VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Samhitha A (1BM23CS293)

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 **Samhitha A (1BM23CS293),** 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

1BM23CS293

INDEX

Name SAMHITHA.A

Standard Section Roll No.

Subject

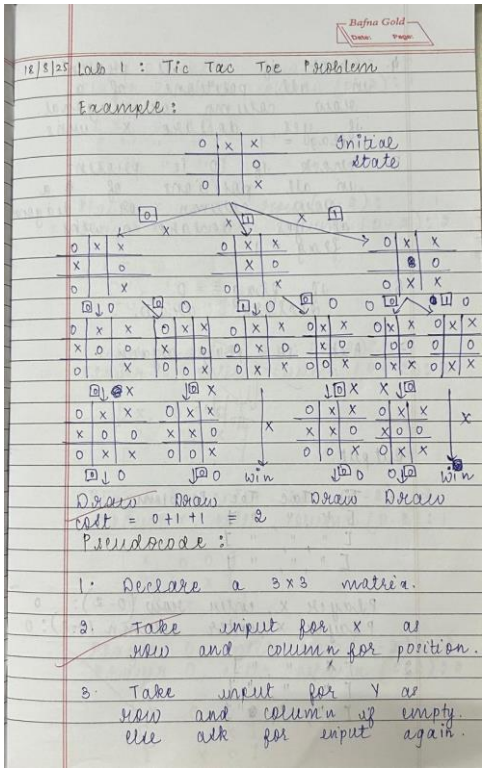
SL No.	Date	Title	Page No.	Teacher Sign / Remarks
1	18/8/25	Tic Tac Toe Problem	10	25/9
2	25/8/25	Vacuum cleaner	10	1/9/25
3	a) 1/9/25	a) BFS (without Heuristic approach)		
	b)	b) BFS (with Heuristic approach)		10
	c)	c) Iterative Deepening		1-9
4.	8/9/25	A* algorithm - Misplaced tiles - Manhattan distance.		10 8-9
5	15/9/25	Hill Climbing (N Queens Problem) Simulated Annealing		10 15-9
6.	22/9/25	Propositional Logic	—	10 22/9
7	29/9/25	Unification Algorithm	—	10
8	13/10/25	First Order Logic	—	10
9	27/10/25	First Order Logic (with KB)		10
10	27/10/25	Adversarial Search	—	27-10

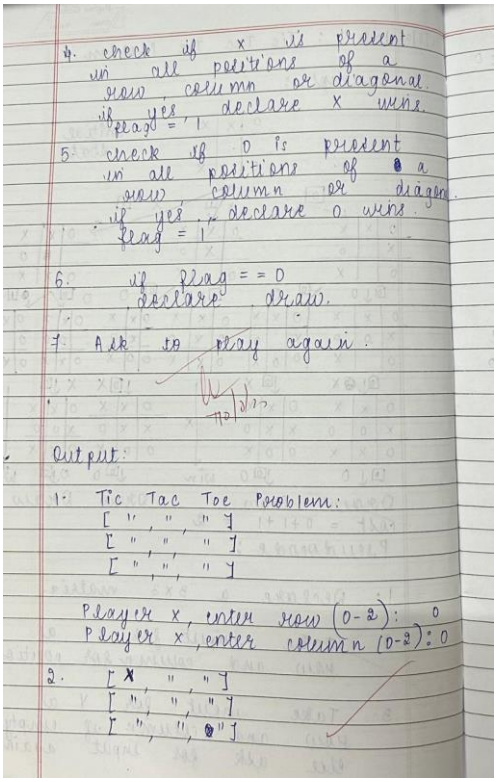
Completed

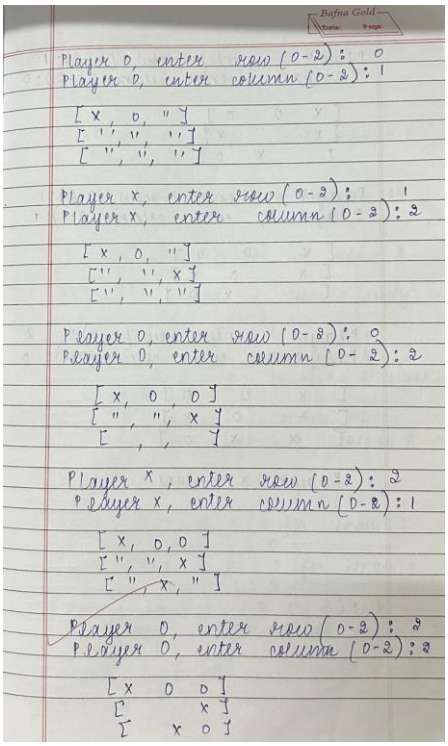
Github Link:
https://github.com/Samhithagit/Samhitha_1BM23CS293_AI_LAB

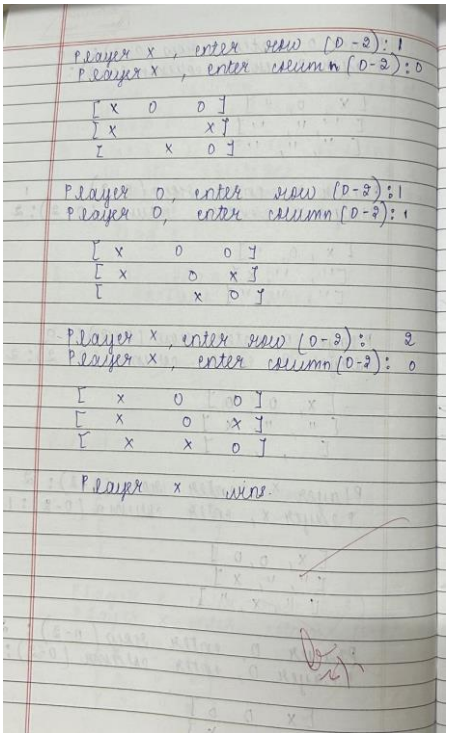
Program 1
Implement Tic –Tac –Toe Game

Algorithm:









Code:

```
def print_board(board):
    for row in board:
        display_row = [(""" if cell == ' ' else cell) for cell in row]
        print("[ " + ", ".join(display_row) + " ]")

def check_winner(board, player):
    for i in range(3):
        if all(s == player for s 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_board_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'
    moves = 0

    while True:
        print_board(board)
        print(f"Player {current_player}'s turn. (Enter row & col between 0-2)")

        while True:
            try:
                row = int(input("Enter row (0-2): "))
                col = int(input("Enter column (0-2): "))
                if row in range(3) and col in range(3):
                    if board[row][col] == ' ':
                        board[row][col] = current_player
                        moves += 1
                        break
                    else:
```

```
                print("That spot is already taken. Try again.")
            else:
                print("Row and column must be between 0 and 2. Try again.")
        except ValueError:
            print("Invalid input. Please enter numbers between 0 and 2.")

    if check_winner(board, current_player):
        print_board(board)
        print(f"Player {current_player} wins!")
        print(f"Cost of path: {moves}")
        break

    if is_board_full(board):
        print_board(board)
        print("It's a tie!")
        print(f"Cost of path: {moves}")
        break

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

if __name__ == "__main__":
    tic_tac_toe()
print("Samhitha A 1BM23CS293")
```

Output case1:

```
[ ', ', ' ]
[ ', ', ' ]
[ ', ', ' ]
Player X's turn. (Enter row & col between 0-2)
Enter row (0-2): 0
Enter column (0-2): 0
[ X, ', ' ]
[ ', ', ' ]
[ ', ', ' ]
Player O's turn. (Enter row & col between 0-2)
Enter row (0-2): 0
Enter column (0-2): 1
[ X, O, ' ]
[ ', ', ' ]
[ ', ', ' ]
```

Player X's turn. (Enter row & col between 0-2)
Enter row (0-2): 1
Enter column (0-2): 2
[X, O, "]
[", ", X]
[", ", "]
Player O's turn. (Enter row & col between 0-2)
Enter row (0-2): 0
Enter column (0-2): 2
[X, O, O]
[", ", X]
[", ", "]
Player X's turn. (Enter row & col between 0-2)
Enter row (0-2): 2
Enter column (0-2): 1
[X, O, O]
[", ", X]
[", X, "]
Player O's turn. (Enter row & col between 0-2)
Enter row (0-2): 2
Enter column (0-2): 2
[X, O, O]
[", ", X]
[", X, O]
Player X's turn. (Enter row & col between 0-2)
Enter row (0-2): 1
Enter column (0-2): 0
[X, O, O]
[X, ", X]
[", X, O]
Player O's turn. (Enter row & col between 0-2)
Enter row (0-2): 1
Enter column (0-2): 1
[X, O, O]
[X, O, X]
[", X, O]
Player X's turn. (Enter row (0-2): 2
Enter column (0-2): 0
[X, O, O]
[X, O, X]
[X, X, O]

Player X wins!
Cost of path: 9
Samhitha A 1BM23CS293

Output case2:

[', ', ']
[', ', ']
[', ', ']
Player X's turn. (Enter row & col between 0-2)
Enter row (0-2): 0
Enter column (0-2): 1
[', X, ']
[', ', ']
[', ', ']
Player O's turn. (Enter row & col between 0-2)
Enter row (0-2): 0
Enter column (0-2): 0
[O, X, ']
[', ', ']
[', ', ']
Player X's turn. (Enter row & col between 0-2)
Enter row (0-2): 2
Enter column (0-2): 2
[O, X, ']
[', ', ']
[', ', X]
Player O's turn. (Enter row & col between 0-2)
Enter row (0-2): 1
Enter column (0-2): 2
[O, X, ']
[', ', O]
[', ', X]
Player X's turn. (Enter row & col between 0-2)
Enter row (0-2): 0
Enter column (0-2): 2
[O, X, X]
[', ', O]
[', ', X]
Player O's turn. (Enter row & col between 0-2)
Enter row (0-2): 2

Enter column (0-2): 0
[O, X, X]
[', ', O]
[O, ', X]
Player X's turn. (Enter row & col between 0-2)
Enter row (0-2):
Invalid input. Please enter numbers between 0 and 2.
Enter row (0-2): 1
Enter column (0-2): 1
[O, X, X]
[', X, O]
[O, ', X]
Player O's turn. (Enter row & col between 0-2)
Enter row (0-2): 1
Enter column (0-2): 0
[O, X, X]
[O, X, O]
[O, ', X]
Player O wins!
Cost of path: 8
Samhitha A 1BM23CS293

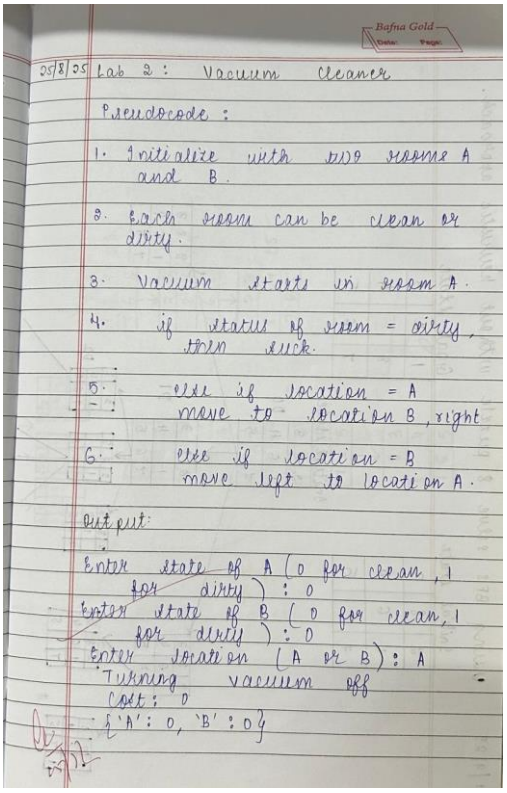
Output case3:

[', ', ']
[', ', ']
[', ', ']
Player X's turn. (Enter row & col between 0-2)
Enter row (0-2): 0
Enter column (0-2): 0
[X, ', ']
[', ', ']
[', ', ']
Player O's turn. (Enter row & col between 0-2)
Enter row (0-2): 1
Enter column (0-2): 0
[X, ', ']
[O, ', ']
[', ', ']
Player X's turn. (Enter row & col between 0-2)
Enter row (0-2): 0
Enter column (0-2): 2
[X, ', X]
[O, ', ']
[', ', ']
Player O's turn. (Enter row & col between 0-2)

Enter row (0-2): 0
Enter column (0-2): 1
[X, O, X]
[O, ", "]
[", ", "]
Player X's turn. (Enter row & col between 0-2)
Enter row (0-2): 1
Enter column (0-2): 2
[X, O, X]
[O, ", X]
[", ", "]
Player O's turn. (Enter row (0-2): 2
Enter column (0-2): 2
[X, O, X]
[O, ", X]
[", ", O]
Player X's turn. (Enter row & col between 0-2)
Enter row (0-2): 1
Enter column (0-2): 1
[X, O, X]
[O, X, X]
[", ", O]
Player O's turn. (Enter row & col between 0-2)
Enter row (0-2): 2
Enter column (0-2): 0
[X, O, X]
[O, X, X]
[O, ", O]
Player X's turn. (Enter row & col between 0-2)
Enter row (0-2): 2
Enter column (0-2): 1
[X, O, X]
[O, X, X]
[O, X, O]
It's a tie!
Cost of path: 9
Samhitha A 1BM23CS293

Implement vacuum cleaner agent

Algorithm:



Code:

```
def vacuum_agent():
    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()

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

    if state['A'] == 0 and state['B'] == 0:
        print("Turning vacuum off")
        print("Cost:", cost)
        print(state)
        return

    if loc == 'A':
        if state['A'] == 0 and state['B'] == 1:
            print("A is clean")
```

```
        print("Moving vacuum right")
        print("Cleaned B.")
        state['B'] = 0; cost += 1
        print(f"Is B clean now? (0 if clean, 1 if dirty): {state['B']}")
        print(f"Is A dirty? (0 if clean, 1 if dirty): {state['A']}")
        print("B is clean")
        print("Moving vacuum left")

    elif state['A'] == 1 and state['B'] == 0:  # <-- Case 3
        print("Cleaned A.")
        state['A'] = 0; cost += 1
        print(f"Is A clean now? (0 if clean, 1 if dirty): {state['A']}")
        print(f"Is B dirty? (0 if clean, 1 if dirty): {state['B']}")
        print("A is clean")
        print("Moving vacuum right")

    elif state['A'] == 1 and state['B'] == 1:
        print("Cleaned A.")
        state['A'] = 0; cost += 1
        print("Moving vacuum right")
        print("Cleaned B.")
        state['B'] = 0; cost += 1
        print(f"Is B clean now? (0 if clean, 1 if dirty): {state['B']}")
        print(f"Is A dirty? (0 if clean, 1 if dirty): {state['A']}")
        print("B is clean")
        print("Moving vacuum left")

elif loc == 'B':
    if state['B'] == 0 and state['A'] == 1:
        print("B is clean")
        print("Moving vacuum left")
        print("Cleaned A.")
        state['A'] = 0; cost += 1
        print(f"Is A clean now? (0 if clean, 1 if dirty): {state['A']}")
        print(f"Is B dirty? (0 if clean, 1 if dirty): {state['B']}")
        print("A is clean")
        print("Moving vacuum right")

    elif state['B'] == 1 and state['A'] == 0:
        print("Cleaned B.")
        state['B'] = 0; cost += 1
```

```
            print(f"Is B clean now? (0 if clean, 1 if dirty): {state['B']}")
            print(f"Is A dirty? (0 if clean, 1 if dirty): {state['A']}")
            print("B is clean")
            print("Moving vacuum left")

        elif state['B'] == 1 and state['A'] == 1:
            print("Cleaned B.")
            state['B'] = 0; cost += 1
            print("Moving vacuum left")
            print("Cleaned A.")
            state['A'] = 0; cost += 1
            print(f"Is A clean now? (0 if clean, 1 if dirty): {state['A']}")
            print(f"Is B dirty? (0 if clean, 1 if dirty): {state['B']}")
            print("A is clean")
            print("Moving vacuum right")

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


vacuum_agent()


print("Samhitha A 1BM23CS293")
```

OUTPUT Case1:

```
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
Turning vacuum off
Cost: 0
{'A': 0, 'B': 0}
Samhitha A 1BM23CS293
```

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}
Samhitha A 1BM23CS293
```

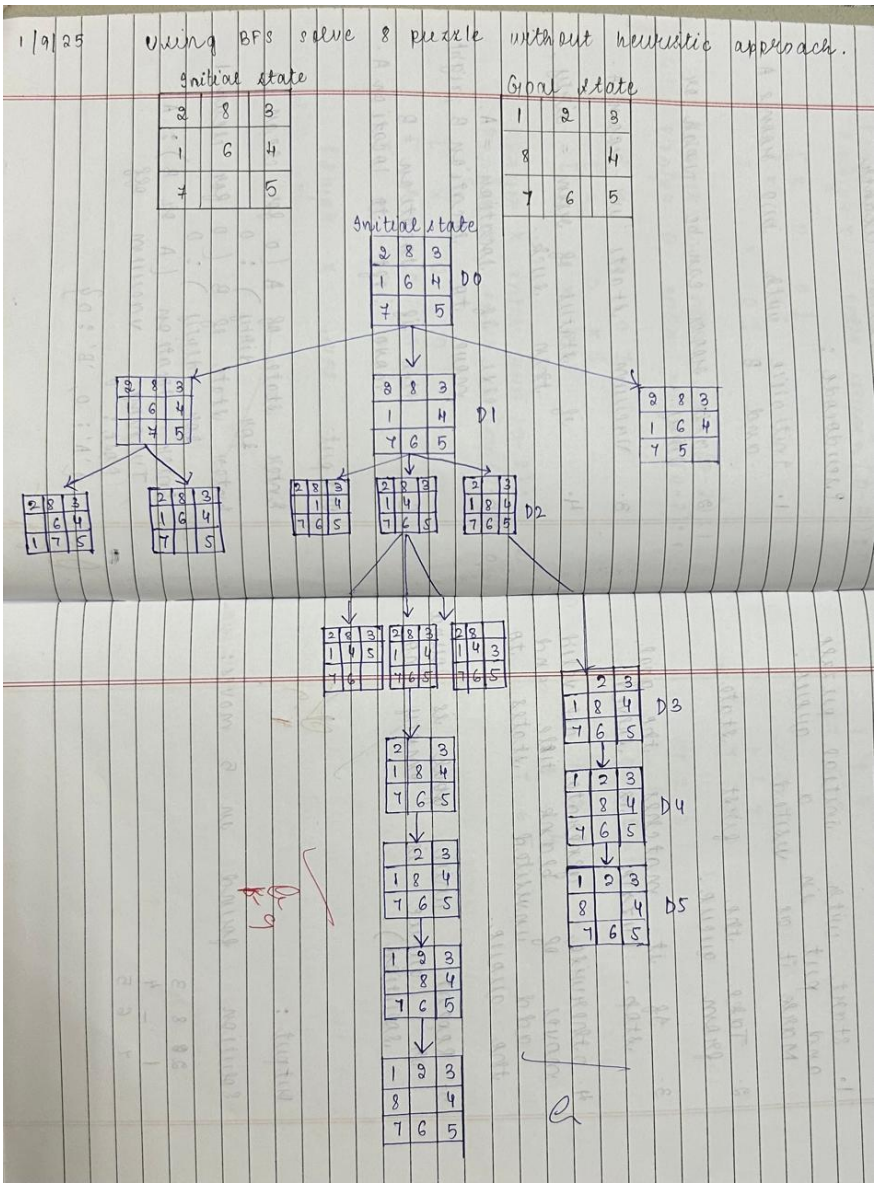
OUTPUT Case3:

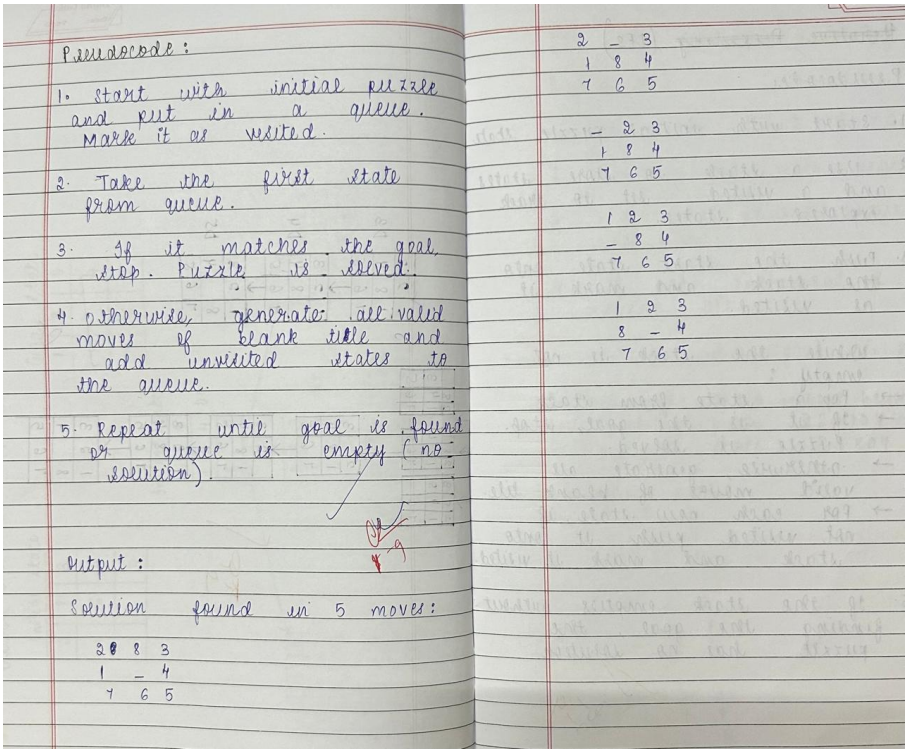
```
Enter state of A (0 for clean, 1 for dirty): 1
Enter state of B (0 for clean, 1 for dirty): 0
Enter location (A or B): A
Cleaned A.
Is A clean now? (0 if clean, 1 if dirty): 0
Is B dirty? (0 if clean, 1 if dirty): 0
A is clean
Moving vacuum right
Cost: 1
{'A': 0, 'B': 0}
Samhitha A 1BM23CS293
```

Program2:

Implement 8 puzzle problems using Breath First Search (BFS)

Algorithm:





Code:

```
from collections import deque

moves = {
    0: [1, 3],
    1: [0, 2, 4],
    2: [1, 5],
    3: [0, 4, 6],
    4: [1, 3, 5, 7],
    5: [2, 4, 8],
    6: [3, 7],
    7: [4, 6, 8],
    8: [5, 7]
}

def bfs(start, goal):
    queue = deque()
    queue.append((start, start.index('0'), []))  # state, index of zero, path
    visited = set()
    visited.add(start)

    all_states = []  # To record all visited states (branches)

    while queue:
        state, zero_pos, path = queue.popleft()
        all_states.append(state)

        if state == goal:
```

```
            return path, all_states

        for move_pos in moves[zero_pos]:
            new_state = list(state)
            new_state[zero_pos], new_state[move_pos] = new_state[move_pos], new_state[zero_pos]
            new_state_str = ''.join(new_state)

            if new_state_str not in visited:
                visited.add(new_state_str)
                queue.append((new_state_str, move_pos, path + [new_state_str]))
    return None, all_states

def print_puzzle(state_str):
    for i in range(0, 9, 3):
        print(' '.join(state_str[i:i+3]).replace('0', '_'))
    print()

def is_valid_state(state):
    return len(state) == 9 and set(state) == set('012345678')

if __name__ == '__main__':
    start_state = input("Enter the initial state (9 digits, 0 for blank): ").strip()
    goal_state = input("Enter the goal state (9 digits, 0 for blank): ").strip()

    if not is_valid_state(start_state):
        print("Invalid initial state! Make sure to enter exactly 9 digits including 0.")
    elif not is_valid_state(goal_state):
        print("Invalid goal state! Make sure to enter exactly 9 digits including 0.")
    else:
        print("\nInitial State:")
        print_puzzle(start_state)
        print("Goal State:")
        print_puzzle(goal_state)

        path, all_states = bfs(start_state, goal_state)

        print(f"Total states explored (branches): {len(all_states)}")
        print("All explored states:")
        for state in all_states:
            print_puzzle(state)

        if path:
            print(f"Solution found in {len(path)} moves:")
            for step in path:
                print_puzzle(step)
        else:
```

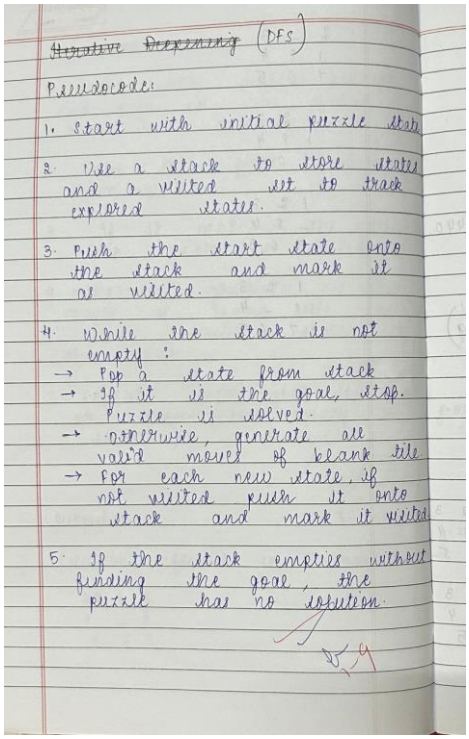
```
        print("No solution found.")

    print("Samhitha A 1BM23CS293")
```

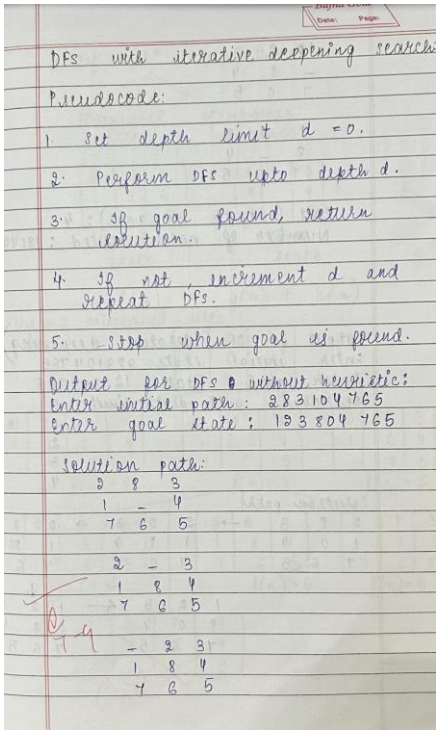
Output:

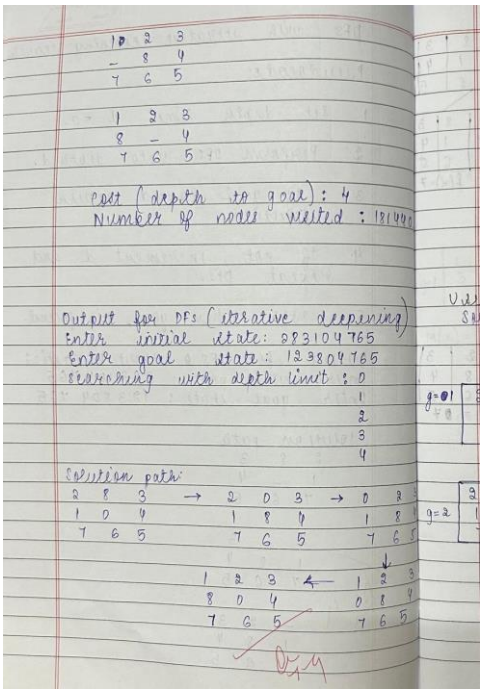
```
Enter initial state (e.g., 54_618732): 2831647_5
Enter goal state (e.g., 12345678_): 1238_4765
Minimum cost: 5

Steps:
283
164
7_5

283
1_4
765

2_3
184
765

_23
184
765

123
_84
765

123
8_4
765
```

Implement 8 puzzle problems using Depth First Search (DFS)

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("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
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.")

print("Samhitha A 1BM23CS293")
```

Output:

```
Enter the initial state (3x3, 0 for empty space):
283104765
Enter the goal state (3x3, 0 for empty space):
123804765
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

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

2 0 3
1 8 4
7 6 5

0 2 3
1 8 4
7 6 5

1 2 3
0 8 4
7 6 5

1 2 3
8 0 4
7 6 5

Samhitha A 1BM23CS293
```

Implement Iterative deepening search 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

# ---- Main program ----
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("Samhitha A 1BM23CS293")
```

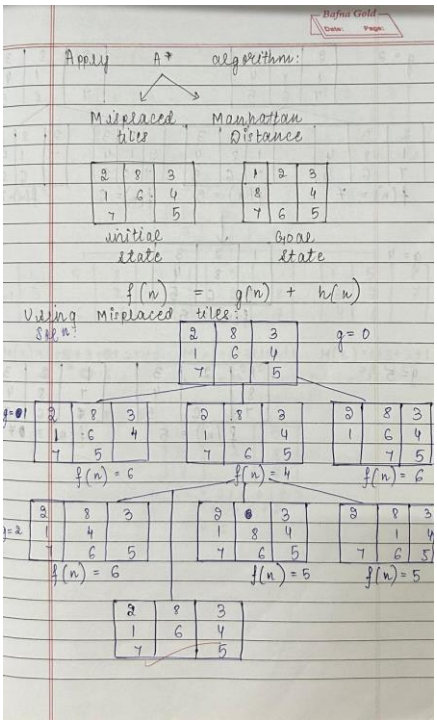
Output:

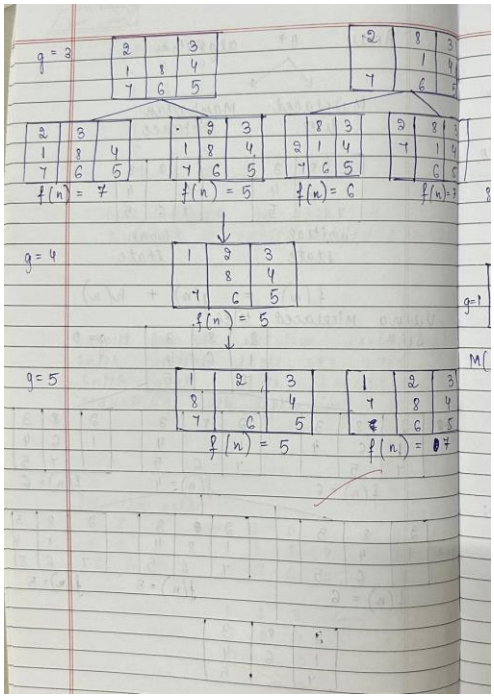
```
Steps (solution path):
283
1_4
765

2_3
184
765

_23
184
765

123
_84
765

123
8_4
765

Cost (depth to goal): 4
Number of nodes visited: 181440
Samhitha A 1BM23CS293
```

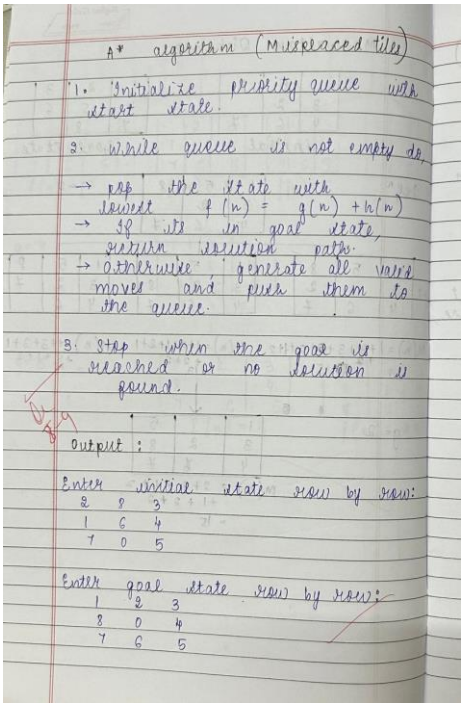
Program3

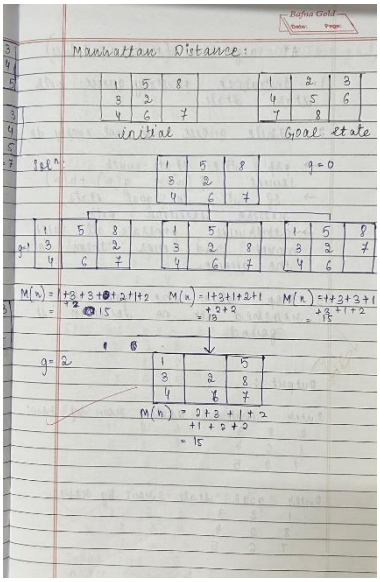
Implement A* search algorithm

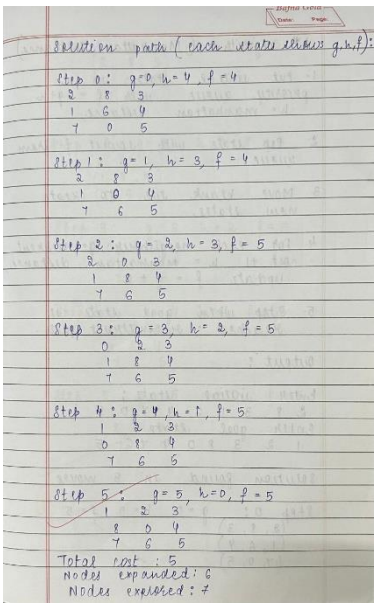
Algorithm:

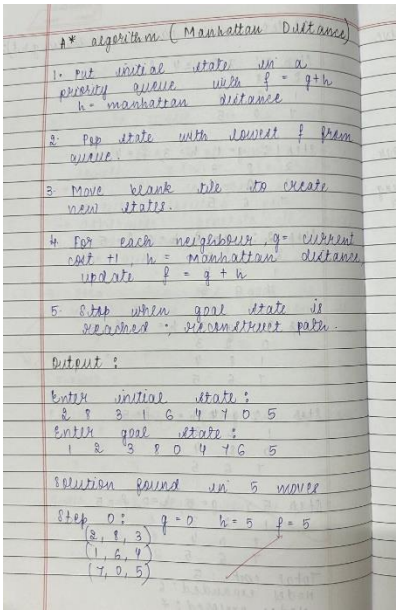


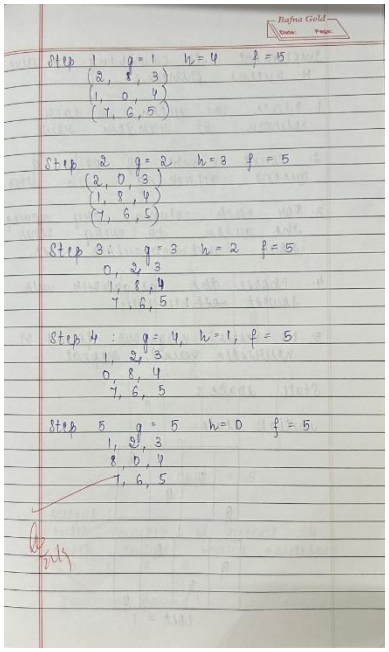












Code:

```
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):
    """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]:
```

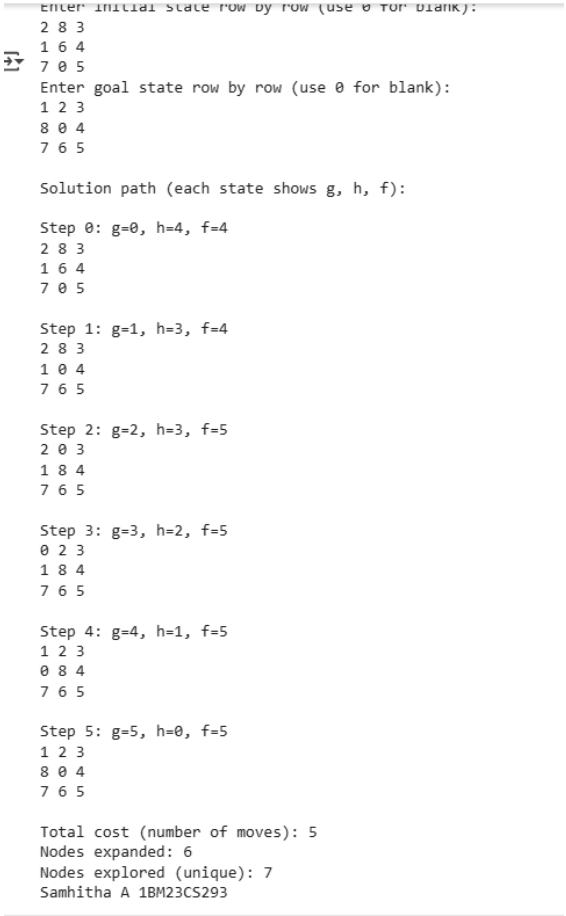
```
                inv += 1
    return inv


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  # 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}")


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


if __name__ == "__main__":
    main()


print("Samhitha A 1BM23CS293")
```

Output:



```
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
Samhitha A 1BM23CS293
```

Code:

```
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("Samhitha A 1BM23CS293")
```

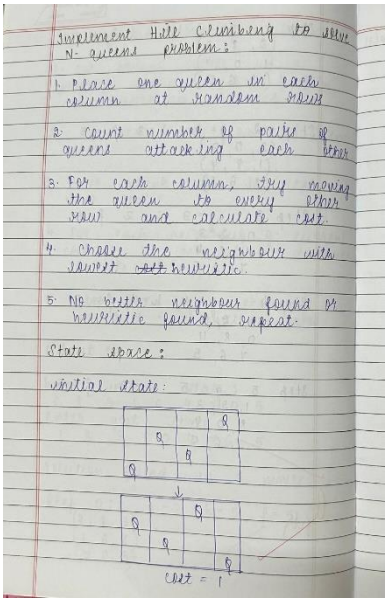
Output:

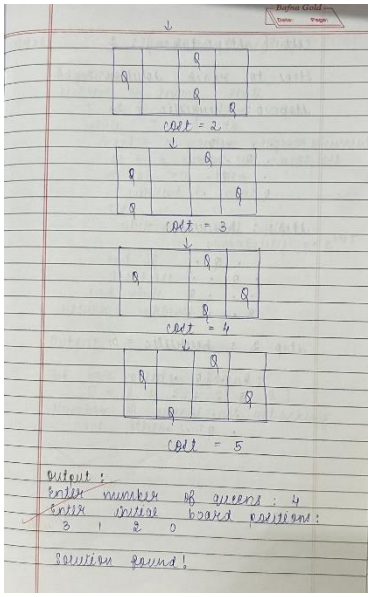
```
Enter initial state (9 numbers, 0 for blank):
2 8 3 1 6 4 7 0 5
Enter goal state (9 numbers, 0 for blank):
1 2 3 8 0 4 7 6 5

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)

Samhitha A 1BM23CS293
```

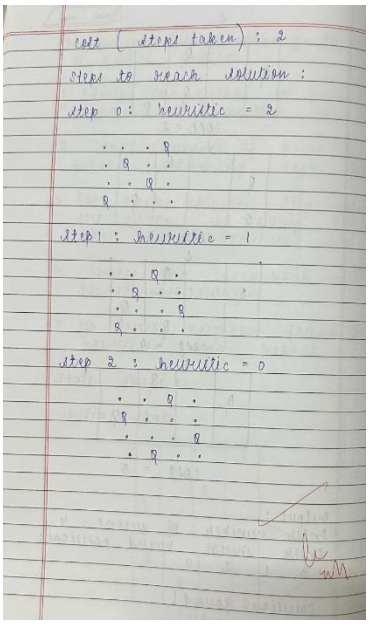
Program4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:







Code:

```
import random

def heuristic(board):
    """Count number of attacking pairs of queens."""
    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):
    """Generate neighbors by swapping the row positions of any two queens."""
    neighbors = []
    n = len(board)
    for i in range(n):
        for j in range(i + 1, n):
            new_board = list(board)
            # Swap queens in columns i and j
            new_board[i], new_board[j] = new_board[j], new_board[i]
```

```
            neighbors.append(new_board)
    return neighbors

def hill_climbing(board, max_sideways=100):
    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:
            # No improvement, reached local minimum
            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("Samhitha A 1BM23CS293")

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

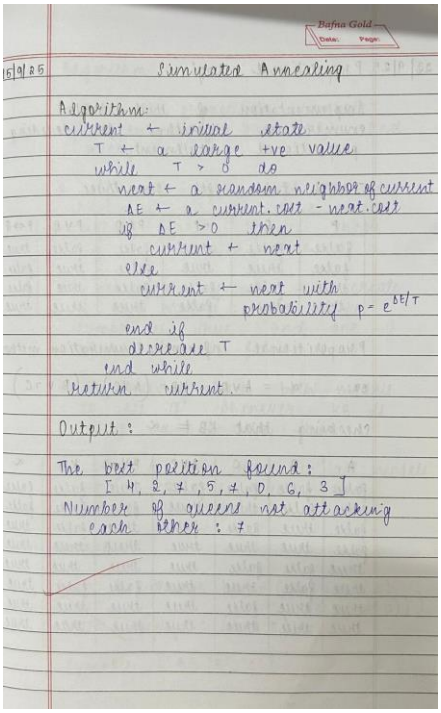
Output:

```
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 1 2 0

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 . .

Samhitha A 1BM23CS293
```

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))
```

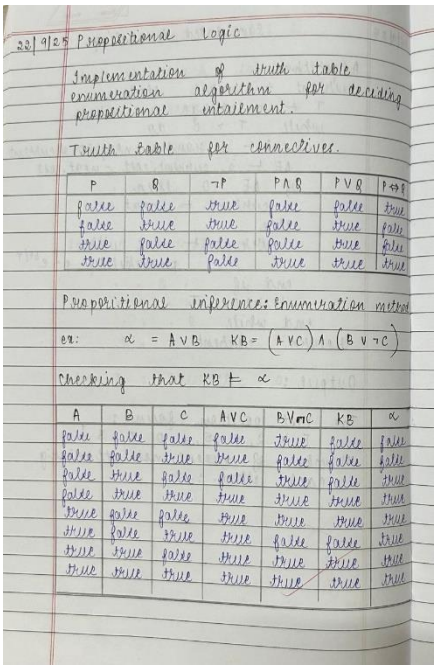
Output:

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

Program 6

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

Algorithm:



23/9/25 Propositional Logic

Implementation of truth table enumeration algorithm for deciding propositional entailment.

Truth table for connectives.

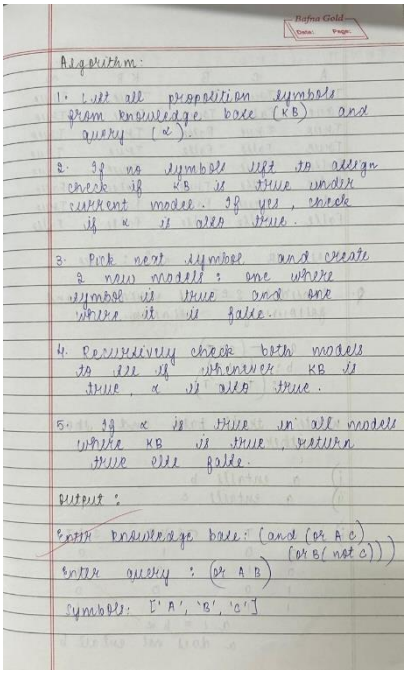
P	Q	¬P	P∧Q	P∨Q	P⇔Q
false	false	true	false	false	true
false	true	true	false	true	false
true	false	false	false	true	false
true	true	false	true	true	true

Propositional inference: Enumeration method

eg: $\alpha = A \vee B$ $KB = (A \vee C) \wedge (B \vee \neg C)$

Checking that $KB \models \alpha$

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



Algorithm:

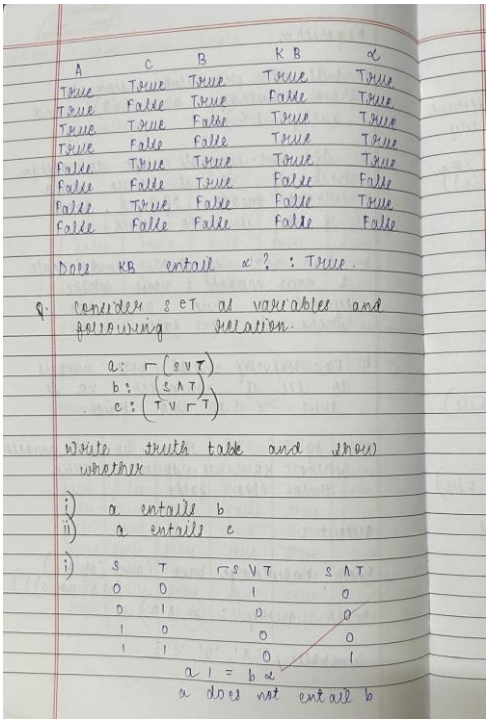
1. List all proposition symbols from knowledge base (KB) and query (α).
2. If no symbols left to assign check if KB is true under current model. If yes, check if α is also true.
3. Pick next symbol and create 2 new models: one where symbol is true and one where it is false.
4. Recursively check both models to see if whenever KB is true, α is also true.
5. If α is true in all models where KB is true, return true else false.

Output:

Enter knowledge base: (and (or A C) (or B (not C)))

Enter query: (or A B)

Symbols: ['A', 'B', 'C']



A	C	B	KB	α
True	True	True	True	True
True	False	True	False	True
True	True	False	True	True
True	False	False	True	True
False	True	True	True	True
False	False	True	False	False
False	True	False	False	True
False	False	False	False	False

Does KB entail α? : True.

9. Consider set of variables and following relation.

a: ¬(S ∨ T)
b: (S ∧ T)
c: (T ∨ ¬T)

Write truth table and show whether

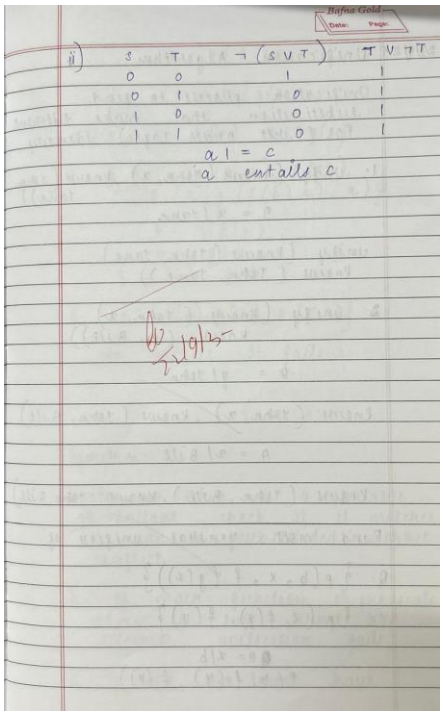
i) a entails b
ii) a entails c

i)

S	T	¬S ∨ T	S ∧ T
0	0	1	0
0	1	0	0
1	0	0	0
1	1	0	1

a 1 = b 0

a does not entail b



ii)

S	T	¬(S ∨ T)	T ∨ ¬T
0	0	1	1
0	1	0	1
1	0	0	1
1	1	0	1

a 1 = c

a entails c

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)

    # Print table header
    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}")

# User input
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}")
print("Samhitha A 1BM23CS293")
```

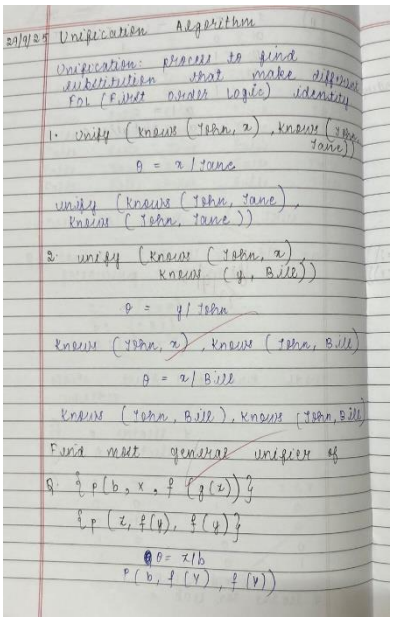
Output:

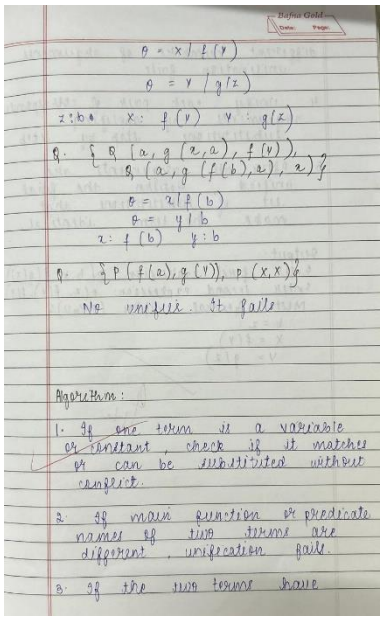
```
Enter knowledge base (e.g. (and A (or B C))): (and (or A C) (or B (not C)))
Enter query (e.g. A): (or A B)
Symbols: ['A', 'C', 'B']
A       C       B       KB      α
True    True    True    True    True
True    False   True    False   True
True    True    False   True    True
True    False   False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

Does KB entail α? : True
Samhitha A 1BM23CS293
```

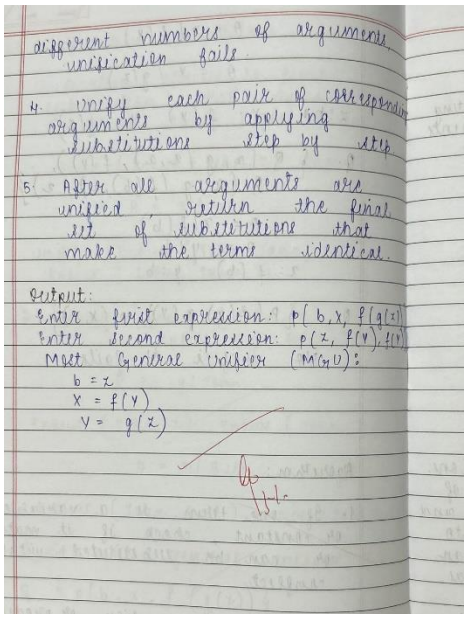
Program 7

Implement unification in first order logic.

Algorithm:







Code:

```
import re

def parse_term(expr):
    expr = expr.replace(' ', '')  # Remove spaces
    tokens = re.findall(r'[A-Za-z0-9_]+|[(),]', expr)  # Tokenize expression

    def parse(tokens):
        if not tokens:
            return None
        token = tokens.pop(0)

        if token == '(':
            # Start of argument list
            args = []
            while tokens[0] != ')':
                args.append(parse(tokens))
                if tokens[0] == ',':
                    tokens.pop(0)  # Remove comma
```

```
            tokens.pop(0)  # Remove ')'
            return args
        elif re.match(r'[A-Za-z_][A-Za-z0-9_]*', token):
            if tokens and tokens[0] == '(':
                tokens.pop(0)  # Remove '('
                args = []
                while tokens[0] != ')':
                    args.append(parse(tokens))
                    if tokens[0] == ',':
                        tokens.pop(0)
                tokens.pop(0)  # Remove ')'
                return [token] + args
            else:
                return token
        else:
            return token

    return parse(tokens)

def is_variable(x):
    # Treat any single letter (upper or lowercase) as a variable
    return isinstance(x, str) and len(x) == 1 and x.isalpha()

def occurs_check(var, term, subst):
    # Prevents variable from being unified with itself or its own term
    if var == term:
        return True
    elif is_variable(term) and term in subst:
        return occurs_check(var, subst[term], subst)
    elif isinstance(term, list):
        return any(occurs_check(var, t, subst) for t in term)
    else:
        return False

def unify(x, y, subst=None):
    if subst is None:
        subst = {}
    x = substitute(x, subst)
    y = substitute(y, subst)

    if x == y:
        return subst
    if is_variable(x):
        return unify_var(x, y, subst)
    if is_variable(y):
        return unify_var(y, x, subst)
    if isinstance(x, list) and isinstance(y, list) and len(x) == len(y):
```

```
        if x[0] != y[0]:  # Ensure the function symbols match
            return None
        for xi, yi in zip(x[1:], y[1:]):
            subst = unify(xi, yi, subst)
            if subst is None:
                return None
        return subst
    return None

def unify_var(var, x, subst):
    # Prevent variable from unifying with itself or its own term
    if var in subst:
        return unify(subst[var], x, subst)
    elif is_variable(x) and x in subst:
        return unify(var, subst[x], subst)
    elif occurs_check(var, x, subst):
        return None
    else:
        subst[var] = x
        return subst

def substitute(term, subst):
    if isinstance(term, list):
        # When term is a function like ['f', 'Y'], make it into f(Y)
        return [term[0]] + [substitute(t, subst) for t in term[1:]]
    elif is_variable(term) and term in subst:
        return substitute(subst[term], subst)
    else:
        return term

def format_term(term):
    # If the term is a list (function), format it correctly
    if isinstance(term, list):
        return f"{term[0]}({', '.join(map(format_term, term[1:]))})"
    return term

# Input
expr1 = input("Enter first expression: ")  # e.g. p(b,X,f(g(Z)))
expr2 = input("Enter second expression: ") # e.g. p(z,f(Y),f(Y))

term1 = parse_term(expr1)
term2 = parse_term(expr2)

result = unify(term1, term2)

if result is None:
    print("No unifier exists.")
```

```
else:
    print("Most General Unifier (MGU):")
    for k, v in result.items():
        print(f"{k} = {format_term(v)}")
print("Samhitha A 1BM23CS293")
```

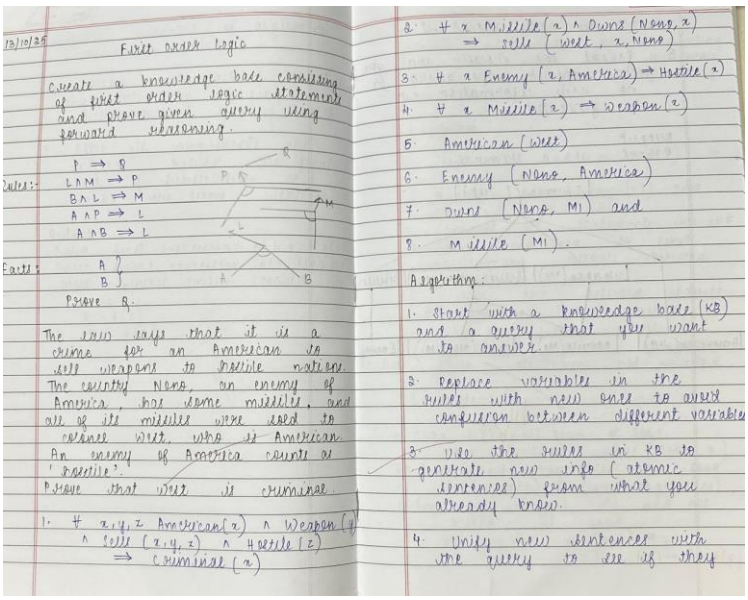
Output:

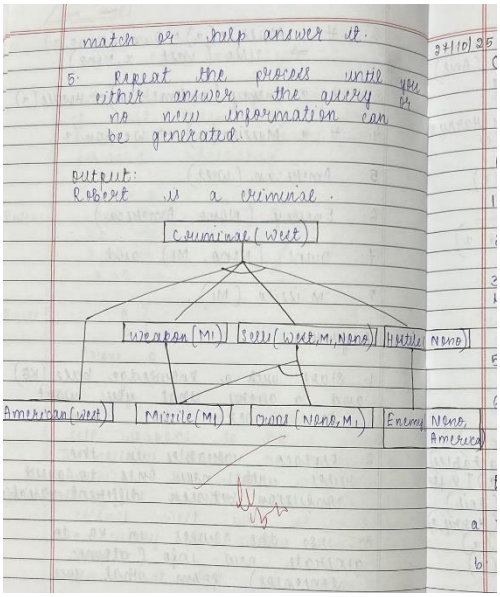
```
Enter first expression: p(b,X,f(g(Z)))
Enter second expression: p(z,f(Y),f(Y))
Most General Unifier (MGU):
b = z
X = f(Y)
Y = g(Z)
Samhitha A 1BM23CS293
```

Program 8

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

Algorithm:



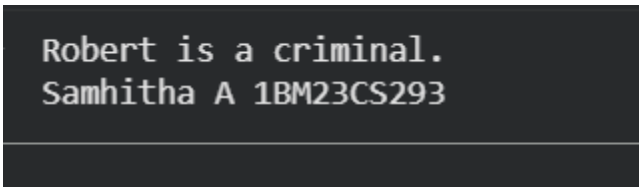


Code:

```
facts = {
    'American(Robert)': True,  # Robert is an American
    'Hostile(A)': True,        # Country A is hostile to America
    'Sells_Weapons(Robert, A)': True  # Robert sold weapons to Country A
}

# Define the law/rule: If American(X) and Hostile(Y) and Sells_Weapons(X, Y), then Crime(X)
def forward_reasoning(facts):
    # Apply the rule: 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  # Robert is a criminal

# Perform forward reasoning to see if we can deduce that Robert is a criminal
forward_reasoning(facts)

# Output the result based on the fact derived
if facts.get('Crime(Robert)', False):
    print("Robert is a criminal.")
    print("Samhitha A 1BM23CS293")
```

```
else:
    print("Robert is not a criminal.")
```

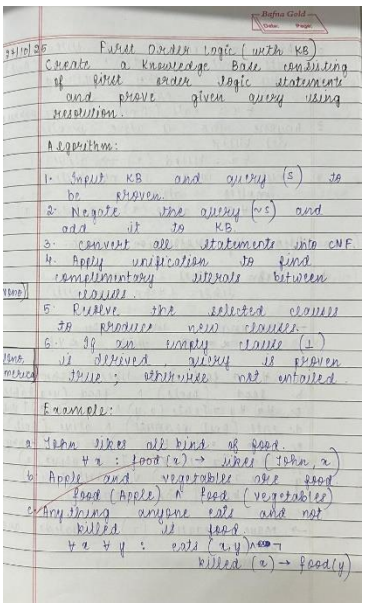
Output:

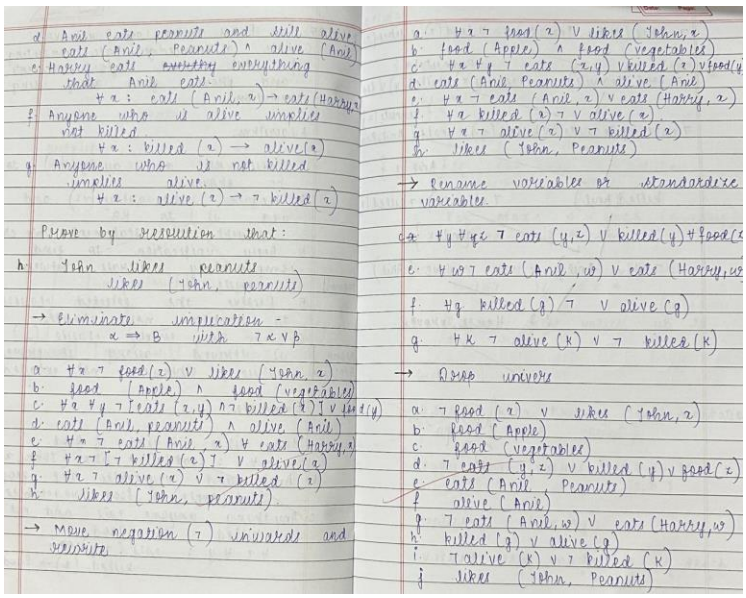


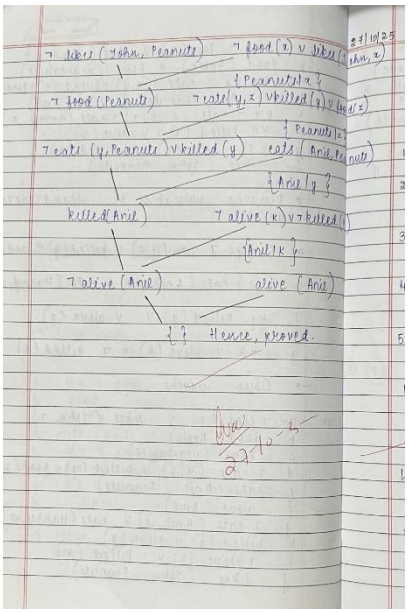
Program 9:

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")



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("Samhitha A 1BM23CS293")
```

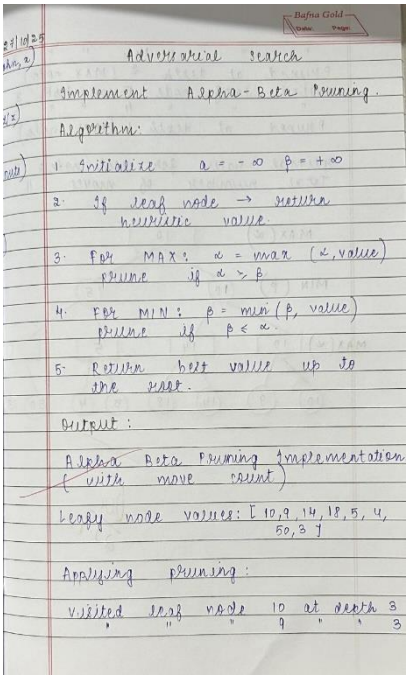
Output:

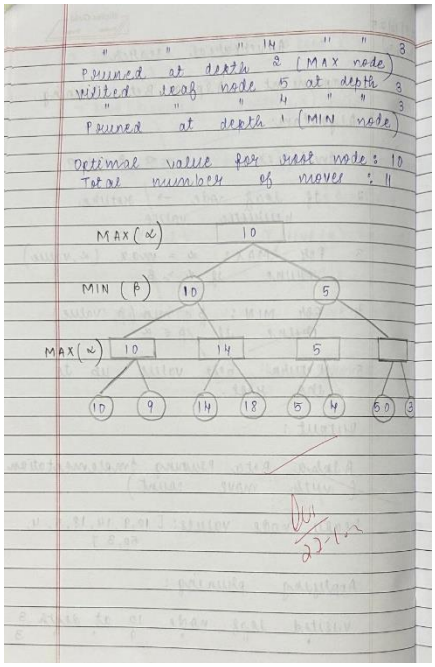
```
 FIRST ORDER LOGIC - RESOLUTION METHOD
Enter the number of statements in the Knowledge Base: 9

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)

============================================================
                RESOLUTION PROCESS
============================================================
Step 1: Convert all implications (→) to CNF (Conjunctive Normal Form).
Step 2: Eliminate all universal quantifiers (∀).
```

```
  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)

============================================================
                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.
============================================================

Samhitha A 1BM23CS293
```

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("Samhitha A 1BM23CS293")
```

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
-----------------------------------------------
Samhitha A 1BM23CS293
```