VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Hrishikesh R Prasad (1BM23CS367)

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

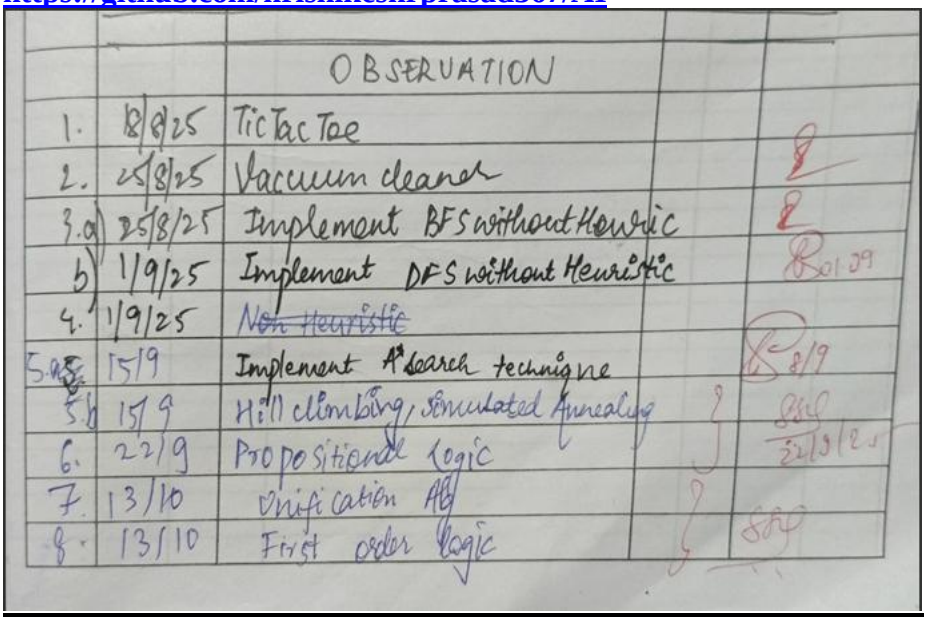
This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by **Hrishikesh R Prasad (1BM23CS367),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** 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.

Prof. Sonika Sharma D Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Index:

Github Link:
https://github.com/hrishikeshrprasad367/AI



		OBSERVATION		
1.	18/8/25	Tic Tac Toe		
2.	25/8/25	Vacuum cleaner		
3.a)	25/8/25	Implement BFS without Heuristic		
b)	1/9/25	Implement DFS without Heuristic		
4.	1/9/25	~~Non Heuristic~~		
5.a)	15/9	Implement A* search technique		
b)	15/9	Hill climbing, simulated Annealing		
6.	22/9	Propositional Logic		
7.	13/10	Unification		
8.	13/10	First order logic		

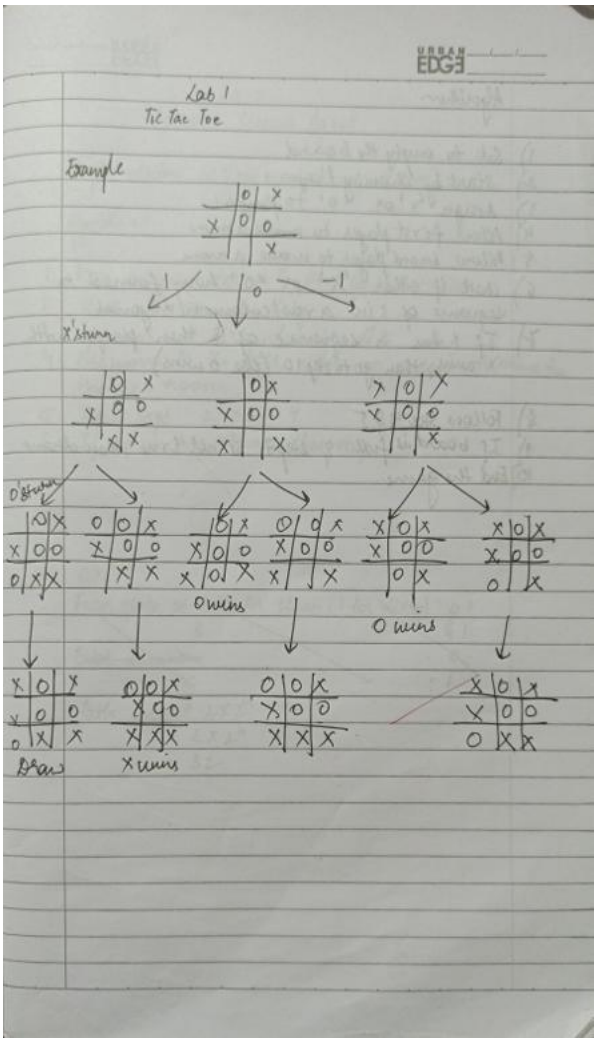
S. No.	Date	Title	Page No.	Teacher's Sign
9.	27/10/25	~~Unification~~ Resolution in FOL	10	[illegible] 6/11
10.	27/10/25	Alpha Beta Pruning	10	[illegible] 6/11

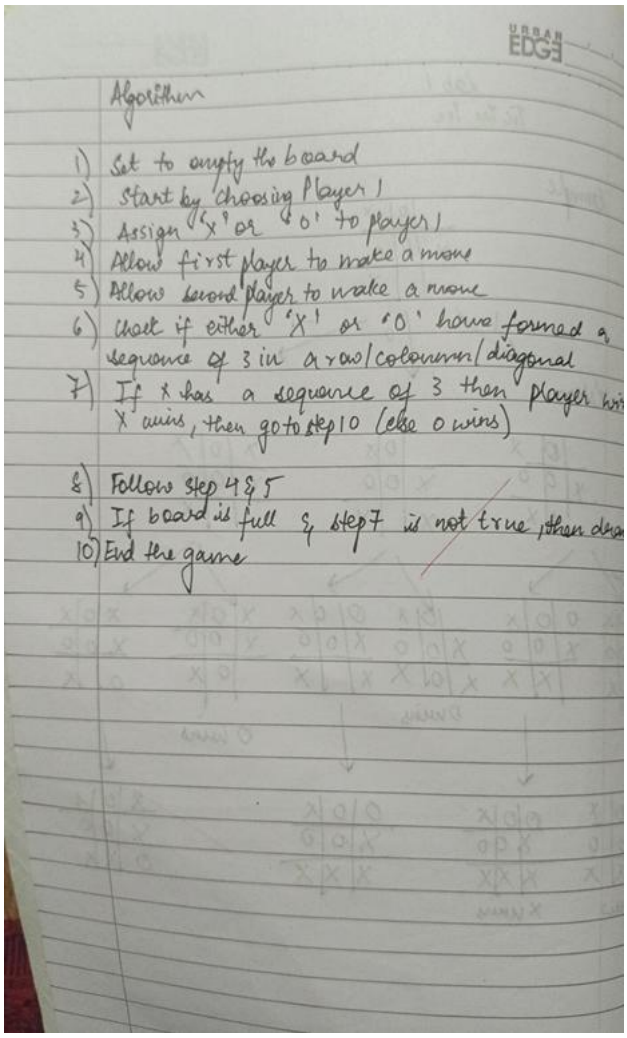
Program 1

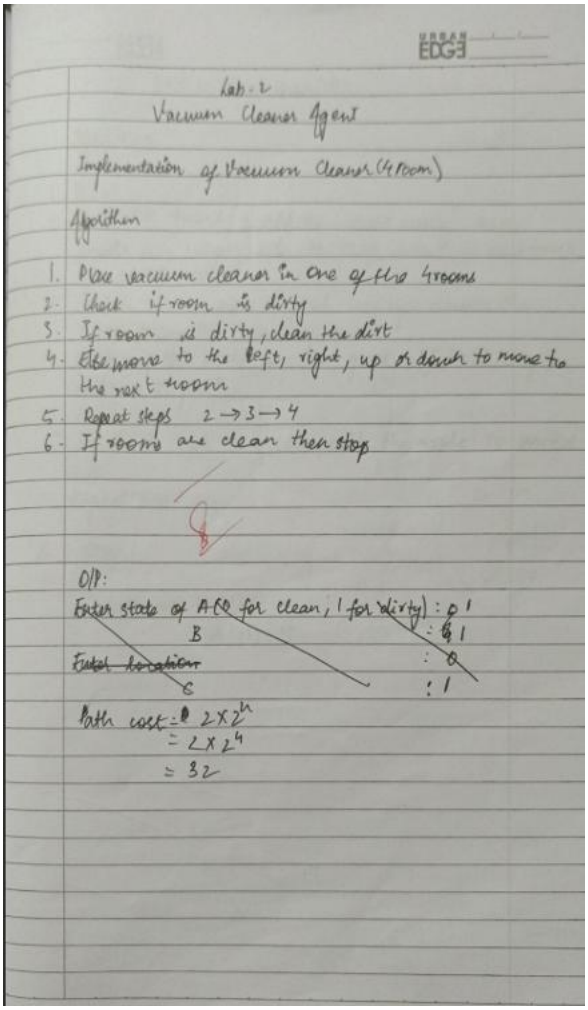
Implement Tic –Tac –Toe Game

Implement vacuum cleaner agent

Algorithm:







Lab-2

Vacuum Cleaner Agent

Implementation of Vacuum Cleaner (4 room)

Algorithm

1. Place vacuum cleaner in one of the 4 rooms
2. Check if room is dirty
3. If room is dirty, clean the dirt
4. Else move to the left, right, up or down to move to the next room
5. Repeat steps 2 → 3 → 4
6. If rooms are clean then stop

O/P:

Enter state of A&B for clean, 1 for dirty) : 0 1

B : 0 1

Enter location

C : 0 1

Path cost = 0 2×2^4

$= 2 \times 2^4$

$= 32$

Code:

TIC TAC TOE Game

```
def show_board(board):
    print("\n  1   2   3")
    for i in range(3):
        row_str = f"{i+1} "
        for j in range(3):
            row_str += board[i][j] if board[i][j] != " " else "."
            if j < 2:
                row_str += " | "
        print(row_str)
        if i < 2:
            print("  -----------")
    print()


def check_winner(board, mark):
    lines = [
        [board[0][0], board[0][1], board[0][2]],
        [board[1][0], board[1][1], board[1][2]],
        [board[2][0], board[2][1], board[2][2]],
        [board[0][0], board[1][0], board[2][0]],
        [board[0][1], board[1][1], board[2][1]],
        [board[0][2], board[1][2], board[2][2]],
        [board[0][0], board[1][1], board[2][2]],
        [board[0][2], board[1][1], board[2][0]]
    ]
    return [mark, mark, mark] in lines


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


def player_move(player, mark, board):
    while True:
        try:
            raw = input(f"Player {player} ({mark}) - Enter row col: ")
            r, c = map(int, raw.split())
            if 1 <= r <= 3 and 1 <= c <= 3:
                if board[r-1][c-1] == " ":
                    board[r-1][c-1] = mark
                    return
                else:
                    print("That spot is taken!")
            else:
                print("Coordinates must be between 1 and 3.")
        except ValueError:
            print("Enter two numbers separated by space.")


def play_game():
    board = [[" " for _ in range(3)] for _ in range(3)]
    players = [(1, "X"), (2, "O")]
    turn = 0

    # show empty board at start
```

```
    show_board(board)

    while True:
        p, m = players[turn % 2]
        player_move(p, m, board)

        # show updated board
        show_board(board)

        if check_winner(board, m):
            print(f"🎉 Player {p} wins!")
            break
        if board_full(board):
            print("🤝 It's a draw!")
            break

        turn += 1


play_game()
```

```
-------------
|   |   |   |
-------------
|   |   |   |
-------------
|   |   |   |
-------------
```

```
Player1, Enter the position to place X->  1 1
```

```
-------------
| X |   |   |
-------------
|   |   |   |
-------------
|   |   |   |
-------------
```

```
Player2, Enter the position to place O->  3 1
```

```
-------------
| X |   |   |
-------------
|   |   |   |
-------------
| O |   |   |
-------------
```

```
Player1, Enter the position to place X->  2 2

-------------
| X |   |   |
-------------
|   | X |   |
-------------
| O |   |   |
-------------

Player2, Enter the position to place O->  3 2

-------------
| X |   |   |
-------------
|   | X |   |
-------------
| O | O |   |
-------------

Player1, Enter the position to place X->  3 3

-------------
| X |   |   |
-------------
|   | X |   |
-------------
| O | O | X |
-------------

Congrats🎉, Player1 wins!!!
```

Vacuum Cleaner Agent

```python
def is_clean(status):
  return status[room_a] and status[room_b]


def simulate(state, choice, status, cost, do_clean=True):
    if is_clean(status):
        print("All rooms are clean")
        return cost

    if choice != 1 and choice != -1:
        print("Invalid choice")
        return cost

    # Vacuum in room A
    if state[0][0]:
        if choice == -1:
            if do_clean and not state[0][1]:
                state[0][1] = True
```

```
                status[room_a] = True
                print("Cleaned room A")
                cost += 1  # Cost of cleaning
            else:
                print("No cleaning in room A")
        elif choice == 1:
          state[0][0] = False
          state[1][0] = True
          print("Moved vacuum from A to B")
        else:
            print("Cannot move from A to B")

    # Vacuum in room B
    elif state[1][0]:
        if choice == 1:
            if do_clean and not state[1][1]:
                state[1][1] = True
                status[room_b] = True
                print("Cleaned room B")
                cost += 1  # Cost of cleaning
            else:
                print("No cleaning in room B")
        elif choice == -1:
            state[1][0] = False
            state[0][0] = True
            print("Moved vacuum from B to A")
        else:
            print("Cannot move from B to A")
    else:
        print("Vacuum is not in any room!")

    return cost


if __name__ == "__main__":
    room_a = 'A'
    room_b = 'B'
    state = [[True, False], [False, False]]
    status = {room_a:False, room_b:False}
    total_cost = 0  # Initialize total cost

    while True:
      if is_clean(status):
        print("All rooms are clean. Exiting.")
        print(f"Total cost: {total_cost}") # Display total cost
        break

      choice = int(input("Enter -1 to act in Room A, 1 to act in Room B: "))
      action = input("Enter 'c' to clean, 'm' to move without cleaning: ").lower()

      if action == 'c':
          total_cost = simulate(state, choice, status, total_cost)
      elif action == 'm':
          total_cost = simulate(state, choice, status, total_cost, False)
      else:
          print("Invalid action choice")
```

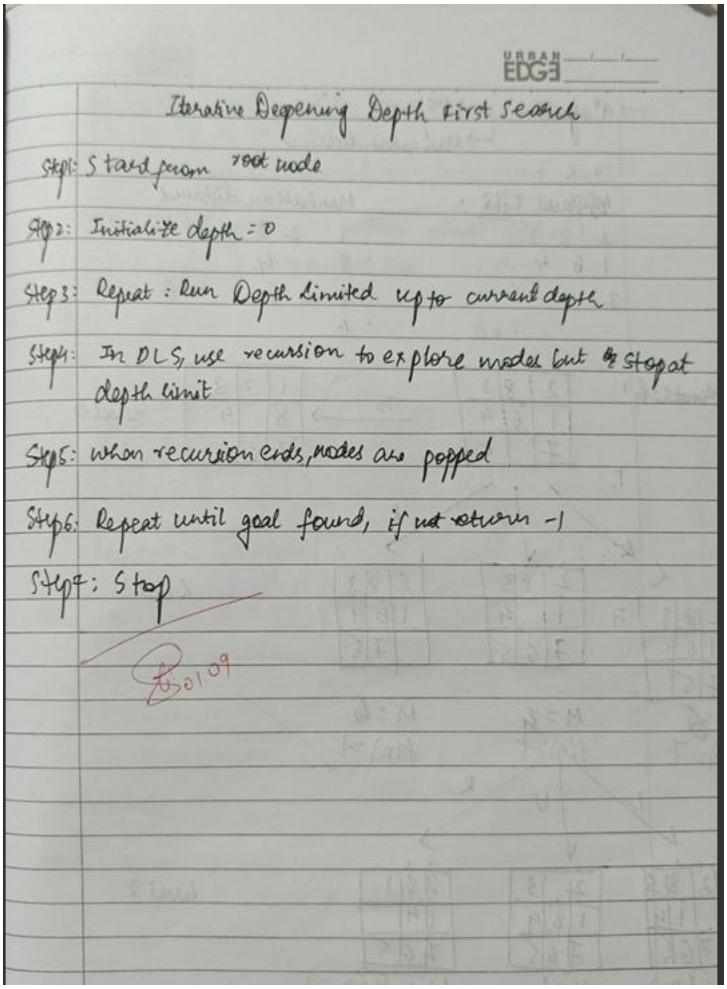
```
Enter -1 to act in Room A, 1 to act in Room B: -1
Enter 'c' to clean, 'm' to move without cleaning: c
```

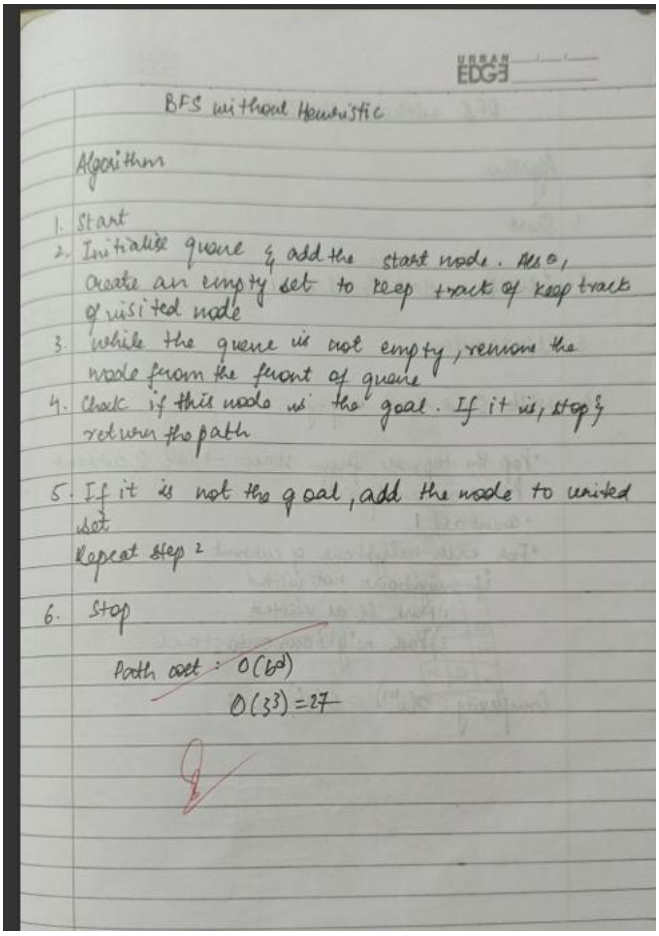
```
Cleaned room A
Enter -1 to act in Room A, 1 to act in Room B: 1
Enter 'c' to clean, 'm' to move without cleaning: m
Moved vacuum from A to B
Enter -1 to act in Room A, 1 to act in Room B: 1
Enter 'c' to clean, 'm' to move without cleaning: c
Cleaned room B
All rooms are clean. Exiting.
Total cost: 2
```

Program 2

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

Algorithm:





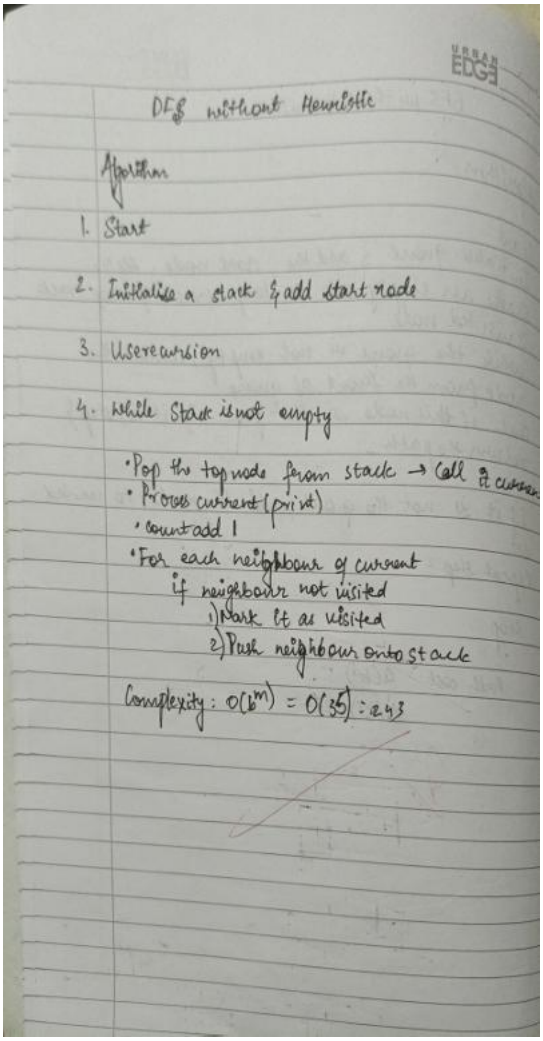
BFS without Heuristic

Algorithm

1. Start
2. Initialise queue & add the start node. Mark it, create an empty set to keep track of keep track of visited node
3. While the queue is not empty, remove the node from the front of queue
4. Check if this node is the goal. If it is, stop & return the path
5. If it is not the goal, add the node to visited set
   Repeat step 2
6. Stop

Path cost: $O(b^d)$

$O(3^3) = 27$



Code:

8 Puzzle using DFS

```
from collections import deque

# Helper to print board in 3x3 format
def print_board(state):
    for i in range(0, 9, 3):
        print(state[i:i+3])
    print()

# Generate possible moves
def get_neighbors(state):
    neighbors = []
    idx = state.index(0)  # blank space
    row, col = divmod(idx, 3)
    moves = []
    if row > 0: moves.append((-1, 0, 'Up'))
    if row < 2: moves.append((1, 0, 'Down'))
    if col > 0: moves.append((0, -1, 'Left'))
    if col < 2: moves.append((0, 1, 'Right'))
```

```
    for dr, dc, action in moves:
        new_row, new_col = row + dr, col + dc
        new_idx = new_row * 3 + new_col
        new_state = list(state)
        new_state[idx], new_state[new_idx] = new_state[new_idx], new_state[idx]
        neighbors.append((tuple(new_state), action))
    return neighbors

# DFS Search
def solve_puzzle(start, goal, max_depth=50):
    stack = [(start, [])]
    explored = set()

    while stack:
        state, path = stack.pop()

        if state in explored:
            continue
        explored.add(state)

        if state == goal:
            return path

        if len(path) < max_depth:
            for neighbor, action in get_neighbors(state):
                if neighbor not in explored:
                    stack.append((neighbor, path + [(action, neighbor)]))
    return None

if __name__ == "__main__":
    print("Enter the initial state (0 for blank, space-separated, 9 numbers):")
    start = tuple(map(int, input().split()))

    print("Enter the goal state (0 for blank, space-separated, 9 numbers):")
    goal = tuple(map(int, input().split()))

    print("\nSolving puzzle with DFS...\n")
    solution = solve_puzzle(start, goal)

    if solution:
        print("Solution found using DFS! (may not be optimal)\n")
        print("Total steps:", len(solution))
        current = start
        print("Initial State:")
        print_board(current)
        for step, state in solution:
            print("Move:", step)
            print_board(state)
    else:
        print("No solution found within depth limit.")
```

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

Solving puzzle with DFS...
```

```
Solution found using DFS! (may not be optimal)

Total steps: 49
Initial State:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

Move: Right
(2, 8, 3)
(1, 6, 4)
(7, 5, 0)

Move: Up
(2, 8, 3)
(1, 6, 0)
(7, 5, 4)

Move: Left
(2, 8, 3)
(1, 0, 6)
(7, 5, 4)

Move: Left
(2, 8, 3)
(0, 1, 6)
(7, 5, 4)

Move: Down
(2, 8, 3)
(7, 1, 6)
(0, 5, 4)

Move: Right
(2, 8, 3)
(7, 1, 6)
(5, 0, 4)

Move: Right
(2, 8, 3)
(7, 1, 6)
(5, 4, 0)

Move: Up
(2, 8, 3)
(7, 1, 0)
(5, 4, 6)

Move: Left
(2, 8, 3)
(7, 0, 1)
(5, 4, 6)

Move: Left
```

```
(2, 8, 3)
(0, 7, 1)
(5, 4, 6)

Move: Down
(2, 8, 3)
(5, 7, 1)
(0, 4, 6)

Move: Right
(2, 8, 3)
(5, 7, 1)
(4, 0, 6)

Move: Right
(2, 8, 3)
(5, 7, 1)
(4, 6, 0)

Move: Up
(2, 8, 3)
(5, 7, 0)
(4, 6, 1)

Move: Left
(2, 8, 3)
(5, 0, 7)
(4, 6, 1)

Move: Left
(2, 8, 3)
(0, 5, 7)
(4, 6, 1)

Move: Down
(2, 8, 3)
(4, 5, 7)
(0, 6, 1)

Move: Right
(2, 8, 3)
(4, 5, 7)
(6, 0, 1)

Move: Right
(2, 8, 3)
(4, 5, 7)
(6, 1, 0)

Move: Up
(2, 8, 3)
(4, 5, 0)
(6, 1, 7)

Move: Left
```

```
(2, 8, 3)
(4, 0, 5)
(6, 1, 7)

Move: Left
(2, 8, 3)
(0, 4, 5)
(6, 1, 7)

Move: Down
(2, 8, 3)
(6, 4, 5)
(0, 1, 7)

Move: Right
(2, 8, 3)
(6, 4, 5)
(1, 0, 7)

Move: Right
(2, 8, 3)
(6, 4, 5)
(1, 7, 0)

Move: Up
(2, 8, 3)
(6, 4, 0)
(1, 7, 5)

Move: Left
(2, 8, 3)
(6, 0, 4)
(1, 7, 5)

Move: Down
(2, 8, 3)
(6, 7, 4)
(1, 0, 5)

Move: Left
(2, 8, 3)
(6, 7, 4)
(0, 1, 5)

Move: Up
(2, 8, 3)
(0, 7, 4)
(6, 1, 5)

Move: Right
(2, 8, 3)
(7, 0, 4)
(6, 1, 5)

Move: Right
```

```
(2, 8, 3)
(7, 4, 0)
(6, 1, 5)

Move: Down
(2, 8, 3)
(7, 4, 5)
(6, 1, 0)

Move: Left
(2, 8, 3)
(7, 4, 5)
(6, 0, 1)

Move: Up
(2, 8, 3)
(7, 0, 5)
(6, 4, 1)

Move: Right
(2, 8, 3)
(7, 5, 0)
(6, 4, 1)

Move: Down
(2, 8, 3)
(7, 5, 1)
(6, 4, 0)

Move: Left
(2, 8, 3)
(7, 5, 1)
(6, 0, 4)

Move: Up
(2, 8, 3)
(7, 0, 1)
(6, 5, 4)

Move: Right
(2, 8, 3)
(7, 1, 0)
(6, 5, 4)

Move: Down
(2, 8, 3)
(7, 1, 4)
(6, 5, 0)

Move: Left
(2, 8, 3)
(7, 1, 4)
(6, 0, 5)

Move: Left
```

```
(2, 8, 3)
(7, 1, 4)
(0, 6, 5)

Move: Up
(2, 8, 3)
(0, 1, 4)
(7, 6, 5)

Move: Right
(2, 8, 3)
(1, 0, 4)
(7, 6, 5)

Move: Up
(2, 0, 3)
(1, 8, 4)
(7, 6, 5)

Move: Left
(0, 2, 3)
(1, 8, 4)
(7, 6, 5)

Move: Down
(1, 2, 3)
(0, 8, 4)
(7, 6, 5)

Move: Right
(1, 2, 3)
(8, 0, 4)
(7, 6, 5)
```

8 Puzzle using Iterative Deepening DFS

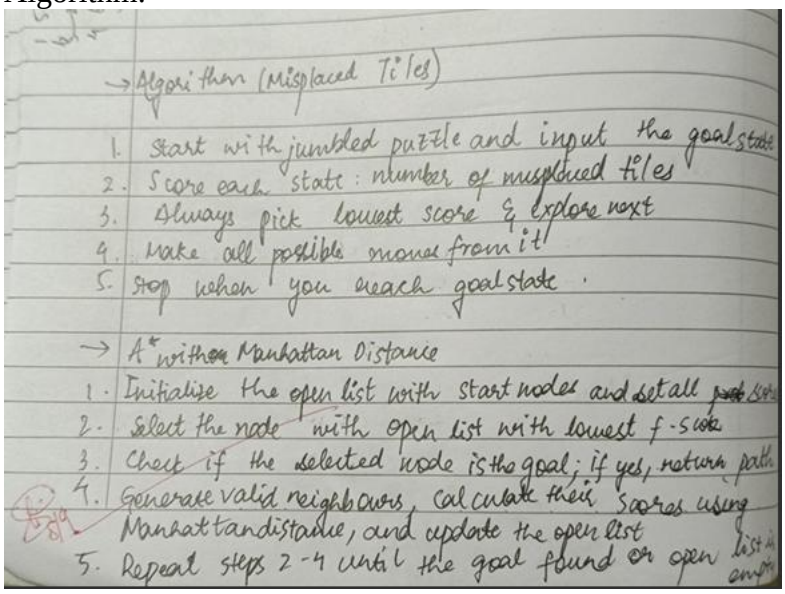
```python
from collections import deque

# Helper to print board in 3x3 format
def print_board(state):
    for i in range(0, 9, 3):
        print(state[i:i+3])
    print()

# Generate possible moves
def get_neighbors(state):
    neighbors = []
    idx = state.index(0)  # blank space
    row, col = divmod(idx, 3)
    moves = []
    if row > 0: moves.append((-1, 0, 'Up'))
    if row < 2: moves.append((1, 0, 'Down'))
```

```
    if col > 0: moves.append((0, -1, 'Left'))
    if col < 2: moves.append((0, 1, 'Right'))

    for dr, dc, action in moves:
        new_row, new_col = row + dr, col + dc
        new_idx = new_row * 3 + new_col
        new_state = list(state)
        new_state[idx], new_state[new_idx] = new_state[new_idx], new_state[idx]
        neighbors.append((tuple(new_state), action))
    return neighbors

# Depth-Limited Search (helper for IDDFS)
def dls(state, goal, depth, path, explored):
    if state == goal:
        return path
    if depth == 0:
        return None
    explored.add(state)
    for neighbor, action in get_neighbors(state):
        if neighbor not in explored:
            result = dls(neighbor, goal, depth-1, path + [(action, neighbor)],
explored)
            if result is not None:
                return result
    return None

# Iterative Deepening DFS
def iddfs(start, goal, max_depth=50):
    for depth in range(max_depth):
        explored = set()
        result = dls(start, goal, depth, [], explored)
        if result is not None:
            return result
    return None

if __name__ == "__main__":
    print("Enter the initial state (0 for blank, space-separated, 9 numbers):")
    start = tuple(map(int, input().split()))

    print("Enter the goal state (0 for blank, space-separated, 9 numbers):")
    goal = tuple(map(int, input().split()))

    print("\nSolving puzzle with Iterative Deepening DFS...\n")
    solution = iddfs(start, goal)

    if solution:
        print("Optimal solution found using IDDFS!\n")
        print("Total steps:", len(solution))
        current = start
        print("Initial State:")
        print_board(current)
        for step, state in solution:
            print("Move:", step)
            print_board(state)
    else:
        print("No solution found within depth limit.")


Enter the initial state (0 for blank, space-separated, 9 numbers):
```

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

Solving puzzle with Iterative Deepening DFS...

Optimal solution found using IDDFS!

Total steps: 5
Initial State:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

Move: Up
(2, 8, 3)
(1, 0, 4)
(7, 6, 5)

Move: Up
(2, 0, 3)
(1, 8, 4)
(7, 6, 5)

Move: Left
(0, 2, 3)
(1, 8, 4)
(7, 6, 5)

Move: Down
(1, 2, 3)
(0, 8, 4)
(7, 6, 5)

Move: Right
(1, 2, 3)
(8, 0, 4)
(7, 6, 5)
```

Program 3:
Implement A* search algorithm

Algorithm:



Code:

A* using misplaced tiles for 8 puzzle

```
import heapq

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

def is_valid(blank, move):
    if move == 'L' and blank % 3 == 0: return False
    if move == 'R' and blank % 3 == 2: return False
    if move == 'U' and blank < 3: return False
    if move == 'D' and blank > 5: return False
    return True

def neighbors(state):
    blank = state.index(0)
    result = []
    for off in moves.values():
        if is_valid(blank, [k for k,v in moves.items() if v == off][0]):
            new_state = list(state)
            swap = blank + off
            new_state[blank], new_state[swap] = new_state[swap], new_state[blank]
            result.append(tuple(new_state))
    return result

def h_misplaced(state, goal):
    return sum(1 for i in range(9) if state[i] != 0 and state[i] != goal[i])

def astar(start, goal):
    open_heap = []
    heapq.heappush(open_heap, (h_misplaced(start, goal), 0, start, [start]))
    visited = set()
    while open_heap:
```

```
        f, g, state, path = heapq.heappop(open_heap)
        if state == goal:
            return path
        if state in visited:
            continue
        visited.add(state)
        for nxt in neighbors(state):
            if nxt not in visited:
                new_g = g + 1
                new_f = new_g + h_misplaced(nxt, goal)
                heapq.heappush(open_heap, (new_f, new_g, nxt, path + [nxt]))
    return None

if __name__ == "__main__":
    init = tuple(map(int, input("Enter initial state (9 numbers, 0 for blank):
").split()))
    goal = tuple(map(int, input("Enter goal state (9 numbers, 0 for blank):
").split()))
    solution = astar(init, goal)
    if solution:
        for i, s in enumerate(solution):
            print("Cost:", i)
            print(s[0:3])
            print(s[3:6])
            print(s[6:9])
            print()
    else:
        print("No solution found.")
```

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

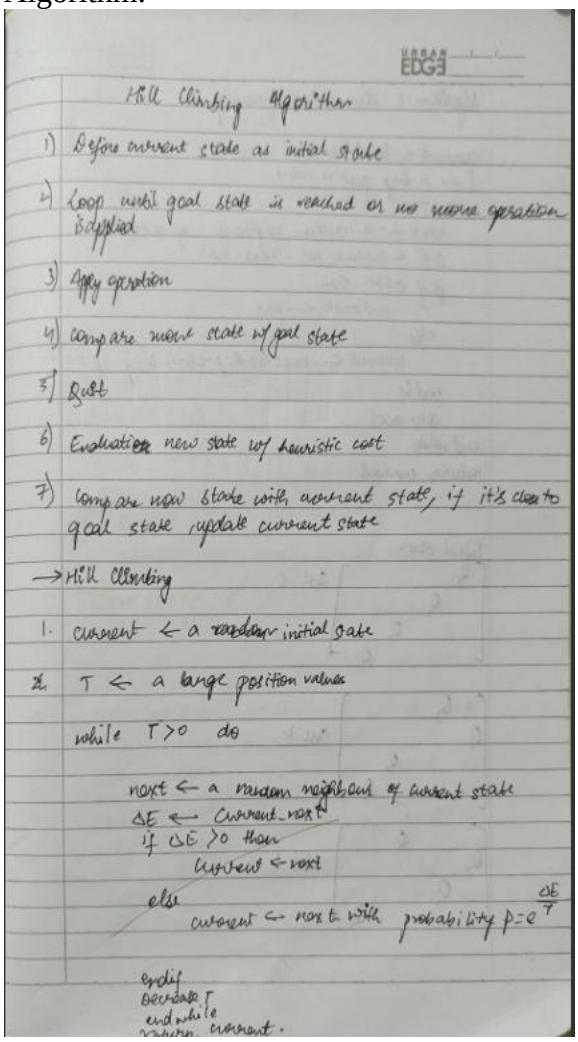
Cost: 1
(1, 2, 3)
(4, 5, 6)
(7, 0, 8)

Cost: 2
(1, 2, 3)
(4, 5, 6)
(7, 8, 0)

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:



Code:

```
import random

def calculate_attacks(state):
    """Heuristic: count number of attacking pairs."""
    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 print_board(state):
    """Print board row by row with '_' for empty and 'Q' for queen."""
    n = len(state)
    for row in range(n):
        print(' '.join('Q' if state[col] == row else '_' for col in range(n)))
    print()
```

```
def get_neighbors(state):
    """Generate all possible neighbor states."""
    neighbors = []
    n = len(state)
    for col in range(n):
        for row in range(n):
            if row != state[col]:
                neighbor = list(state)
                neighbor[col] = row
                neighbors.append(neighbor)
    return neighbors

def hill_climbing_verbose(n, initial_state=None, max_iter=1000):
    if initial_state is None:
        current = [random.randint(0, n - 1) for _ in range(n)]
    else:
        current = list(initial_state)

    current_attacks = calculate_attacks(current)
    step = 0

    print(f"\nInitial state (Step {step}): cost = {current_attacks}")
    print_board(current)

    while step < max_iter:
        neighbors = get_neighbors(current)
        if not neighbors:
            break

        best = min(neighbors, key=calculate_attacks)
        best_attacks = calculate_attacks(best)

        if best_attacks >= current_attacks:
            print("\nNo better neighbor found. Local optimum reached.")
            break

        step += 1
        current, current_attacks = best, best_attacks

        print(f"Step {step}: cost = {current_attacks}")
        print_board(current)

        if current_attacks == 0:
            print("Found a solution!\n")
            break

    print("Final state (array form):", current)
    print("Final attacking pairs (cost):", current_attacks)

if __name__ == "__main__":
    N = int(input("Enter N (number of queens): ").strip())
    s = input("Enter initial state as N space-separated integers (rows 0..N-1), or
press Enter for random initial state:\n").strip()
    initial = None
    if s:
        parts = list(map(int, s.split()))
        if len(parts) == N and all(0 <= x < N for x in parts):
            initial = parts
        else:
```

```
            print("Invalid initial state, using random.")
    hill_climbing_verbose(N, initial_state=initial)
```

Enter N (number of queens): 4
Enter initial state as N space-separated integers (rows 0..N-1), or press Enter for random initial state:

Initial state (Step 0): cost = 4
_ _ _ _
_ _ _ _
Q _ _ Q
_ Q Q _

Step 1: cost = 2
Q _ _ _
_ _ _ _
_ _ _ Q
_ Q Q _

Step 2: cost = 1
Q _ Q _
_ _ _ _
_ _ _ Q
_ Q _ _

Step 3: cost = 0
_ _ Q _
Q _ _ _
_ _ _ Q
_ Q _ _

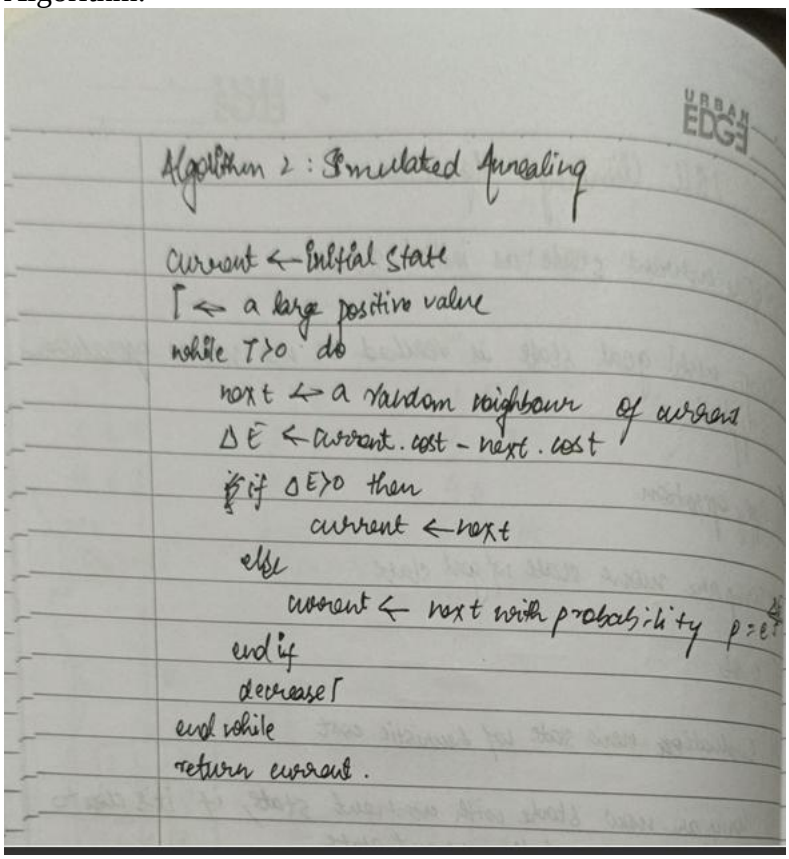
Found a solution!

Final state (array form): [1, 3, 0, 2]
Final attacking pairs (cost): 0

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:



Code:

```
import random
import math

def calculate_attacks(state):
    """Heuristic: number of attacking queen pairs."""
    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 random_neighbor(state):
    """Generate a random neighbor by moving one queen."""
    n = len(state)
    neighbor = list(state)
    col = random.randint(0, n - 1)
    row = random.randint(0, n - 1)
    while row == state[col]:
        row = random.randint(0, n - 1)
    neighbor[col] = row
    return neighbor
```

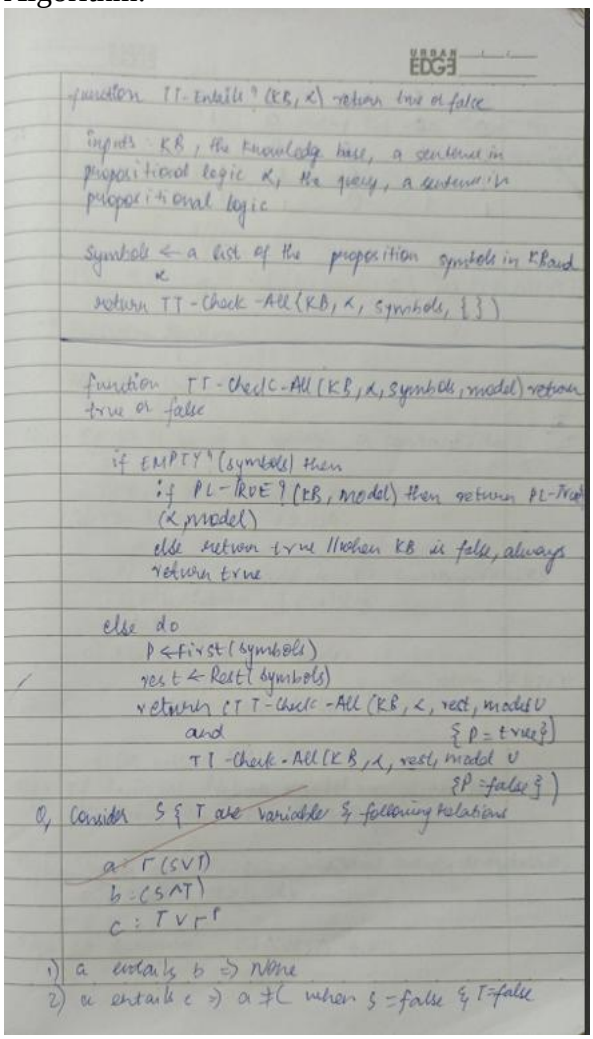
```
def simulated_annealing(n, max_steps=10000, initial_temp=100, cooling_rate=0.99):
    """Solve N-Queens using simulated annealing."""
    current = [random.randint(0, n - 1) for _ in range(n)]
    current_attacks = calculate_attacks(current)

    temperature = initial_temp

    for step in range(max_steps):
        if current_attacks == 0:
            return current, current_attacks

        neighbor = random_neighbor(current)
        neighbor_attacks = calculate_attacks(neighbor)

        delta_e = current_attacks - neighbor_attacks

        if delta_e > 0 or random.random() < math.exp(delta_e / temperature):
            current, current_attacks = neighbor, neighbor_attacks

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

    return current, current_attacks

if __name__ == "__main__":
    N = 8
    solution, attacks = simulated_annealing(N)
    print("Final State:", solution)
    print("Attacking pairs:", attacks)
    if attacks == 0:
        print("Found a solution!")
    else:
        print("Did not find solution (stuck).")
```

Final State: [6, 2, 0, 5, 7, 4, 1, 3]
Attacking pairs: 0
Found a solution!

Program 6:

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

Algorithm:



Code:

```
import itertools
import pandas as pd
import re

def replace_implications(expr):
    """
    Replace every X => Y with (not X or Y).
    This uses regex with a callback to avoid partial string overwrites.
    """
    # Pattern: capture left side and right side around =>
    # Made more flexible to handle various expressions
    pattern = r'([^=><]+?)\s*=>\s*([^=><]+?)(?=\s|$|[&|)])'
    while re.search(pattern, expr):
        expr = re.sub(pattern,
                      lambda m: f"(not {m.group(1).strip()} or
{m.group(2).strip()})",
                      expr,
                      count=1)
```

```
    return expr

def pl_true(sentence, model):
    expr = sentence.strip()
    expr = expr.replace("<=>", "==")
    expr = replace_implications(expr)

    # Replace propositional symbols with their truth values safely
    for sym, val in model.items():
        expr = re.sub(rf'\b{sym}\b', str(val), expr)

    # Clean up spacing and add proper spacing for boolean operators
    expr = re.sub(r'\s+', ' ', expr)  # Remove extra spaces
    expr = expr.replace(" and ", " and ").replace(" or ", " or ").replace(" not ", "
not ")

    return eval(expr)

def get_symbols(KB, alpha):
    symbols = set()
    for sentence in KB + [alpha]:
        # Find all alphabetic tokens (propositional variables)
        for token in re.findall(r'\b[A-Za-z]+\b', sentence):
            if token not in ['and', 'or', 'not']:  # Exclude boolean operators
                symbols.add(token)
    return sorted(list(symbols))

def tt_entails(KB, alpha):
    symbols = get_symbols(KB, alpha)
    rows = []
    entails = True

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

        try:
            kb_val = all(pl_true(sentence, model) for sentence in KB)
            alpha_val = pl_true(alpha, model)

            rows.append({**model, "KB": kb_val, "alpha": alpha_val})

            if kb_val and not alpha_val:
                entails = False
        except Exception as e:
            print(f"Error evaluating with model {model}: {e}")
            return False

    df = pd.DataFrame(rows)

    # Create a beautiful formatted table
    print("\n" + "="*50)
    print("                TRUTH TABLE")
    print("="*50)

    # Get column widths for proper alignment
    col_widths = {}
    for col in df.columns:
        col_widths[col] = max(len(str(col)), df[col].astype(str).str.len().max())
```

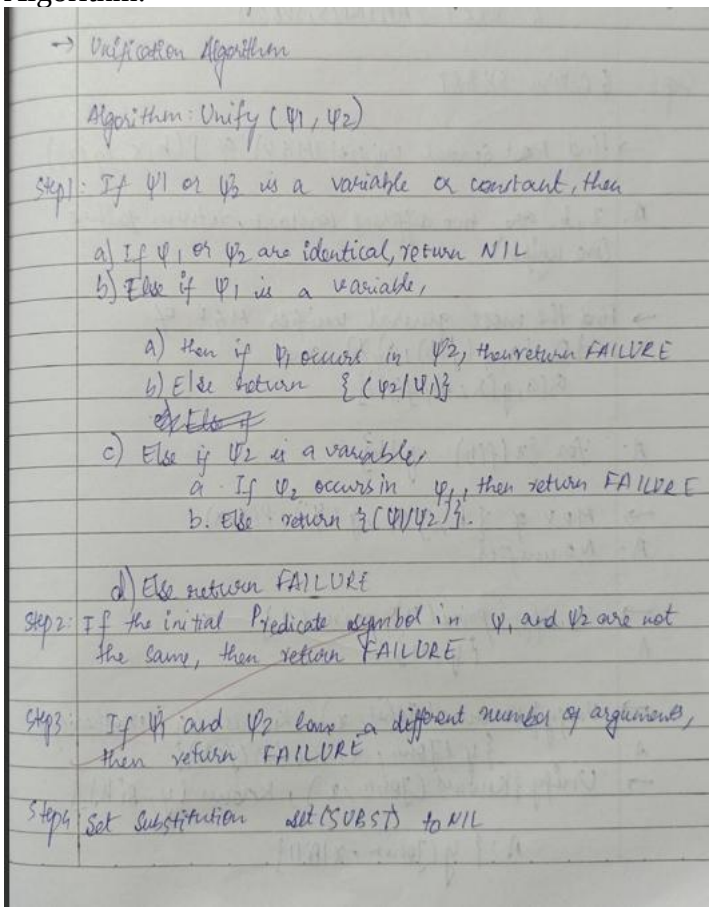
```
    # Calculate total table width
    table_width = sum(col_widths.values()) + len(df.columns) * 3 - 1

    # Print top border
    print("┌" + "─" * table_width + "┐")

    # Print header
    header = "│"
    for col in df.columns:
        header += f" {col:^{col_widths[col]}} │"
    print(header)

    # Print separator
    separator = "├"
    for col in df.columns:
        separator += "─" * (col_widths[col] + 2) + "┼"
    separator = separator[:-1] + "┤"
    print(separator)

    # Print rows
    for _, row in df.iterrows():
        row_str = "│"
        for col in df.columns:
            value = str(row[col])
            row_str += f" {value:^{col_widths[col]}} │"
        print(row_str)

    # Print bottom border
    print("└" + "─" * table_width + "┘")

    # Print result with styling
    print("\n" + "="*50)
    result_text = f"KB ENTAILS ALPHA: {'✓ YES' if entails else '✗ NO'}"
    print(f"{result_text:^50}")
    print("="*50)
    return entails

# --- Interactive input ---
print("Enter Knowledge Base (KB) sentences, separated by commas.")
print("Use symbols like A, B, C and operators: and, or, not, =>, <=>")
kb_input = input("KB: ").strip()
KB = [x.strip() for x in kb_input.split(",")]
alpha = input("Enter query (alpha): ").strip()
result = tt_entails(KB, alpha)
print(f"Result: {result}")
#
Enter Knowledge Base (KB) sentences, separated by commas.
Use symbols like A, B, C and operators: and, or, not, =>, <=>
KB: A => C, B
Enter query (alpha): B or C

==================================================
                TRUTH TABLE
==================================================
┌──────────────────────────────────────────┐
│   A    │   B    │   C    │   KB   │ alpha │
├────────┼────────┼────────┼────────┼───────┤
│ True   │ True   │ True   │ True   │ True  │
```

```
| True  | True  | False | False | True  |
| True  | False | True  | False | True  |
| True  | False | False | False | False |
| False | True  | True  | True  | True  |
| False | True  | False | True  | True  |
| False | False | True  | False | True  |
| False | False | False | False | False |
└─────────────────────────────────────────┘

==================================================
             KB ENTAILS ALPHA: ✓ YES
==================================================
Result: True
```

Program 7

Implement unification in first order logic

Algorithm:



→ Unification Algorithm

Algorithm: Unify (ψ_1, ψ_2)

Step1: If ψ_1 or ψ_2 is a variable or constant, then

a) If ψ_1 or ψ_2 are identical, return NIL
b) Else if ψ_1 is a variable,

a) then if ψ_1 occurs in ψ_2, then return FAILURE
b) Else return $\{(\psi_2/\psi_1)\}$

~~c) Else if~~

c) Else if ψ_2 is a variable,

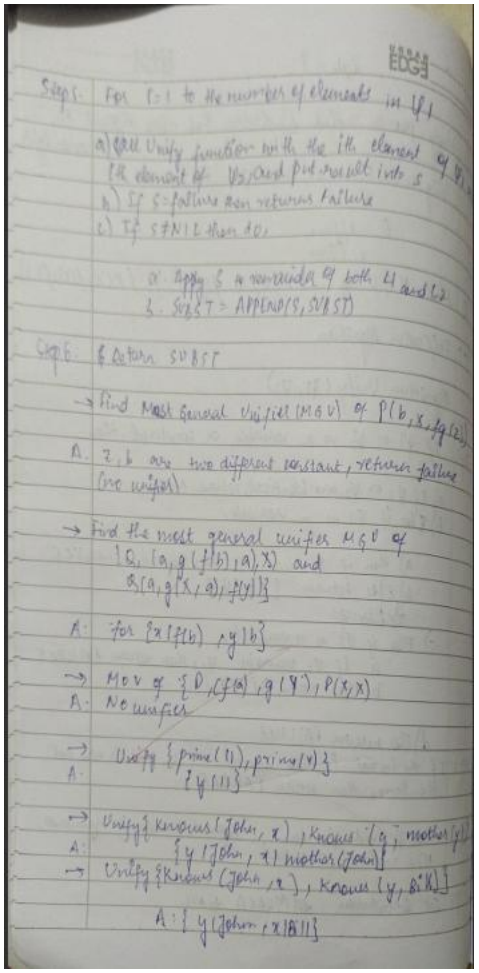
a. If ψ_2 occurs in ψ_1, then return FAILURE
b. Else return $\{(\psi_1/\psi_2)\}$.

d) Else return FAILURE

Step2: If the initial Predicate symbol in ψ_1 and ψ_2 are not the same, then return FAILURE

Step3: If ψ_1 and ψ_2 have a different number of arguments, then return FAILURE

Step4: Set substitution set (SUBST) to NIL



Code:

```
class Term:
    """Base class for terms in first-order logic"""
    pass

class Constant(Term):
    """Represents a constant"""
    def __init__(self, name):
        self.name = name

    def __eq__(self, other):
        return isinstance(other, Constant) and self.name == other.name

    def __repr__(self):
        return self.name

    def __hash__(self):
        return hash(('Constant', self.name))

class Variable(Term):
    """Represents a variable"""
    def __init__(self, name):
        self.name = name

    def __eq__(self, other):
```

```
        return isinstance(other, Variable) and self.name == other.name

    def __repr__(self):
        return self.name

    def __hash__(self):
        return hash(('Variable', self.name))

class Predicate(Term):
    """Represents a predicate with arguments"""
    def __init__(self, name, args):
        self.name = name
        self.args = args if isinstance(args, list) else [args]

    def __eq__(self, other):
        return (isinstance(other, Predicate) and
                self.name == other.name and
                len(self.args) == len(other.args) and
                all(a == b for a, b in zip(self.args, other.args)))

    def __repr__(self):
        return f"{self.name}({', '.join(str(arg) for arg in self.args)})"

def occurs_check(var, term, subst):
    """Check if variable occurs in term (prevents infinite structures)"""
    if var == term:
        return True
    elif isinstance(term, Variable) and term in subst:
        return occurs_check(var, subst[term], subst)
    elif isinstance(term, Predicate):
        return any(occurs_check(var, arg, subst) for arg in term.args)
    return False

def apply_substitution(term, subst):
    """Apply substitution to a term"""
    if isinstance(term, Variable):
        if term in subst:
            return apply_substitution(subst[term], subst)
        return term
    elif isinstance(term, Predicate):
        new_args = [apply_substitution(arg, subst) for arg in term.args]
        return Predicate(term.name, new_args)
    else:
        return term

def unify(term1, term2, subst=None):
    """
    Unification Algorithm
    Returns substitution set if unification succeeds, None if it fails
    """
    if subst is None:
        subst = {}

    # Apply existing substitutions
    term1 = apply_substitution(term1, subst)
```

```
    term2 = apply_substitution(term2, subst)

    # Step 1: If term1 or term2 is a variable or constant
    # Step 1a: If both are identical
    if term1 == term2:
        return subst

    # Step 1b: If term1 is a variable
    elif isinstance(term1, Variable):
        if occurs_check(term1, term2, subst):
            return None  # FAILURE
        else:
            new_subst = subst.copy()
            new_subst[term1] = term2
            return new_subst

    # Step 1c: If term2 is a variable
    elif isinstance(term2, Variable):
        if occurs_check(term2, term1, subst):
            return None  # FAILURE
        else:
            new_subst = subst.copy()
            new_subst[term2] = term1
            return new_subst

    # Step 1d: Both are constants but not equal
    elif isinstance(term1, Constant) or isinstance(term2, Constant):
        return None  # FAILURE

    # Step 2: Check if both are predicates with same name
    elif isinstance(term1, Predicate) and isinstance(term2, Predicate):
        if term1.name != term2.name:
            return None  # FAILURE

        # Step 3: Check if they have same number of arguments
        if len(term1.args) != len(term2.args):
            return None  # FAILURE

        # Step 4 & 5: Unify arguments recursively
        current_subst = subst.copy()
        for arg1, arg2 in zip(term1.args, term2.args):
            current_subst = unify(arg1, arg2, current_subst)
            if current_subst is None:  # If unification fails
                return None

        return current_subst

    else:
        return None  # FAILURE

def print_substitution(subst):
    """Pretty print substitution set"""
    if subst is None:
        print("FAILURE: Unification failed")
    elif not subst:
```

```
        print("NIL: Terms are already unified")
    else:
        print("Substitution:")
        for var, term in subst.items():
            print(f"  {var} -> {term}")

def parse_term(term_str):
    """Parse a string representation of a term into Term objects"""
    term_str = term_str.strip()

    # Check if it's a predicate (contains parentheses)
    if '(' in term_str:
        paren_idx = term_str.index('(')
        pred_name = term_str[:paren_idx].strip()

        # Extract arguments between parentheses
        args_str = term_str[paren_idx+1:term_str.rindex(')')].strip()

        # Split arguments by comma (handle nested predicates)
        args = []
        depth = 0
        current_arg = ""
        for char in args_str:
            if char == ',' and depth == 0:
                args.append(parse_term(current_arg))
                current_arg = ""
            else:
                if char == '(':
                    depth += 1
                elif char == ')':
                    depth -= 1
                current_arg += char

        if current_arg.strip():
            args.append(parse_term(current_arg))

        return Predicate(pred_name, args)

    # Check if it's a variable (lowercase first letter or starts with ?)
    elif term_str[0].islower() or term_str[0] == '?':
        return Variable(term_str)

    # Otherwise it's a constant (uppercase first letter)
    else:
        return Constant(term_str)

def run_interactive():
    """Interactive mode for user input"""
    print("=== Unification Algorithm (Interactive Mode) ===")
    print("Enter terms to unify. Use:")
    print("  - Variables: lowercase letters (x, y, z) or ?x, ?y")
    print("  - Constants: uppercase letters (John, Mary, A)")
    print("  - Predicates: Name(arg1, arg2, ...) e.g., P(x, y)")
    print("  - Type 'quit' to exit\n")
```

```
    while True:
        print("-" * 50)
        term1_str = input("Enter first term: ").strip()

        if term1_str.lower() == 'quit':
            print("Exiting...")
            break

        term2_str = input("Enter second term: ").strip()

        if term2_str.lower() == 'quit':
            print("Exiting...")
            break

        try:
            term1 = parse_term(term1_str)
            term2 = parse_term(term2_str)

            print(f"\nUnifying: {term1} and {term2}")
            result = unify(term1, term2)
            print_substitution(result)
            print()

        except Exception as e:
            print(f"Error parsing terms: {e}")
            print("Please check your input format.\n")

def run_examples():
    """Run predefined examples"""
    print("=== Unification Algorithm Examples ===\n")

    # Example 1: Unifying variables
    print("Example 1: Unify(x, y)")
    x = Variable('x')
    y = Variable('y')
    result = unify(x, y)
    print_substitution(result)
    print()

    # Example 2: Unifying variable with constant
    print("Example 2: Unify(x, John)")
    x = Variable('x')
    john = Constant('John')
    result = unify(x, john)
    print_substitution(result)
    print()

    # Example 3: Unifying predicates
    print("Example 3: Unify(P(x, y), P(John, z))")
    p1 = Predicate('P', [Variable('x'), Variable('y')])
    p2 = Predicate('P', [Constant('John'), Variable('z')])
    result = unify(p1, p2)
    print_substitution(result)
    print()
```

```
    # Example 4: Unifying complex predicates
    print("Example 4: Unify(P(x, f(y)), P(a, f(b)))")
    p1 = Predicate('P', [Variable('x'), Predicate('f', [Variable('y')])])
    p2 = Predicate('P', [Constant('a'), Predicate('f', [Constant('b')])])
    result = unify(p1, p2)
    print_substitution(result)
    print()

    # Example 5: Failure case - occurs check
    print("Example 5: Unify(x, f(x)) - Occurs Check")
    x = Variable('x')
    fx = Predicate('f', [x])
    result = unify(x, fx)
    print_substitution(result)
    print()

    # Example 6: Failure case - different predicates
    print("Example 6: Unify(P(x), Q(x)) - Different Predicates")
    p1 = Predicate('P', [Variable('x')])
    p2 = Predicate('Q', [Variable('x')])
    result = unify(p1, p2)
    print_substitution(result)
    print()

    # Example 7: Failure case - different constants
    print("Example 7: Unify(John, Mary) - Different Constants")
    john = Constant('John')
    mary = Constant('Mary')
    result = unify(john, mary)
    print_substitution(result)

# Main program
if __name__ == "__main__":
    print("Choose mode:")
    print("1. Run predefined examples")
    print("2. Interactive mode (enter your own terms)")

    choice = input("\nEnter choice (1 or 2): ").strip()
    print()

    if choice == '1':
        run_examples()
    elif choice == '2':
        run_interactive()
    else:
        print("Invalid choice. Running examples by default...\n")
        run_examples()
```

Choose mode:
1. Run predefined examples
2. Interactive mode (enter your own terms)

Enter choice (1 or 2): 2

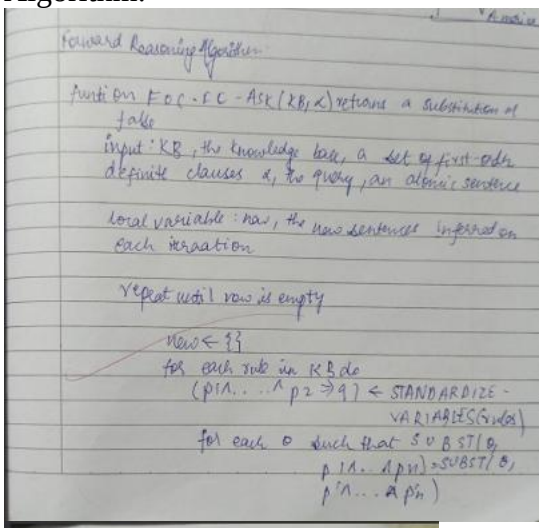
```
=== Unification Algorithm (Interactive Mode) ===
Enter terms to unify. Use:
  - Variables: lowercase letters (x, y, z) or ?x, ?y
  - Constants: uppercase letters (John, Mary, A)
  - Predicates: Name(arg1, arg2, ...) e.g., P(x, y)
  - Type 'quit' to exit

--------------------------------------------------
Enter first term: p(B,x,f(g(z)))
Enter second term: p(z,f(y),f(y))

Unifying: p(B, x, f(g(z))) and p(z, f(y), f(y))
Substitution:
  z -> B
  x -> f(y)
  y -> g(B)

--------------------------------------------------
Enter first term: Q(A,g(x,A),f(y))
Enter second term: Q(A,g(f(B),A),x)

Unifying: Q(A, g(x, A), f(y)) and Q(A, g(f(B), A), x)
Substitution:
  x -> f(B)
  y -> B

--------------------------------------------------
Enter first term: p(f(A),g(y))
Enter second term: p(x,x)

Unifying: p(f(A), g(y)) and p(x, x)
FAILURE: Unification failed

--------------------------------------------------
Enter first term: prime(N11)
Enter second term: prime(y)

Unifying: prime(N11) and prime(y)
Substitution:
  y -> N11

--------------------------------------------------
Enter first term: knows(John,x)
Enter second term: knows(y,mother(y))

Unifying: knows(John, x) and knows(y, mother(y))
Substitution:
  y -> John
```

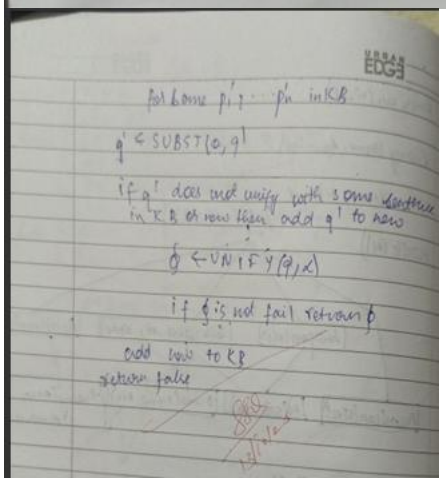
x -> mother(John)

Program 8

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

Algorithm:





Code:

```
class Term:
    """Base class for terms in first-order logic"""
    pass

class Constant(Term):
    """Represents a constant"""
    def __init__(self, name):
        self.name = name

    def __eq__(self, other):
        return isinstance(other, Constant) and self.name == other.name

    def __repr__(self):
        return self.name

    def __hash__(self):
        return hash(('Constant', self.name))
```

```
class Variable(Term):
    """Represents a variable"""
    def __init__(self, name):
        self.name = name

    def __eq__(self, other):
        return isinstance(other, Variable) and self.name == other.name

    def __repr__(self):
        return self.name

    def __hash__(self):
        return hash(('Variable', self.name))

class Predicate(Term):
    """Represents a predicate with arguments"""
    def __init__(self, name, args):
        self.name = name
        self.args = args if isinstance(args, list) else [args]

    def __eq__(self, other):
        return (isinstance(other, Predicate) and
                self.name == other.name and
                len(self.args) == len(other.args) and
                all(a == b for a, b in zip(self.args, other.args)))

    def __repr__(self):
        return f"{self.name}({', '.join(str(arg) for arg in self.args)})"

    def __hash__(self):
        return hash((self.name, tuple(self.args)))

class Rule:
    """Represents an implication rule: premises => conclusion"""
    def __init__(self, premises, conclusion):
        self.premises = premises if isinstance(premises, list) else [premises]
        self.conclusion = conclusion

    def __repr__(self):
        premises_str = ' ∧ '.join(str(p) for p in self.premises)
        return f"{premises_str} => {self.conclusion}"

# Variable counter for standardization
_var_counter = 0

def get_new_variable():
    """Generate a new unique variable"""
    global _var_counter
    _var_counter += 1
    return Variable(f"v{_var_counter}")

def standardize_variables(rule):
    """Replace all variables in rule with new unique variables"""
    var_mapping = {}
```

```
    def replace_vars(term):
        if isinstance(term, Variable):
            if term not in var_mapping:
                var_mapping[term] = get_new_variable()
            return var_mapping[term]
        elif isinstance(term, Predicate):
            new_args = [replace_vars(arg) for arg in term.args]
            return Predicate(term.name, new_args)
        else:
            return term

    new_premises = [replace_vars(p) for p in rule.premises]
    new_conclusion = replace_vars(rule.conclusion)
    return Rule(new_premises, new_conclusion)

def occurs_check(var, term, subst):
    """Check if variable occurs in term"""
    if var == term:
        return True
    elif isinstance(term, Variable) and term in subst:
        return occurs_check(var, subst[term], subst)
    elif isinstance(term, Predicate):
        return any(occurs_check(var, arg, subst) for arg in term.args)
    return False

def apply_substitution(term, subst):
    """Apply substitution to a term"""
    if isinstance(term, Variable):
        if term in subst:
            return apply_substitution(subst[term], subst)
        return term
    elif isinstance(term, Predicate):
        new_args = [apply_substitution(arg, subst) for arg in term.args]
        return Predicate(term.name, new_args)
    else:
        return term

def unify(term1, term2, subst=None):
    """Unification algorithm"""
    if subst is None:
        subst = {}

    term1 = apply_substitution(term1, subst)
    term2 = apply_substitution(term2, subst)

    if term1 == term2:
        return subst
    elif isinstance(term1, Variable):
        if occurs_check(term1, term2, subst):
            return None
        else:
            new_subst = subst.copy()
            new_subst[term1] = term2
            return new_subst
```

```
    elif isinstance(term2, Variable):
        if occurs_check(term2, term1, subst):
            return None
        else:
            new_subst = subst.copy()
            new_subst[term2] = term1
            return new_subst
    elif isinstance(term1, Constant) or isinstance(term2, Constant):
        return None
    elif isinstance(term1, Predicate) and isinstance(term2, Predicate):
        if term1.name != term2.name or len(term1.args) != len(term2.args):
            return None

        current_subst = subst.copy()
        for arg1, arg2 in zip(term1.args, term2.args):
            current_subst = unify(arg1, arg2, current_subst)
            if current_subst is None:
                return None
        return current_subst
    else:
        return None

def unify_all(premises, kb_facts, subst=None):
    """Try to unify all premises with facts in KB"""
    if subst is None:
        subst = {}

    if not premises:
        return [subst]

    first_premise = premises[0]
    remaining_premises = premises[1:]

    all_substitutions = []

    for fact in kb_facts:
        theta = unify(first_premise, fact, subst.copy())
        if theta is not None:
            # Apply substitution to remaining premises
            substituted_remaining = [apply_substitution(p, theta) for p in remaining
_premises]
            # Recursively unify remaining premises
            result_substs = unify_all(substituted_remaining, kb_facts, theta)
            all_substitutions.extend(result_substs)

    return all_substitutions

def fol_fc_ask(kb_facts, kb_rules, query, max_iterations=100):
    """
    Forward Chaining Algorithm for First-Order Logic

    Args:
        kb_facts: List of atomic sentences (facts) in KB
        kb_rules: List of implication rules in KB
        query: The query to prove (atomic sentence)
```

```
    max_iterations: Maximum number of iterations to prevent infinite loops

Returns:
    Substitution if query can be proved, None otherwise
"""
print("=== Forward Chaining Algorithm ===\n")
print(f"Query: {query}\n")
print("Initial KB Facts:")
for fact in kb_facts:
    print(f"  {fact}")
print("\nKB Rules:")
for rule in kb_rules:
    print(f"  {rule}")
print("\n" + "="*50 + "\n")

iteration = 0

while iteration < max_iterations:
    iteration += 1
    new = []

    print(f"Iteration {iteration}:")

    # For each rule in KB
    for rule in kb_rules:
        # Standardize variables in the rule
        std_rule = standardize_variables(rule)

        # Try to find substitutions that satisfy all premises
        substitutions = unify_all(std_rule.premises, kb_facts)

        # For each valid substitution
        for theta in substitutions:
            # Apply substitution to conclusion
            inferred = apply_substitution(std_rule.conclusion, theta)

            # Check if this fact is new
            if inferred not in kb_facts and inferred not in new:
                new.append(inferred)
                print(f"  Inferred: {inferred}")
                print(f"    From rule: {std_rule}")
                print(f"    With substitution: {theta}")

                # Check if inferred fact unifies with query
                result = unify(inferred, query)
                if result is not None:
                    print(f"\n*** Query proved! ***")
                    print(f"Substitution: {result}")
                    return result

    # If no new facts inferred, we're done
    if not new:
        print("  No new facts inferred.")
        print("\nForward chaining completed. Query cannot be proved.")
        return None
```

```
        # Add new facts to KB
        kb_facts.extend(new)
        print()

    print(f"Maximum iterations ({max_iterations}) reached.")
    return None

def parse_term(term_str):
    """Parse a string into a Term object"""
    term_str = term_str.strip()

    if '(' in term_str:
        paren_idx = term_str.index('(')
        pred_name = term_str[:paren_idx].strip()
        args_str = term_str[paren_idx+1:term_str.rindex(')')].strip()

        args = []
        depth = 0
        current_arg = ""
        for char in args_str:
            if char == ',' and depth == 0:
                args.append(parse_term(current_arg))
                current_arg = ""
            else:
                if char == '(':
                    depth += 1
                elif char == ')':
                    depth -= 1
                current_arg += char

        if current_arg.strip():
            args.append(parse_term(current_arg))

        return Predicate(pred_name, args)
    elif term_str[0].islower():
        return Variable(term_str)
    else:
        return Constant(term_str)

def parse_rule(rule_str):
    """Parse a rule string like 'P(x) ∧ Q(x) => R(x)'"""
    if '=>' in rule_str:
        parts = rule_str.split('=>')
        conclusion_str = parts[1].strip()
        premises_str = parts[0].strip()

        # Split premises by ∧ or AND
        premise_parts = [p.strip() for p in premises_str.replace('AND', '∧').split('
∧')]

        premises = [parse_term(p) for p in premise_parts]
        conclusion = parse_term(conclusion_str)

        return Rule(premises, conclusion)
```

```
    else:
        # It's just a fact
        return parse_term(rule_str)

# Example usage
if __name__ == "__main__":
    print("Choose mode:")
    print("1. Run example (Animal reasoning)")
    print("2. Interactive mode")

    choice = input("\nEnter choice (1 or 2): ").strip()
    print()

    if choice == '1':
        # Example: Animal reasoning
        # Facts
        kb_facts = [
            Predicate('Animal', [Constant('Dog')]),
            Predicate('Animal', [Constant('Cat')]),
            Predicate('Loves', [Constant('John'), Constant('Dog')]),
            Predicate('Owns', [Constant('John'), Constant('Dog')])
        ]

        # Rules
        kb_rules = [
            # Animal(x) ∧ Loves(y, x) => Loves(x, y)
            Rule([Predicate('Animal', [Variable('x')]),
                  Predicate('Loves', [Variable('y'), Variable('x')])],
                 Predicate('Loves', [Variable('x'), Variable('y')])),

            # Owns(x, y) ∧ Animal(y) => KeepsAsPet(x, y)
            Rule([Predicate('Owns', [Variable('x'), Variable('y')]),
                  Predicate('Animal', [Variable('y')])],
                 Predicate('KeepsAsPet', [Variable('x'), Variable('y')]))
        ]

        # Query: Does Dog love John?
        query = Predicate('Loves', [Constant('Dog'), Constant('John')])

        result = fol_fc_ask(kb_facts, kb_rules, query)

    elif choice == '2':
        print("=== Interactive Forward Chaining ===")
        print("Enter facts and rules for the knowledge base.\n")

        kb_facts = []
        kb_rules = []

        # Input facts
        print("Enter facts (one per line, empty line to finish):")
        print("Example: Animal(Dog), Loves(John, Dog)")
        while True:
            fact_str = input("Fact: ").strip()
            if not fact_str:
                break
```

```
            try:
                fact = parse_term(fact_str)
                kb_facts.append(fact)
            except Exception as e:
                print(f"Error parsing fact: {e}")

        # Input rules
        print("\nEnter rules (one per line, empty line to finish):")
        print("Example: Animal(x) ∧ Loves(y,x) => Loves(x,y)")
        print("You can also use 'AND' instead of ∧")
        while True:
            rule_str = input("Rule: ").strip()
            if not rule_str:
                break
            try:
                rule = parse_rule(rule_str)
                kb_rules.append(rule)
            except Exception as e:
                print(f"Error parsing rule: {e}")

        # Input query
        print("\nEnter query:")
        query_str = input("Query: ").strip()
        try:
            query = parse_term(query_str)
            result = fol_fc_ask(kb_facts, kb_rules, query)
        except Exception as e:
            print(f"Error parsing query: {e}")

    else:
        print("Invalid choice.")
```

Choose mode:
1. Run example (Animal reasoning)
2. Interactive mode

Enter choice (1 or 2): 1

=== Forward Chaining Algorithm ===

Query: Loves(Dog, John)

Initial KB Facts:
Animal(Dog)
Animal(Cat)
Loves(John, Dog)
Owns(John, Dog)

KB Rules:
Animal(x) ∧ Loves(y, x) => Loves(x, y)
Owns(x, y) ∧ Animal(y) => KeepsAsPet(x, y)

==========

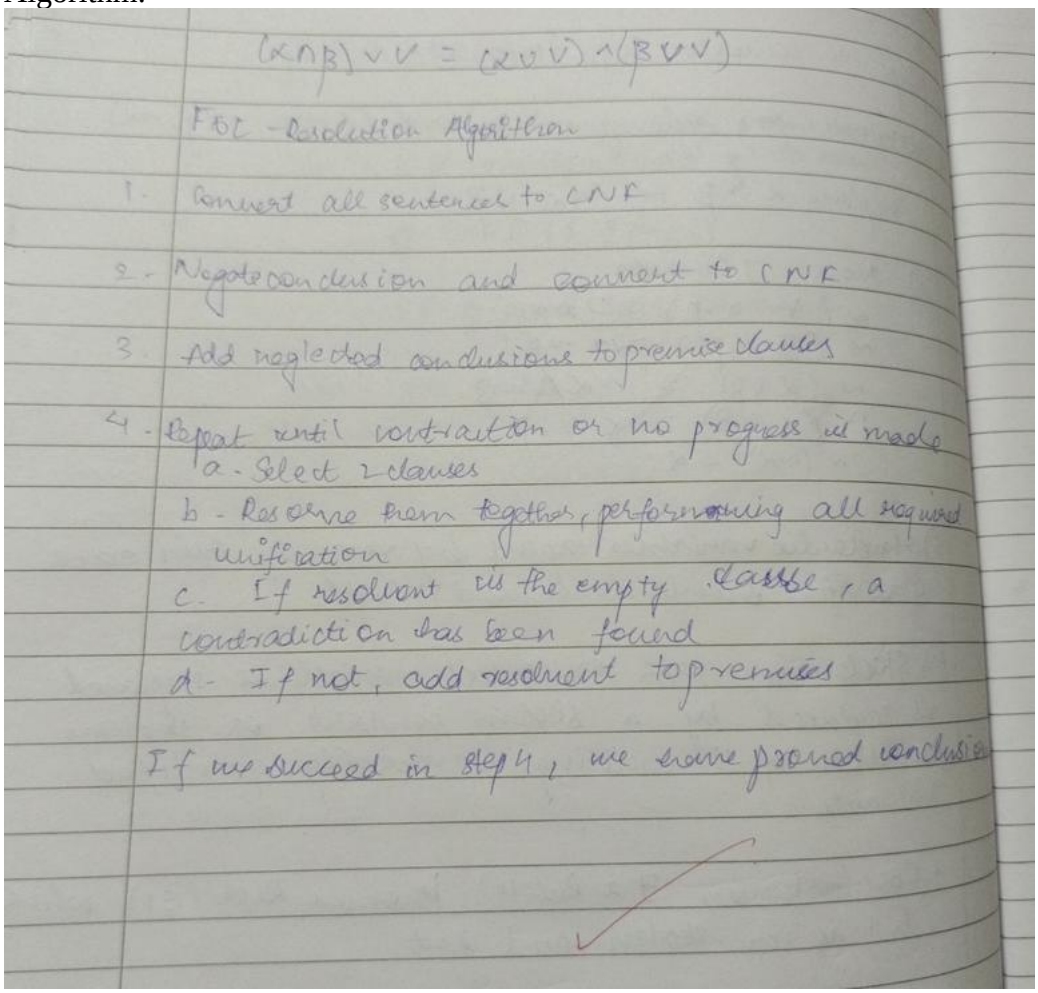
Iteration 1:
Inferred: Loves(Dog, John)
From rule: Animal(v1) ∧ Loves(v2, v1) => Loves(v1, v2)
With substitution: {v1: Dog, v2: John}

***** Query proved! *****
Substitution: {}

Program 9

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

Algorithm:



Code:

```
from itertools import combinations

def get_clauses():
    n = int(input("Enter number of clauses in Knowledge Base: "))
    clauses = []
    for i in range(n):
        clause = input(f"Enter clause {i+1}: ")
        clause_set = set(clause.replace(" ", "").split("v"))
        clauses.append(clause_set)
    return clauses

def resolve(ci, cj):
    resolvents = []
    for di in ci:
        for dj in cj:
            if di == ('~' + dj) or dj == ('~' + di):
                new_clause = (ci - {di}) | (cj - {dj})
```

```
                resolvents.append(new_clause)
    return resolvents

def resolution_algorithm(kb, query):
    kb.append(set(['~' + query]))
    derived = []
    clause_id = {frozenset(c): f"C{i+1}" for i, c in enumerate(kb)}

    step = 1
    while True:
        new = []
        for (ci, cj) in combinations(kb, 2):
            resolvents = resolve(ci, cj)
            for res in resolvents:
                if res not in kb and res not in new:
                    cid_i, cid_j = clause_id[frozenset(ci)], clause_id[frozenset(cj)
]
                    clause_name = f"R{step}"
                    derived.append((clause_name, res, cid_i, cid_j))
                    clause_id[frozenset(res)] = clause_name
                    new.append(res)
                    print(f"[Step {step}] {clause_name} = Resolve({cid_i}, {cid_j})
→ {res or '{}'}")
                    step += 1

                    # If empty clause found → proof complete
                    if res == set():
                        print("\n✅ Query is proved by resolution (empty clause fou
nd).")
                        print("\n--- Proof Tree ---")
                        print_tree(derived, clause_name)
                        return True
        if not new:
            print("\n❌ Query cannot be proved by resolution.")
            return False
        kb.extend(new)

def print_tree(derived, goal):
    tree = {name: (parents, clause) for name, clause, *parents in [(r[0], r[1], r[2:
][0], r[2:][1]) for r in derived]}

    def show(node, indent=0):
        if node not in tree:
            print(" " * indent + node)
            return
        parents, clause = tree[node]
        print(" " * indent + f"{node}: {set(clause) or '{}'}")
        for p in parents:
            show(p, indent + 4)

    show(goal)

# --- MAIN PROGRAM ---
print("=== FOL Resolution Demo with Proof Tree ===")
```

```
kb = get_clauses()
query = input("Enter query to prove: ")
resolution_algorithm(kb, query)
```

=== FOL Resolution Demo with Proof Tree ===
Enter number of clauses in Knowledge Base: 3
Enter clause 1: A v B
Enter clause 2: ~B v C
Enter clause 3: ~C
Enter query to prove: A
[Step 1] R1 = Resolve(C1, C2) → {'A', 'C'}
[Step 2] R2 = Resolve(C1, C4) → {'B'}
[Step 3] R3 = Resolve(C2, C3) → {'~B'}
[Step 4] R4 = Resolve(C1, R3) → {'A'}
[Step 5] R5 = Resolve(C2, R2) → {'C'}
[Step 6] R6 = Resolve(R2, R3) → {}

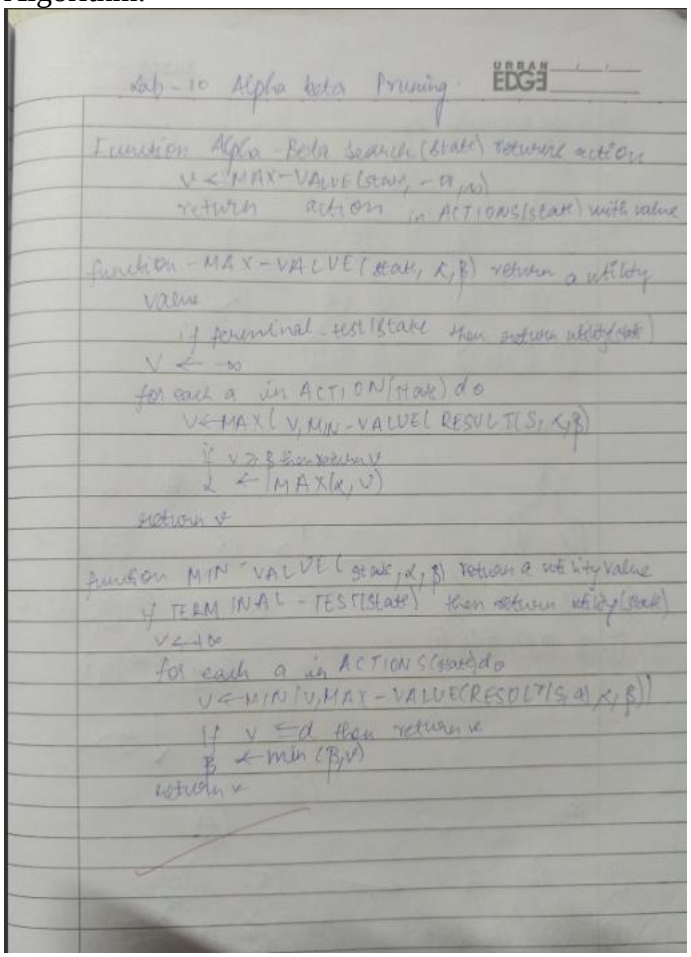
☑ Query is proved by resolution (empty clause found).

--- Proof Tree ---
R6: {}
R2: {'B'}
C1
C4
R3: {'~B'}
C2
C3
True

Program 10

Implement Alpha-Beta Pruning.

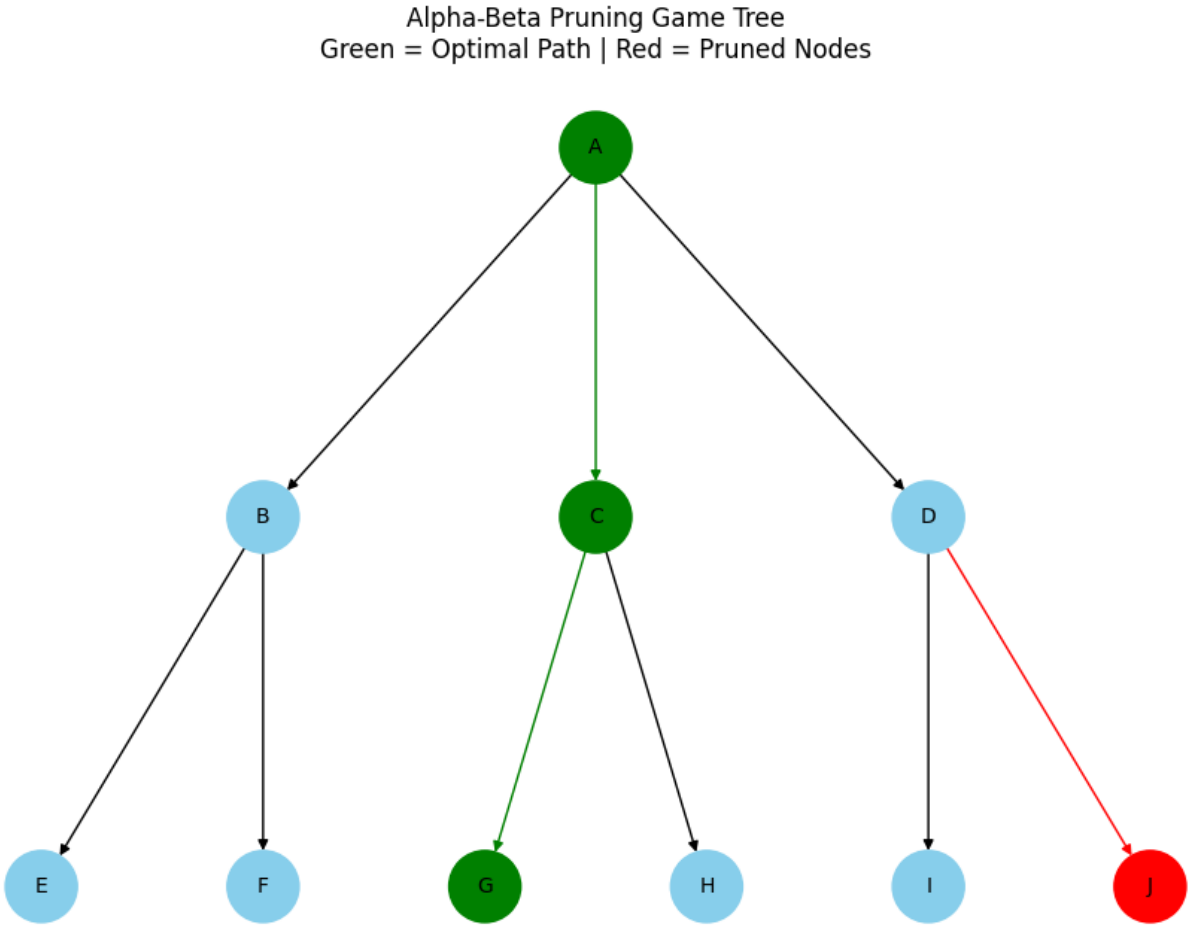
Algorithm:



Code:

```
class Node:
    def __init__(self, name):
        self.name = name
        self.children = []
        self.value = None
        self.pruned = False

def alpha_beta(node, depth, maximizing, values, alpha, beta, index):
    # Terminal node
    if depth == 3:
        node.value = values[index[0]]
        index[0] += 1
        return node.value

    if maximizing:
        best = float('-inf')
        for i in range(2):  # 2 children
            child = Node(f"{node.name}{i}")
            node.children.append(child)
            val = alpha_beta(child, depth + 1, False, values, alpha, beta, index)
```

```
            best = max(best, val)
            alpha = max(alpha, best)
            if beta <= alpha:
                node.pruned = True
                break
        node.value = best
        return best
    else:
        best = float('inf')
        for i in range(2):
            child = Node(f"{node.name}{i}")
            node.children.append(child)
            val = alpha_beta(child, depth + 1, True, values, alpha, beta, index)
            best = min(best, val)
            beta = min(beta, best)
            if beta <= alpha:
                node.pruned = True
                break
        node.value = best
        return best

def print_tree(node, indent=0):
    prune_mark = " [PRUNED]" if node.pruned else ""
    val = f" = {node.value}" if node.value is not None else ""
    print(" " * indent + f"{node.name}{val}{prune_mark}")
    for child in node.children:
        print_tree(child, indent + 4)

# --- main ---
print("=== Alpha-Beta Pruning with Tree ===")
values = list(map(int, input("Enter 8 leaf node values separated by spaces:
").split()))

root = Node("R")
alpha_beta(root, 0, True, values, float('-inf'), float('inf'), [0])

print("\n--- Game Tree ---")
print_tree(root)

print("\nOptimal Value at Root:", root.value)


Enter number of non-leaf nodes: 4

Enter parent node: A
Enter children of A (space separated): B C D

Enter parent node: B
Enter children of B (space separated): E F

Enter parent node: C
Enter children of C (space separated): G H

Enter parent node: D
Enter children of D (space separated): I J

Enter number of leaf nodes: 6
Enter leaf node and its value (e.g. E 3): E 3
```




```
Enter leaf node and its value (e.g. E 3): F 5
Enter leaf node and its value (e.g. E 3): G 6
Enter leaf node and its value (e.g. E 3): H 9
Enter leaf node and its value (e.g. E 3): I 1
Enter leaf node and its value (e.g. E 3): J 2

Enter root node: A
Enter total depth of tree: 3

--------------------------------
Final Optimal Value: 6
Pruned Nodes: ['J']
--------------------------------
```