visited, path + [matrix], depth + 1, max_depth):
                    return True

    visited.remove(tuple(map(tuple, matrix)))  # Unmark the state for other paths
    return False

# Initial configuration
initial = [
    [1, 2, 3],
    [0, 4, 6],
    [7, 5, 8]
]

# Final configuration
goal = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 0]
]

# Starting position of the empty tile
empty_pos = (1, 0)

# Solve the puzzle using DFS
visited = set()  # To keep track of visited states
path = []  # To store the solution path
max_depth = 20  # Set a reasonable maximum depth

if not dfs(initial, empty_pos, goal, visited, path, 0, max_depth):
    print("No solution found.")
```

Output :



2 : Iterative deepening search

```
class PuzzleState:
    def __init__(self, board, empty_tile_pos, depth=0, path=[]):
        self.board = board
        self.empty_tile_pos = empty_tile_pos  # (row, col)
        self.depth = depth
        self.path = path  # Keep track of the path taken to reach this state

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

    def generate_moves(self):
        row, col = self.empty_tile_pos
        moves = []
        directions = [(-1, 0, 'Up'), (1, 0, 'Down'), (0, -1, 'Left'), (0, 1, 'Right')]  # up, down, left, right
        for dr, dc, move_name in directions:
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                new_board = self.board[:]
                new_board[row * 3 + col], new_board[new_row * 3 + new_col] = new_board[new_row *
```

```
3 + new_col], new_board[row * 3 + col]
                new_path = self.path + [move_name]  # Update the path with the new move
                moves.append(PuzzleState(new_board, (new_row, new_col), self.depth + 1, new_path))
        return moves

    def display(self):
        # Display the board in a matrix form
        for i in range(0, 9, 3):
            print(self.board[i:i + 3])
        print(f"Moves: {self.path}")  # Display the moves taken to reach this state
        print()  # Newline for better readability

def iddfs(initial_state, goal, max_depth):
    for depth in range(max_depth + 1):
        print(f"Searching at depth: {depth}")
        found = dls(initial_state, goal, depth)
        if found:
            print(f"Goal found at depth: {found.depth}")
            found.display()
            return found
    print("Goal not found within max depth.")
    return None

def dls(state, goal, depth):
    if state.is_goal(goal):
        return state

    if depth <= 0:
        return None

    for move in state.generate_moves():
        print("Current state:")
        move.display()  # Display the current state
        result = dls(move, goal, depth - 1)
        if result is not None:
            return result
    return None

def main():
    # User input for initial state, goal state, and maximum depth
    initial_state_input = input("Enter initial state (0 for empty tile, space-separated, e.g. '1 2 3 4 5 6 7 8
0'): ")
    goal_state_input = input("Enter goal state (0 for empty tile, space-separated, e.g. '1 2 3 4 5 6 7 8
0'): ")
```

```
    max_depth = int(input("Enter maximum depth: "))

    initial_board = list(map(int, initial_state_input.split()))
    goal_board = list(map(int, goal_state_input.split()))
    empty_tile_pos = initial_board.index(0) // 3, initial_board.index(0) % 3  # Calculate the position of
the empty tile

    initial_state = PuzzleState(initial_board, empty_tile_pos)

    solution = iddfs(initial_state, goal_board, max_depth)

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

Output :

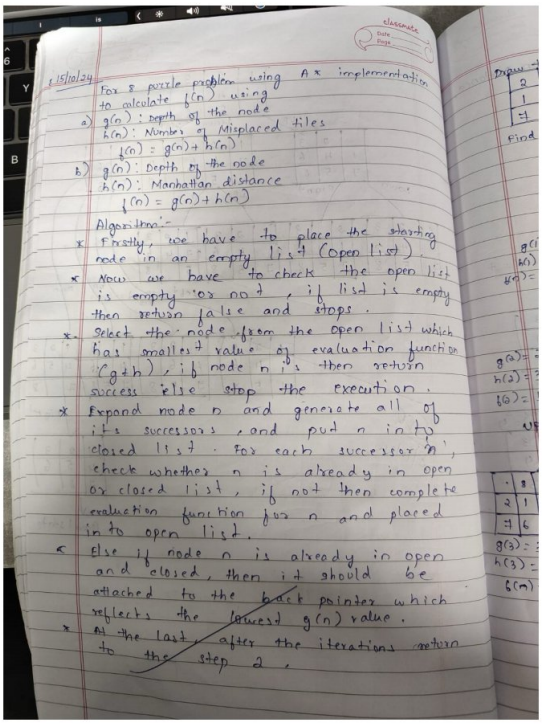
```
Enter initial state (0 for empty tile, space-separated, e.g. '1 2 3 4 5 6 7 8 0'): 1 2 3 0 4 6 7 5 8
Enter goal state (0 for empty tile, space-separated, e.g. '1 2 3 4 5 6 7 8 0'): 1 2 3 4 5 6 7 8 0
Enter maximum depth: 2
Searching at depth: 0
Searching at depth: 1
```

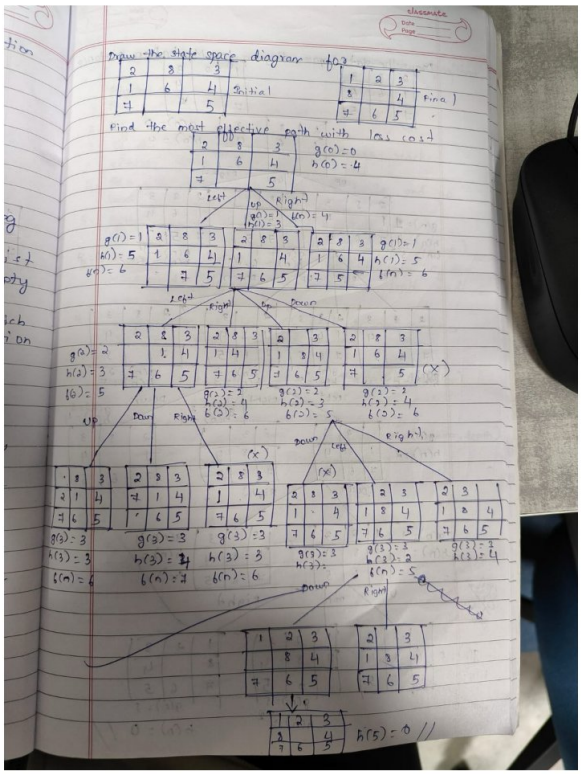
```
Current state:
[0, 2, 3]
[1, 4, 6]
[7, 5, 8]
Moves: ['Up']

Current state:
[1, 2, 3]
[7, 4, 6]
[0, 5, 8]
Moves: ['Down']

Current state:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]
Moves: ['Right']

Searching at depth: 2
Current state:
[0, 2, 3]
[1, 4, 6]
[7, 5, 8]
Moves: ['Up']

Current state:
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]
Moves: ['Up', 'Down']

Current state:
[2, 0, 3]
[1, 4, 6]
[7, 5, 8]
Moves: ['Up', 'Right']

Current state:
[1, 2, 3]
[7, 4, 6]
[0, 5, 8]
Moves: ['Down']

Current state:
[1, 2, 3]
[7, 4, 6]
[0, 5, 8]
Moves: ['Down']

Current state:
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]
Moves: ['Down', 'Up']

Current state:
[1, 2, 3]
[7, 4, 6]
[5, 0, 8]
Moves: ['Down', 'Right']

Current state:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]
Moves: ['Right']

Current state:
[1, 0, 3]
[4, 2, 6]
[7, 5, 8]
Moves: ['Right', 'Up']

Current state:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
Moves: ['Right', 'Down']

Current state:
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]
Moves: ['Right', 'Left']

Current state:
[1, 2, 3]
[4, 6, 0]
[7, 5, 8]
Moves: ['Right', 'Right']

Goal not found within max depth.
```

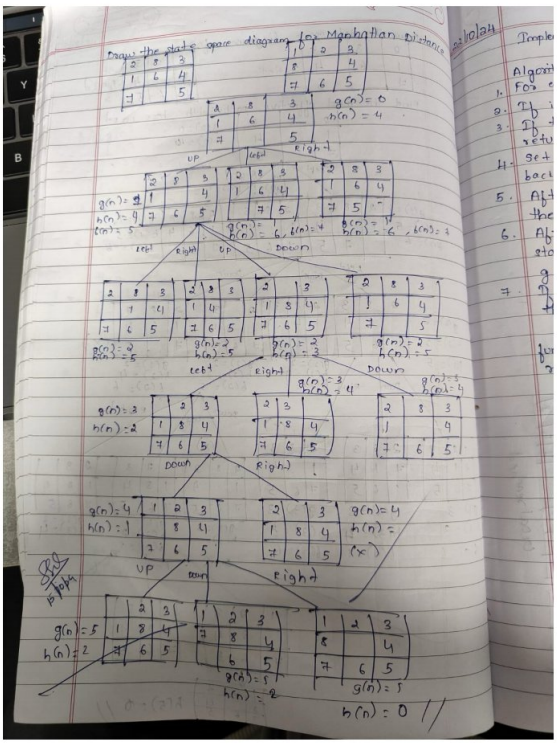
Program 3

Implement A* search algorithm

Algorithm:







Code:

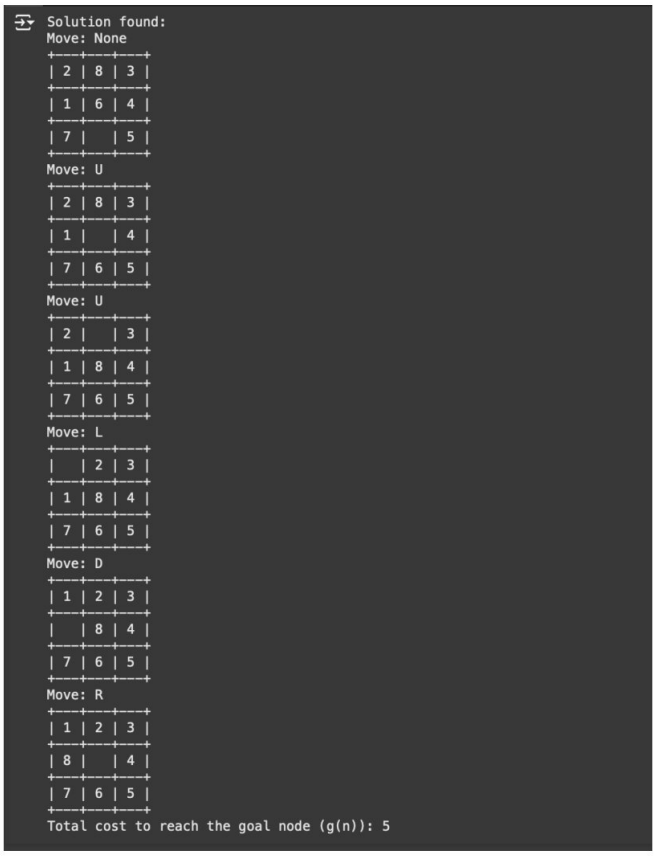
Misplaced Tiles :

```
import heapq
from termcolor import colored

class PuzzleState:
    def __init__(self, board, parent, move, depth):
        self.board = board
        self.parent = parent
        self.move = move
        self.depth = depth
        self.heuristic = misplaced_tiles(board)
        self.cost = self.depth + self.heuristic

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

def print_board(board):
    print("+---+---+---+")
    for row in range(0, 9, 3):
        row_visual = "|"
        for tile in board[row:row + 3]:
            if tile == 0:
                row_visual += f" {colored(' ', 'cyan')} |"
```

```
            else:
                row_visual += f" {colored(str(tile), 'yellow')} |"
        print(row_visual)
        print("+---+---+---+")

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

moves = {
    'U': -3,
    'D': 3,
    'L': -1,
    'R': 1
}

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

def move_tile(board, move, blank_pos):
    new_board = board[:]
    new_blank_pos = blank_pos + moves[move]
    new_board[blank_pos], new_board[new_blank_pos] = new_board[new_blank_pos],
new_board[blank_pos]
    return new_board

def a_star(start_state):
    open_list = []
    closed_list = set()
    heapq.heappush(open_list, PuzzleState(start_state, None, None, 0))

    while open_list:
        current_state = heapq.heappop(open_list)

        if current_state.board == goal_state:
            return current_state

        closed_list.add(tuple(current_state.board))

        blank_pos = current_state.board.index(0)

        for move in moves:
            if move == 'U' and blank_pos < 3:
                continue
            if move == 'D' and blank_pos > 5:
                continue
```

```
            if move == 'L' and blank_pos % 3 == 0:
                continue
            if move == 'R' and blank_pos % 3 == 2:
                continue

            new_board = move_tile(current_state.board, move, blank_pos)

            if tuple(new_board) in closed_list:
                continue

            new_state = PuzzleState(new_board, current_state, move, current_state.depth + 1)
            heapq.heappush(open_list, new_state)

    return None

def print_solution(solution):
    path = []
    current = solution
    while current:
        path.append(current)
        current = current.parent
    path.reverse()

    for step in path:
        print(f"Move: {step.move}")
        print_board(step.board)

    total_cost = solution.depth
    print(colored(f"Total cost to reach the goal node (g(n)): {total_cost}", "green"))

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

solution = a_star(initial_state)

if solution:
    print(colored("Solution found:", "green"))
    print_solution(solution)
else:
    print(colored("No solution exists.", "red"))
```

Output :



Code :

Manhattan distance approach

```
#Manhattan approach
import heapq
from termcolor import colored

class PuzzleState:
    def __init__(self, board, parent, move, depth):
        self.board = board
        self.parent = parent
        self.move = move
        self.depth = depth
        self.heuristic = heuristic(board)
        self.cost = self.depth + self.heuristic

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

def print_board(board):
    print("+---+---+---+")
```

```
    for row in range(0, 9, 3):
        row_visual = "|"
        for tile in board[row:row + 3]:
            if tile == 0:
                row_visual += f" {colored(' ', 'cyan')} |"
            else:
                row_visual += f" {colored(str(tile), 'yellow')} |"
        print(row_visual)
        print("+---+---+---+")

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

moves = {
    'U': -3,
    'D': 3,
    'L': -1,
    'R': 1
}

def heuristic(board):
    distance = 0
    for i in range(9):
        if board[i] != 0:
            x1, y1 = divmod(i, 3)
            x2, y2 = divmod(board[i] - 1, 3)
            distance += abs(x1 - x2) + abs(y1 - y2)
    return distance

def move_tile(board, move, blank_pos):
    new_board = board[:]
    new_blank_pos = blank_pos + moves[move]
    new_board[blank_pos], new_board[new_blank_pos] = new_board[new_blank_pos], new_board[blank_pos]
    return new_board

def a_star(start_state):
    open_list = []
    closed_list = set()
    heapq.heappush(open_list, PuzzleState(start_state, None, None, 0))

    while open_list:
        current_state = heapq.heappop(open_list)

        if current_state.board == goal_state:
            return current_state

        closed_list.add(tuple(current_state.board))

        blank_pos = current_state.board.index(0)

        for move in moves:
            if move == 'U' and blank_pos < 3:
                continue
            if move == 'D' and blank_pos > 5:
                continue
            if move == 'L' and blank_pos % 3 == 0:
```

```
            continue
        if move == 'R' and blank_pos % 3 == 2:
            continue

        new_board = move_tile(current_state.board, move, blank_pos)

        if tuple(new_board) in closed_list:
            continue

        new_state = PuzzleState(new_board, current_state, move, current_state.depth + 1)
        heapq.heappush(open_list, new_state)

    return None

def print_solution(solution):
    path = []
    current = solution
    while current:
        path.append(current)
        current = current.parent
    path.reverse()

    for step in path:
        print(f"Move: {step.move}")
        print_board(step.board)

    total_cost = solution.depth
    print(colored(f"Total cost to reach the goal node (g(n)): {total_cost}", "green"))

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

solution = a_star(initial_state)

if solution:
    print(colored("Solution found:", "green"))
    print_solution(solution)
else:
    print(colored("No solution exists.", "red"))
print("Goal reached")
```

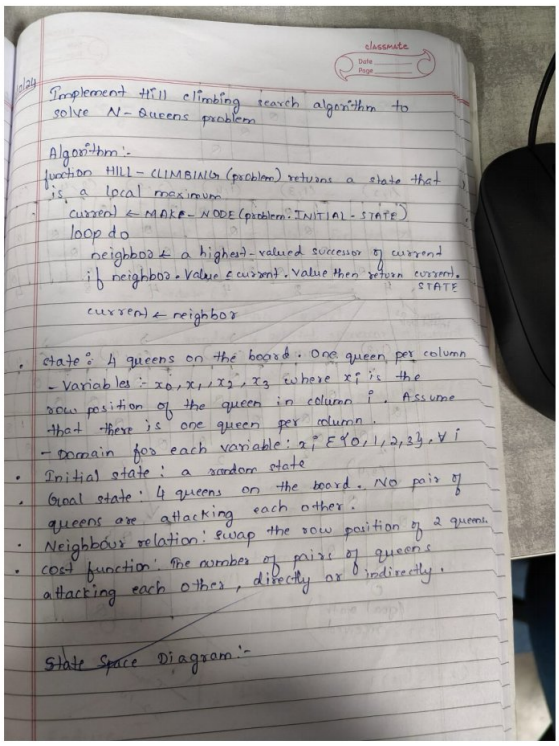
Output :

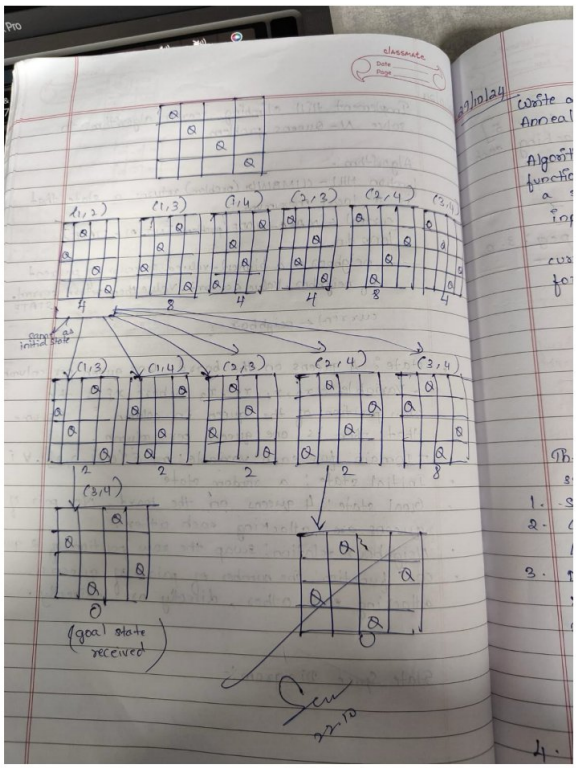
```
Solution found:
Move: None
+---+---+---+
| 2 | 8 | 3 |
+---+---+---+
| 1 | 6 | 4 |
+---+---+---+
| 7 |   | 5 |
+---+---+---+
Move: U
+---+---+---+
| 2 | 8 | 3 |
+---+---+---+
| 1 |   | 4 |
+---+---+---+
| 7 | 6 | 5 |
+---+---+---+
Move: U
+---+---+---+
| 2 |   | 3 |
+---+---+---+
| 1 | 8 | 4 |
+---+---+---+
| 7 | 6 | 5 |
+---+---+---+
Move: L
+---+---+---+
|   | 2 | 3 |
+---+---+---+
| 1 | 8 | 4 |
+---+---+---+
| 7 | 6 | 5 |
+---+---+---+
Move: D
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
|   | 8 | 4 |
+---+---+---+
| 7 | 6 | 5 |
+---+---+---+
Move: R
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 8 |   | 4 |
+---+---+---+
| 7 | 6 | 5 |
+---+---+---+
Total cost to reach the goal node (g(n)): 5
Goal reached
```

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:





Code:

```
import random
def calculate_cost(board):
    n = len(board)
    attacks = 0
    for i in range(n):
        for j in range(i + 1, n):
            if board[i] == board[j]:  # Same column
                attacks += 1
            if abs(board[i] - board[j]) == abs(i - j):  # Same diagonal
                attacks += 1
    return attacks


def get_neighbors(board):
    neighbors = []
    n = len(board)
    for col in range(n):
        for row in range(n):
            if row != board[col]:  # Only change the row of the queen
                new_board = board[:]
                new_board[col] = row
                neighbors.append(new_board)
    return neighbors
```

```
def hill_climb(board, max_restarts=100):
    current_cost = calculate_cost(board)
    print("Initial board configuration:")
    print_board(board, current_cost)
    iteration = 0
    restarts = 0

    while restarts < max_restarts:  # Add limit to the number of restarts
        while current_cost != 0:  # Continue until cost is zero
            neighbors = get_neighbors(board)
            best_neighbor = None
            best_cost = current_cost

            for neighbor in neighbors:
                cost = calculate_cost(neighbor)
                if cost < best_cost:  # Looking for a lower cost
                    best_cost = cost
                    best_neighbor = neighbor

            if best_neighbor is None:  # No better neighbor found
                break  # Break the loop if we are stuck at a local minimum
```

```
            board = best_neighbor
            current_cost = best_cost
            iteration += 1
            print(f"Iteration {iteration}:")
            print_board(board, current_cost)

        if current_cost == 0:
            break  # We found the solution, no need for further restarts
        else:
            # Restart with a new random configuration
            board = [random.randint(0, len(board)-1) for _ in range(len(board))]
            current_cost = calculate_cost(board)
            restarts += 1
            print(f"Restart {restarts}:")
            print_board(board, current_cost)

    return board, current_cost

def print_board(board, cost):
    n = len(board)
    display_board = [['.'] * n for _ in range(n)]  # Create an empty board
```

```
    for col in range(n):
        display_board[board[col]][col] = 'Q'  # Place queens on the board
    for row in range(n):
        print(' '.join(display_board[row]))  # Print the board
    print(f"Cost: {cost}\n")


if __name__ == "__main__":
    n = int(input("Enter the number of queens (N): "))  # User input for N
    initial_state = list(map(int, input(f"Enter the initial state (row numbers for each column, space-separated): ").split()))
    if len(initial_state) != n or any(r < 0 or r >= n for r in initial_state):
        print("Invalid initial state. Please ensure it has N elements with values from 0 to N-1.")
    else:
        solution, cost = hill_climb(initial_state)
        if cost == 0:
            print(f"Solution found with no conflicts:")
        else:
            print(f"No solution found within the restart limit:")
        print_board(solution, cost)
```

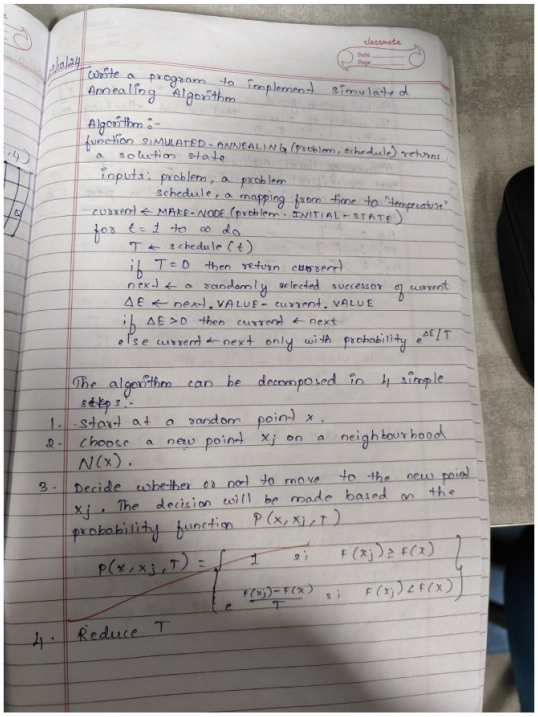
Output :

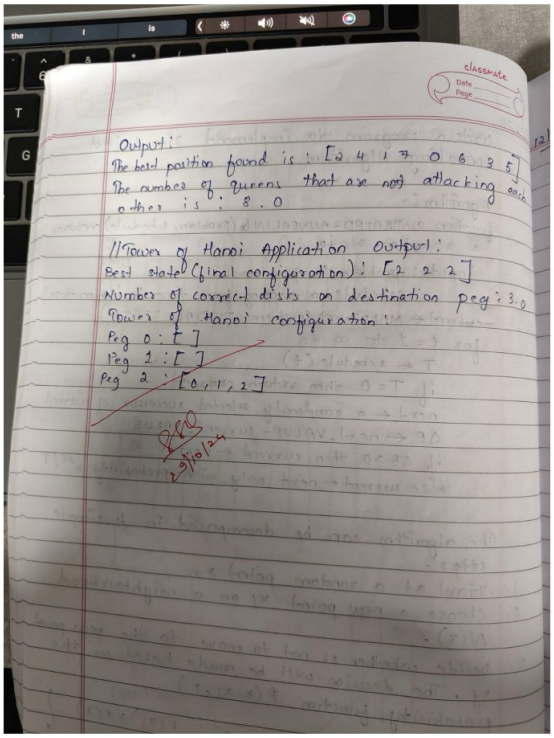
```
Enter the number of queens (N): 4
Enter the initial state (row numbers for each column, space-separated): 0 1 2 3
Initial board configuration:
Q . . .
. Q . .
. . Q .
. . . Q
Cost: 6

Iteration 1:
. . . .
Q Q . .
. . Q .
. . . Q
Cost: 4

Iteration 2:
. Q . .
Q . . .
. . Q .
. . . Q
Cost: 2

Restart 1:
. Q Q Q
. . . .
. . . .
Q . . .
Cost: 4

Iteration 3:
. Q . Q
. . . .
. . Q .
Q . . .
Cost: 2

Iteration 4:
. Q . .
. . . Q
. . Q .
Q . . .
Cost: 1

Restart 2:
. . . Q
. Q . .
. . . .
Q . Q .
Cost: 2

Iteration 6:
. . . .
. Q . .
. . Q Q
Q . . .
Cost: 2

Iteration 7:
. . Q .
. Q . .
. . . Q
Q . . .
Cost: 1

Restart 4:
Q . . .
. Q . Q
. . Q .
. . . .
Cost: 5

Iteration 8:
Q . . .
. Q . Q
. . . .
. . Q .
Cost: 2

Iteration 9:
Q Q . .
. . . Q
. . . .
. . Q .
Cost: 1

Iteration 10:
. Q . .
. . . Q
Q . . .
. . Q .
Cost: 0

Solution found with no conflicts:
. Q . .
. . . Q
Q . . .
. . Q .
Cost: 0
```

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm





Code:

```
#!pip install mlrose-hiive joblib
#!pip install --upgrade joblib
#!pip install joblib==1.1.0
import mlrose_hiive as mlrose
import numpy as np

def queens_max(position):
    no_attack_on_j = 0
    queen_not_attacking = 0
    for i in range(len(position) - 1):
        no_attack_on_j = 0
        for j in range(i + 1, len(position)):
            if (position[j] != position[i]) and (position[j] != position[i] + (j - i)) and (position[j] !=
            position[i] - (j - i)):
                no_attack_on_j += 1
        if (no_attack_on_j == len(position) - 1 - i):
            queen_not_attacking += 1
    if (queen_not_attacking == 7):
        queen_not_attacking += 1
    return queen_not_attacking

objective = mlrose.CustomFitness(queens_max)

problem = mlrose.DiscreteOpt(length=8, fitness_fn=objective, maximize=True, max_val=8)
T = mlrose.ExpDecay()

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

#The simulated_annealing function returns 3 values, we need to capture all 3
best_position, best_objective, fitness_curve = mlrose.simulated_annealing(problem=problem,
schedule=T, max_attempts=500,
                                                  init_state=initial_position)

print('The best position found is:', best_position)
print('The number of queens that are not attacking each other is:', best_objective)
```

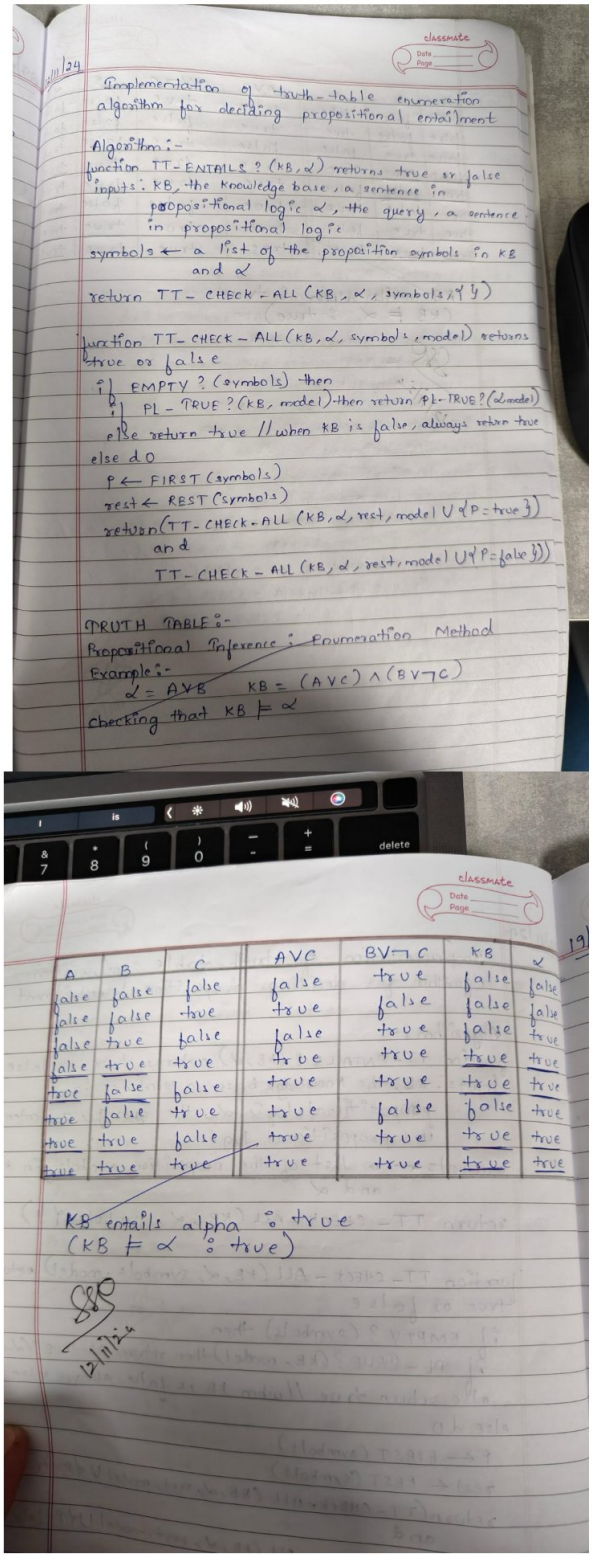
Output :

```
The best position found is: [2 4 1 7 0 6 3 5]
The number of queens that are not attacking each other is: 8.0
```

Program 6

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

Algorithm:



Implementation of truth-table enumeration algorithm for deciding propositional entailment

Algorithm:-

function TT-ENTAILS? (KB, α) returns true or false
inputs: KB, the knowledge base, a sentence in propositional logic
α, the query, a sentence in propositional logic

symbols ← a list of the proposition symbols in KB and α

return TT-CHECK-ALL (KB, α, symbols, {})

function TT-CHECK-ALL (KB, α, symbols, model) returns true or false

if EMPTY? (symbols) then
if PL-TRUE? (KB, model) then return PL-TRUE? (α, model)
else return true // when KB is false, always return true
else do
P ← FIRST (symbols)
rest ← REST (symbols)
return (TT-CHECK-ALL (KB, α, rest, model ∪ {P = true}) and TT-CHECK-ALL (KB, α, rest, model ∪ {P = false}))

TRUTH TABLE:-
Propositional Inference: Enumeration Method
Example:-
α = A∨B KB = (A∨C) ∧ (B∨¬C)
Checking that KB ⊨ α

A	B	C	A∨C	B∨¬C	KB	α
false	false	false	false	true	false	false
false	false	true	true	false	false	false
false	true	false	false	true	false	true
false	true	true	true	true	true	true
true	false	false	true	true	true	true
true	false	true	true	false	false	true
true	true	false	true	true	true	true
true	true	true	true	true	true	true

KB entails alpha : true
(KB ⊨ α : true)

12/12/24

Code:

```
import itertools

def pl_true(sentence, model):
    A = model.get('A', False)
    B = model.get('B', False)
    C = model.get('C', False)

    if sentence == "A or B":
        return A or B
    elif sentence == "(A or C) and (B or not C)":
        return (A or C) and (B or not C)
    return False

def tt_entails(kb, alpha):
    symbols = ['A', 'B', 'C']
    return tt_check_all(kb, alpha, symbols, {})

def tt_check_all(kb, alpha, symbols, model):
    if not symbols:
        if pl_true(kb, model):
            return pl_true(alpha, model)
        else:
            return True
    else:
        p = symbols[0]
        rest = symbols[1:]
        model_true = model.copy()
        model_false = model.copy()
        model_true[p] = True
        model_false[p] = False

        return (tt_check_all(kb, alpha, rest, model_true) and
                tt_check_all(kb, alpha, rest, model_false))

kb = "(A or C) and (B or not C)"
alpha = "A or B"

result = tt_entails(kb, alpha)
print(f"KB entails α: {result}\n")

def generate_truth_table():
    print(f"{'A':<10}{'B':<10}{'C':<10}{'A∨C':<10}{'B∨¬C':<10}{'KB':<10}{'α (A∨B)':<10}
Highlight")
```

```
    for A, B, C in itertools.product([False, True], repeat=3):
        A_or_C = A or C
        B_or_not_C = B or not C
        KB = (A or C) and (B or not C)
        alpha = A or B
        highlight = "*" if KB and alpha else ""  # Mark rows where KB and α are both True
        print(f"{str(A):<10}{str(B):<10}{str(C):<10}{str(A_or_C):<10}{str(B_or_not_C):<10}
{str(KB):<10}{str(alpha):<10}  {highlight}")

generate_truth_table()
```

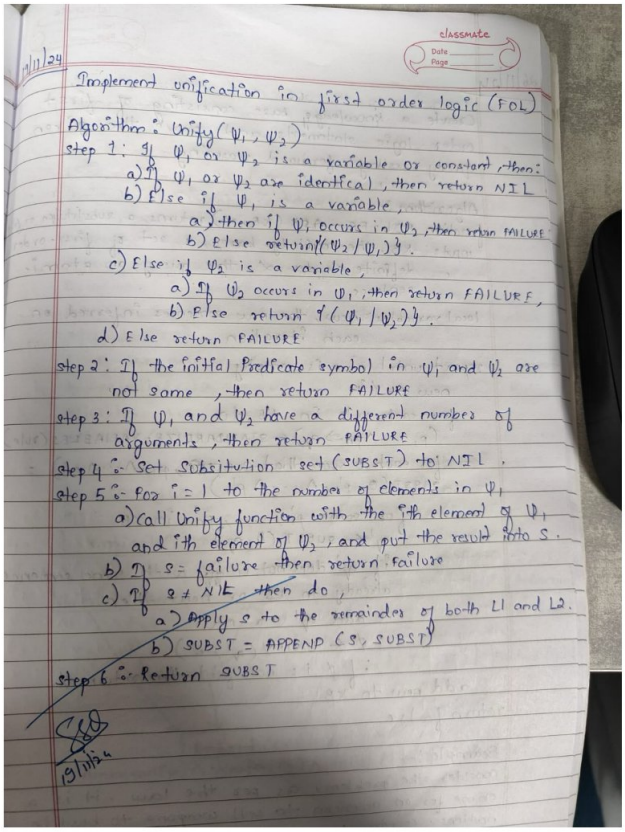
Output :

```
KB entails α: True

A         B         C         AvC       Bv¬C      KB        α (AvB)     Highlight
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        *
```

Program 7

Implement unification in first order logic

Algorithm:



Code:

```
def unify(x, y, subst={}):
    if subst is None:
        return None
    elif x == y:
        return subst
    elif isinstance(x, str) and x.islower():
        return unify_variable(x, y, subst)
    elif isinstance(y, str) and y.islower():
        return unify_variable(y, x, subst)
    elif isinstance(x, list) and isinstance(y, list):
        if len(x) != len(y):
            return None
        for xi, yi in zip(x, y):
            subst = unify(xi, yi, subst)
```

```
        return subst
    else:
        return None

def unify_variable(var, x, subst):
    if var in subst:
        return unify(subst[var], x, subst)
    elif x in subst:
        return unify(var, subst[x], subst)
    elif occurs_check(var, x):
        return None
    else:
        subst[var] = x
        return subst

def occurs_check(var, x):
    if var == x:
        return True
    elif isinstance(x, list):
        return any(occurs_check(var, arg) for arg in x)
    return False

def test_unification():
    terms = [
        (['P', 'x', 'f(y)'], ['P', 'a', 'f(g(x))']),
        (['P', 'X', 'b'], ['P', 'a', 'b']),
        (['P', 'X', 'b'], ['P', 'f(X)', 'b']),
        (['P', 'a', 'b'], ['Q', 'a', 'b']),
        (['P', 'a'], ['P', 'a', 'b']),
    ]

    for t1, t2 in terms:
        subst = unify(t1, t2)
        print(f"Unify({t1}, {t2}) -> {'Unification Possible' if subst else 'Unification Not Possible'}")

test_unification()
```

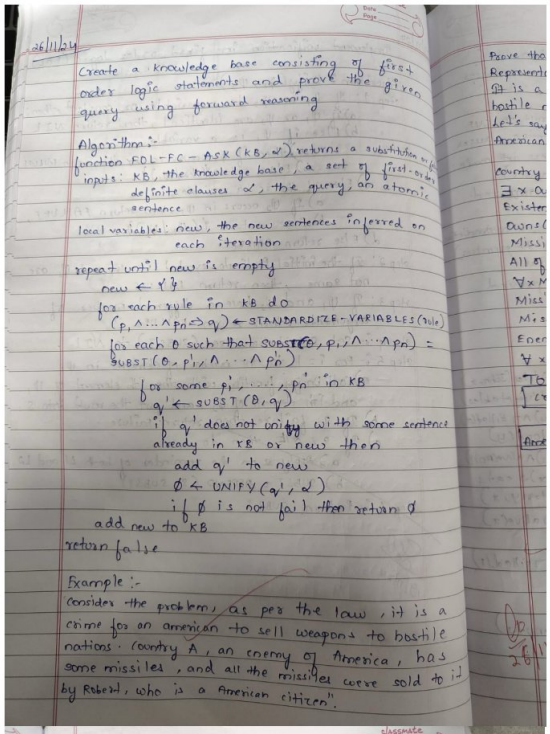
Output :

```
Unify(['P', 'x', 'f(y)'], ['P', 'a', 'f(g(x))']) -> Unification Possible
Unify(['P', 'X', 'b'], ['P', 'a', 'b']) -> Unification Possible
Unify(['P', 'X', 'b'], ['P', 'f(X)', 'b']) -> Unification Not Possible
Unify(['P', 'a', 'b'], ['Q', 'a', 'b']) -> Unification Not Possible
Unify(['P', 'a'], ['P', 'a', 'b']) -> Unification Not Possible
```

Program 8

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

Algorithm:



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

Algorithm:-

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, p1' ∧ ... ∧ pn')

for some p1', ..., pn' in KB

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)

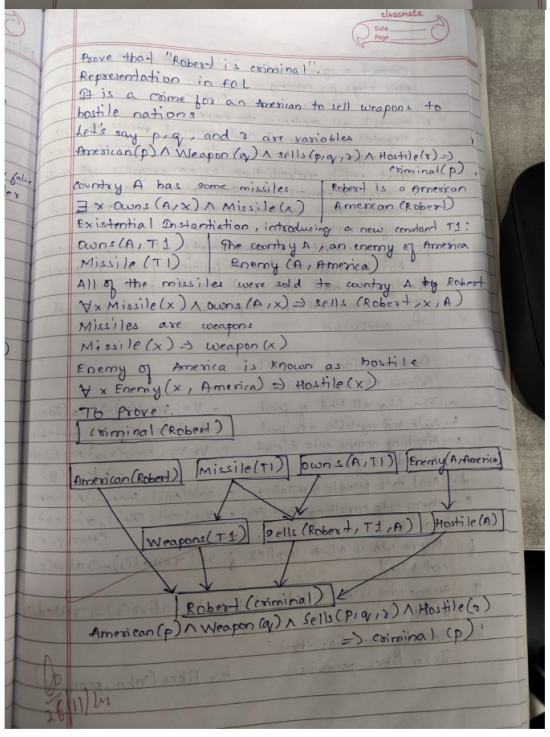
if φ is not fail then return φ

add new to KB

return false

Example:-

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



Prove that "Robert is criminal".

Representation in FOL

It is a crime for an American to sell weapons to hostile nations

Let's say p, q, and r are variables

American(p) ∧ Weapon(q) ∧ sells(p, q, r) ∧ Hostile(r) ⇒ Criminal(p)

Country A has some missiles. | Robert is a American

∃ x Owns(A, x) ∧ Missile(x) | American(Robert)

Existential Instantiation, introducing a new constant T1:

Owns(A, T1) | The country A, an enemy of America

Missile(T1) | Enemy(A, America)

All of the missiles were sold to country A by Robert

∀ x Missile(x) ∧ Owns(A, x) ⇒ Sells(Robert, x, A)

Missiles are weapons

Missile(x) ⇒ Weapon(x)

Enemy of America is known as hostile

∀ x Enemy(x, America) ⇒ Hostile(x)

To Prove:

Criminal(Robert)

American(Robert) | Missile(T1) | Owns(A, T1) | Enemy(A, America)

Weapons(T1) | Sells(Robert, T1, A) | Hostile(A)

Robert(criminal)

American(p) ∧ Weapon(q) ∧ Sells(p, q, r) ∧ Hostile(r) ⇒ Criminal(p)

Code:

```
class KnowledgeBase:
    def __init__(self):
        self.facts = set()   # Set of known facts
        self.rules = []      # List of rules

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

    def add_rule(self, rule):
        self.rules.append(rule)

    def infer(self):
        inferred = True
        while inferred:
            inferred = False
            for rule in self.rules:
                if rule.apply(self.facts):
                    inferred = True

# Define the Rule class
class Rule:
    def __init__(self, premises, conclusion):
        self.premises = premises  # List of conditions
        self.conclusion = conclusion  # Conclusion to add if premises are met

    def apply(self, facts):
        if all(premise in facts for premise in self.premises):
            if self.conclusion not in facts:
                facts.add(self.conclusion)
                print(f"Inferred: {self.conclusion}")
                return True
        return False

# Initialize the knowledge base
kb = KnowledgeBase()

# Facts in the problem
kb.add_fact("American(Robert)")
kb.add_fact("Missile(T1)")
kb.add_fact("Owns(A, T1)")
kb.add_fact("Enemy(A, America)")

# Rules based on the problem
```

```
# 1. Missile(x) implies Weapon(x)
kb.add_rule(Rule(["Missile(T1)"], "Weapon(T1)"))

# 2. Enemy(x, America) implies Hostile(x)
kb.add_rule(Rule(["Enemy(A, America)"], "Hostile(A)"))

# 3. Missile(x) and Owns(A, x) imply Sells(Robert, x, A)
kb.add_rule(Rule(["Missile(T1)", "Owns(A, T1)"], "Sells(Robert, T1, A)"))

# 4. American(p) and Weapon(q) and Sells(p, q, r) and Hostile(r) imply Criminal(p)
kb.add_rule(Rule(["American(Robert)", "Weapon(T1)", "Sells(Robert, T1, A)", "Hostile(A)"],
"Criminal(Robert)"))

# Infer new facts based on the rules
kb.infer()

# Check if Robert is a criminal
if "Criminal(Robert)" in kb.facts:
    print("Conclusion: Robert is a criminal.")
else:
    print("Conclusion: Unable to prove Robert is a criminal.")
```

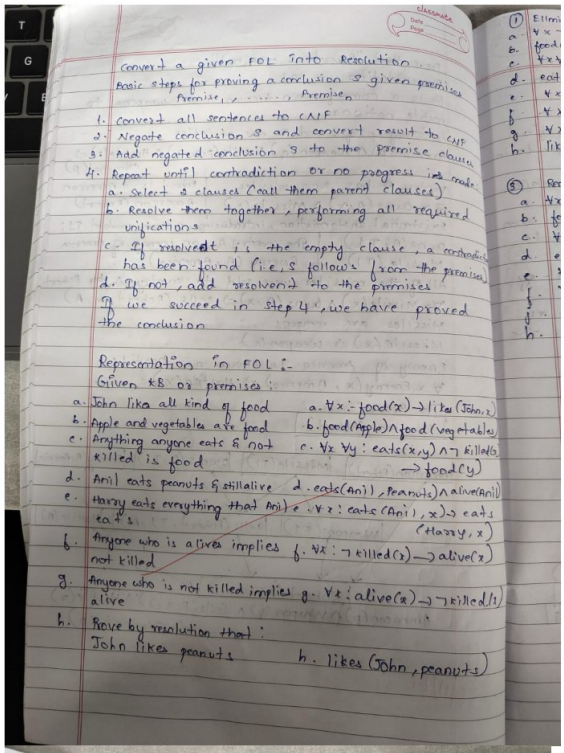
Output :

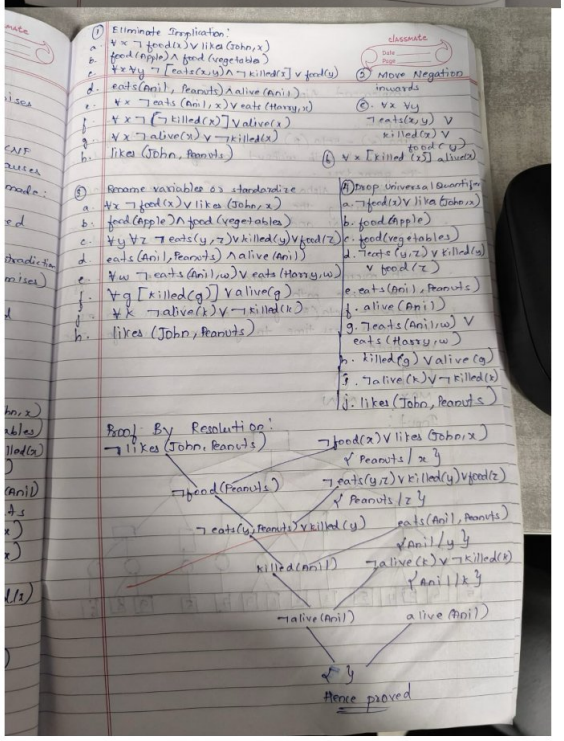
```
Inferred: Weapon(T1)
Inferred: Hostile(A)
Inferred: Sells(Robert, T1, A)
Inferred: Criminal(Robert)
Conclusion: Robert is a criminal.
```

Program 9

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

Algorithm:





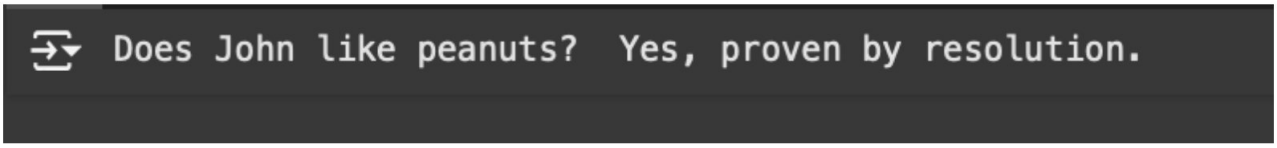
Code:

```
from sympy import symbols, And, Or, Not, Implies, to_cnf

# Define constants (entities in the problem)
John, Anil, Harry, Apple, Vegetables, Peanuts, x, y = symbols('John Anil Harry Apple Vegetables
Peanuts x y')

# Define predicates as symbols (this works as a workaround)
Food = symbols('Food')
Eats = symbols('Eats')
Likes = symbols('Likes')
Alive = symbols('Alive')
Killed = symbols('Killed')

# Knowledge Base (Premises) in First-Order Logic
premises = [
    # 1. John likes all kinds of food: Food(x) → Likes(John, x)
    Implies(Food, Likes),

    # 2. Apples and vegetables are food: Food(Apple) ∧ Food(Vegetables)
    And(Food, Food),

    # 3. Anything anyone eats and is not killed is food: (Eats(y, x) ∧ ¬Killed(y)) → Food(x)
    Implies(And(Eats, Not(Killed)), Food),

    # 4. Anil eats peanuts and is still alive: Eats(Anil, Peanuts) ∧ Alive(Anil)
    And(Eats, Alive),

    # 5. Harry eats everything that Anil eats: Eats(Anil, x) → Eats(Harry, x)
    Implies(Eats, Eats),

    # 6. Anyone who is alive implies not killed: Alive(x) → ¬Killed(x)
    Implies(Alive, Not(Killed)),

    # 7. Anyone who is not killed implies alive: ¬Killed(x) → Alive(x)
    Implies(Not(Killed), Alive),
]

# Negated conclusion to prove: ¬Likes(John, Peanuts)
negated_conclusion = Not(Likes)

# Convert all premises and the negated conclusion to Conjunctive Normal Form (CNF)
cnf_clauses = [to_cnf(premise, simplify=True) for premise in premises]
```

```
cnf_clauses.append(to_cnf(negated_conclusion, simplify=True))

# Function to resolve two clauses
def resolve(clause1, clause2):
    """
    Resolve two CNF clauses to produce resolvents.
    """
    clause1_literals = clause1.args if isinstance(clause1, Or) else [clause1]
    clause2_literals = clause2.args if isinstance(clause2, Or) else [clause2]
    resolvents = []

    for literal in clause1_literals:
        if Not(literal) in clause2_literals:
            # Remove the literal and its negation and combine the rest
            new_clause = Or(
                *[l for l in clause1_literals if l != literal],
                *[l for l in clause2_literals if l != Not(literal)]
            ).simplify()
            resolvents.append(new_clause)

    return resolvents

# Function to perform resolution on the set of CNF clauses
def resolution(cnf_clauses):
    """
    Perform resolution on CNF clauses to check for a contradiction.
    """
    clauses = set(cnf_clauses)
    new_clauses = set()

    while True:
        clause_list = list(clauses)
        for i in range(len(clause_list)):
            for j in range(i + 1, len(clause_list)):
                resolvents = resolve(clause_list[i], clause_list[j])
                if False in resolvents:  # Empty clause found
                    return True  # Contradiction found; proof succeeded
                new_clauses.update(resolvents)

        if new_clauses.issubset(clauses):  # No new information
            return False  # No contradiction; proof failed

        clauses.update(new_clauses)
```

```
# Perform resolution to check if the conclusion follows
result = resolution(cnf_clauses)
print("Does John like peanuts? ", "Yes, proven by resolution." if result else "No, cannot be proven.")
```

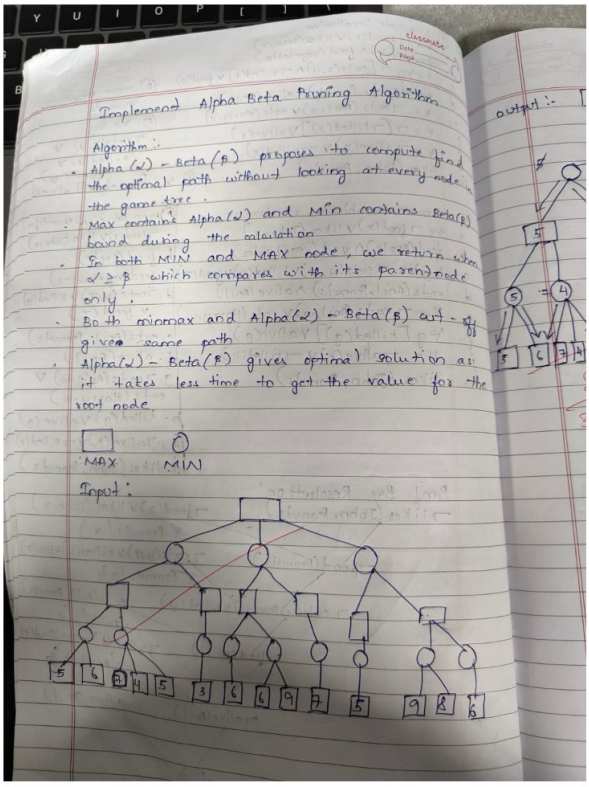
Output:

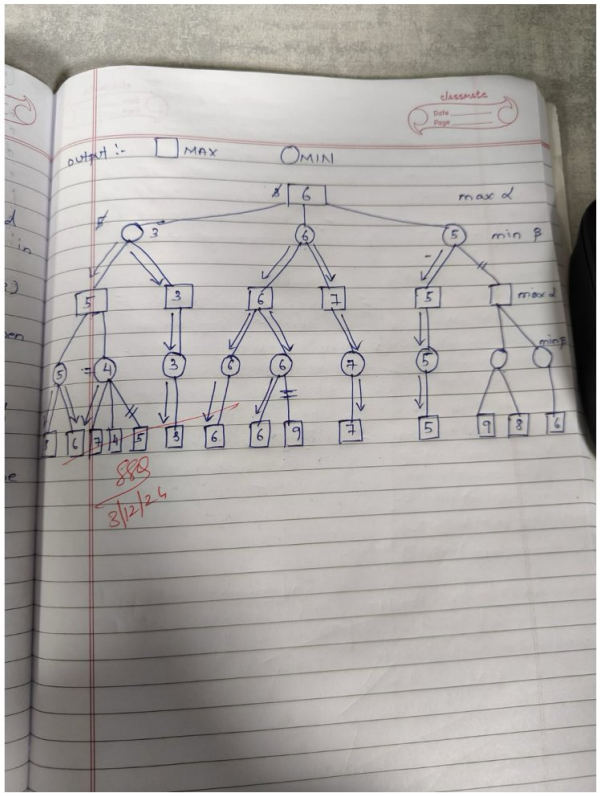


Program 10

Implement Alpha-Beta Pruning.

Algorithm:





Code:

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

    Parameters:
    node (dict): The current node in the decision tree, with the following structure:
        {
            'value': int,
            'left': dict or None,
            'right': dict or None
        }
    depth (int): The current depth in the decision tree.
    is_maximizing (bool): Flag to indicate whether the current player is the maximizing player.

    Returns:
    int: The utility value of the current node.
    """
    # Base case: Leaf node
    if node['left'] is None and node['right'] is None:
        return node['value']

    # Recursive case
    if is_maximizing:
        best_value = -math.inf
        if node['left']:
            best_value = max(best_value, minimax(node['left'], depth + 1, False))
        if node['right']:
            best_value = max(best_value, minimax(node['right'], depth + 1, False))
        return best_value
    else:
        best_value = math.inf
        if node['left']:
            best_value = min(best_value, minimax(node['left'], depth + 1, True))
        if node['right']:
            best_value = min(best_value, minimax(node['right'], depth + 1, True))
        return best_value
```

```
# Example usage
decision_tree = {
    'value': 5,
    'left': {
        'value': 6,
        'left': {
            'value': 7,
            'left': {
                'value': 4,
                'left': None,
                'right': None
            },
            'right': {
                'value': 5,
                'left': None,
                'right': None
            }
        },
        'right': {
            'value': 3,
            'left': {
                'value': 6,
                'left': None,
                'right': None
            },
            'right': {
                'value': 9,
                'left': None,
                'right': None
            }
        }
    },
    'right': {
        'value': 8,
        'left': {
            'value': 7,
            'left': {
```

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

# Find the best move for the maximizing player
best_value = minimax(decision_tree, 0, True)
print(f"The best value for the maximizing player is: {best_value}")
```

Output: