VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Samriddhi Singh (1BM23CS295)

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 **Samriddhi Singh (1BM23CS295),** 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.

Seema Patil Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link: https://github.com/samriddhisingh05/SamriddhiSingh_1BM23CS295-AILab.git

USN → 1BM23CS295

INDEX

Name : Samriddhi Singh
Sub :
Std :
Div :
Roll No :
Telephone No :
E-mail ID :
Blood Group :
Birth Day :

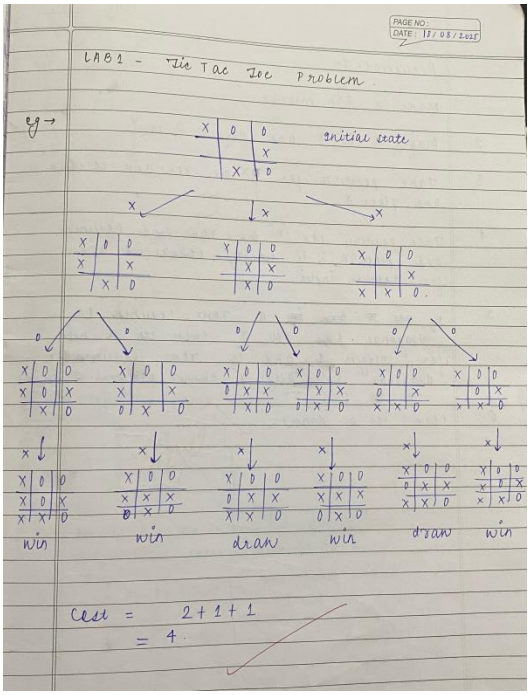
Sr. No	Title	Page No.	Sign/Remarks	date
1.	Tic-Tac-Toe	1-4	10	18/8/25
2.	Vacuum Cleaner	5	10	25/8/25
3.	a) BFS (without Heuristic approach)	6-7	10	1/9/25
	b) BFS (with Heuristic approach)			
	c) Iterative deepening (DFS)	8-9		
4.	A* algorithm - misplaced tiles		10	8/9/25
	- Manhattan Distance	8-9		
5.	Hill climbing algorithm.			15/9/25
	Stimulated Annealing	22-25	10	
6.	Propositional logic	26-31	10	22/9/25
7.	Unification algorithm	32-34	10	29/9/25
8.	First order logic	35-40	—	13/10/25
9.	FOL (with KB)	41-43	10	27/10/25
10.	Adversarial Search	44-47		27/10/25

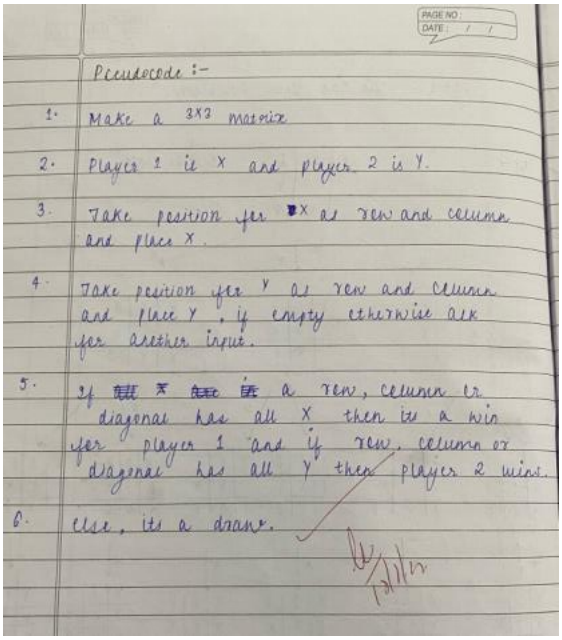
Completed

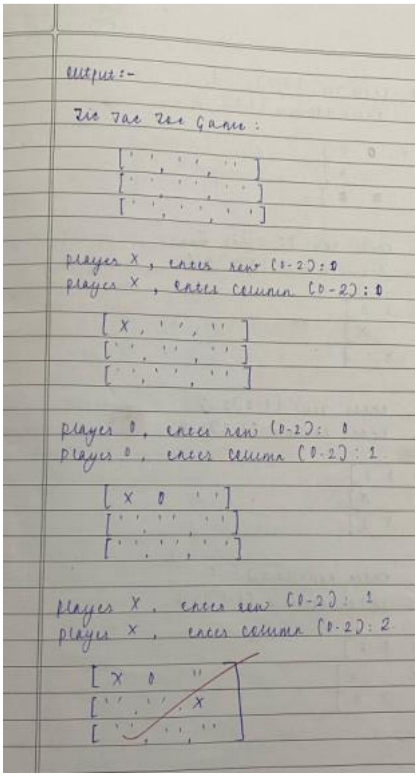
Program 1

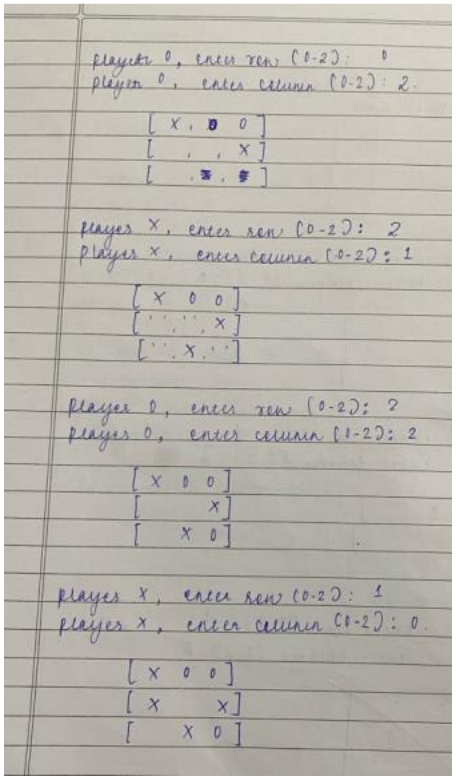
Implement Tic –Tac –Toe Game

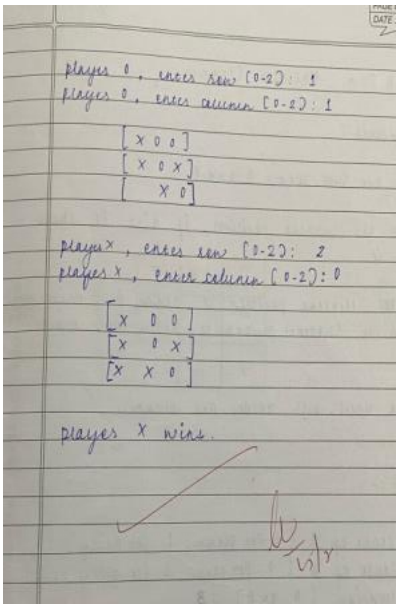
Algorithm:











Code:

```
def print_board(board):
    for row in board:
        print(row)

def check_winner(board, player):

    for i in range(3):
        if all(board[i][j] == player for j in range(3)):
            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 tic_tac_toe():
    board = [[" " for _ in range(3)] for _ in range(3)]
    players = ["X", "O"]
    turn = 0
    path_cost = 0

    for move in range(9):
        print("\nCurrent Board:")
        print_board(board)

        player = players[turn % 2]
        print(f"\nPlayer {player}'s turn:")


        row = int(input("Enter row (0-2): "))
        col = int(input("Enter col (0-2): "))


        if board[row][col] == " ":
            board[row][col] = player
            path_cost += 1
        else:
            print("Spot taken! Try again.")
            continue


        if check_winner(board, player):
            print("\nFinal Board:")
            print_board(board)
            print(f"\nPlayer {player} wins!")
            print(f"Path Cost = {path_cost}")
            return

        turn += 1

    print("\nFinal Board:")
    print_board(board)
    print("\nIt's a draw!")
    print(f"Path Cost = {path_cost}")
```

tic_tac_toe()

print("Samriddhi Singh 1BM23CS295")

OPUTPUT-Tic-tac-toe

Case 1-

```
Current Board:
[' ', ' ', ' ']
[' ', ' ', ' ']
[' ', ' ', ' ']
Player X's turn:
Enter row (0-2): 0
Enter col (0-2): 0
Current Board:
['X', ' ', ' ']
[' ', ' ', ' ']
[' ', ' ', ' ']
Player O's turn:
Enter row (0-2): 0
Enter col (0-2): 1
Current Board:
['X', 'O', ' ']
[' ', ' ', ' ']
[' ', ' ', ' ']
Player X's turn:
Enter row (0-2): 1
Enter col (0-2): 1
Current Board:
['X', 'O', ' ']
[' ', 'X', ' ']
[' ', ' ', ' ']
Player O's turn:
Enter row (0-2): 1
Enter col (0-2): 2
Current Board:
['X', 'O', ' ']
[' ', 'X', 'O']
[' ', ' ', ' ']
Player X's turn:
Enter row (0-2): 2
Enter col (0-2): 2
Final Board:
['X', 'O', ' ']
[' ', 'X', 'O']
[' ', ' ', 'X']
```

Player X wins!
Path Cost = 5
Samriddhi Singh 1BM23CS295

Case 2-

```
Current Board:
[' ', ' ', ' ']
[' ', ' ', ' ']
[' ', ' ', ' ']
Player X's turn:
Enter row (0-2): 0
Enter col (0-2): 0
Current Board:
['X', ' ', ' ']
[' ', ' ', ' ']
[' ', ' ', ' ']
Player O's turn:
Enter row (0-2): 0
Enter col (0-2): 1
Current Board:
['X', 'O', ' ']
[' ', ' ', ' ']
[' ', ' ', ' ']
Player X's turn:
Enter row (0-2): 1
Enter col (0-2): 0
Current Board:
['X', 'O', ' ']
['X', ' ', ' ']
[' ', ' ', ' ']
Player O's turn:
Enter row (0-2): 1
Enter col (0-2): 1
Current Board:
['X', 'O', ' ']
['X', 'O', ' ']
[' ', ' ', ' ']
Player X's turn:
Enter row (0-2): 0
Enter col (0-2): 2
Current Board:
['X', 'O', 'X']
['X', 'O', ' ']
[' ', ' ', ' ']
Player O's turn:
Enter row (0-2): 2
Enter col (0-2): 2
```

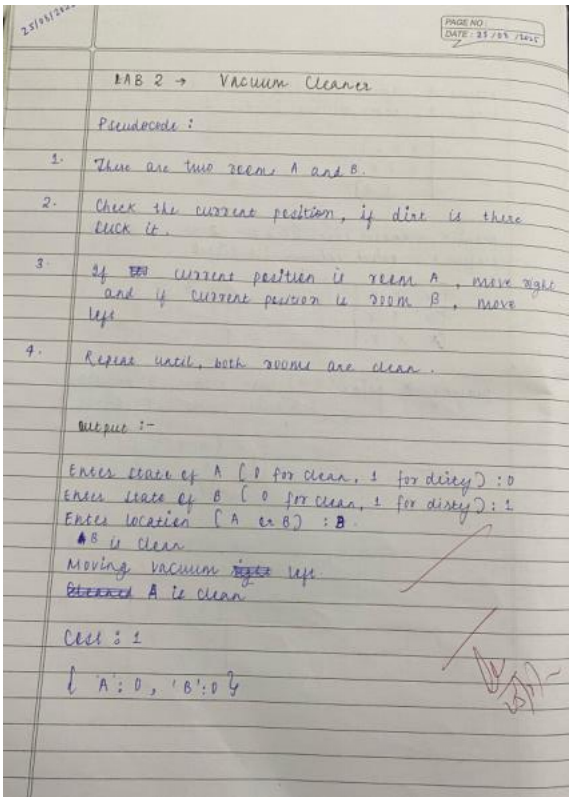
Current Board:
['X', 'O', 'X']['X', 'O', ' ']
[' ', ' ', 'O']
Player X's turn:
Enter row (0-2): 1
Enter col (0-2): 2
Current Board:
['X', 'O', 'X']
['X', 'O', 'X']
[' ', ' ', 'O']
Player O's turn:
Enter row (0-2): 2
Enter col (0-2): 1
Final Board:
['X', 'O', 'X']
['X', 'O', 'X']
[' ', 'O', 'O']
Player O wins!
Path Cost = 8
Samriddhi Singh 1BM23CS295

Case 3-
Current Board:
[' ', ' ', ' ']
[' ', ' ', ' ']
[' ', ' ', ' ']
Player X's turn:
Enter row (0-2): 0
Enter col (0-2): 0
Current Board:
['X', ' ', ' ']
[' ', ' ', ' ']
[' ', ' ', ' ']
Player O's turn:
Enter row (0-2): 1
Enter col (0-2): 0
Current Board:
['X', ' ', ' ']
['O', ' ', ' ']
[' ', ' ', ' ']
Player X's turn:
Enter row (0-2): 0
Enter col (0-2): 2
Current Board:
['X', ' ', 'X']
['O', ' ', ' '][' ', ' ', ' ']
Player O's turn:

```
Enter row (0-2): 0
Enter col (0-2): 1
Current Board:
['X', 'O', 'X']
['O', ' ', ' ']
[' ', ' ', ' ']
Player X's turn:
Enter row (0-2): 1
Enter col (0-2): 2
Current Board:
['X', 'O', 'X']
['O', ' ', 'X']
[' ', ' ', ' ']
Player O's turn:
Enter row (0-2): 2
Enter col (0-2): 2
Current Board:
['X', 'O', 'X']
['O', ' ', 'X']
[' ', ' ', 'O']
Player X's turn:
Enter row (0-2): 1
Enter col (0-2): 1
Current Board:
['X', 'O', 'X']
['O', 'X', 'X']
[' ', ' ', 'O']
Player O's turn:
Enter row (0-2): 2
Enter col (0-2): 0
Current Board:
['X', 'O', 'X']
['O', 'X', 'X']
['O', ' ', 'O']
Player X's turn:
Enter row (0-2): 2
Enter col (0-2): 1
Final Board:
['X', 'O', 'X']
['O', 'X', 'X']
['O', 'X', 'O']
It's a draw!
Path Cost = 9
```

Implement vacuum cleaner agent

Algorithm:



LAB 2 → Vacuum Cleaner

Pseudocode :

1. There are two rooms, A and B.
2. Check the current position, if dirt is there suck it.
3. If current position is room A, move right and if current position is room B, move left
4. Repeat until, both rooms are clean.

Output :-

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) : B
B is clean
Moving vacuum left
A is clean

Cost : 1

{'A': 0, 'B': 0}

Code:

```
def vacuum_world():
    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): "))
    loc = input("Enter location (A or B): ").upper()
    cost, s = 0, {'A': A, 'B': B}

    if loc == 'A':
        print("Cleaned A." if s['A'] else "A is clean"); cost += s['A']; s['A'] = 0
        print("Is A clean now?:", s['A']); print("Is B dirty?:", s['B'])
        print("Moving vacuum right")
        print("Cleaned B." if s['B'] else "B is clean"); cost += s['B']; s['B'] = 0
    elif loc == 'B':
        print("Cleaned B." if s['B'] else "B is clean"); cost += s['B']; s['B'] = 0
        print("Is B clean now?:", s['B']); print("Is A dirty?:", s['A'])
        print("Moving vacuum left")
        print("Cleaned A." if s['A'] else "A is clean"); cost += s['A']; s['A'] = 0
    else:
        print("Turning vacuum off")

    print("Cost:", cost, "\n", s)
```

```
    print("Samriddhi Singh 1BM23CS295")

vacuum_world()
```

OUTPUT Case1:

```
Enter state of A (0 for clean, 1 for dirty): 1
Enter state of B (0 for clean, 1 for dirty): 0
Enter location (A or B): A
Cleaned A.
Is A clean now?: 0
Is B dirty?: 0
Moving vacuum right
B is clean
Cost: 1
 {'A': 0, 'B': 0}
Samriddhi Singh 1BM23CS295
```

OUTPUT Case2:

```
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
Is A clean now?: 0
Is B dirty?: 0
Moving vacuum right
B is clean
Cost: 0
 {'A': 0, 'B': 0}
Samriddhi Singh 1BM23CS295
```

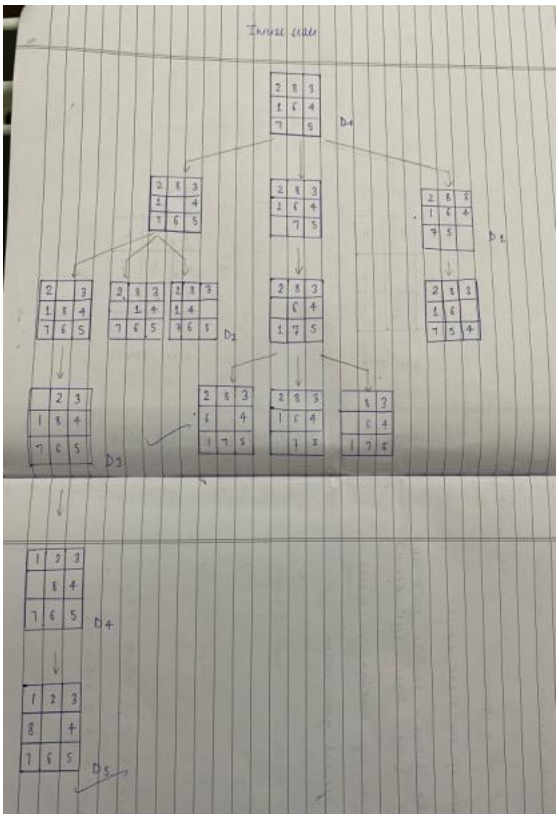
OUTPUT Case3:

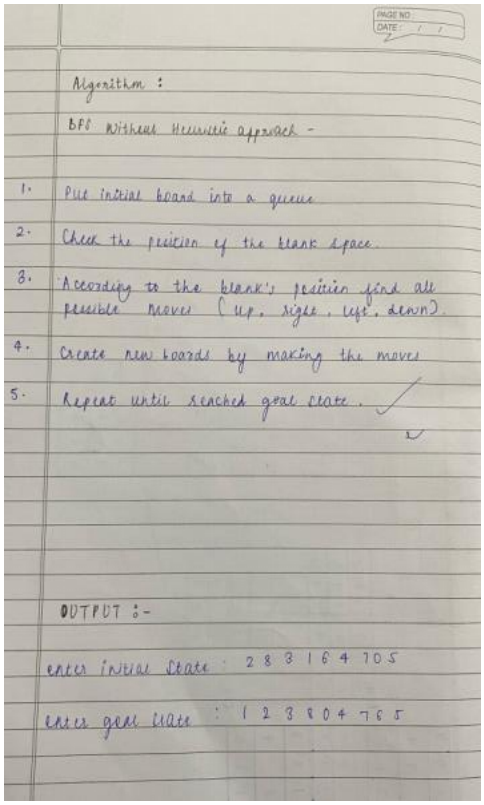
```
Enter state of A (0 for clean, 1 for dirty): 0
Enter state of B (0 for clean, 1 for dirty): 1
Enter location (A or B): B
Cleaned B.
Is B clean now?: 0
Is A dirty?: 0
Moving vacuum left
A is clean
Cost: 1
 {'A': 0, 'B': 0}
Samriddhi Singh 1BM23CS295
```

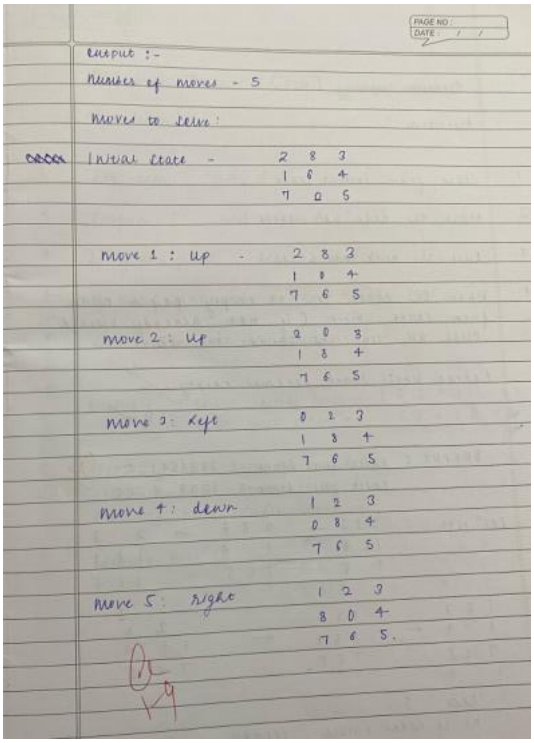
Program2

Implement 8 puzzle problems using Breath First Search (BFS)

Algorithm:







Code:

```
from collections import deque

moves = {
    'Up': -3,
    'Down': 3,
    'Left': -1,
    'Right': 1
}

def is_valid_move(pos, move):
    if move == 'Left' and pos % 3 == 0:
        return False
    if move == 'Right' and pos % 3 == 2:
        return False
    if move == 'Up' and pos < 3:
        return False
    if move == 'Down' and pos > 5:
        return False
    return True

def get_neighbors(state):
    neighbors = []
    zero_pos = state.index(0)
    for move, pos_change in moves.items():
        if is_valid_move(zero_pos, move):
            new_zero_pos = zero_pos + pos_change
            new_state = list(state)
            new_state[zero_pos], new_state[new_zero_pos] = new_state[new_zero_pos],
new_state[zero_pos]
            neighbors.append((tuple(new_state), move))
    return neighbors

def bfs(start_state, goal_state):
    queue = deque()
    queue.append((start_state, []))
    visited = set()
    visited.add(start_state)
    explored_states = []

    while queue:
        current_state, path = queue.popleft()
        explored_states.append(current_state)
        if current_state == goal_state:
            return path, explored_states

        for neighbor, move in get_neighbors(current_state):
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append((neighbor, path + [move]))
    return None, explored_states
```

```
def input_state(prompt):
    while True:
        raw = input(prompt).strip().split()
        if len(raw) != 9:
            print("Please enter exactly 9 numbers separated by spaces.")
            continue
        try:
            numbers = [int(x) for x in raw]
        except ValueError:
            print("Please enter valid integers only.")
            continue
        if set(numbers) != set(range(9)):
            print("Numbers must be from 0 to 8 without repetition.")
            continue
        return tuple(numbers)

def print_state(state):
    for i in range(3):
        print(state[3*i:3*i+3])
    print()

def main():
    start_state = input_state("Enter the initial state (9 numbers from 0 to 8, space separated): ")
    goal_state = input_state("Enter the goal state (9 numbers from 0 to 8, space separated): ")

    print("\nStarting BFS...\n")
    path, explored_states = bfs(start_state, goal_state)

    print(f"Total states explored: {len(explored_states)}\n")

    print("States explored in order:")
    for idx, state in enumerate(explored_states, 1):
        print(f"State {idx}:")
        print_state(state)

    if path is None:
        print("No solution found.")
    else:
        print(f"Number of moves to solve: {len(path)}")
        print("Moves to solve:")
        current = start_state
        print("Initial state:")
        print_state(current)
        for i, move in enumerate(path, 1):
            print(f"Move {i}: {move}")
            zero_pos = current.index(0)
            pos_change = moves[move]
            new_zero_pos = zero_pos + pos_change
            new_state = list(current)
            new_state[zero_pos], new_state[new_zero_pos] = new_state[new_zero_pos],
new_state[zero_pos]
            current = tuple(new_state)
```

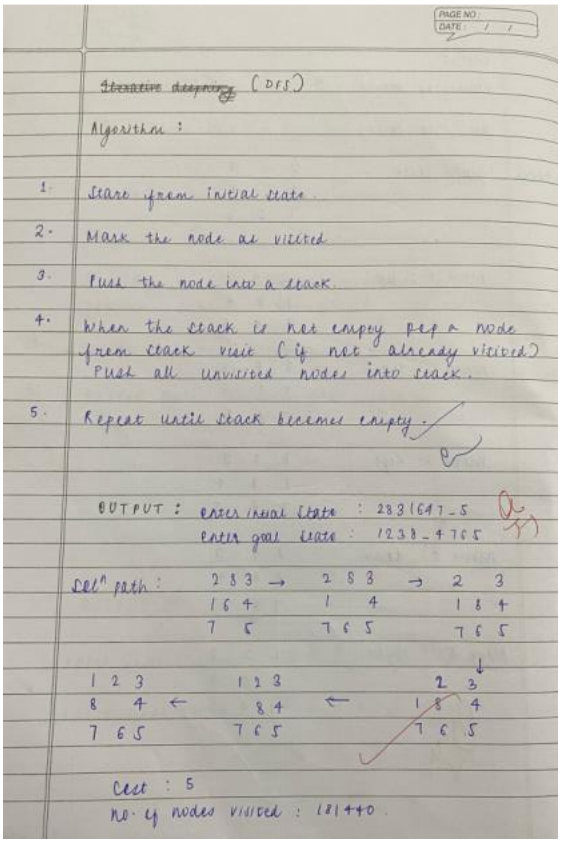
```
        print_state(current)

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

Output:

```
Number of moves to solve: 5
Moves to solve:
Initial state:(2, 8, 3)
(1, 6, 4)
(7, 0, 5)
Move 1: Up
(2, 8, 3)
(1, 0, 4)
(7, 6, 5)
Move 2: Up
(2, 0, 3)
(1, 8, 4)
(7, 6, 5)
Move 3: Left
(0, 2, 3)
(1, 8, 4)
(7, 6, 5)
Move 4: Down
(1, 2, 3)
(0, 8, 4)
(7, 6, 5)
Move 5: Right
(1, 2, 3)
(8, 0, 4)
(7, 6, 5)
Samriddhi Singh-1BM23CS295
```

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 = []
```

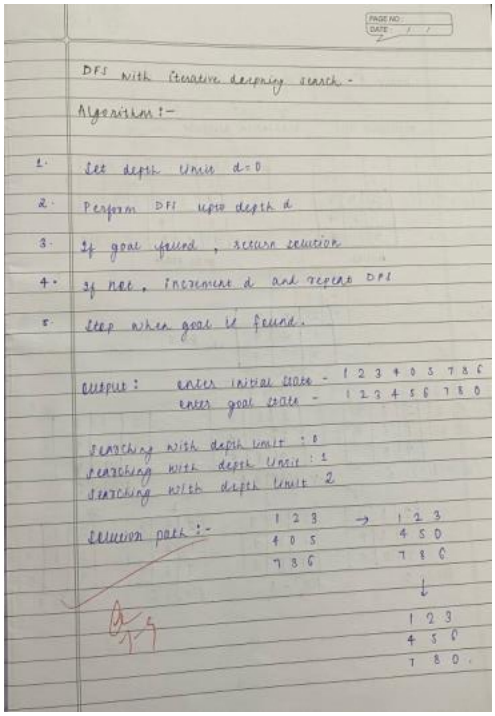
```
    while stack:
        state, cost = stack.pop()
        order.append(state)
        if state == goal:
            path = []
            while state:
                path.append(state)
                state = parent[state]
            path.reverse()
            return path, cost, order, visited
        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))

print("Samriddhi Singh,1BM23CS295")
```

Output:

```
Steps (solution path):
283
164
7_5

283
1_4
765

2_3
184
765

_23
184
765

123
_84
765

123
8_4
765

Cost (depth to goal): 5
Number of nodes visited: 181440
Samriddhi Singh,1BM23CS295
```

Implement Iterative deepening search algorithm

Algorithm:



Code:

```
def get_neighbors(state):
    neighbors = []
    idx = state.index("0")
    x, y = divmod(idx, 3)
```

```
    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_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([start_state])
    parent = {start_state: None}

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

        if depth < limit:
            for neighbor in get_neighbors(current_state):
                if neighbor not in visited:
                    visited.add(neighbor)
                    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("Samriddhi Singh 1BM23CS295")
print("Enter the initial state (3x3, 0 for empty space):")
initial_state = "".join(input().split())
print("Enter the goal state (3x3, 0 for empty space):")
goal_state = "".join(input().split())


max_depth = 50  # You can adjust this as needed
solution = iddfs(initial_state, goal_state, max_depth)

if solution:
```

```
    print("\nSolution path:")
    for state in solution:

        for i in range(0, 9, 3):
            print(" ".join(state[i:i+3]))
        print()
else:
    print("\nNo solution found.")
```

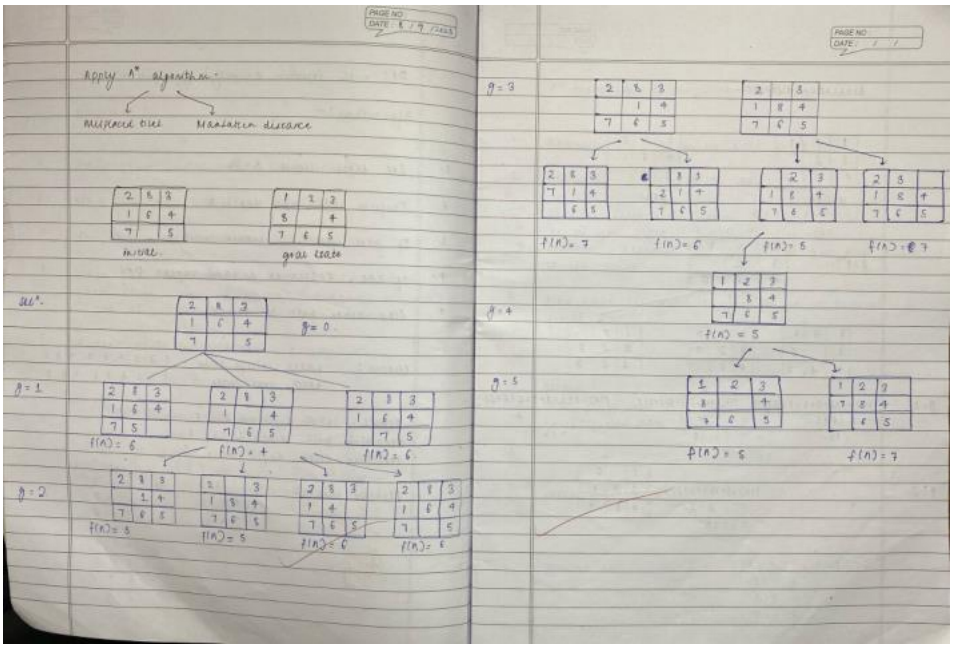
Output:

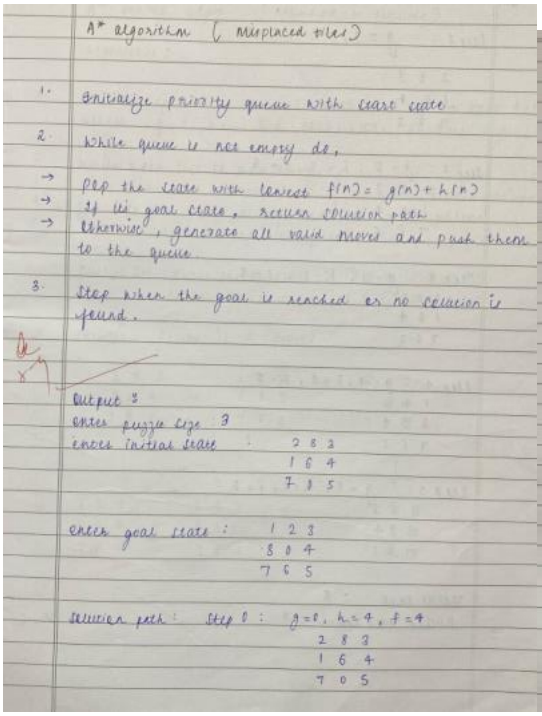
```
Samriddhi Singh 1BM23CS295
Enter the initial state (3x3, 0 for empty space):
1 2 3 4 0 5 7 8 6
Enter the goal state (3x3, 0 for empty space):
1 2 3 4 5 6 7 8 0
Searching with depth limit: 0
Searching with depth limit: 1
Searching with depth limit: 2

Solution path:
1 2 3
4 0 5
7 8 6

1 2 3
4 5 0
7 8 6

1 2 3
4 5 6
7 8 0
```

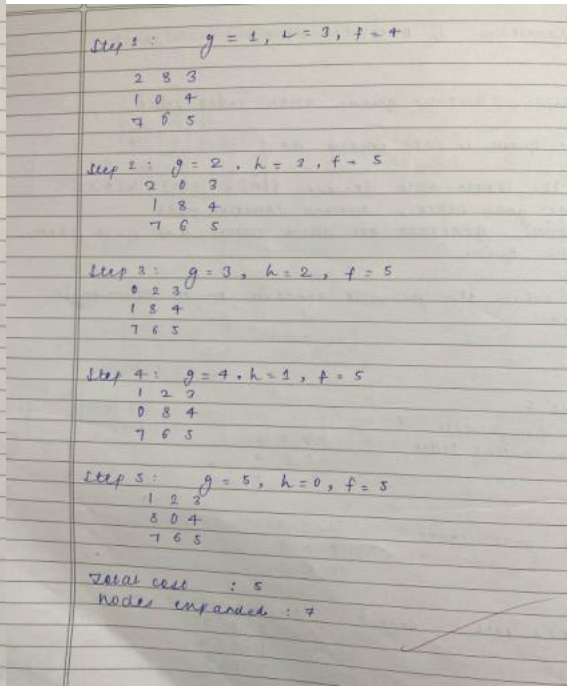
Program3

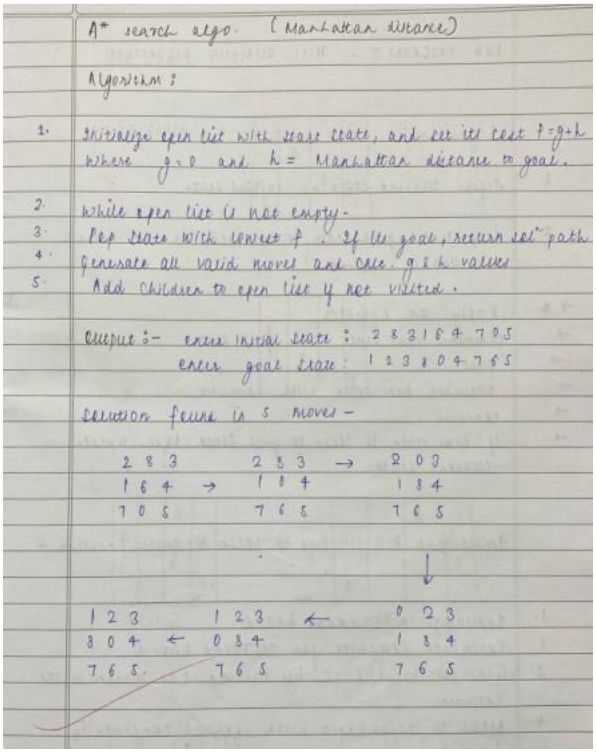
Implement A* search algorithm

Algorithm:









Code:

```
#misplaced tiles
import heapq
from itertools import count


def misplaced_heuristic(board, goal):
    """h(n): number of tiles not in their goal position (excluding blank 0)."""
    n = len(board)
    misplaced = 0
    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 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  # 1-indexed from bottom
    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)

    # Map values to goal indices to compute relative order
    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.")

    # Validate same tile multiset
    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}")
        print("Samriddhi Singh,1BM23CS29517")

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


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

Output:

```
Enter puzzle size n (e.g., 3 for 3x3): 3
Enter initial state row by row (use 0 for blank):
2 8 3
1 6 4
7 0 5
Enter goal state row by row (use 0 for blank):
1 2 3
8 0 4
7 6 5

Solution path (each state shows g, h, f):

Step 0: g=0, h=4, f=4
2 8 3
1 6 4
7 0 5

Step 1: g=1, h=3, f=4
2 8 3
1 0 4
7 6 5

Step 2: g=2, h=3, f=5
2 0 3
1 8 4
7 6 5

Step 3: g=3, h=2, f=5
0 2 3
1 8 4
7 6 5

Step 4: g=4, h=1, f=5
1 2 3
0 8 4
7 6 5

Step 5: g=5, h=0, f=5
1 2 3
8 0 4
7 6 5

Total cost (number of moves): 5
Nodes expanded: 6
Nodes explored (unique): 7
Samriddhi Singh,1BM23CS29517
```

Code:

#MANHATTAN DISTANCE

```
import heapq

def manhattan(state, goal):
```

```
    dist = 0
    for i in range(9):
        if state[i] != 0:
            x1, y1 = divmod(i, 3)
            j = goal.index(state[i])
            x2, y2 = divmod(j, 3)
            dist += abs(x1 - x2) + abs(y1 - y2)
    return dist

def get_neighbors(state):
    neighbors = []
    i = state.index(0)
    x, y = divmod(i, 3)
    moves = [(-1,0),(1,0),(0,-1),(0,1)]
    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            j = nx*3 + ny
            new_state = list(state)
            new_state[i], new_state[j] = new_state[j], new_state[i]
            neighbors.append(tuple(new_state))
    return neighbors


def a_star(start, goal):
    open_heap = [(manhattan(start, goal), 0, start, [])]
    visited = set()
    while open_heap:
        f, g, state, path = heapq.heappop(open_heap)
        if state == goal:
            return path + [state], g
        if state in visited: continue
        visited.add(state)
        for nb in get_neighbors(state):
            if nb not in visited:
                new_g = g + 1
                new_f = new_g + manhattan(nb, goal)
                heapq.heappush(open_heap, (new_f, new_g, nb, path + [state]))
    return None, -1


print("Enter initial state (9 numbers, 0 for blank):")
start = tuple(map(int, input().split()))
print("Enter goal state (9 numbers, 0 for blank):")
goal = tuple(map(int, input().split()))

path, cost = a_star(start, goal)

if path:
    print("\nSolution found in", cost, "moves\n")
    for step, p in enumerate(path):
        print("Step", step, " g=", step, " h=", manhattan(p, goal), " f=", step + manhattan(p, goal))
```

```
        for i in range(0, 9, 3):
            print(p[i:i+3])
        print()
else:
    print("No solution found")
print("Samriddhi Singh,1BM23CS295")
```

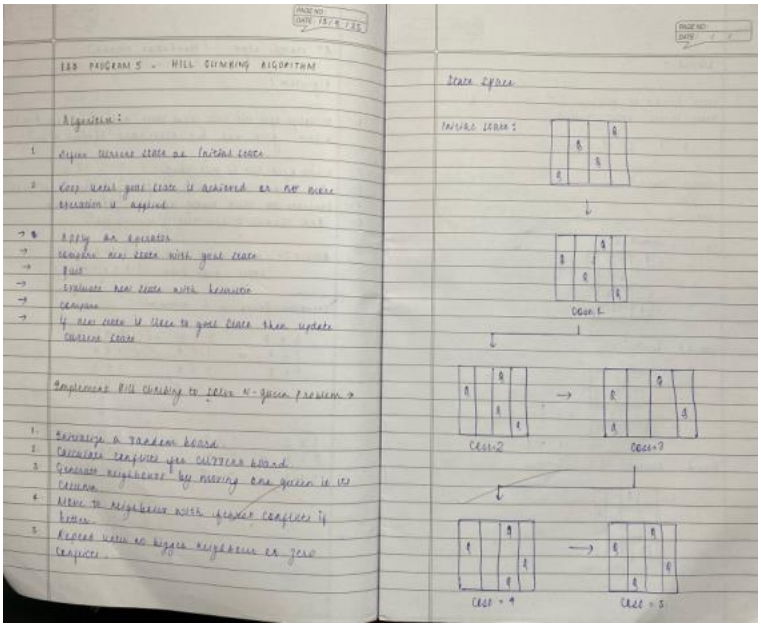
Output:

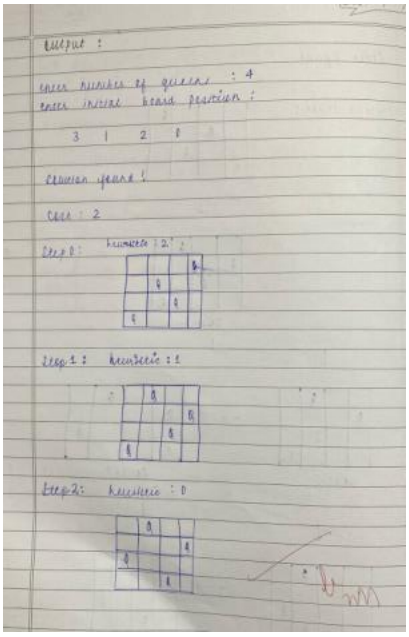
```
Enter initial state (9 numbers, 0 for blank):
2 8 3 1 6 4 7 0 5
Enter goal state (9 numbers, 0 for blank):
1 2 3 8 0 4 7 6 5

Solution found in 5 moves

Step 0  g= 0  h= 5  f= 5
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

Step 1  g= 1  h= 4  f= 5
(2, 8, 3)
(1, 0, 4)
(7, 6, 5)

Step 2  g= 2  h= 3  f= 5
(2, 0, 3)
(1, 8, 4)
(7, 6, 5)

Step 3  g= 3  h= 2  f= 5
(0, 2, 3)
(1, 8, 4)
(7, 6, 5)

Step 4  g= 4  h= 1  f= 5
(1, 2, 3)
(0, 8, 4)
(7, 6, 5)

Step 5  g= 5  h= 0  f= 5
(1, 2, 3)
(8, 0, 4)
(7, 6, 5)

Samriddhi Singh,1BM23CS295
```

Program4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:





Code:

```
import random

def heuristic(board):

    attacks = 0
    n = len(board)
    for i in range(n):
        for j in range(i + 1, n):
            if board[i] == board[j] or abs(board[i] - board[j]) == j - i:
                attacks += 1
    return attacks

def get_neighbors(board):

    neighbors = []
    n = len(board)
    for i in range(n):
        for j in range(i + 1, n):
            new_board = list(board)

            new_board[i], new_board[j] = new_board[j], new_board[i]
            neighbors.append(new_board)
```

```
    return neighbors

def hill_climbing(board, max_sideways=0):  # sideways moves disabled
    steps = 0
    sideways_moves = 0
    current_heur = heuristic(board)
    path = [board[:]]

    while True:
        if current_heur == 0:
            return board, steps, path

        neighbors = get_neighbors(board)
        neighbor_heuristics = [(neighbor, heuristic(neighbor)) for neighbor in neighbors]

        best_heur = min(h for _, h in neighbor_heuristics)
        best_neighbors = [nb for nb, h in neighbor_heuristics if h == best_heur]

        if best_heur > current_heur:

            return None, steps, path

        next_board = random.choice(best_neighbors)

        if best_heur < current_heur:
            sideways_moves = 0
        elif best_heur == current_heur:
            sideways_moves += 1
            if sideways_moves > max_sideways:
                return None, steps, path

        board = next_board
        current_heur = best_heur
        path.append(board[:])
        steps += 1

def print_board(board):
    n = len(board)
    for row in range(n):
        line = ""
        for col in range(n):
            line += "Q " if board[col] == row else ". "
        print(line)
    print()

def main():
    n = int(input("Enter the number of queens (N): "))
```

```
    print(f"Enter the initial board positions (row for each queen in column 0 to {n-1}):")
    print(f"Rows should be between 0 and {n-1}, space separated.")
    board_input = input()

    try:
        board = list(map(int, board_input.strip().split()))
    except ValueError:
        print("Invalid input format.")
        return

    if len(board) != n or any(r < 0 or r >= n for r in board):
        print("Invalid board input.")
        return

    solution, cost, path = hill_climbing(board)

    if solution:
        print("\nSolution found!\n")
    else:
        print("\nNo solution found (stuck in local minimum).\n")

    print(f"Cost (steps taken): {cost}\n")
    print("Steps to reach solution:")

    for step_num, state in enumerate(path):
        print(f"Step {step_num}: heuristic = {heuristic(state)}")
        print_board(state)

    print("Samriddhi Singh,1BM23CS295")

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

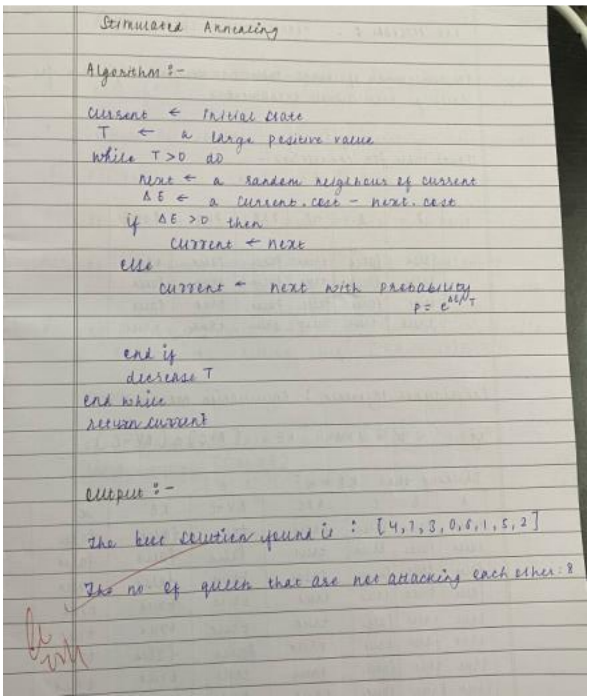
Output:

```
Enter the number of queens (N): 4
Enter the initial board positions (row for each queen in column 0 to 3):
Rows should be between 0 and 3, space separated.
3 2 0 1

Solution found!

Cost (steps taken): 2

Steps to reach solution:
Step 0: heuristic = 2
. . Q .
. . . Q
. Q . .
Q . . .

Step 1: heuristic = 1
. Q . .
. . . Q
. . Q .
Q . . .

Step 2: heuristic = 0
. Q . .
. . . Q
Q . . .
. . Q .

Samriddhi Singh,1BM23CS295
```

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:



Code:

```
import random, math

def cost(state):
    attacks = 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):
                attacks += 1
    return attacks

def neighbor(state):
    n = len(state)
    new_state = 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, T=1000, cooling=0.99):
```

```
    current = [random.randint(0, n-1) for _ in range(n)]

    while T > 1e-6:
        next_state = neighbor(current)
        deltaE = cost(current) - cost(next_state)


        if deltaE > 0:
            current = next_state
        else:

            if random.random() < math.exp(deltaE / T):
                current = next_state

        T *= cooling


        if cost(current) == 0:
            break

    return current


solution = simulated_annealing(8)
print("The best position found is:", solution)
print("The number of queens that are not attacking each other is:", 8 if cost(solution) == 0 else 8 - cost(solution))

print("Samriddhi Singh,1BM23CS295")
```

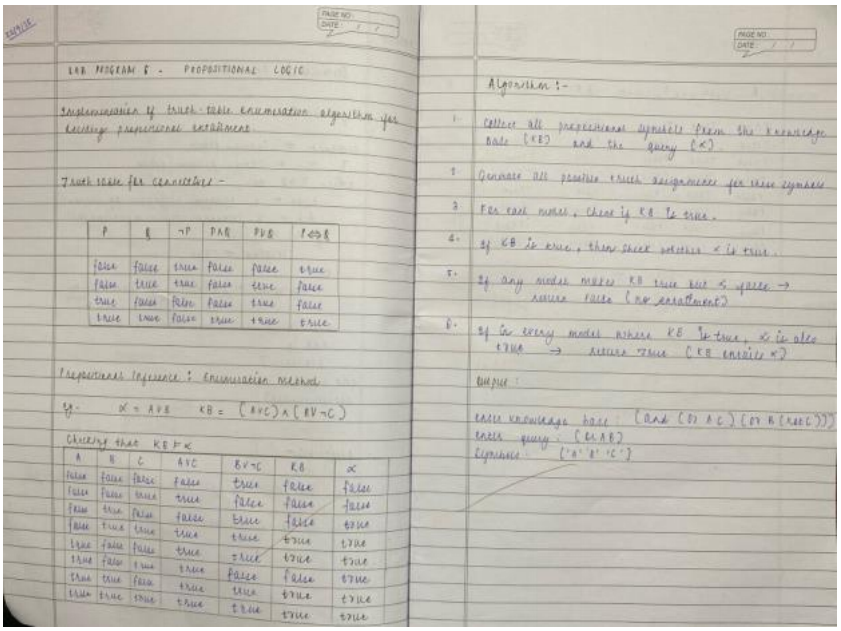
Output:

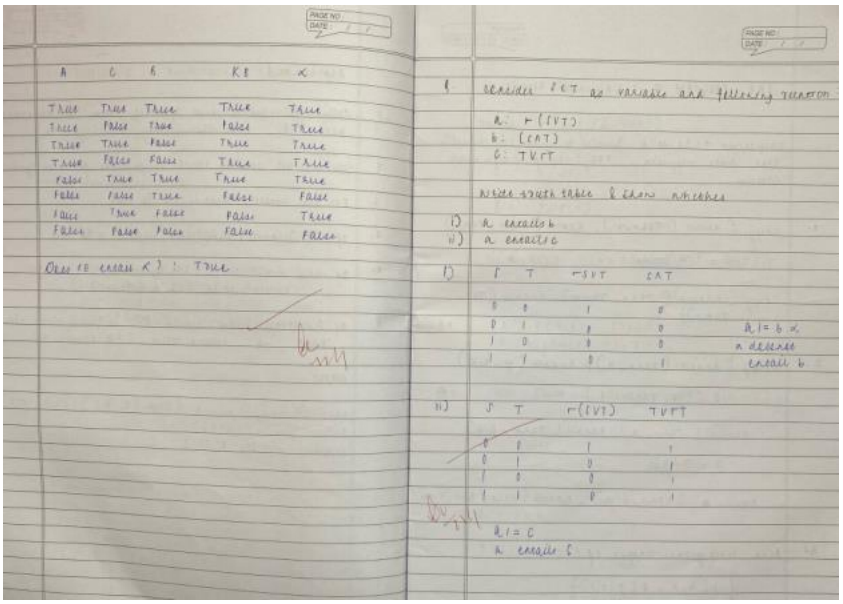
```
The best position found is: [4, 7, 3, 0, 6, 1, 5, 2]
The number of queens that are not attacking each other is: 8
Samriddhi Singh,1BM23CS295
```

Program 6

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

Algorithm:





Code:

```
def parse_expr(tokens):
    token = tokens.pop(0)
    if token == '(':
        op = tokens.pop(0)
```

```
        args = []
        while tokens[0] != ')':
            args.append(parse_expr(tokens))
        tokens.pop(0)  # Remove ')'
        return (op, *args)
    else:
        return token

def tokenize(s):
    return s.replace('(', ' ( ').replace(')', ' ) ').split()

def tt_entails(kb, alpha):
    symbols = list(get_symbols(kb) | get_symbols(alpha))
    print("Symbols:", symbols)


    header = symbols + ['KB', 'α']
    print("\t".join(header))

    # Start recursive check
    result = tt_check_all(kb, alpha, symbols, {})
    return result

def tt_check_all(kb, alpha, symbols, model):
    if not symbols:
        kb_val = pl_true(kb, model)
        alpha_val = pl_true(alpha, model)
        # Print current model and values
        row = [str(model.get(s, False)) for s in sorted(model.keys())] + [str(kb_val), str(alpha_val)]
        print("\t".join(row))

        if kb_val:
            return alpha_val
        else:
            return True
    else:
        rest = symbols[1:]
        symbol = symbols[0]
        model_true = model.copy()
        model_true[symbol] = True
        model_false = model.copy()
        model_false[symbol] = False
        return (tt_check_all(kb, alpha, rest, model_true) and
                tt_check_all(kb, alpha, rest, model_false))

def get_symbols(expr):
    if isinstance(expr, str):
        return {expr}
    elif isinstance(expr, tuple):
        symbols = set()
        for part in expr[1:] if expr[0] != 'not' else [expr[1]]:
            symbols |= get_symbols(part)
```

```
        return symbols
    else:
        return set()

def pl_true(expr, model):
    if isinstance(expr, str):
        return model.get(expr, False)
    op = expr[0]
    if op == 'and':
        return all(pl_true(arg, model) for arg in expr[1:])
    elif op == 'or':
        return any(pl_true(arg, model) for arg in expr[1:])
    elif op == 'not':
        return not pl_true(expr[1], model)
    elif op == 'implies':
        return (not pl_true(expr[1], model)) or pl_true(expr[2], model)
    else:
        raise ValueError(f"Unknown operator: {op}")


kb_input = input("Enter knowledge base (e.g. (and A (or B C))): ")
alpha_input = input("Enter query (e.g. A): ")

kb = parse_expr(tokenize(kb_input))
alpha = parse_expr(tokenize(alpha_input))

result = tt_entails(kb, alpha)
print(f"\nDoes KB entail α? : {result}")
```

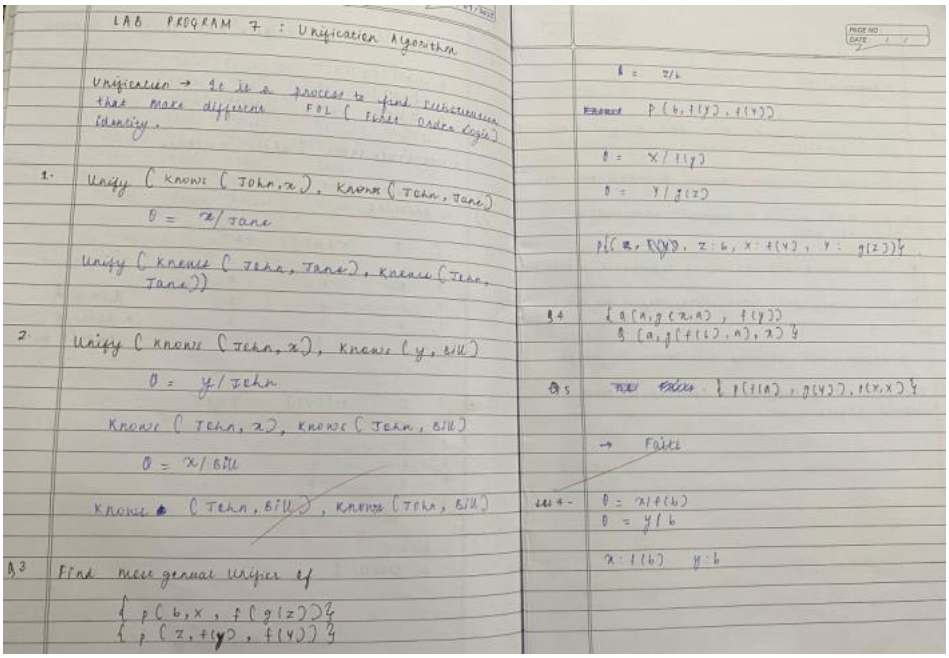
Output:

```
Enter knowledge base (e.g. (and A (or B C))): (and(or A C) (or B(not C)))
Enter query (e.g. A): (or  A B)
Symbols: ['C', 'A', 'B']
C       A       B       KB      α
True    True    True    True    True
True    False   True    False   True
False   True    True    True    True
False   False   True    False   False
True    True    False   True    True
True    False   False   True    True
False   True    False   False   True
False   False   False   False   False

Does KB entail α? : True
Samriddhi Singh,1BM23CS295
```

Program 7

Implement unification in first order logic

Algorithm:


LAB PROGRAM 7 : Unification Algorithm

Unification → It is a process to find substitutions that make different FOL (First Order Logic) identity.

1. Unify (Knows (John, x), Knows (John, Jane))
   θ = x/Jane
   Unify (Knows (John, Jane), Knows (John, Jane))
2. Unify (Knows (John, x), Knows (y, Bill))
   θ = y/John
   Knows (John, x), Knows (John, Bill)
   θ = x/Bill
   Knows (John, Bill), Knows (John, Bill)

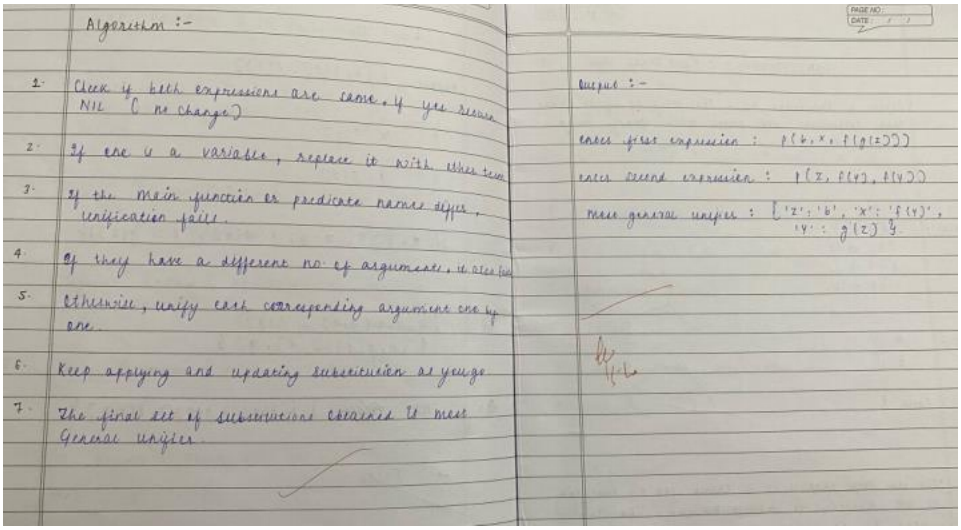
Q3 Find most general unifier of
{ p(b, x, f(g(z)) }
{ p(z, f(y), f(y)) }

θ = z/b
p(b, f(y), f(y))
θ = x/f(y)
θ = y/g(z)
p((z, f(y)), z:b, x:f(y), y: g(z))}

Q4 { a(a, g(z, a), f(y))
a(a, g(f(b), a), x) }

Q5 { p(f(a), g(y)), p(x, x) }
→ False

Q4 - θ = x/f(b)
θ = y/b
x: f(b) y: b


Algorithm :-

1. Check if both expressions are same. If yes return NIL (no change)
2. If one is a variable, replace it with other term
3. If the main function or predicate names differ, unification fails.
4. If they have a different no of arguments, unification fails
5. Otherwise, unify each corresponding argument one by one.
6. Keep applying and updating substitutions as you go.
7. The final set of substitutions obtained is most General unifier.

Output :-

Given first expression : p(b, x, f(g(z)))
Given second expression : p(z, f(y), f(y))
Most general unifier : {'z': 'b', 'x': 'f(y)', 'y': g(z)}

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)
print("Samriddhi Singh,1BM23CS295")
```

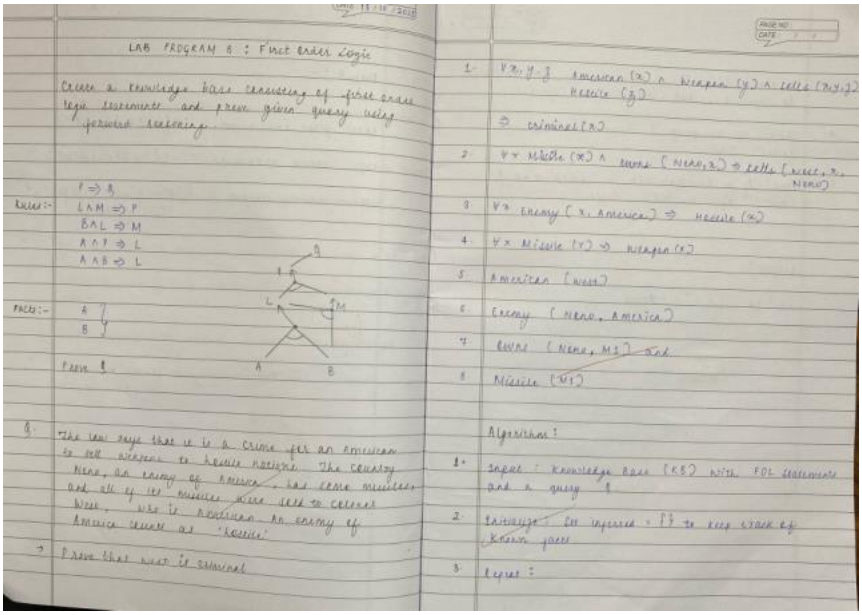
Output:

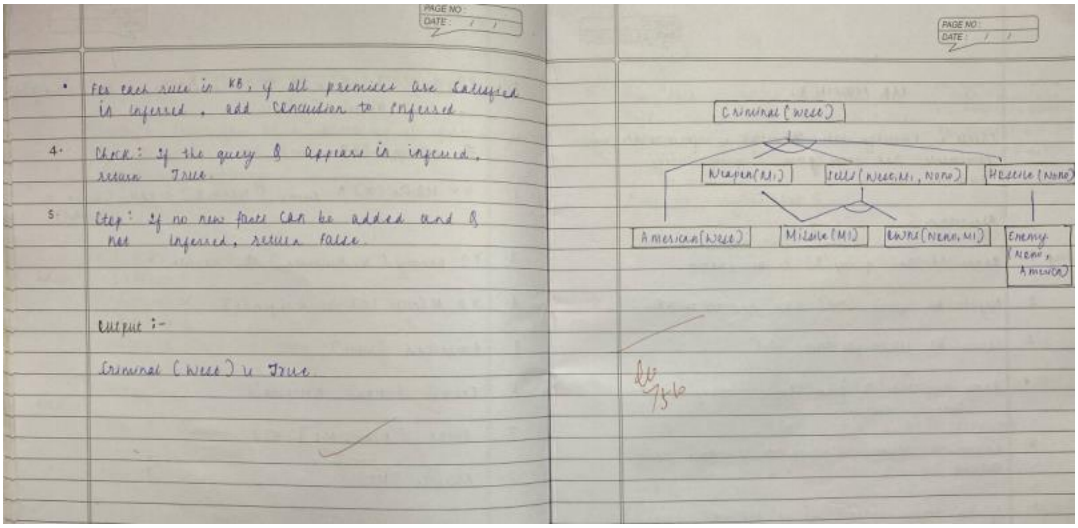
```
Enter first expression: p(b,X,f(g(Z)))
Enter second expression: p(Z,f(Y),f(Y))
Most General Unifier (MGU): {'Z': 'b', 'X': 'f(Y)', 'Y': 'g(Z)'}
Samriddhi Singh,1BM23CS295
```

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.")

print("Samriddhi Singh,1BM23CS295")
```

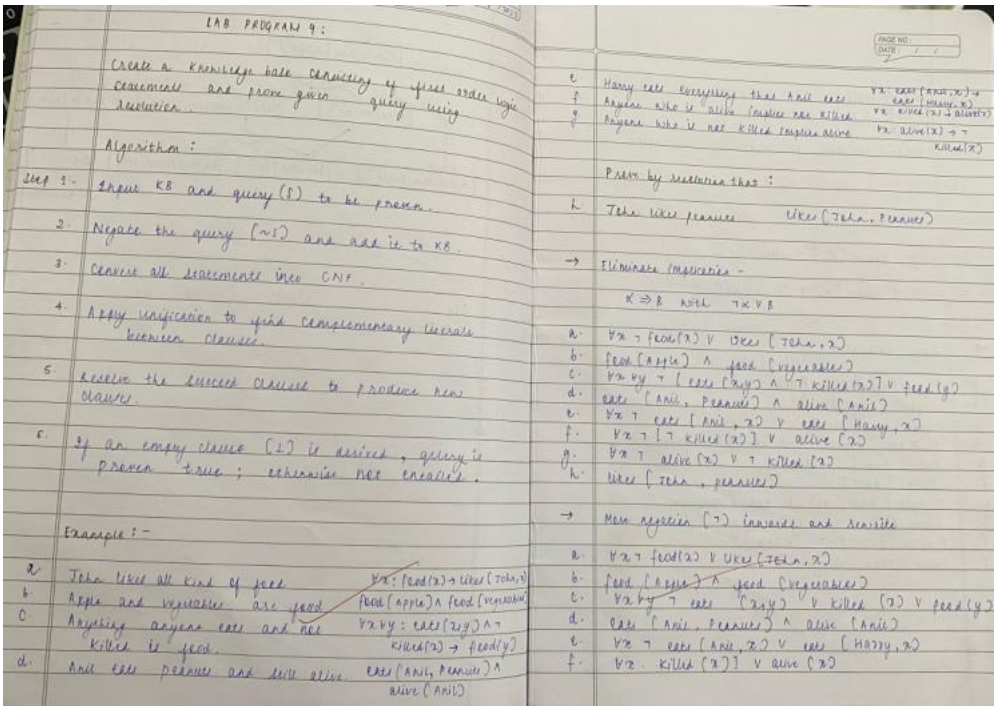
Output:

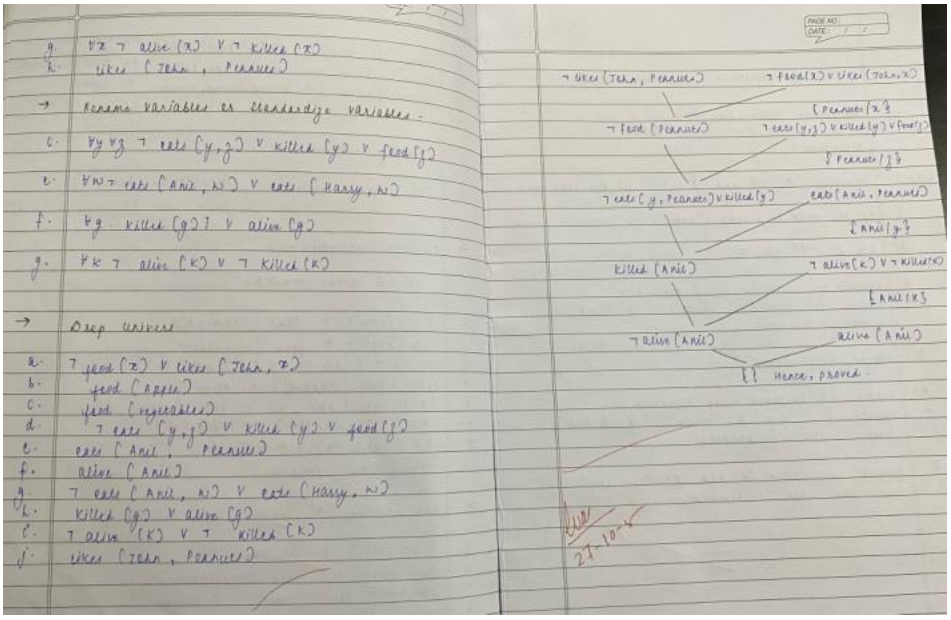
```
Robert is a criminal.
Samriddhi Singh 1BM23CS295
```

Program 9

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

Algorithm:





Code:

```
def fol_resolution(kb, query):
    print("\n" + "="*55)
    print(" KNOWLEDGE BASE")
    print("="*55)
    for i, clause in enumerate(kb, start=1):
        print(f"  {i}. {clause}")

    print("\n" + "="*55)
```

```
    print("                    QUERY")
    print("="*55)
    print(f"  Prove: {query}")
    print(f"  Negated Query: ~{query}\n")

    print("="*55)
    print("               RESOLUTION PROCESS")
    print("="*55)
    print("Step 1: Convert all implications (→) to CNF (Conjunctive Normal Form).")
    print("Step 2: Eliminate all universal quantifiers (∀).")
    print("Step 3: Add negated query (~Query) to the KB.")
    print("Step 4: Apply resolution rule between matching clauses.")
    print("Step 5: Continue until the empty clause (⊥) is found.\n")

    # Simulated resolution steps for John likes peanuts problem
    print("="*55)
    print("               RESOLUTION TREE")
    print("="*55)
    print("""
                    [~Likes(John, Peanuts)]
                            |
                    [Food(Peanuts) → Likes(John, Peanuts)]
                            |
                    [Eats(Anil, Peanuts) ∧ ¬Killed(Anil) → Food(Peanuts)]
                            |
                    [Alive(Anil) → ¬Killed(Anil)]
                            |
                    [Alive(Anil)]
                            ↓
                      ⊥ (Contradiction Found)
    """)

    print("="*55)
    print(f"Therefore, the query '{query}' is PROVEN by Resolution.")
    print("="*55 + "\n")


print("\n FIRST ORDER LOGIC - RESOLUTION METHOD")
print("----------------------------------------")

n = int(input("Enter the number of statements in the Knowledge Base: "))
kb = []
print("\nEnter each statement (e.g., '∀x: Food(x) → Likes(John, x)'):")
for i in range(n):
    stmt = input(f"KB[{i+1}]: ")
    kb.append(stmt)

query = input("\nEnter the query to prove: ")

fol_resolution(kb, query)
print("Samriddhi Singh 1BM23CS295")
```

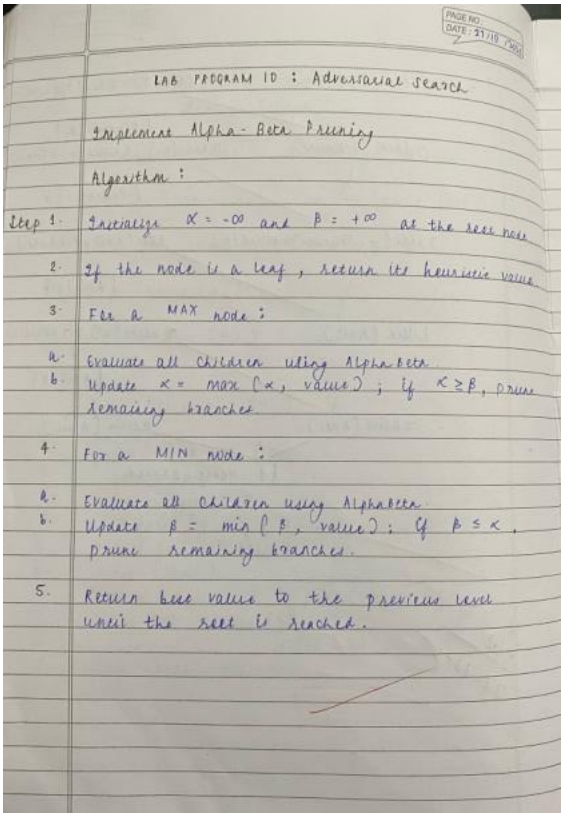
Output:

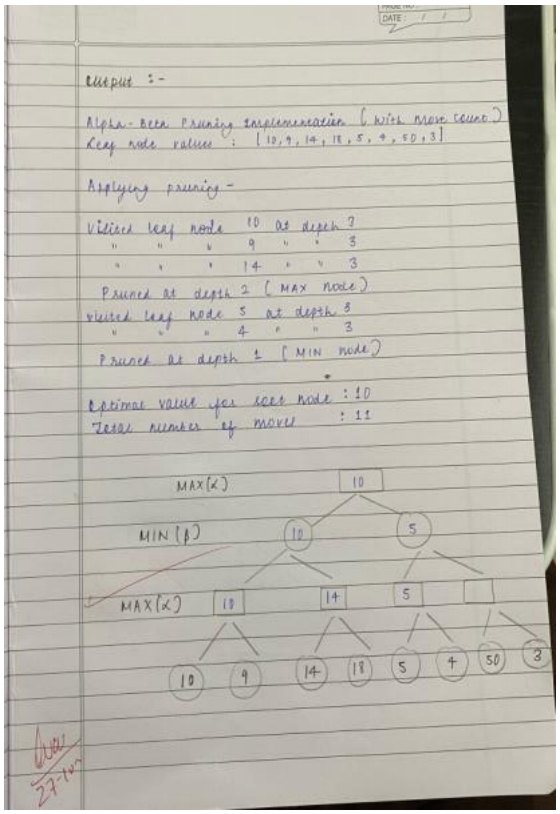
```
Enter the number of statements in the Knowledge Base: 9

Enter each statement (e.g., '∀x: Food(x) → Likes(John, x)'):
KB[1]: ∀x: Food(x) → Likes(John, x)
KB[2]: Food(Apple)
KB[3]:  Food(Vegetables)
KB[4]: ∀x, y: (Eats(x, y) ∧ ¬Killed(x)) → Food(y
KB[5]: Eats(Anil, Peanuts)
KB[6]: Alive(Anil)
KB[7]: ∀x, y: Eats(Harry, y) ← Eats(Anil, y)
KB[8]: ∀x: Alive(x) → ¬Killed(x)
KB[9]: ∀x: ¬Killed(x) → Alive(x)

Enter the query to prove: Likes(John, Peanuts)

============================================================
                    KNOWLEDGE BASE
============================================================
  1. ∀x: Food(x) → Likes(John, x)
  2. Food(Apple)
  3.  Food(Vegetables)
  4. ∀x, y: (Eats(x, y) ∧ ¬Killed(x)) → Food(y
  5. Eats(Anil, Peanuts)
  6. Alive(Anil)
  7. ∀x, y: Eats(Harry, y) ← Eats(Anil, y)
  8. ∀x: Alive(x) → ¬Killed(x)
  9. ∀x: ¬Killed(x) → Alive(x)

============================================================
                          QUERY
============================================================
  Prove: Likes(John, Peanuts)
  Negated Query: ~Likes(John, Peanuts)

============================================================
                          QUERY
============================================================
  Prove: Likes(John, Peanuts)
  Negated Query: ~Likes(John, Peanuts)

============================================================
                    RESOLUTION PROCESS
============================================================
Step 1: Convert all implications (→) to CNF (Conjunctive Normal Form).
Step 2: Eliminate all universal quantifiers (∀).
Step 3: Add negated query (~Query) to the KB.
Step 4: Apply resolution rule between matching clauses.
Step 5: Continue until the empty clause (⊥) is found.

============================================================
                    RESOLUTION TREE
============================================================

                          [~Likes(John, Peanuts)]
                                    |
                          [Food(Peanuts) → Likes(John, Peanuts)]
                                    |
                          [Eats(Anil, Peanuts) ∧ ¬Killed(Anil) → Food(Peanuts)]
                                    |
                          [Alive(Anil) → ¬Killed(Anil)]
                                    |
                          [Alive(Anil)]
                                    ↓
                            ⊥ (Contradiction Found)

============================================================
✅ Therefore, the query 'Likes(John, Peanuts)' is PROVEN by Resolution.
============================================================

Samriddhi Singh 1BM23CS295
```

Program 10

Implement Alpha-Beta Pruning.

Algorithm:





Code:

```
moves = 0

def alphabeta(depth, nodeIndex, maximizingPlayer, values, alpha, beta):
    global moves
    moves += 1

    if depth == 3:
        print(f"Visited leaf node {values[nodeIndex]} at depth {depth}")
        return values[nodeIndex]

    if maximizingPlayer:
        best = float('-inf')
        for i in range(2):
            val = alphabeta(depth + 1, nodeIndex * 2 + i, False, values, alpha, beta)
            best = max(best, val)
            alpha = max(alpha, best)
            if beta <= alpha:
                print(f" Pruned at depth {depth} (MAX node)")
                break
        return best
    else:
        best = float('inf')
        for i in range(2):
            val = alphabeta(depth + 1, nodeIndex * 2 + i, True, values, alpha, beta)
            best = min(best, val)
            beta = min(beta, best)
            if beta <= alpha:
                print(f" Pruned at depth {depth} (MIN node)")
                break
        return best

print("Alpha–Beta Pruning Implementation (With Move Count)")

values = [10, 9, 14, 18, 5, 4, 50, 3]

print("Leaf node values:", values)

alpha = float('-inf')
beta = float('inf')

print("\nApplying Alpha–Beta Pruning...\n")

optimal_value = alphabeta(0, 0, True, values, alpha, beta)

print("\n----------------------------------------------")
```

```
print(f" Optimal value for the root node: {optimal_value}")
print(f" Total number of moves (nodes evaluated): {moves}")
print("-----------------------------------------------")
print("Samriddhi Singh 1BM23CS295")
```

Output:

```
Alpha-Beta Pruning Implementation (With Move Count)
Leaf node values: [10, 9, 14, 18, 5, 4, 50, 3]

Applying Alpha-Beta Pruning...

Visited leaf node 10 at depth 3
Visited leaf node 9 at depth 3
Visited leaf node 14 at depth 3
 Pruned at depth 2 (MAX node)
Visited leaf node 5 at depth 3
Visited leaf node 4 at depth 3
 Pruned at depth 1 (MIN node)

-----------------------------------------------
 Optimal value for the root node: 10
 Total number of moves (nodes evaluated): 11
-----------------------------------------------
Samriddhi Singh 1BM23CS295
```