VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Anjana Manoj (1BM23CS038)

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
Aug 2025 to Dec 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 **Anjana Manoj (1BM23CS038),** 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.

Dr. K. R. Mamatha Associate Professor Department of ISE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

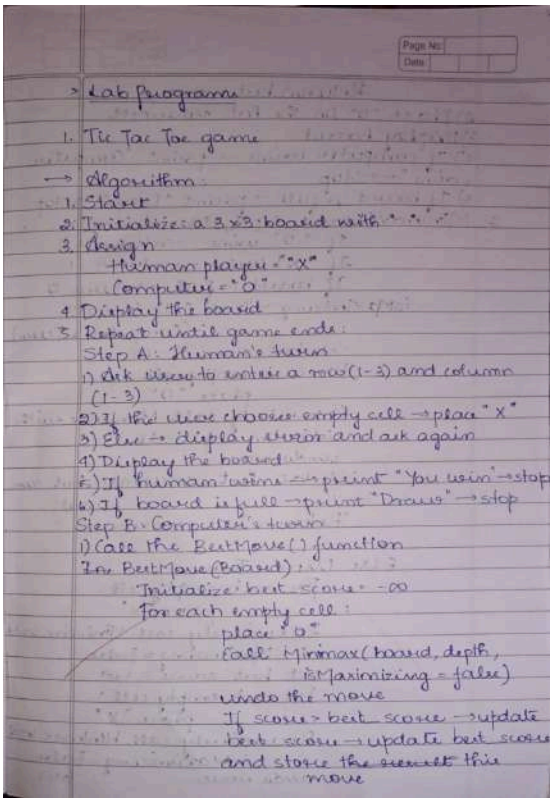
Index

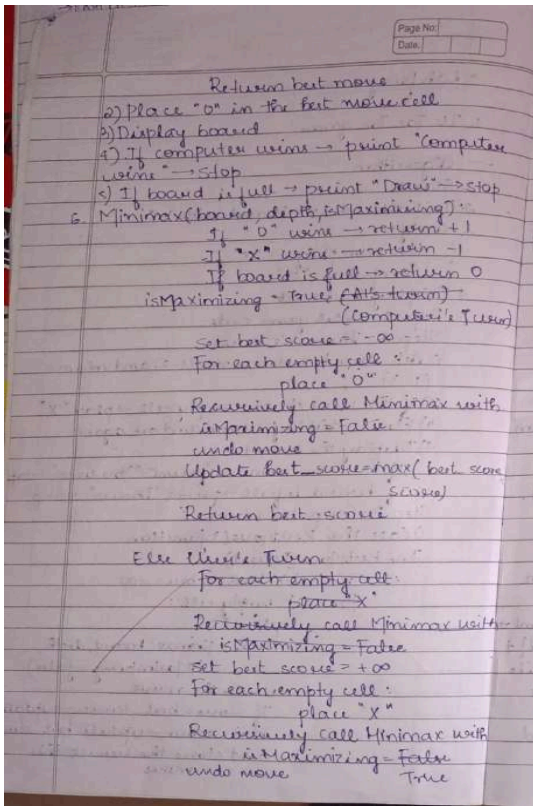
Github Link:
https://github.com/AnjanaManoj-05/AI_LAB

Program 1

Implement Tic –Tac –Toe Game

Algorithm:





Code:

```
def print_board(board):
    print("\nCurrent Board:")
    for row in board:
        print(row)
    print()


def check_winner(board, player):
    for i in range(3):
        if all(cell == player for cell in board[i]):
            return True
        if all(board[j][i] == player for j in range(3)):
            return True


    if all(board[i][i] == player for i in range(3)):
        return True
    if all(board[i][2 - i] == player for i in range(3)):
        return True

    return False


def is_full(board):
    return all(cell != " " for row in board for cell in row)


def tic_tac_toe():
    board = [[" " for _ in range(3)] for _ in range(3)]
    current_player = "X"
    move_count = 0

    print("Tic-Tac-Toe Game (3x3 Matrix Format)\n")
    print_board(board)

    while True:
        try:
            row = int(input(f"Player {current_player}, enter row (0-2): "))
            col = int(input(f"Player {current_player}, enter col (0-2): "))
        except ValueError:
            print("Please enter integers between 0 and 2.")
            continue

        if not (0 <= row <= 2 and 0 <= col <= 2):
```

```
            print("Invalid position. Try again.")
            continue
        if board[row][col] != " ":
            print("Cell already filled. Choose another.")
            continue

        board[row][col] = current_player
        move_count += 1
        print_board(board)

        if check_winner(board, current_player):
            print(f"Player {current_player} wins!")
            break

        if is_full(board):
            print("Game is a draw.")
            break

        current_player = "O" if current_player == "X" else "X"

    print(f"Total moves (cost): {move_count}")

tic_tac_toe()
```

Output case1:

```
Tic-Tac-Toe Game (3x3 Matrix Format)


Current Board:
[' ', ' ', ' ']
[' ', ' ', ' ']
[' ', ' ', ' ']

Player X, enter row (0-2): 1
Player X, enter col (0-2): 1

Current Board:
[' ', ' ', ' ']
[' ', 'X', ' ']
[' ', ' ', ' ']

Player O, enter row (0-2): 0
Player O, enter col (0-2): 2

Current Board:
[' ', ' ', 'O']
[' ', 'X', ' ']
[' ', ' ', ' ']

Player X, enter row (0-2): 1
Player X, enter col (0-2): 0
```

```
Current Board:
[' ', ' ', 'O']
['X', 'X', ' ']
[' ', ' ', ' ']

Player O, enter row (0-2): 2
Player O, enter col (0-2): 1

Current Board:
[' ', ' ', 'O']
['X', 'X', ' ']
[' ', 'O', ' ']

Player X, enter row (0-2): 2
Player X, enter col (0-2): 2

Current Board:
[' ', ' ', 'O']
['X', 'X', ' ']
[' ', 'O', 'X']

Player O, enter row (0-2): 2
Player O, enter col (0-2): 0

Current Board:
[' ', ' ', 'O']
['X', 'X', ' ']
['O', 'O', 'X']

Player X, enter row (0-2): 0
Player X, enter col (0-2): 1

Current Board:
[' ', 'X', 'O']
['X', 'X', ' ']
['O', 'O', 'X']

Player O, enter row (0-2): 1
Player O, enter col (0-2): 2

Current Board:
[' ', 'X', 'O']
['X', 'X', 'O']
['O', 'O', 'X']

Player X, enter row (0-2): 0
Player X, enter col (0-2): 0

Current Board:
['X', 'X', 'O']
['X', 'X', 'O']
['O', 'O', 'X']

Player X wins!
Total moves (cost): 9
```

Output case2:

```
Tic-Tac-Toe Game (3x3 Matrix Format)


Current Board:
[' ', ' ', ' ']
[' ', ' ', ' ']
[' ', ' ', ' ']

Player X, enter row (0-2): 0
Player X, enter col (0-2): 2

Current Board:
[' ', ' ', 'X']
[' ', ' ', ' ']
[' ', ' ', ' ']

Player O, enter row (0-2): 2
Player O, enter col (0-2): 1

Current Board:
[' ', ' ', 'X']
[' ', ' ', ' ']
[' ', 'O', ' ']

Player X, enter row (0-2): 0
Player X, enter col (0-2): 0

Current Board:
['X', ' ', 'X']
[' ', ' ', ' ']
[' ', 'O', ' ']

Player O, enter row (0-2): 0
Player O, enter col (0-2): 1

Current Board:
['X', 'O', 'X']
[' ', ' ', ' ']
[' ', 'O', ' ']

Player X, enter row (0-2): 2
Player X, enter col (0-2): 0

Current Board:
['X', 'O', 'X']
[' ', ' ', ' ']
['X', 'O', ' ']

Player O, enter row (0-2): 1
Player O, enter col (0-2): 1

Current Board:
['X', 'O', 'X']
[' ', 'O', ' ']
```

```
['X', 'O', ' ']

Player O wins!
Total moves (cost): 6
```

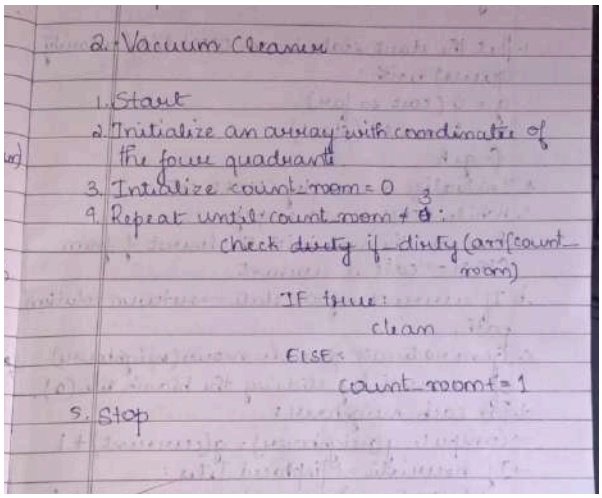
Output case3:

```
Tic-Tac-Toe Game (3x3 Matrix Format):

Current Board:
[' ', ' ', ' ']
[' ', ' ', ' ']
[' ', ' ', ' ']


Player X, enter row (0-2): 1
Player X, enter col (0-2): 0
Current Board:
[' ', ' ', ' ']
['X', ' ', ' ']
[' ', ' ', ' ']

Player O, enter row (0-2): 0
Player O, enter col (0-2): 2

Current Board:
[' ', ' ', 'O']
['X', ' ', ' ']
[' ', ' ', ' ']

Player X, enter row (0-2): 2
Player X, enter col (0-2): 0

Current Board:
[' ', ' ', 'O']
['X', ' ', ' ']
['X', ' ', ' ']

Player O, enter row (0-2): 0
Player O, enter col (0-2): 0

Current Board:
['O', ' ', 'O']
['X', ' ', ' ']
['X', ' ', ' ']

Player X, enter row (0-2): 0
Player X, enter col (0-2): 1

Current Board:
['O', 'X', 'O']
['X', ' ', ' ']
['X', ' ', ' ']

Player O, enter row (0-2): 2
Player O, enter col (0-2): 1
```

```
Current Board:
['O', 'X', 'O']
['X', ' ', ' ']
['X', 'O', ' ']

Player X, enter row (0-2): 2
Player X, enter col (0-2): 2

Current Board:
['O', 'X', 'O']
['X', ' ', ' ']
['X', 'O', 'X']

Player O, enter row (0-2): 1
Player O, enter col (0-2): 1

Current Board:
['O', 'X', 'O']
['X', 'O', ' ']
['X', 'O', 'X']

Player X, enter row (0-2): 1
Player X, enter col (0-2): 2

Current Board:
['O', 'X', 'O']
['X', 'O', 'X']
['X', 'O', 'X']

Game is a draw.
Total moves (cost): 9
```

Implement vacuum cleaner agent

Algorithm:



Code:

```
def vacuum_cleaner()
    A = int(input("Enter state of A (0 for clean, 1 for dirty): "))
    B = int(input("Enter state of B (0 for clean, 1 for dirty): "))
    location = input("Enter location (A or B): ").upper()

    cost = 0
    state = {'A': A, 'B': B}

    if location == 'A':
        if state['A'] == 1:  # If A is dirty
            print("Cleaned A.")
            state['A'] = 0
            cost += 1
        else:
            print("A is clean")

        if state['B'] == 1:  # If B is dirty
            print("Moving vacuum right")
            print("Cleaned B.")
            state['B'] = 0
            cost += 1
            print("Is B clean now? (0 if clean, 1 if dirty):", state['B'])
            print("Is A dirty? (0 if clean, 1 if dirty):", state['A'])
            print("B is clean")
            print("Moving vacuum left")
        else:
            print("Turning vacuum off")

    elif location == 'B':
        if state['B'] == 1:  # If B is dirty
            print("Cleaned B.")
```

```
            state['B'] = 0
            cost += 1
        else:
            print("B is clean")

        if state['A'] == 1:  # If A is dirty
            print("Moving vacuum left")
            print("Cleaned A.")
            state['A'] = 0
            cost += 1
            print("Is A clean now? (0 if clean, 1 if dirty):", state['A'])
            print("Is B dirty? (0 if clean, 1 if dirty):", state['B'])
            print("A is clean")
            print("Moving vacuum right")
        else:
            print("Turning vacuum off")

    print("Cost:", cost)
    print(state)

vacuum_cleaner()
```

OUTPUT Case1:

```
Enter state of A (0 for clean, 1 for dirty): 1
Enter state of B (0 for clean, 1 for dirty): 1
Enter location (A or B): A
Cleaned A.
Moving vacuum right
Cleaned B.
Is B clean now? (0 if clean, 1 if dirty): 0
Is A dirty? (0 if clean, 1 if dirty): 0
B is clean
Moving vacuum left
Cost: 2
{'A': 0, 'B': 0}
```

OUTPUT Case2:

```
Enter state of A (0 for clean, 1 for dirty): 0
Enter state of B (0 for clean, 1 for dirty): 1
Enter location (A or B): A
A is clean
Moving vacuum right
Cleaned B.
Is B clean now? (0 if clean, 1 if dirty): 0
Is A dirty? (0 if clean, 1 if dirty): 0
B is clean
Moving vacuum left
Cost: 1
{'A': 0, 'B': 0}
```

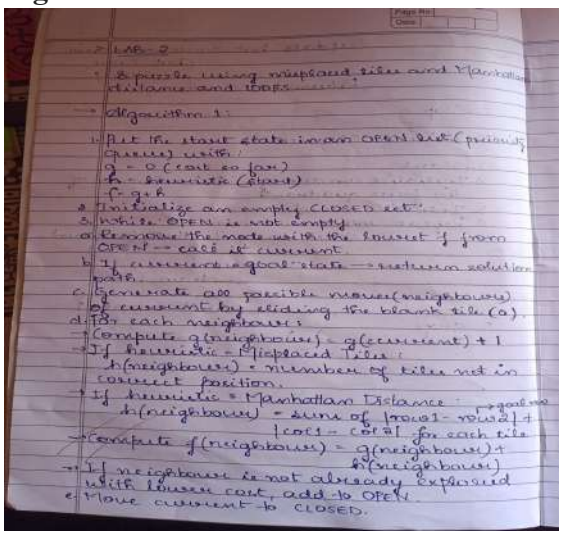
OUTPUT Case3:

```
Enter state of A (0 for clean, 1 for dirty): 0
Enter state of B (0 for clean, 1 for dirty): 0
Enter location (A or B): A
A is clean
Turning vacuum off
Cost: 0
{'A': 0, 'B': 0}
```

Program2

Implement 8 puzzle problems using Depth First Search (DFS)

Algorithm:

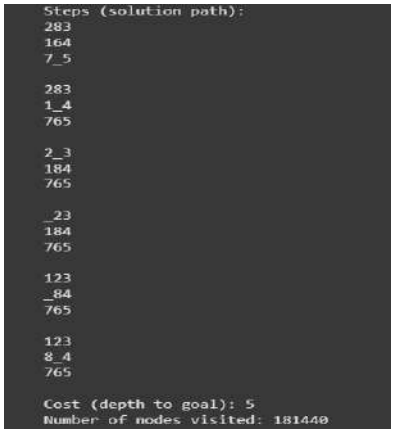


Code:

```
def get_moves(state):
    idx = state.index("_")
    x, y = divmod(idx, 3)
    moves = []
    for dx, dy in [(-1,0),(1,0),(0,-1),(0,1)]:
        nx, ny = x+dx, y+dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            nidx = nx*3 + ny
            lst = list(state)
            lst[idx], lst[nidx] = lst[nidx], lst[idx]
            moves.append("".join(lst))
    return moves

def dfs(start, goal):
    stack = [(start, 0)]
    parent = {start: None}
    visited = {start}
    order = []

    while stack:
        state, cost = stack.pop()
        order.append(state)
        if state == goal:
            path = []
            while state:
                path.append(state)
                state = parent[state]
            path.reverse()
            return path, cost, order, visited
        for move in reversed(get_moves(state)):
            if move not in visited:
                visited.add(move)
                parent[move] = state
                stack.append((move, cost+1))
    return None, -1, order, visited

start = input("Enter initial state (e.g., 54_618732): ")
goal = input("Enter goal state (e.g., 12345678_): ")
path, cost, visited_order, visited_set = dfs(start, goal)

print("Visited nodes (till goal found):")
for v in visited_order:
    for i in range(0, 9, 3):
        print(v[i:i+3])
    print()
    if v == goal:
        break

print("Steps (solution path):")
for p in path:
    for i in range(0, 9, 3):
```

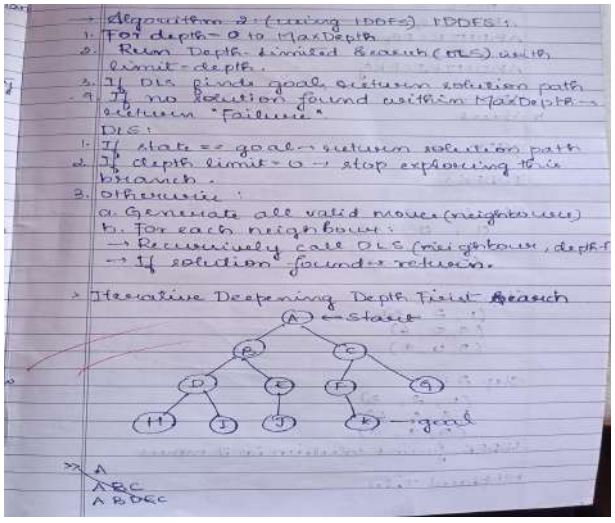
```
        print(p[i:i+3])
    print()

print("Cost (depth to goal):", cost)
print("Number of nodes visited:", len(visited_set))
```

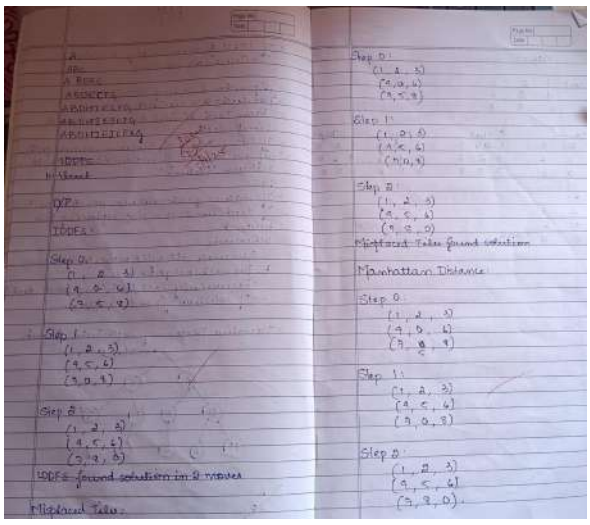
Output:



Implement Iterative deepening search algorithm

Algorithm:





Code:

```
def get_neighbors(state):
    neighbors = []
    idx = state.index("0")
```

```
    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    x, y = divmod(idx, 3)

    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_idx = nx * 3 + ny
            state_list = list(state)
            state_list[idx], state_list[new_idx] = state_list[new_idx], state_list[idx]
            neighbors.append("".join(state_list))
    return neighbors

def dfs_limit(start_state, goal_state, limit):
    stack = [(start_state, 0)]
    visited = set()
    parent = {start_state: None}
    path = []

    while stack:
        current_state, depth = stack.pop()

        if current_state == goal_state:
            while current_state:
                path.append(current_state)
                current_state = parent[current_state]
            return path[::-1]

        if depth < limit and current_state not in visited:
            visited.add(current_state)
            neighbors = get_neighbors(current_state)
            neighbors.reverse()  # Maintain consistent exploration order
            for neighbor in neighbors:
                if neighbor not in visited:
                    parent[neighbor] = current_state
                    stack.append((neighbor, depth + 1))
    return None

def iddfs(start_state, goal_state, max_depth):
    for limit in range(max_depth + 1):
        print(f"Searching with depth limit: {limit}")
        solution = dfs_limit(start_state, goal_state, limit)
        if solution:
            return solution
    return None

print("Enter the initial state (enter 3 digits per row, separated by spaces, 0 for empty):")
initial_state_rows = []
for i in range(3):
    row = input(f"Row {i+1}: ").split()
    initial_state_rows.extend(row)
initial_state = "".join(initial_state_rows)
```

```
print("\nEnter the goal state (enter 3 digits per row, separated by spaces, 0 for empty):")
goal_state_rows = []
for i in range(3):
    row = input(f"Row {i+1}: ").split()
    goal_state_rows.extend(row)
goal_state = "".join(goal_state_rows)

max_depth = 50

solution = iddfs(initial_state, goal_state, max_depth)

if solution:
    print("\nIDDFS solution path:")
    for s in solution:
        print(s[:3])
        print(s[3:6])
        print(s[6:])
        print()
else:
    print(f"\nNo solution found within the maximum depth of {max_depth}.")
```

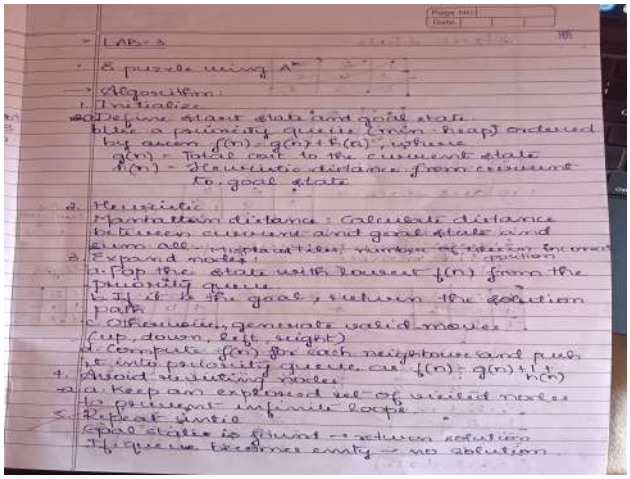
Output:

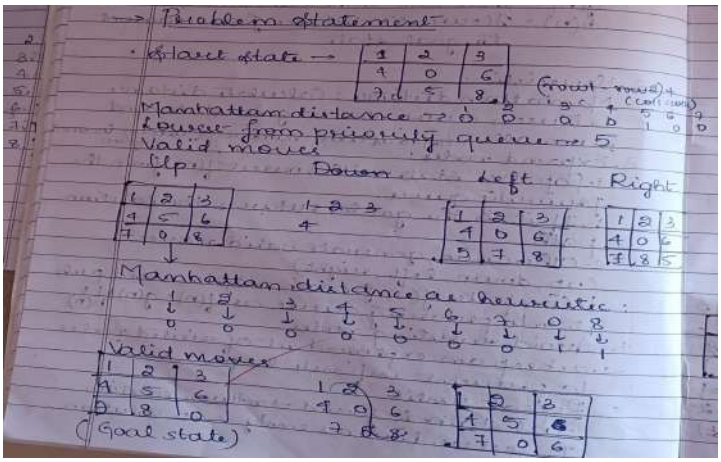
```
Sharada Koundinya 1BM23CS310
Enter the initial state (enter 3 digits per row, separated by spaces, 0 for empty):
Row 1: 283
Row 2: 164
Row 3: 705

Enter the goal state (enter 3 digits per row, separated by spaces, 0 for empty):
Row 1: 123
Row 2: 804
Row 3: 765
Searching with depth limit: 0
Searching with depth limit: 1
Searching with depth limit: 2
Searching with depth limit: 3
Searching with depth limit: 4
Searching with depth limit: 5

IDDFS solution path:
283
164
705

283
104
765

203
184
765

023
184
765

123
084
765

123
804
765
```

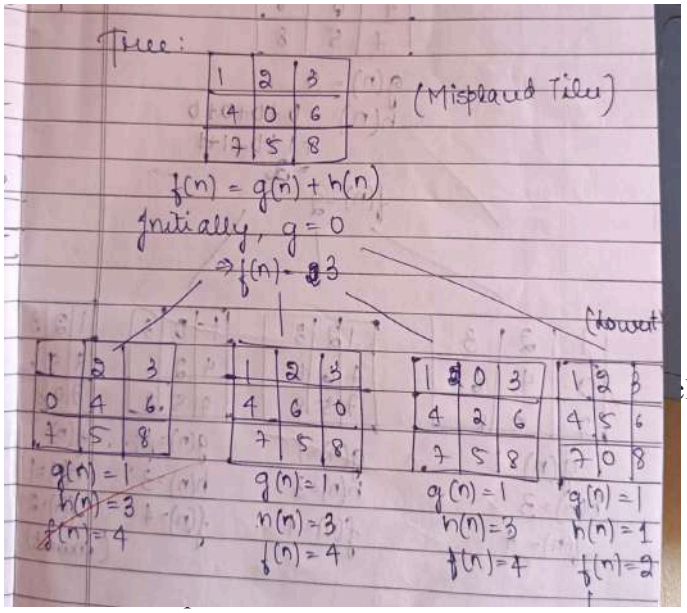
Program3

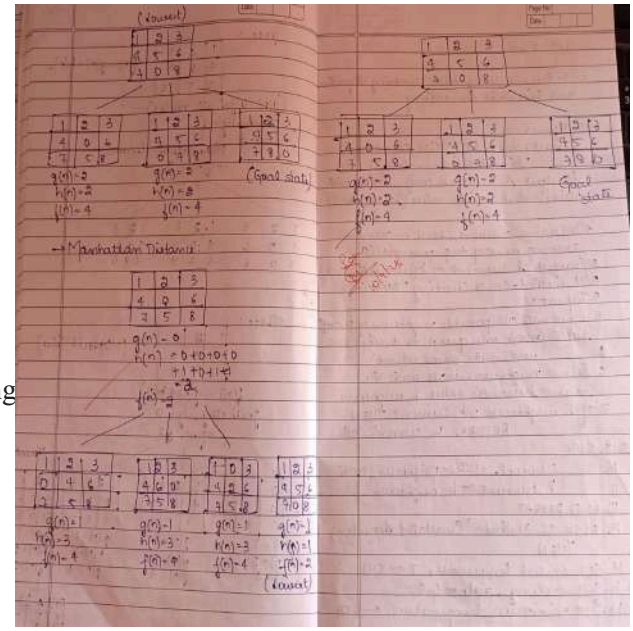
Implement A* search algorithm

Algorithm:









cluding

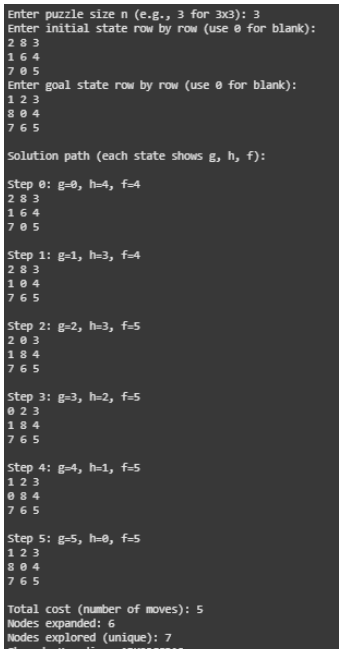
```
    return misplaced

def find_blank(board):
    n = len(board)
    for i in range(n):
        for j in range(n):
            if board[i][j] == 0:
                return i, j
    raise ValueError("Board does not contain a blank tile (0)")

def neighbors(board):
    """Generate neighboring boards by sliding one tile into the blank."""
    n = len(board)
    x, y = find_blank(board)
    dirs = [(0,1),(0,-1),(1,0),(-1,0)]
    res = []
    for dx, dy in dirs:
```

```
            nx, ny = x + dx, y + dy
            if 0 <= nx < n and 0 <= ny < n:
                b = [list(row) for row in board]
                b[x][y], b[nx][ny] = b[nx][ny], b[x][y]
                res.append(tuple(tuple(row) for row in b))
    return res

def flatten(board):
    return [x for row in board for x in row]

def inversion_count(seq):
    arr = [x for x in seq if x != 0]
    inv = 0
    for i in range(len(arr)):
        for j in range(i+1, len(arr)):
            if arr[i] > arr[j]:
                inv += 1
    return inv

def blank_row_from_bottom(board):
    n = len(board)
    for i in range(n):
        for j in range(n):
            if board[i][j] == 0:
                return n - i
    raise ValueError("Board does not contain a blank tile (0)")

def is_solvable(start, goal):
    """General n-puzzle solvability test (odd/even width)."""
    n = len(start)
    start_flat = flatten(start)
    goal_flat = flatten(goal)

    pos = {val: idx for idx, val in enumerate(goal_flat)}
    start_perm = [pos[val] for val in start_flat]

    inv = inversion_count(start_perm)

    if n % 2 == 1:
        # odd grid: inversions parity must be even
        return inv % 2 == 0
    else:
        # even grid: blank row from bottom parity matters
        blank_row = blank_row_from_bottom(start)
        goal_blank_row = blank_row_from_bottom(goal)
        # When using relative permutation to goal, parity of blank rows must match
        return (inv + blank_row) % 2 == (0 + goal_blank_row) % 2

def reconstruct_path(came_from, current):
    path = [current]
    while current in came_from:
        current = came_from[current]
```

```
        path.append(current)
    path.reverse()
    return path

def a_star_misplaced(start, goal):
    start = tuple(tuple(row) for row in start)
    goal = tuple(tuple(row) for row in goal)

    if len(start) != len(start[0]) or len(goal) != len(goal[0]) or len(start) != len(goal):
        raise ValueError("Initial and goal must be square boards of the same size.")

    start_vals = sorted(flatten(start))
    goal_vals = sorted(flatten(goal))
    if start_vals != goal_vals:
        raise ValueError("Initial and goal must contain the same set of tiles.")

    if not is_solvable(start, goal):
        return None, None, 0, 0  # unsolvable

    counter = count()  # tie-breaker

    h0 = misplaced_heuristic(start, goal)
    g_score = {start: 0}
    f0 = h0

    open_heap = [(f0, next(counter), start)]
    open_set = {start: f0}
    closed = set()
    came_from = {}

    expansions = 0

    while open_heap:
        _, _, current = heapq.heappop(open_heap)
        if current in closed:
            continue
        closed.add(current)

        if current == goal:
            path = reconstruct_path(came_from, current)
            return path, g_score[current], expansions, len(closed)

        expansions += 1

        for nb in neighbors(current):
            tentative_g = g_score[current] + 1
            if nb in closed:
                continue
            if nb not in g_score or tentative_g < g_score[nb]:
                came_from[nb] = current
                g_score[nb] = tentative_g
                h = misplaced_heuristic(nb, goal)
```

```
            f = tentative_g + h
            if nb not in open_set or f < open_set[nb]:
                heapq.heappush(open_heap, (f, next(counter), nb))
                open_set[nb] = f

    return None, None, expansions, len(closed)

def read_board(n, prompt):
    print(prompt)
    board = []
    for i in range(n):
        row = list(map(int, input().split()))
        if len(row) != n:
            raise ValueError(f"Row {i+1} must contain exactly {n} integers.")
        board.append(row)
    return board

def print_board(board):
    for row in board:
        print(" ".join(f"{x}" for x in row))

def main():
    try:
        n = int(input("Enter puzzle size n (e.g., 3 for 3x3): ").strip())
        initial = read_board(n, "Enter initial state row by row (use 0 for blank):")
        goal = read_board(n, "Enter goal state row by row (use 0 for blank):")

        result = a_star_misplaced(initial, goal)
        path, cost, expansions, explored = result

        if path is None:
            print("No solution (unsolvable with given start/goal).")
            return

        print("\nSolution path (each state shows g, h, f):\n")
        for idx, state in enumerate(path):
            g = idx  # each step costs 1
            h = misplaced_heuristic(state, tuple(tuple(r) for r in goal))
            f = g + h
            print(f"Step {idx}: g={g}, h={h}, f={f}")
            print_board(state)
            print()

        print(f"Total cost (number of moves): {cost}")
        print(f"Nodes expanded: {expansions}")
        print(f"Nodes explored (unique): {explored}")

    except Exception as e:
        print("Error:", e)

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

Output:



Code:

```
#MANHATTAN DISTANCE
import heapq
from itertools import count

def misplaced_heuristic(board, goal):
    misplaced = 0
    n = len(board)
    for i in range(n):
        for j in range(n):
            if board[i][j] != 0 and board[i][j] != goal[i][j]:
                misplaced += 1
    return misplaced

def manhattan_heuristic(board, goal):
    n = len(board)
    # Map goal positions for each tile
    goal_pos = {}
    for i in range(n):
        for j in range(n):
            goal_pos[goal[i][j]] = (i, j)

    dist = 0
    for i in range(n):
        for j in range(n):
            val = board[i][j]
            if val != 0:
                gi, gj = goal_pos[val]
```

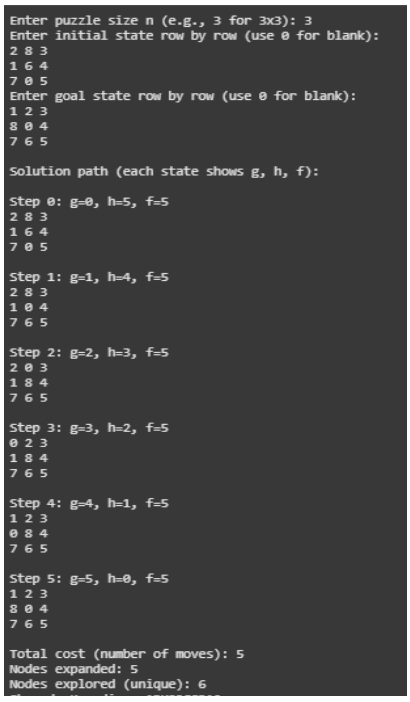
```
                dist += abs(i - gi) + abs(j - gj)
    return dist

def find_blank(board):
    n = len(board)
    for i in range(n):
        for j in range(n):
            if board[i][j] == 0:
                return i, j
    raise ValueError("Board does not contain a blank tile (0)")

def neighbors(board):
    n = len(board)
    x, y = find_blank(board)
    dirs = [(0,1),(0,-1),(1,0),(-1,0)]
    res = []
    for dx, dy in dirs:
        nx, ny = x + dx, y + dy
        if 0 <= nx < n and 0 <= ny < n:
            b = [list(row) for row in board]
            b[x][y], b[nx][ny] = b[nx][ny], b[x][y]
            res.append(tuple(tuple(row) for row in b))
    return res

def flatten(board):
    return [x for row in board for x in row]

def inversion_count(seq):
    arr = [x for x in seq if x != 0]
    inv = 0
    for i in range(len(arr)):
        for j in range(i+1, len(arr)):
            if arr[i] > arr[j]:
                inv += 1
    return inv

def blank_row_from_bottom(board):
    n = len(board)
    for i in range(n):
        for j in range(n):
            if board[i][j] == 0:
                return n - i
    raise ValueError("Board does not contain a blank tile (0)")

def is_solvable(start, goal):
    n = len(start)
    start_flat = flatten(start)
    goal_flat = flatten(goal)

    pos = {val: idx for idx, val in enumerate(goal_flat)}
    start_perm = [pos[val] for val in start_flat]
```

```
    inv = inversion_count(start_perm)

    if n % 2 == 1:
        return inv % 2 == 0
    else:
        blank_row = blank_row_from_bottom(start)
        goal_blank_row = blank_row_from_bottom(goal)
        return (inv + blank_row) % 2 == (0 + goal_blank_row) % 2

def reconstruct_path(came_from, current):
    path = [current]
    while current in came_from:
        current = came_from[current]
        path.append(current)
    path.reverse()
    return path

def a_star_manhattan(start, goal):
    start = tuple(tuple(row) for row in start)
    goal = tuple(tuple(row) for row in goal)

    if len(start) != len(start[0]) or len(goal) != len(goal[0]) or len(start) != len(goal):
        raise ValueError("Initial and goal must be square boards of the same size.")

    start_vals = sorted(flatten(start))
    goal_vals = sorted(flatten(goal))
    if start_vals != goal_vals:
        raise ValueError("Initial and goal must contain the same set of tiles.")

    if not is_solvable(start, goal):
        return None, None, 0, 0

    counter = count()
    h0 = manhattan_heuristic(start, goal)
    g_score = {start: 0}
    f0 = h0

    open_heap = [(f0, next(counter), start)]
    open_set = {start: f0}
    closed = set()
    came_from = {}

    expansions = 0

    while open_heap:
        _, _, current = heapq.heappop(open_heap)
        if current in closed:
            continue
        closed.add(current)

        if current == goal:
            path = reconstruct_path(came_from, current)
```

```
            return path, g_score[current], expansions, len(closed)

        expansions += 1

        for nb in neighbors(current):
            tentative_g = g_score[current] + 1
            if nb in closed:
                continue
            if nb not in g_score or tentative_g < g_score[nb]:
                came_from[nb] = current
                g_score[nb] = tentative_g
                h = manhattan_heuristic(nb, goal)
                f = tentative_g + h
                if nb not in open_set or f < open_set[nb]:
                    heapq.heappush(open_heap, (f, next(counter), nb))
                    open_set[nb] = f

    return None, None, expansions, len(closed)

def read_board(n, prompt):
    print(prompt)
    board = []
    for i in range(n):
        row = list(map(int, input().split()))
        if len(row) != n:
            raise ValueError(f"Row {i+1} must contain exactly {n} integers.")
        board.append(row)
    return board

def print_board(board):
    for row in board:
        print(" ".join(f"{x}" for x in row))

def main():
    try:
        n = int(input("Enter puzzle size n (e.g., 3 for 3x3): ").strip())
        initial = read_board(n, "Enter initial state row by row (use 0 for blank):")
        goal = read_board(n, "Enter goal state row by row (use 0 for blank):")

        result = a_star_manhattan(initial, goal)
        path, cost, expansions, explored = result

        if path is None:
            print("No solution (unsolvable with given start/goal).")
            return

        print("\nSolution path (each state shows g, h, f):\n")
        for idx, state in enumerate(path):
            g = idx
            h = manhattan_heuristic(state, tuple(tuple(r) for r in goal))
            f = g + h
            print(f"Step {idx}: g={g}, h={h}, f={f}")
```

```
            print_board(state)
            print()

        print(f"Total cost (number of moves): {cost}")
        print(f"Nodes expanded: {expansions}")
        print(f"Nodes explored (unique): {explored}")

    except Exception as e:
        print("Error:", e)

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

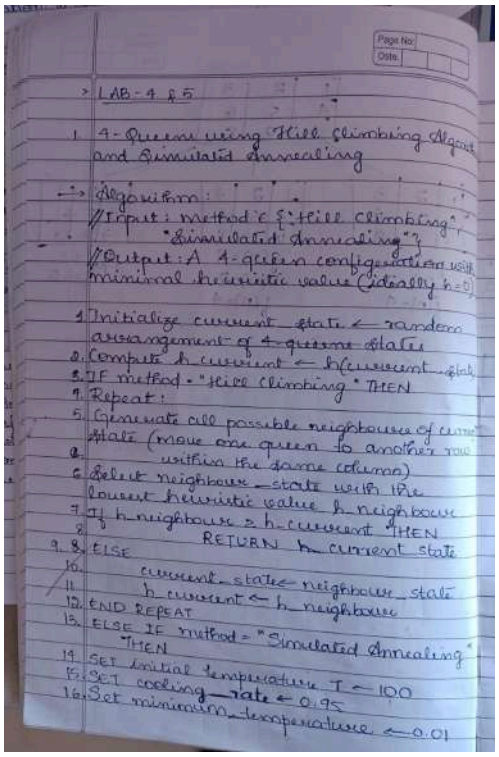
Output:

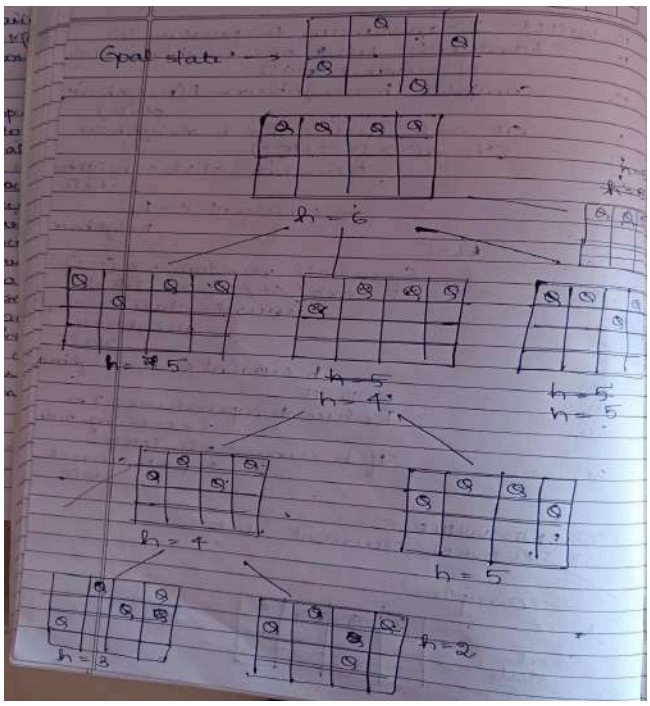


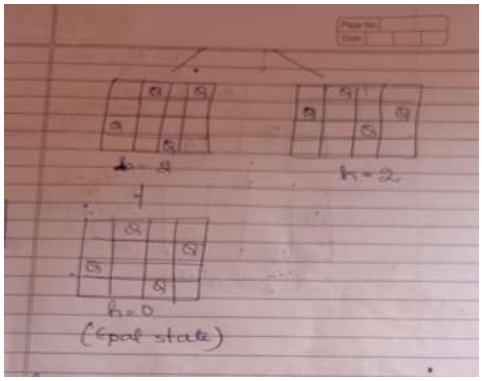
Program4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:







Code:

```
def calculate_cost(state):
    cost = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                cost += 1
    return cost

def generate_neighbors(state):
    neighbors = []
    n = len(state)
    for col in range(n):
```

```
        for row in range(n):
            if state[col] != row:  # move queen
                new_state = list(state)
                new_state[col] = row
                neighbors.append(new_state)
    return neighbors

def hill_climbing(initial_state):
    current = initial_state
    current_cost = calculate_cost(current)
    step = 0

    print(f"Step {step}: State = {current}, Cost = {current_cost}")

    while True:
        neighbors = generate_neighbors(current)
        neighbor_costs = [(n, calculate_cost(n)) for n in neighbors]

        # Print state space for this step
        print("\nNeighbors and their costs:")
        for n, c in neighbor_costs:
            print(f"   {n} -> Cost = {c}")

        # Pick the best neighbor (lowest cost)
        best_neighbor, best_cost = min(neighbor_costs, key=lambda x: x[1])

        if best_cost >= current_cost:
            break

        step += 1
        current, current_cost = best_neighbor, best_cost
        print(f"\nStep {step}: Move to {current}, Cost = {current_cost}")

        if current_cost == 0:
            print("\nGoal reached! Solution found.")
            break

initial_state = [3, 1, 2, 0]

hill_climbing(initial_state)

]
```

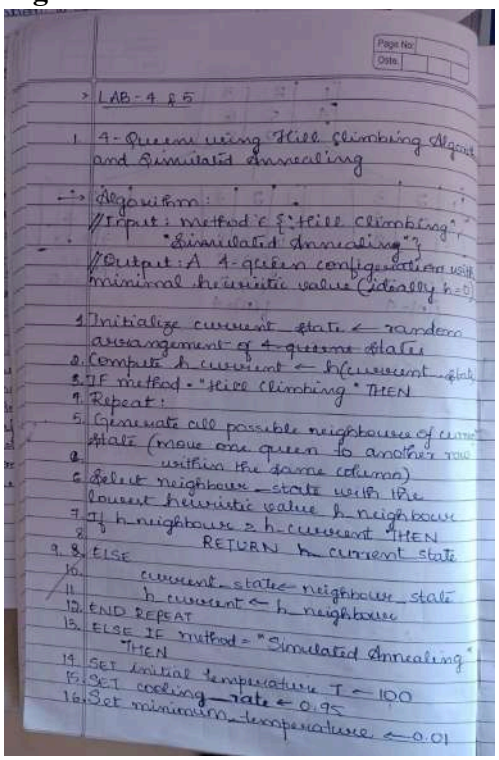
Output:

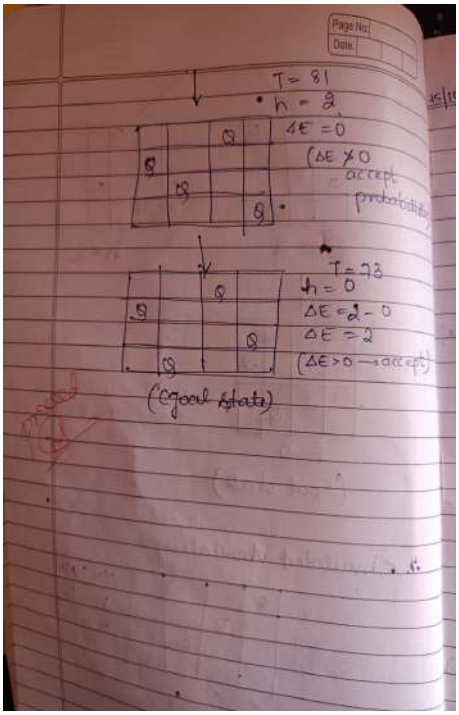
```
Week 7
Step 0: State = [3, 1, 2, 0], Cost = 2

Neighbors and their costs:
   [0, 1, 2, 0] -> Cost = 4
   [1, 1, 2, 0] -> Cost = 2
   [2, 1, 2, 0] -> Cost = 3
   [3, 0, 2, 0] -> Cost = 2
   [3, 2, 2, 0] -> Cost = 4
   [3, 3, 2, 0] -> Cost = 3
   [3, 1, 0, 0] -> Cost = 3
   [3, 1, 1, 0] -> Cost = 4
   [3, 1, 3, 0] -> Cost = 2
   [3, 1, 2, 1] -> Cost = 3
   [3, 1, 2, 2] -> Cost = 2
   [3, 1, 2, 3] -> Cost = 4
```

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:





Code:

```
import random
import math
```

```
def calculate_cost(state):
    cost = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                cost += 1
    return cost

def get_random_neighbor(state):
    n = len(state)
    new_state = list(state)
    col = random.randint(0, n - 1)
    row = random.randint(0, n - 1)
    new_state[col] = row
    return new_state

def simulated_annealing(n=8, max_iterations=10000, initial_temp=100.0, cooling_rate=0.99):

    current = [random.randint(0, n - 1) for _ in range(n)]
    current_cost = calculate_cost(current)
    best = current
    best_cost = current_cost
    temperature = initial_temp

    for _ in range(max_iterations):
        if current_cost == 0:
            break

        neighbor = get_random_neighbor(current)
        neighbor_cost = calculate_cost(neighbor)
        delta = neighbor_cost - current_cost

        if delta < 0 or random.random() < math.exp(-delta / temperature):
            current, current_cost = neighbor, neighbor_cost

            if current_cost < best_cost:
                best, best_cost = current, current_cost

        temperature *= cooling_rate
        if temperature < 1e-6:
            break

    return best, best_cost

best_state, best_cost = simulated_annealing()

print("The best position found:", best_state)
print("cost =", best_cost)
```

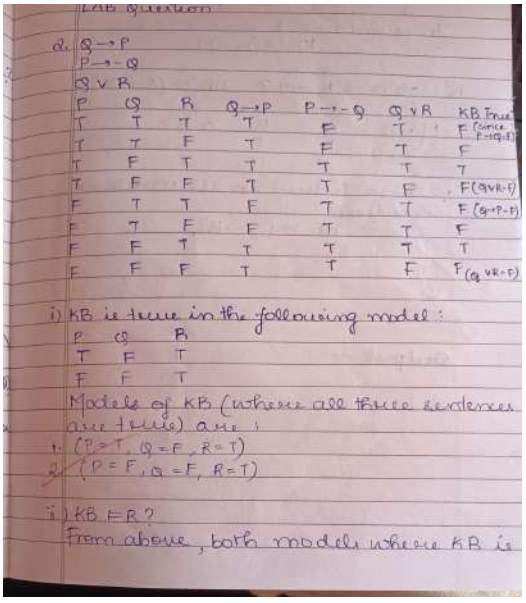
Output:

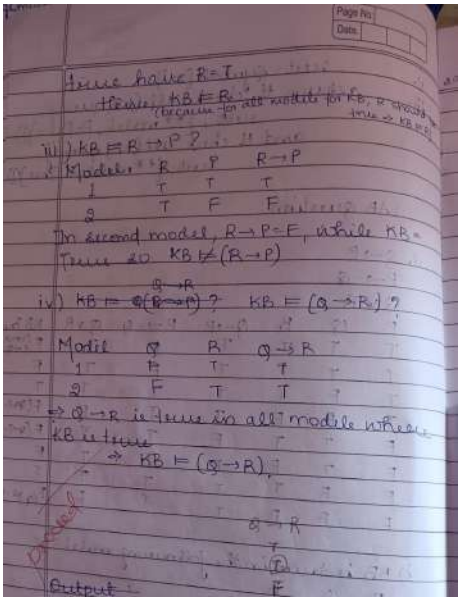
```
The best position found: [5, 2, 6, 1, 7, 4, 0, 3]
cost = 0
```

Program 6

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

Algorithm:





Code:

```
import itertools

def eval_expr(expr, model):
    try:
        return eval(expr, {}, model)
    except:
        return False

def tt_entails(KB, query):
    symbols = sorted(set([ch for ch in KB + query if ch.isalpha()]))

    print("\nTruth Table:")
    print(" | ".join(symbols) + " | KB | Query")
    print("-" * (6 * len(symbols) + 20))

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

```
        kb_val = eval_expr(KB, model)
        query_val = eval_expr(query, model)

        row = " | ".join(["T" if model[s] else "F" for s in symbols])
        print(f"{row} |  {kb_val}  |  {query_val}")

        if kb_val and not query_val:
            entails = False

    return entails

KB = input("Enter Knowledge Base (use &, |, ~ for AND, OR, NOT): ")
query = input("Enter Query: ")

result = tt_entails(KB, query)

print("\nResult:")
if result:
    print("KB entails Query (True in all cases).")
else:
    print("KB does NOT entail Query.")
```

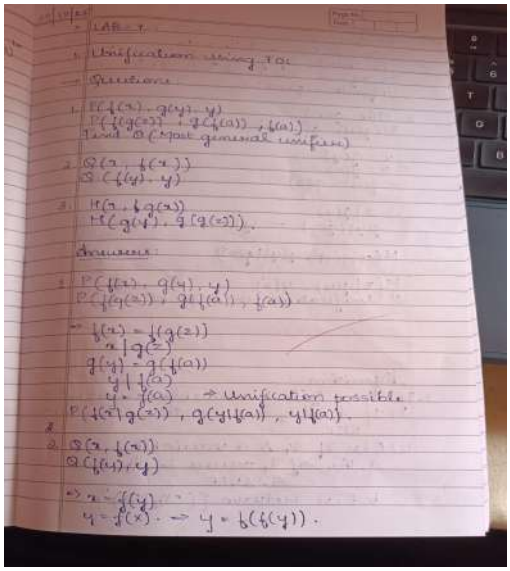
Output:

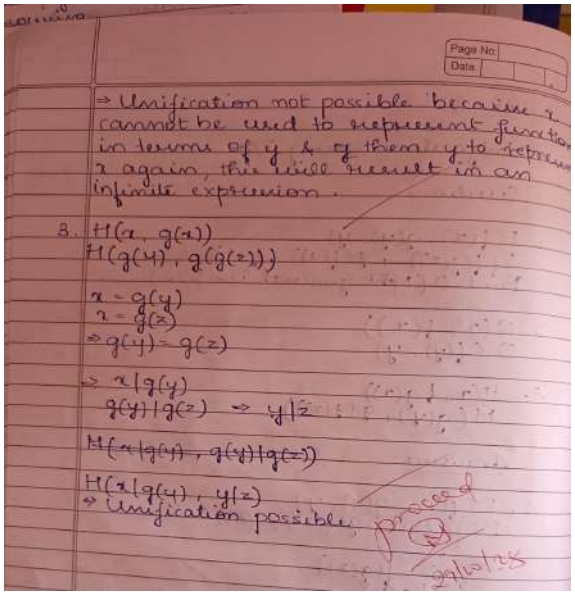
```
Enter Knowledge Base (use &, |, ~ for AND, OR, NOT): (A|C)&(B|~C)
Enter Query:  A|B

Truth Table:
A | B | C | KB | Query
-------------------------------------
F | F | F |  0  |  False
F | F | T |  0  |  False
F | T | F |  0  |  True
F | T | T |  1  |  True
T | F | F |  1  |  True
T | F | T |  0  |  True
T | T | F |  1  |  True
T | T | T |  1  |  True

Result:
KB entails Query (True in all cases).
```

Program 7

Implement unification in first order logic

Algorithm:





Code:

```
def occurs_check(var, term, subst):
    if var == term:
        return True
    elif isinstance(term, tuple):
        return any(occurs_check(var, t, subst) for t in term)
    elif term in subst:
        return occurs_check(var, subst[term], subst)
    return False

def unify(x, y, subst):
    if subst is None:
        return None
    elif x == y:
        return subst
    elif isinstance(x, str) and x.isupper():
        return unify_var(x, y, subst)
    elif isinstance(y, str) and y.isupper():
        return unify_var(y, x, subst)
    elif isinstance(x, tuple) and isinstance(y, tuple):
        if x[0] != y[0] or len(x) != len(y):
            return None
        for a, b in zip(x[1:], y[1:]):
            subst = unify(a, b, subst)
            if subst is None:
                return None
        return subst
```

```
    else:
        return None

def unify_var(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, subst):
        return None
    else:
        subst[var] = x
        return subst

def parse_expr(s):
    s = s.replace(" ", "")
    if '(' not in s:
        return s
    name_end = s.index('(')
    name = s[:name_end]
    args = []
    depth = 0
    current = ""
    for c in s[name_end+1:-1]:
        if c == ',' and depth == 0:
            args.append(parse_expr(current))
            current = ""
        else:
            if c == '(':
                depth += 1
            elif c == ')':
                depth -= 1
            current += c
    if current:
        args.append(parse_expr(current))
    return tuple([name] + args)

def expr_to_str(expr):
    if isinstance(expr, tuple):
        return expr[0] + "(" + ",".join(expr_to_str(e) for e in expr[1:]) + ")"
    else:
        return expr

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

expr1 = parse_expr(expr1_input)
expr2 = parse_expr(expr2_input)

subst = unify(expr1, expr2, {})

if subst:
```

```
    formatted_subst = {var: expr_to_str(val) for var, val in subst.items()}
else:
    formatted_subst = None

print("Most General Unifier (MGU):", formatted_subst)
```

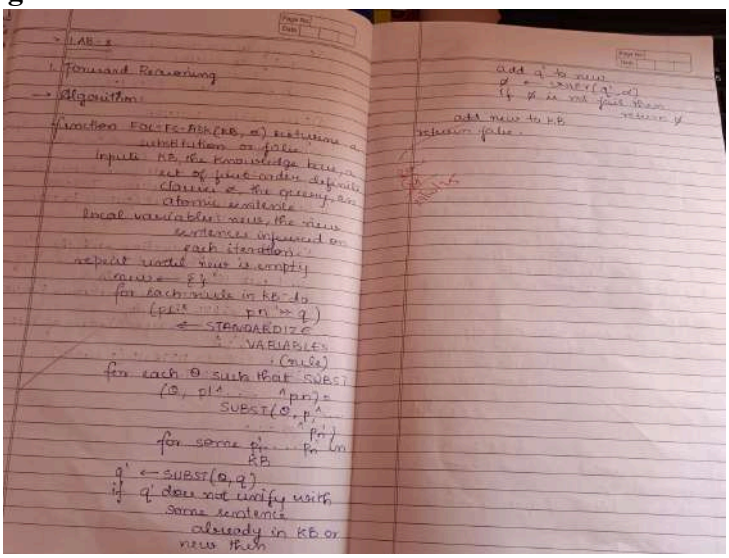
Output:

```
Enter first expression: p(b,X,f(g(Z)))
Enter second expression: p(Z,f(Y),f(Y))
Most General Unifier (MGU): {'Z': 'b', 'X': 'f(Y)', 'Y': 'g(Z)'}
```

Program 8

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

Algorithm:



Code:

```
facts = {
    'American(Robert)': True,
    'Hostile(A)': True,
    'Sells_Weapons(Robert, A)': True
}

If American(X) and Hostile(Y) and Sells_Weapons(X, Y), then Crime(X)
def forward_reasoning(facts):
```

If American(X) and Hostile(Y) and Sells_Weapons(X, Y), then Crime(X)

```
        if facts.get('American(Robert)', False) and facts.get('Hostile(A)', False) and facts.get('Sells_Weapons(Robert, A)', False):
    facts['Crime(Robert)'] = True

forward_reasoning(facts)

if facts.get('Crime(Robert)', False):
   print("Robert is a criminal.")
else:
   print("Robert is not a criminal.")
```

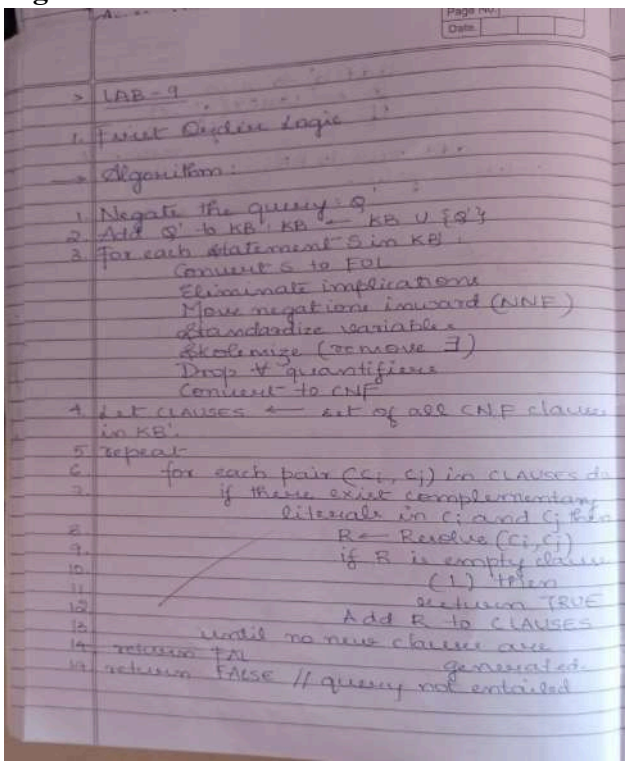
Output:

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 itertools import combinations

def get_literals(clause):
```

```
    """Split a clause into literals (handles OR operator)."""
    return [literal.strip() for literal in clause.split("OR")]

def negate_literal(literal):
    """Return the negation of a literal."""
    literal = literal.strip()
    if literal.startswith("~"):
        return literal[1:]
    else:
        return "~" + literal

def resolve(ci, cj):
    """Try to resolve two clauses and return resolvents."""
    resolvents = set()
    lits_i = get_literals(ci)
    lits_j = get_literals(cj)

    for li in lits_i:
        for lj in lits_j:
            if li == negate_literal(lj):
                new_clause = set(lits_i + lits_j)
                new_clause.discard(li)
                new_clause.discard(lj)
                if not new_clause:
                    resolvents.add("NIL")
                else:
                    resolvents.add(" OR ".join(sorted(new_clause)))
    return resolvents


def resolution_algorithm(kb, query):
    print("\n========== RESOLUTION IN FIRST ORDER LOGIC ==========\n")

    # Negate the query and add it to KB
    negated_query = negate_literal(query)
    print(f"Original Query     : {query}")
    print(f"Negated Query      : {negated_query}")
    clauses = kb + [negated_query]
    print("\nInitial Knowledge Base:")
    for i, c in enumerate(clauses, 1):
        print(f"  Clause {i}: {c}")

    print("\n========== Resolution Steps ==========\n")
    new = set()
    step = 1
    while True:
        pairs = list(combinations(clauses, 2))
```

```
        for (ci, cj) in pairs:
            resolvents = resolve(ci, cj)
            for res in resolvents:
                if res == "NIL":
                    print(f"Step {step}: Resolving [{ci}] and [{cj}] gives NIL ⇒ Contradiction found!")
                    print("\n✅ Query is PROVED by Resolution.\n")
                    return True
                if res not in clauses and res not in new:
                    print(f"Step {step}: Resolving [{ci}] and [{cj}] ⇒ New clause: {res}")
                    new.add(res)
                    step += 1
        if new.issubset(set(clauses)):
            print("\n❌ No new clauses can be generated.")
            print("Query CANNOT be proved by Resolution.\n")
            return False
        for c in new:
            if c not in clauses:
                clauses.append(c)


# -------------------------
# Main Program (User Input)
# -------------------------

if __name__ == "__main__":
    print("=== Resolution in First Order Logic (Simplified) ===")
    n = int(input("Enter number of clauses in Knowledge Base: "))
    kb = []
    for i in range(n):
        clause = input(f"Enter clause {i+1}: ").strip()
        kb.append(clause)

    query = input("Enter the query to prove: ").strip()

    # Run Resolution Algorithm
    resolution_algorithm(kb, query)
```

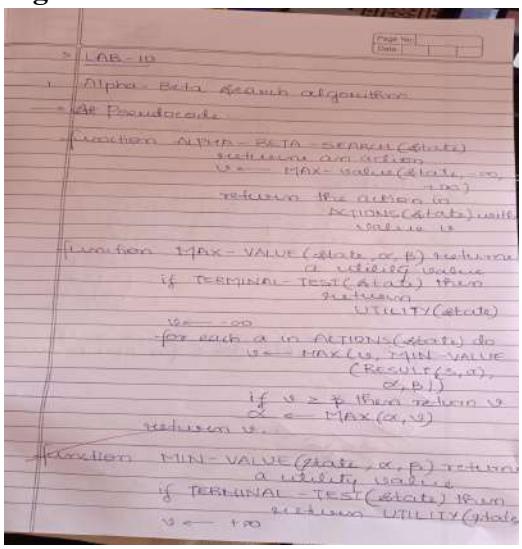
Output:

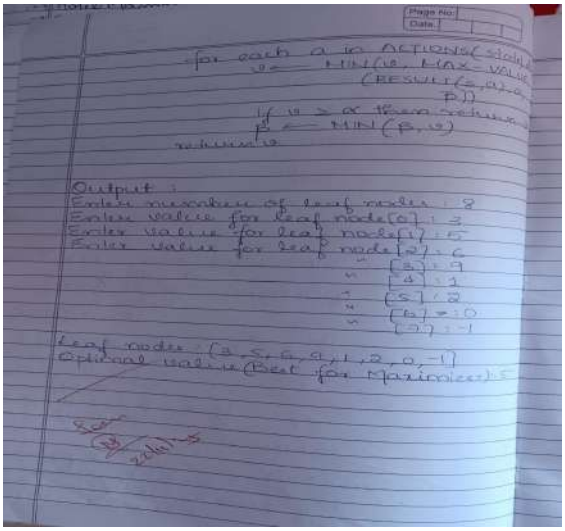
```
========== RESTART: C:/Users/BMSCECSE/Desktop/Lab-9.py ==========
=== Resolution in First Order Logic (Simplified) ===
Enter number of clauses in Knowledge Base: 4
Enter clause 1: P OR Q
Enter clause 2: ~Q OR R
Enter clause 3: ~R
Enter clause 4: ~P
Enter the query to prove: R

========== RESOLUTION IN FIRST ORDER LOGIC ==========

Original Query     : R
Negated Query      : ~R

Initial Knowledge Base:
  Clause 1: P OR Q
  Clause 2: ~Q OR R
  Clause 3: ~R
  Clause 4: ~P
  Clause 5: ~R

---------- Resolution Steps ----------

Step 1: Resolving [P OR Q] and [~Q OR R] ⇒ New clause: P OR R
Step 2: Resolving [P OR Q] and [~P] ⇒ New clause: Q
Step 3: Resolving [~Q OR R] and [~R] ⇒ New clause: ~Q
Step 4: Resolving [P OR Q] and [~Q] ⇒ New clause: P
Step 5: Resolving [~Q OR R] and [Q] ⇒ New clause: R
Step 6: Resolving [Q] and [~Q] gives NIL ⇒ Contradiction found!

☑ Query is PROVED by Resolution.
```

Program 10

Implement Alpha-Beta Pruning

Algorithm:





Code:

```
import math

# Recursive function for Alpha–Beta Pruning
def alpha_beta_pruning(depth, node_index, is_maximizing, values, alpha, beta, max_depth):
    indent = "    " * depth  # for visual indentation
    if depth == max_depth:
        print(f"{indent}Reached leaf node[{node_index}] = {values[node_index]}")
        return values[node_index]

    if is_maximizing:
        best = -math.inf
        print(f"{indent}Maximizer at depth {depth}, Alpha={alpha}, Beta={beta}")
        for i in range(2):  # each node has 2 children
                val = alpha_beta_pruning(depth + 1, node_index * 2 + i, False, values, alpha, beta,
max_depth)
            best = max(best, val)
            alpha = max(alpha, best)
                    print(f"{indent}Max node[{node_index}] updated Best={best}, Alpha={alpha},
Beta={beta}")
            if beta <= alpha:
                    print(f"{indent}-- Pruning remaining branches at Max node[{node_index}] (Beta <=
Alpha) --")
                break
        return best

    else:  # Minimizer's turn
```

```
        best = math.inf
        print(f"{indent}Minimizer at depth {depth}, Alpha={alpha}, Beta={beta}")
        for i in range(2):
                val = alpha_beta_pruning(depth + 1, node_index * 2 + i, True, values, alpha, beta,
max_depth)
            best = min(best, val)
            beta = min(beta, best)
                        print(f"{indent}Min node[{node_index}] updated Best={best}, Alpha={alpha},
Beta={beta}")
            if beta <= alpha:
                    print(f"{indent}-- Pruning remaining branches at Min node[{node_index}] (Beta <=
Alpha) --")
                break
        return best


# ----------------------------------------
# Main Program
# ----------------------------------------
if __name__ == "__main__":
    print("=== Alpha–Beta Pruning (Cutoff) Search Algorithm ===\n")

    # Input: leaf node values
    n = int(input("Enter number of leaf nodes (power of 2): "))
    values = []
    for i in range(n):
        val = int(input(f"Enter value for leaf node[{i}]: "))
        values.append(val)

    print("\nLeaf Nodes (Terminal Values):", values)
    max_depth = int(math.log2(n))  # since binary tree

    print("\n========== Search Trace ==========\n")
    best_value = alpha_beta_pruning(0, 0, True, values, -math.inf, math.inf, max_depth)

    print("\n========== RESULT ==========")
    print(f"Optimal Value (Best for Maximizer): {best_value}")
```

Output:

```
========== RESTART: C:/Users/HARSHEKH/Desktop/Lab-10.py ==========
=== Alpha-Beta Pruning (Cutoff) Search Algorithm ===

Enter number of leaf nodes (power of 2): 8
Enter value for leaf node[0]: 3
Enter value for leaf node[1]: 5
Enter value for leaf node[2]: 6
Enter value for leaf node[3]: 9
Enter value for leaf node[4]: 1
Enter value for leaf node[5]: 2
Enter value for leaf node[6]: 0
Enter value for leaf node[7]: -1

Leaf Nodes (Terminal Values): [3, 5, 6, 9, 1, 2, 0, -1]

========= Search Trace =========

Maximizer at depth 0, Alpha=-inf, Beta=inf
  Minimizer at depth 1, Alpha=-inf, Beta=inf
    Maximizer at depth 2, Alpha=-inf, Beta=inf
      Reached leaf node[0] = 3
    Max node[0] updated Best=3, Alpha=3, Beta=inf
      Reached leaf node[1] = 5
    Max node[0] updated Best=5, Alpha=5, Beta=inf
  Min node[0] updated Best=5, Alpha=-inf, Beta=5
    Maximizer at depth 2, Alpha=-inf, Beta=5
      Reached leaf node[2] = 6
    Max node[1] updated Best=6, Alpha=6, Beta=5
    -- Pruning remaining branches at Max node[1] (Beta <= Alpha) --
  Min node[0] updated Best=5, Alpha=-inf, Beta=5
Max node[0] updated Best=5, Alpha=5, Beta=inf
  Minimizer at depth 1, Alpha=5, Beta=inf
    Maximizer at depth 2, Alpha=5, Beta=inf
      Reached leaf node[4] = 1
    Max node[2] updated Best=1, Alpha=5, Beta=inf
      Reached leaf node[5] = 2
    Max node[2] updated Best=2, Alpha=5, Beta=inf
  Min node[1] updated Best=2, Alpha=5, Beta=2
  -- Pruning remaining branches at Min node[1] (Beta <= Alpha) --
Max node[0] updated Best=5, Alpha=5, Beta=inf

========= RESULT =========
Optimal Value (Best for Maximizer): 5
>>>
```