VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Shreya Sathynarayana(1BM23CS318)

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

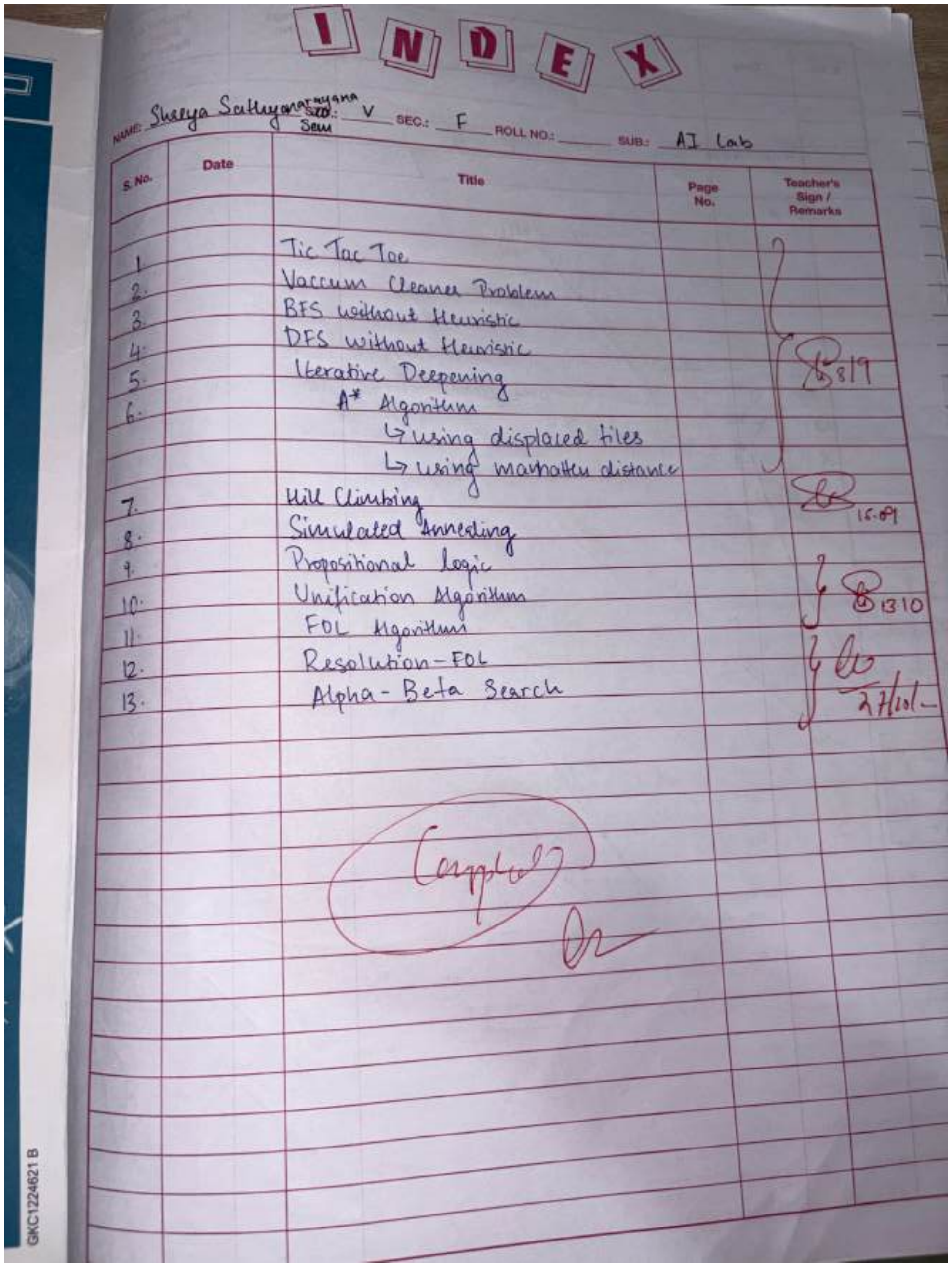
B.M.S. College of Engineering,
Bull Temple Road, Bangalore 560019
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by **Shreya Sathyanarayana(1BM23CS318),** 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.

Surabhi S Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index



INDEX

NAME: Shreya Sathyanarayana STD.: V Sem SEC.: F ROLL NO.: SUB.: AI Lab

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
1		Tic Tac Toe		
2.		Vaccum Cleaner Problem		
3.		BFS without Heuristic		
4.		DFS without Heuristic		
5.		Iterative Deepening		[illegible] 8/9
6.		A* Algorithm		
		↳ using displaced tiles		
		↳ using manhatten distance		
7.		Hill Climbing		[illegible] 15.09
8.		Simulated Annealing		
9.		Propositional logic		
10.		Unification Algorithm		[illegible] 13.10
11.		FOL Algorithm		
12.		Resolution - FOL		[illegible]
13.		Alpha - Beta Search		27/10

Completed

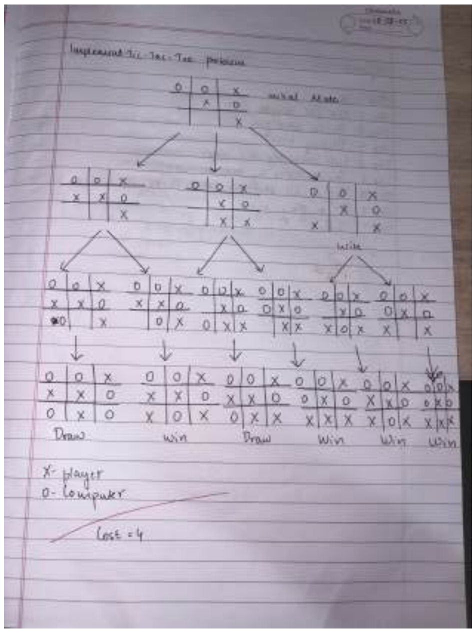
[illegible]

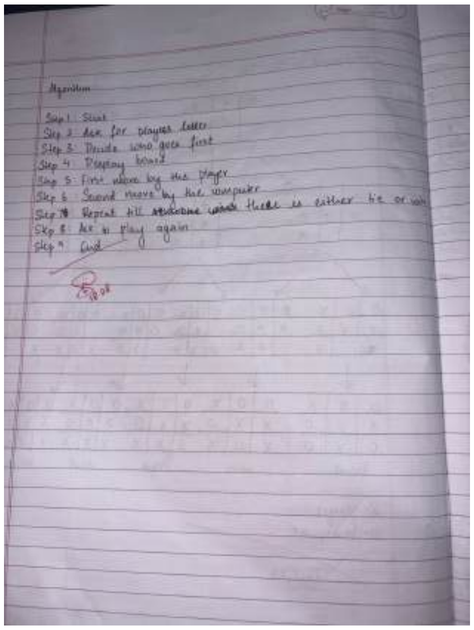
GKC1224621 B

Github Link:
https://github.com/shreyasathyanarayana1/AI_Lab

Program 1
Implement Tic –Tac –Toe Game

Algorithm:





Algorithm

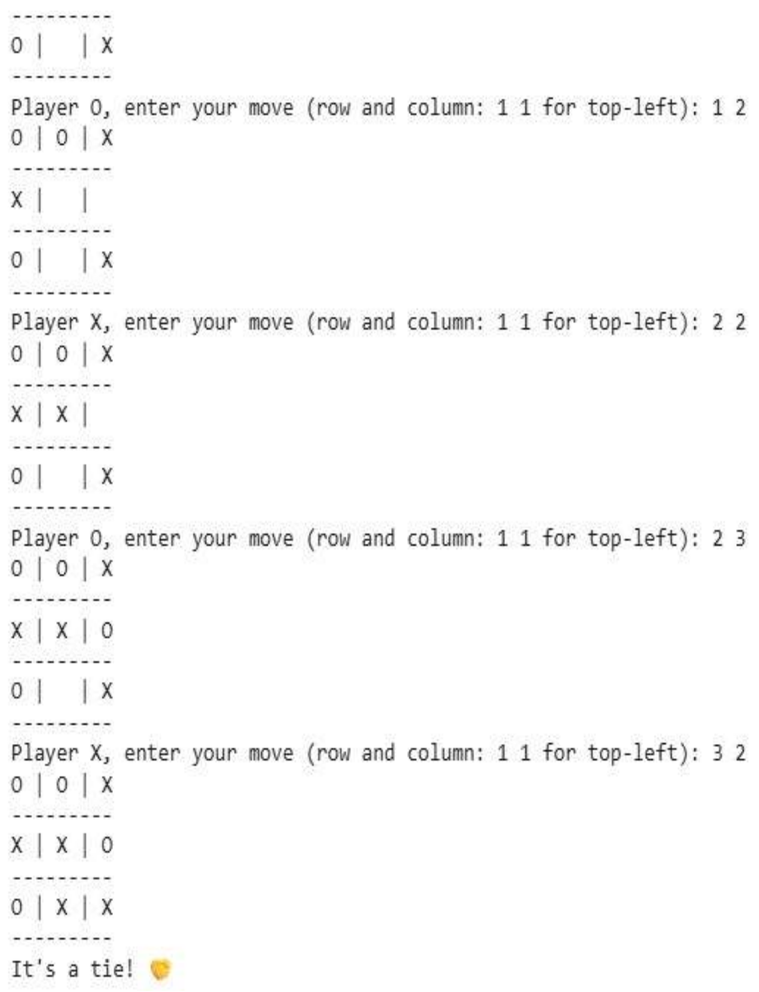
```
Step 1: Start
Step 2: Ask for players letter
Step 3: Decide who goes first
Step 4: Display board
Step 5: First move by the player
Step 6: Second move by the computer
Step 7: Repeat till [illegible] there is either tie or win
Step 8: Ask to play again
Step 9: End
```

Code:

```
def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("---------")



def check_winner(board):
    # Check rows
    for row in board:
        if row[0] == row[1] == row[2] != " ":
            return True
    # Check columns
    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col] != " ":
            return True
    # Check diagonals
    if board[0][0] == board[1][1] == board[2][2] != " ":
        return True
    if board[0][2] == board[1][1] == board[2][0] != " ":
        return True
    return False
```

```
def is_full(board):
    for row in board:
        if " " in row:
            return False
    return True


def tic_tac_toe():
    print("Welcome to Tic Tac Toe!")
    board = [[" " for _ in range(3)] for _ in range(3)]
    print_board(board)
    current_player = "X"

    while True:
        try:
            row, col = map(int, input(f"Player {current_player}, enter your move (row and column: 1 1 for top-
left): ").split())

            if row < 1 or row > 3 or col < 1 or col > 3:
                print("Invalid position! Enter numbers between 1 and 3.")
                continue

            if board[row - 1][col - 1] != " ":
                print("Cell already taken! Try again.")
                continue

            board[row - 1][col - 1] = current_player
            print_board(board)

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

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

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

        except ValueError:
            print("Invalid input! Enter row and column numbers separated by a space.")


tic_tac_toe()
```

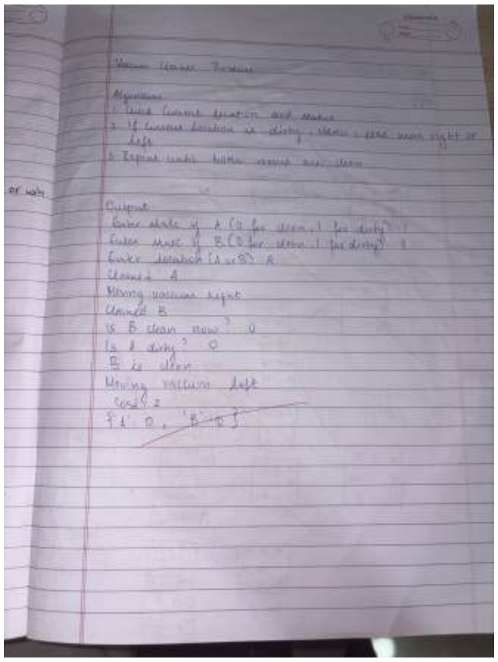
OUTPUT:



```
---------
O |   | X
---------
Player O, enter your move (row and column: 1 1 for top-left): 1 2
O | O | X
---------
X |   |
---------
O |   | X
---------
Player X, enter your move (row and column: 1 1 for top-left): 2 2
O | O | X
---------
X | X |
---------
O |   | X
---------
Player O, enter your move (row and column: 1 1 for top-left): 2 3
O | O | X
---------
X | X | O
---------
O |   | X
---------
Player X, enter your move (row and column: 1 1 for top-left): 3 2
O | O | X
---------
X | X | O
---------
O | X | X
---------
It's a tie! 🤝
```

Implement vacuum cleaner agent

Algorithm:



Vacuum Cleaner Problem

Algorithm
1. Check current location and status
2. If current location is dirty, clean, else move right or left
3. Repeat until both rooms are clean

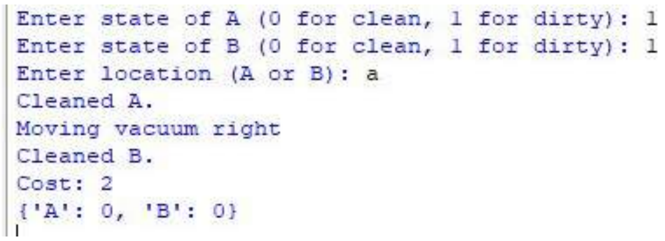
Output:
```
Enter state of A (0 for clean, 1 for dirty) 1
Enter state of B (0 for clean, 1 for dirty) 1
Enter location (A or B) A
Cleaned A
Moving vacuum right
Cleaned B
Is B clean now? 0
Is A dirty? 0
B is clean
Moving vacuum left
Cost: 2
{'A': 0, 'B': 0}
```

Code:

VACCUM CLEANER FOR 2 ROOMS

```
def vacuum_world():
    # Take input for room states
    state = {
        '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): "))
    }

    # Take input for current location
    location = input("Enter location (A or B): ").strip().upper()
    cost = 0

    # Logic for location A
    if location == 'A':
        if state['A'] == 1:
            print("Cleaned A.")
            state['A'] = 0
            cost += 1
        else:
            print("A is already clean.")
        if state['B'] == 1:
            print("Moving vacuum right →")
            cost += 1
            print("Cleaned B.")
            state['B'] = 0
        else:
            print("Moving vacuum right →")
            print("B is already clean.")

    # Logic for location B
    elif location == 'B':
        if state['B'] == 1:
            print("Cleaned B.")
            state['B'] = 0
            cost += 1
        else:
            print("B is already clean.")

        if state['A'] == 1:
            print("Moving vacuum left ←")
            cost += 1
            print("Cleaned A.")
            state['A'] = 0
        else:
            print("Moving vacuum left ←")
            print("A is already clean.")

    print(f"\nTotal Cost: {cost}")
    print(f"Final State: {state}")


vacuum_world()
```

OUTPUT :

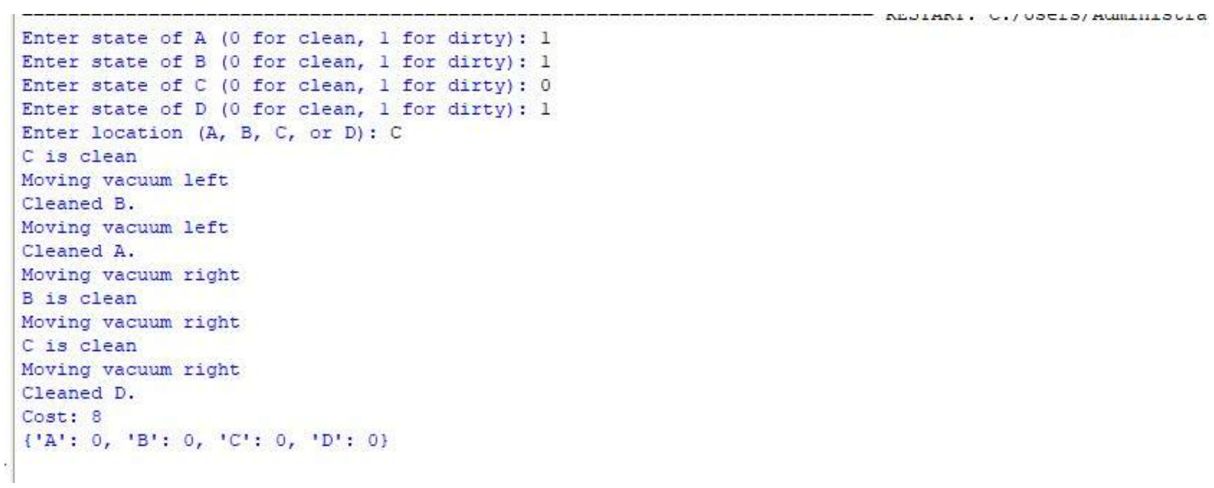


```
Enter state of A (0 for clean, 1 for dirty): 1
Enter state of B (0 for clean, 1 for dirty): 1
Enter location (A or B): a
Cleaned A.
Moving vacuum right
Cleaned B.
Cost: 2
{'A': 0, 'B': 0}
```

VACCUM CLEANER FOR 4 ROOMS

```
def vacuum_world_4_locations():
    # Take input for room states
    state = {
        '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): ")),
        'C': int(input("Enter state of C (0 for clean, 1 for dirty): ")),
        'D': int(input("Enter state of D (0 for clean, 1 for dirty): "))
    }

    # Take input for starting location
    location = input("Enter starting location (A, B, C, or D): ").strip().upper()
    cost = 0
    locations = ['A', 'B', 'C', 'D']
    current_index = locations.index(location)

    print("\n--- Vacuum Cleaner Simulation Start ---\n")

    # Continue until all rooms are clean
    while any(value == 1 for value in state.values()):
        current_loc = locations[current_index]

        # Clean the current room if dirty
        if state[current_loc] == 1:
            print(f"Cleaned Room {current_loc}.")
            state[current_loc] = 0
            cost += 1
        else:
            print(f"Room {current_loc} is already clean.")

        # Check if any rooms are still dirty
        if any(value == 1 for value in state.values()):
            dirty_indices = [i for i, loc in enumerate(locations) if state[loc] == 1]
            # Find nearest dirty room
            nearest_dirty = min(dirty_indices, key=lambda i: abs(i - current_index))

            # Move towards the nearest dirty room
            if nearest_dirty > current_index:
                print("Moving vacuum right →")
                current_index += 1
                cost += 1
            elif nearest_dirty < current_index:
                print("Moving vacuum left ←")
                current_index -= 1
                cost += 1
        else:
            break

    print("\n--- All Rooms Cleaned ---")
    print(f"Total Cost: {cost}")
    print(f"Final Room States: {state}")


# Run the simulation
vacuum_world_4_locations()
vacuum_world_4_locations()
```

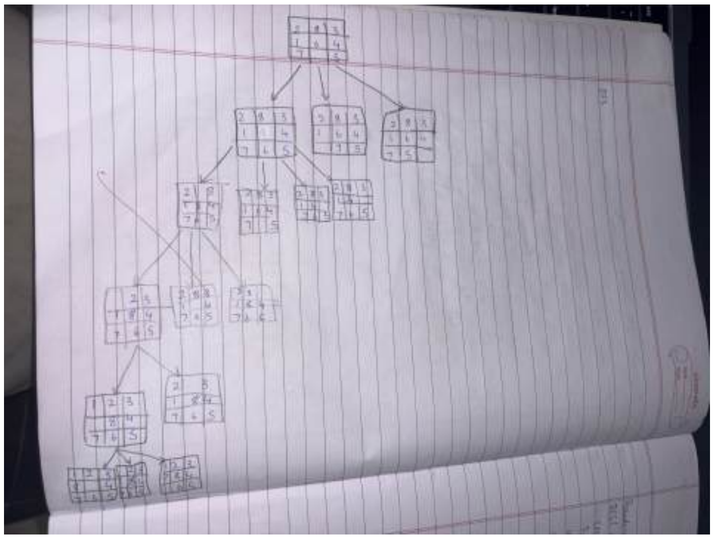
OUTPUT:

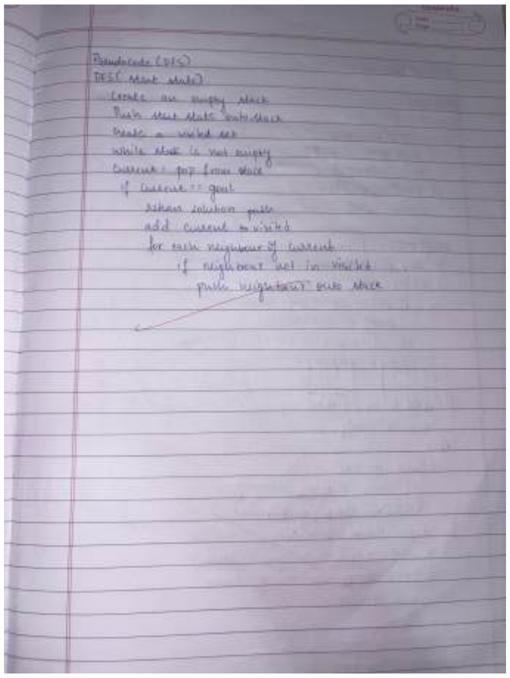


```
========================================================================= RESTART: C:/Users/Administra
Enter state of A (0 for clean, 1 for dirty): 1
Enter state of B (0 for clean, 1 for dirty): 1
Enter state of C (0 for clean, 1 for dirty): 0
Enter state of D (0 for clean, 1 for dirty): 1
Enter location (A, B, C, or D): C
C is clean
Moving vacuum left
Cleaned B.
Moving vacuum left
Cleaned A.
Moving vacuum right
B is clean
Moving vacuum right
C is clean
Moving vacuum right
Cleaned D.
Cost: 8
{'A': 0, 'B': 0, 'C': 0, 'D': 0}
```

<u>Program 2</u>
Implement 8 puzzle problems using Depth First Search (DFS)

Algorithm:





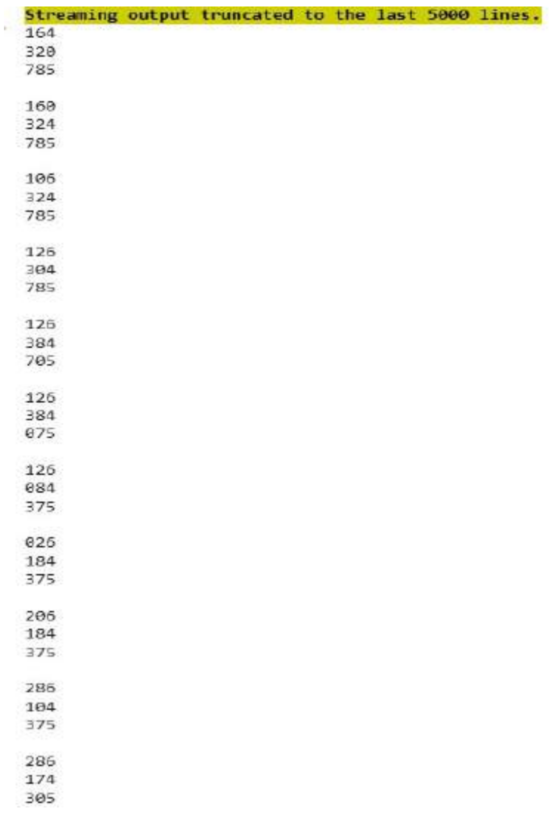
```
Pseudocode (DFS)
DFS( start state)
  Create an empty stack
  Push start state onto stack
  Create a visited set
  while stack is not empty
  current = pop from stack
   if current == goal
      return solution path
      add current to visited
      for each neighbour of current
          if neighbour not in visited
             push neighbour onto stack
```

Code:

```
def get_neighbors_dfs(state: str):
    idx = state.index('0')
    r, c = divmod(idx, 3)
    moves = [(0, 1), (1, 0), (0, -1), (-1, 0)]
    neighbors = []
    for dr, dc in moves:
        nr, nc = r + dr, c + dc
        if 0 <= nr < 3 and 0 <= nc < 3:
            nidx = nr * 3 + nc
            s_list = list(state)
            s_list[idx], s_list[nidx] = s_list[nidx], s_list[idx]
            neighbors.append("".join(s_list))
    return neighbors


def is_solvable(state: str) -> bool:
    nums = [int(x) for x in state if x != '0']
    inv = 0
    for i in range(len(nums)):
        for j in range(i + 1, len(nums)):
            if nums[i] > nums[j]:
```

```
            inv += 1
    return inv % 2 == 0


def dfs(start_state: str, goal_state: str):
    stack = [start_state]
    visited = set()
    parent = {start_state: None}
    while stack:
        current = stack.pop()
        if current == goal_state:
            path = []
            while current is not None:
                path.append(current)
                current = parent[current]
            return path[::-1]
        if current in visited:
            continue
        visited.add(current)
        neighbors = get_neighbors_dfs(current)
        neighbors.reverse()
        for neighbor in neighbors:
            if neighbor not in visited and neighbor not in parent:
                parent[neighbor] = current
                stack.append(neighbor)
    return None


def print_state_as_grid(state: str):
    print(state[:3])
    print(state[3:6])
    print(state[6:])
    print()


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

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

if len(initial_state) != 9 or len(goal_state) != 9 or \
   not all(ch in "0123456789" for ch in initial_state) or \
   not all(ch in "0123456789" for ch in goal_state):
    print("\nInvalid input. Please enter digits 0–8, three per row.")
else:
    if not is_solvable(initial_state):
        print("\nThis initial configuration is not solvable for a standard 8-puzzle.")
    else:
        solution = dfs(initial_state, goal_state)
        if solution:
            print("\nDFS solution path:")
            for s in solution:
                print_state_as_grid(s)
            print(f"Moves: {len(solution) - 1}")
        else:
            print("\nNo solution found.")
```

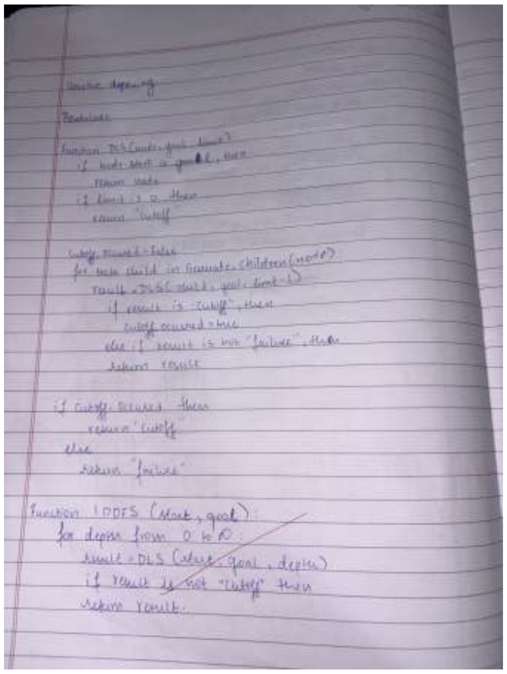
OUTPUT:



```
Streaming output truncated to the last 5000 lines.
164
320
785

160
324
785

106
324
785

126
304
785

126
384
705

126
384
075

126
084
375

026
184
375

206
184
375

286
104
375

286
174
305
```

Implement Iterative deepening search algorithm

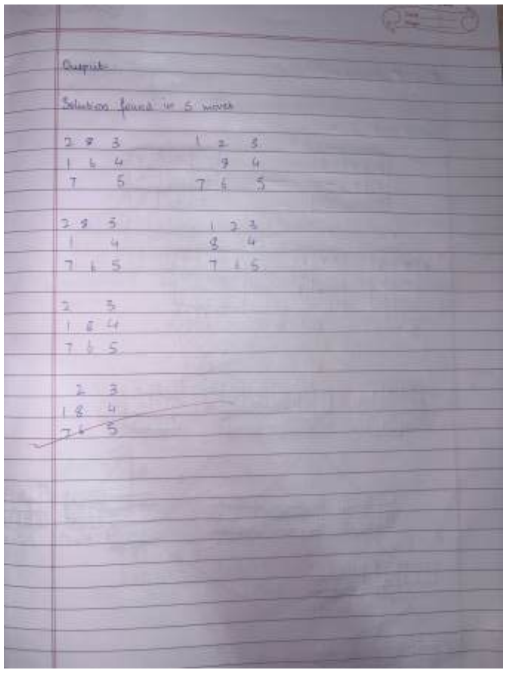
Algorithm:



Iterative deepening

Pseudocode

```
function DLS(node, goal, limit)
  if node.state is goal, then
    return node
  if limit = 0 then
    return "cutoff"

  cutoff_occurred = false
  for each child in Generate_children(node)
    result = DLS(child, goal, limit-1)
      if result is "cutoff", then
        cutoff occurred = true
      else if result is not "failure", then
      return result

if cutoff occurred then
   return "cutoff"
else
   return "failure"

function IDDFS (start, goal)
  for depth from 0 to ∞:
    result = DLS (start, goal, depth)
    if result is not "cutoff" then
    return result.
```



Output:

Solutions found in 5 moves

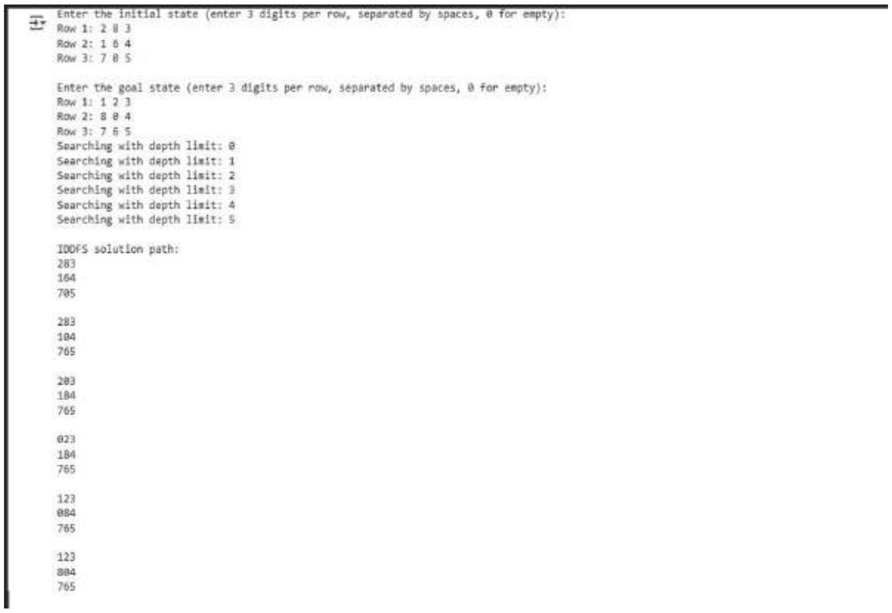
```
2 8 3        1 2 3
1 6 4          8 4
7   5        7 6 5

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

2   3
1 8 4
7 6 5

  2 3
1 8 4
7 6 5
```

Code:

```
def get_neighbors(state):
    neighbors = []
    idx = state.index("0")
    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # up, down, left, right
    x, y = divmod(idx, 3)

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



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

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

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

        if depth < limit and current_state not in visited:
            visited.add(current_state)
            neighbors = get_neighbors(current_state)
            neighbors.reverse()  # maintain consistent exploration order

            for neighbor in neighbors:
                if neighbor not in visited:
                    parent[neighbor] = current_state
                    stack.append((neighbor, depth + 1))
    return None
```

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



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


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


max_depth = 50
solution = iddfs(initial_state, goal_state, max_depth)


if solution:
    print("\nIDDFS solution path:")
    for s in solution:
        print(s[:3])
        print(s[3:6])
        print(s[6:])
        print()
    print(f"Total moves: {len(solution) - 1}")
else:
    print(f"\nNo solution found within the maximum depth of {max_depth}.")
```

OUTPUT:



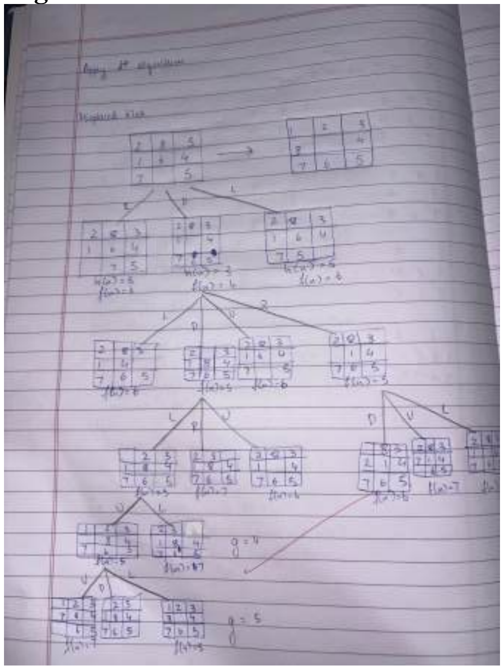
```
Enter the initial state (enter 3 digits per row, separated by spaces, 0 for empty):
Row 1: 2 8 3
Row 2: 1 6 4
Row 3: 7 0 5

Enter the goal state (enter 3 digits per row, separated by spaces, 0 for empty):
Row 1: 1 2 3
Row 2: 8 0 4
Row 3: 7 6 5
Searching with depth limit: 0
Searching with depth limit: 1
Searching with depth limit: 2
Searching with depth limit: 3
Searching with depth limit: 4
Searching with depth limit: 5

IDDFS solution path:
283
164
705

283
104
765

203
184
765

023
184
765

123
084
765

123
804
765
```

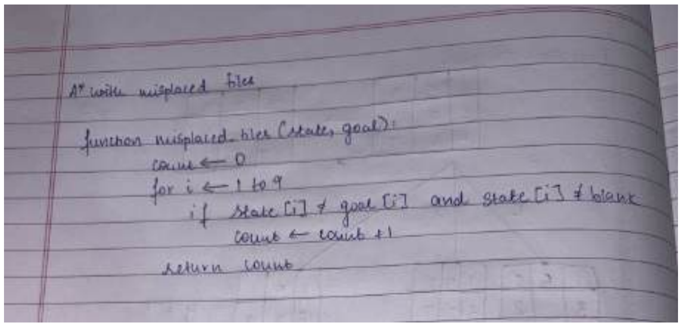
Program 3

Implement A* search algorithm

For misplaced tiles:

Algorithm:





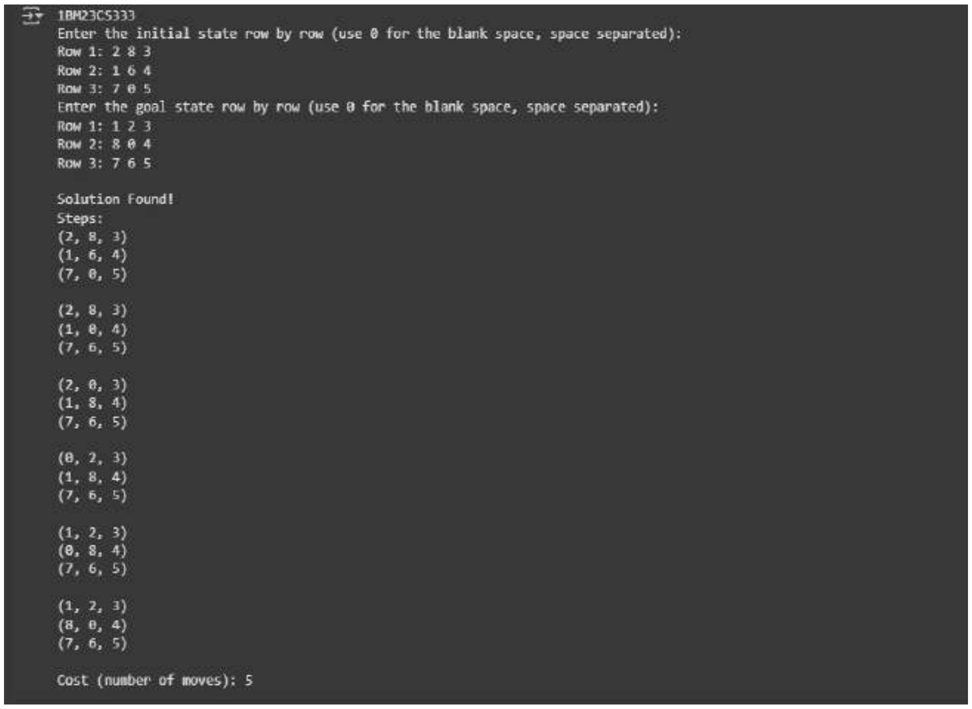
A* with misplaced tiles

```
function misplaced_tiles (state, goal):
    count ← 0
    for i ← 1 to 9
        if state [i] ≠ goal [i]  and state [i] ≠ blank
            count ← count +1
    return count
```

Code:

```
import heapq


def misplaced_tiles_heuristic(state, goal_state):
    """Counts tiles out of place (ignores 0). state & goal_state are 3x3 tuple-of-tuples/lists."""
    misplaced_count = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0 and state[i][j] != goal_state[i][j]:
                misplaced_count += 1
    return misplaced_count



def get_blank_position(state):
    """Returns (row, col) of the blank (0)."""
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
    return -1, -1  # should not happen if input is valid


def get_neighbors(state):
    """All states reachable by sliding one tile into the blank. Returns list of tuple-of-tuples."""
    neighbors = []
    row, col = get_blank_position(state)
    moves = [(0, 1), (0, -1), (1, 0), (-1, 0)]  # Right, Left, Down, Up

    for dr, dc in moves:
        new_row, new_col = row + dr, col + dc
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_state = [list(r) for r in state]  # make a mutable copy
            new_state[row][col], new_state[new_row][new_col] = new_state[new_row][new_col],
new_state[row][col]
            neighbors.append(tuple(tuple(r) for r in new_state))  # back to tuple-of-tuples
    return neighbors


def a_star(initial_state, goal_state):
    """
    A* with misplaced tiles heuristic.
    Returns (path, cost) where path is a list of states (tuple-of-tuples).
    """
    initial_state = tuple(tuple(row) for row in initial_state)
    goal_state = tuple(tuple(row) for row in goal_state)

    # open_set stores tuples: (f_cost, g_cost, state, path_so_far)
    start_h = misplaced_tiles_heuristic(initial_state, goal_state)
    open_set = [(start_h, 0, initial_state, [])]
    closed_set = set()

    while open_set:
        f_cost, g_cost, current_state, path = heapq.heappop(open_set)

        if current_state == goal_state:
            return path + [current_state], g_cost  # solved

        if current_state in closed_set:
            continue
        closed_set.add(current_state)

        for neighbor in get_neighbors(current_state):
            if neighbor in closed_set:
                continue
            new_g = g_cost + 1
            new_f = new_g + misplaced_tiles_heuristic(neighbor, goal_state)
            heapq.heappush(open_set, (new_f, new_g, neighbor, path + [current_state]))

    return None, -1  # no solution found


def get_user_input_state(state_name):
    """Reads a 3x3 state from user input (0–8, space-separated; use 0 for blank)."""
    print(f"Enter the {state_name} state row by row (use 0 for the blank, space separated):")
    state = []
    for i in range(3):
        while True:
            try:
                row = list(map(int, input(f"Row {i+1}: ").split()))
                if len(row) == 3 and all(0 <= x <= 8 for x in row):
                    state.append(row)
                    break
                else:
                    print("Invalid input. Please enter exactly 3 numbers between 0 and 8.")
            except ValueError:
                print("Invalid input. Please enter numbers separated by spaces.")
    return state


if __name__ == "__main__":
    # Get initial and goal states
    initial_state = get_user_input_state("initial")
    goal_state = get_user_input_state("goal")

    # Solve
    path, cost = a_star(initial_state, goal_state)

    # Output
    if path:
        print("\nSolution Found!")
        print("Steps:")
        for step_index, step in enumerate(path):
            print(f"Step {step_index}:")
            for row in step:
                print(list(row))
            print()
        print(f"Cost (number of moves): {cost}")
        print(f"Total states in path: {len(path)}")
    else:
        print("\nNo solution found for the given states.")
```

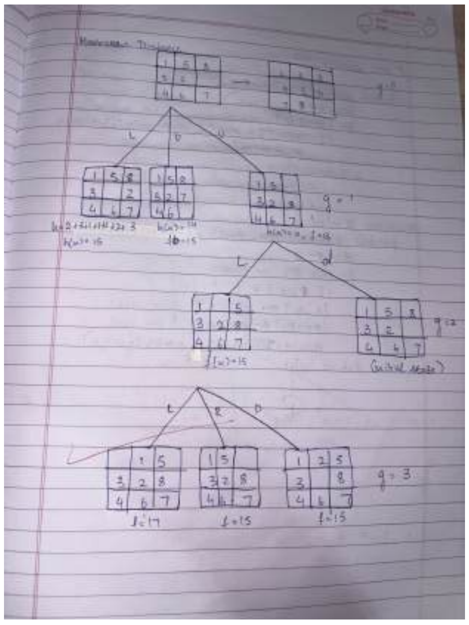
OUTPUT:

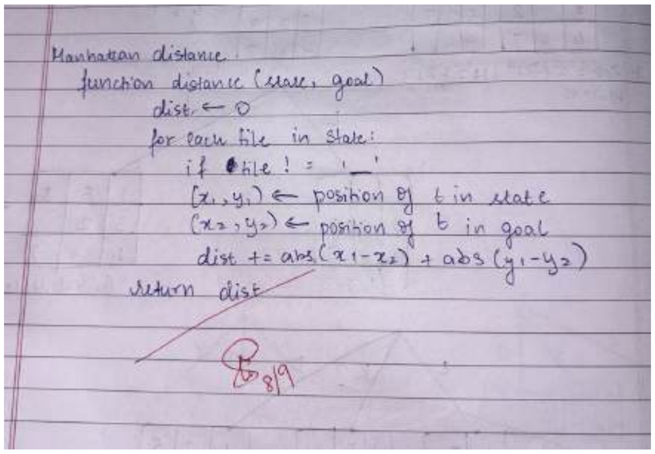


```
1BM23CS333
Enter the initial state row by row (use 0 for the blank space, space separated):
Row 1: 2 8 3
Row 2: 1 6 4
Row 3: 7 0 5
Enter the goal state row by row (use 0 for the blank space, space separated):
Row 1: 1 2 3
Row 2: 8 0 4
Row 3: 7 6 5

Solution Found!
Steps:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

(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)

Cost (number of moves): 5
```

Using Manhattan distance:

Algorithm:





```
Manhattan distance
  function distance (state, goal)
      dist ← 0
      for each tile in state:
          if tile ! = '_'
          (x1, y1) ← position of t in state
          (x2, y2) ← position of t in goal
          dist += abs(x1-x2) + abs(y1-y2)
    return dist
```

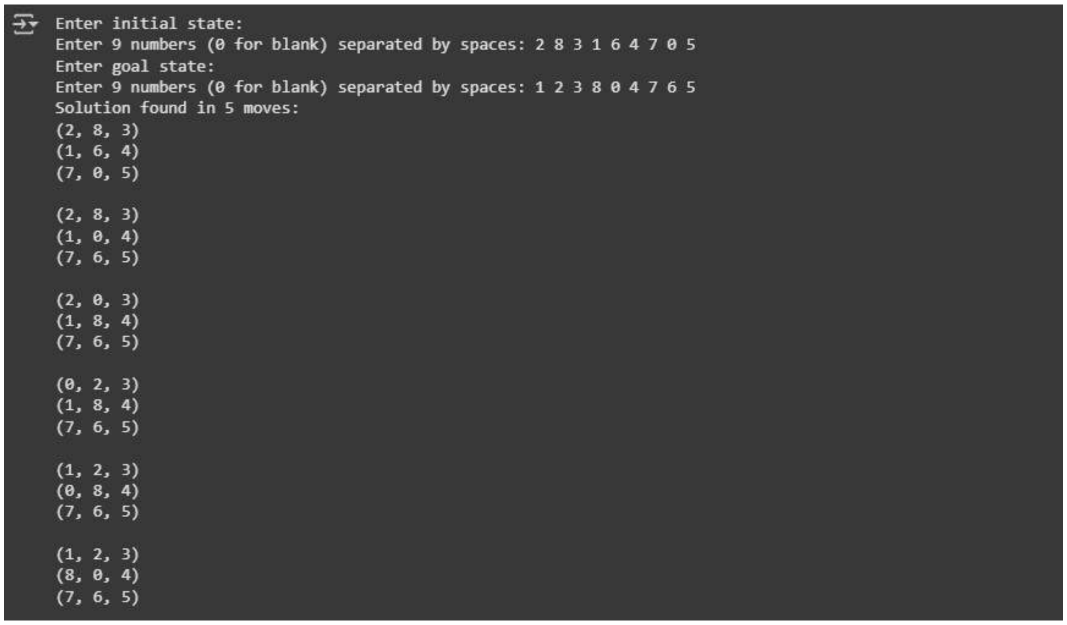
[illegible] 8/9

Code:

```
import heapq
from itertools import count

class PuzzleState:
    def __init__(self, board, moves=0, previous=None):
        """
        board: tuple of length 9 for the 3x3 board; 0 is the blank.
        moves: g-cost (number of moves from start)
        previous: parent PuzzleState for path reconstruction
        """
        self.board = board
        self.moves = moves
        self.previous = previous
        self.size = 3  # 3x3 puzzle

    def __eq__(self, other):
        return isinstance(other, PuzzleState) and self.board == other.board

    def __hash__(self):
        return hash(self.board)

    def get_neighbors(self):
        """Return list of PuzzleState neighbors by sliding the blank."""
        neighbors = []
        zero_index = self.board.index(0)
        x, y = divmod(zero_index, self.size)
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # U, D, L, R
        for dx, dy in directions:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < self.size and 0 <= new_y < self.size:
                new_zero_index = new_x * self.size + new_y
                new_board = list(self.board)
                new_board[zero_index], new_board[new_zero_index] = (
                    new_board[new_zero_index],
                    new_board[zero_index],
                )
                neighbors.append(PuzzleState(tuple(new_board), self.moves + 1, self))
        return neighbors

    def manhattan_distance(self, goal_positions):
        """Compute Manhattan distance to the goal using a precomputed tile->goal_index map."""
        dist = 0
        for idx, tile in enumerate(self.board):
            if tile == 0:
                continue
            goal_idx = goal_positions[tile]
            x1, y1 = divmod(idx, self.size)
            x2, y2 = divmod(goal_idx, self.size)
            dist += abs(x1 - x2) + abs(y1 - y2)
        return dist

    def __lt__(self, other):
        # Only for heap tie-breaking; actual ordering is done via f-cost and a counter.
        return False


def inversion_parity(board):
    """Return inversion parity (0 even, 1 odd) ignoring the blank."""
    arr = [x for x in board 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 % 2


def is_solvable_relative(start_board, goal_board):
    """
    For 3x3 (odd width) puzzles, start is solvable to goal iff inversion parity matches.
    """
    return inversion_parity(start_board) == inversion_parity(goal_board)
```

```
def a_star(start, goal):
    """
    A* search using Manhattan distance heuristic.
    Returns the path (list of PuzzleState) from start to goal, or None if not found.
    """
    # Precompute goal positions for fast Manhattan lookup
    goal_positions = {tile: idx for idx, tile in enumerate(goal.board)}

    open_heap = []
    entry_counter = count()  # tie-breaker for heap

    start_h = start.manhattan_distance(goal_positions)
    # g(start) = 0, so f = h
    heapq.heappush(open_heap, (start_h, next(entry_counter), start))

    g_score = {start: 0}
    closed_set = set()

    while open_heap:
        _, _, current = heapq.heappop(open_heap)

        if current == goal:
            # reconstruct path
            path = []
            node = current
            while node is not None:
                path.append(node)
                node = node.previous
            path.reverse()
            return path

        if current in closed_set:
            continue
        closed_set.add(current)

        for neighbor in current.get_neighbors():
            if neighbor in closed_set:
                continue

            tentative_g = g_score[current] + 1

            # If this path to neighbor is better, record it
            if neighbor not in g_score or tentative_g < g_score[neighbor]:
                neighbor.previous = current  # keep parent pointer in sync
                g_score[neighbor] = tentative_g
                h = neighbor.manhattan_distance(goal_positions)
                f = tentative_g + h
            heapq.heappush(open_heap, (f, next(entry_counter), neighbor))

    return None


def print_path(path):
    for state in path:
        for i in range(3):
            print(state.board[i * 3 : (i + 1) * 3])
        print()


def get_input_state(prompt):
    print(prompt)
    while True:
        try:
            values = list(
                map(int, input("Enter 9 numbers (0 for blank) separated by spaces: ").strip().split())
            )
            if len(values) != 9 or sorted(values) != list(range(9)):
                raise ValueError
            return tuple(values)
        except ValueError:
            print("Invalid input. Please enter numbers 0 to 8 without duplicates.")


if __name__ == "__main__":
    # Read start and goal
    start_board = get_input_state("Enter initial state:")
    goal_board = get_input_state("Enter goal state:")

    if not is_solvable_relative(start_board, goal_board):
        print("The given initial state is NOT solvable to the given goal. Exiting.")
    else:
        start_state = PuzzleState(start_board)
        goal_state = PuzzleState(goal_board)

        solution_path = a_star(start_state, goal_state)

        if solution_path:
            print(f"Solution found in {len(solution_path) - 1} moves:")
            print_path(solution_path)
        else:
            print("No solution found.")
```

OUTPUT:

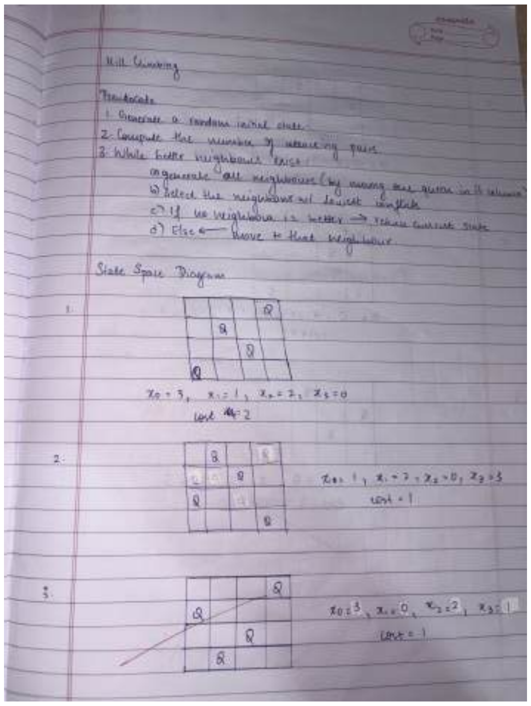


```
Enter initial state:
Enter 9 numbers (0 for blank) separated by spaces: 2 8 3 1 6 4 7 0 5
Enter goal state:
Enter 9 numbers (0 for blank) separated by spaces: 1 2 3 8 0 4 7 6 5
Solution found in 5 moves:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

(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)
```

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:



Hill Climbing

Pseudocode

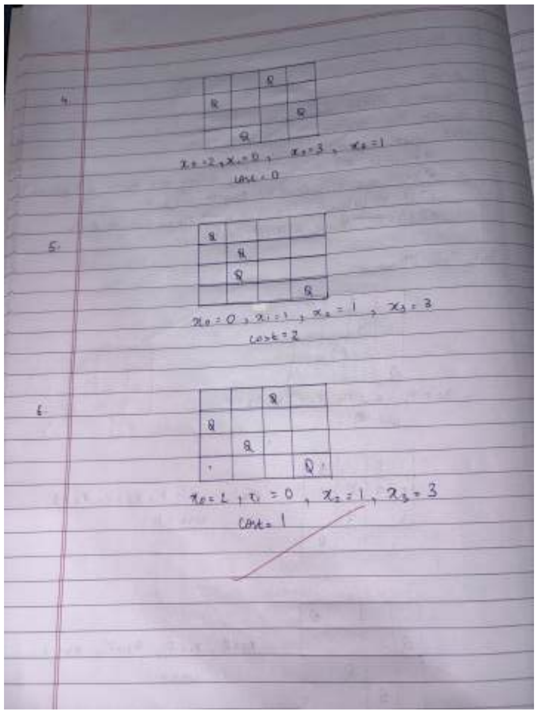
```
1. Generate a random initial state.
2. Compute the number of attacking pairs.
3. While better neighbours exist:
      a) generate all neighbours (by moving one queen in its column)
      b) Select the neighbour with lowest conflicts
      c) If no neighbour is better → return current state
      d) Else ← move to that neighbour
```

State Space Diagram

1. $x_0 = 3$, $x_1 = 1$, $x_2 = 2$, $x_3 = 0$
   cost = 2

2. $x_0 = 1$, $x_1 = 2$, $x_2 = 0$, $x_3 = 3$
   cost = 1

3. $x_0 = 3$, $x_1 = 0$, $x_2 = 2$, $x_3 = 1$
   cost = 1



4. $x_0 = 2$, $x_1 = 0$, $x_2 = 3$, $x_3 = 1$
   cost = 0

5. $x_0 = 0$, $x_1 = 1$, $x_2 = 1$, $x_3 = 3$
   cost = 2

6. $x_0 = 2$, $x_1 = 0$, $x_2 = 1$, $x_3 = 3$
   cost = 1

Code:

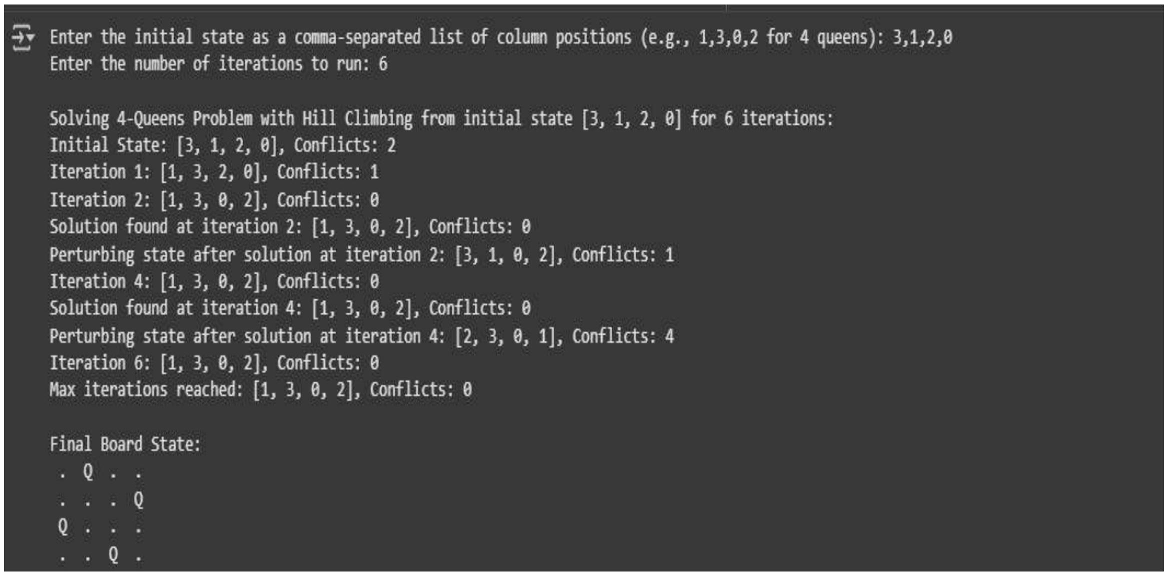
```
import random
def initial_state(n):
    """Generates a random initial state (a permutation of columns)."""
    state = list(range(n))
    random.shuffle(state)
    return state


def conflicts(state):
    """
    Calculates the number of conflicts (attacking queens) in a state.
    Representation: state[row] = column of queen in that row.
    Since it's a permutation, no row/column conflicts; only diagonal conflicts are counted.
    """
    n = len(state)
    conflict_count = 0
    for i in range(n):
        for j in range(i + 1, n):
            # Diagonal conflicts only
            if abs(state[i] - state[j]) == abs(i - j):
                conflict_count += 1
    return conflict_count

def get_neighbors(state):
    """
    Generates neighboring states by swapping positions (columns) of two rows.
    This preserves the permutation structure.
    """
    n = len(state)
    neighbors = []
    for i in range(n):
        for j in range(i + 1, n):
            neighbor = list(state)
            neighbor[i], neighbor[j] = neighbor[j], neighbor[i]
            neighbors.append(neighbor)
    return neighbors

def hill_climbing(initial_state_list, max_iterations=1000):
    """
    Hill Climbing to reduce conflicts in N-Queens.
    Continues for max_iterations; if a solution is found, it prints it and
    performs a small perturbation to potentially explore other solutions.
    """
    current_state = list(initial_state_list)
    current_conflicts = conflicts(current_state)
```

```
print(f"Initial State: {current_state}, Conflicts: {current_conflicts}")

for i in range(max_iterations):
    if current_conflicts == 0:
        print(f"Solution found at iteration {i}: {current_state}, Conflicts: {current_conflicts}")
        # Perturb to explore other solutions (if iterations remain)
        if i < max_iterations - 1:
            n = len(current_state)
            if n > 1:
                idx1, idx2 = random.sample(range(n), 2)
                current_state[idx1], current_state[idx2] = current_state[idx2], current_state[idx1]
                current_conflicts = conflicts(current_state)
                print(
                    f"Perturbing state after solution at iteration {i}: "
                    f"{current_state}, Conflicts: {current_conflicts}"
                )
        continue  # Keep iterating as requested

    neighbors = get_neighbors(current_state)
    if not neighbors:
        print(f"No neighbors to explore at iteration {i}: {current_state}, Conflicts: {current_conflicts}")
        for j in range(i + 1, max_iterations):
            print(f"Iteration {j}: {current_state}, Conflicts: {current_conflicts}")
        return current_state

    best_neighbor = current_state
    best_conflicts = current_conflicts

    for neighbor in neighbors:
        neighbor_conflicts = conflicts(neighbor)
```

```
            if neighbor_conflicts < best_conflicts:
                best_conflicts = neighbor_conflicts
                best_neighbor = neighbor

        print(f"Iteration {i + 1}: {best_neighbor}, Conflicts: {best_conflicts}")

        if best_conflicts >= current_conflicts:
            # Local optimum or plateau reached
            for j in range(i + 1, max_iterations):
                print(f"Iteration {j + 1}: {current_state}, Conflicts: {current_conflicts}")
            return current_state

        current_state = best_neighbor
        current_conflicts = best_conflicts

    print(f"Max iterations reached: {current_state}, Conflicts: {current_conflicts}")
    return current_state

def print_board(state):
    """Prints the N-Queens board."""
    if state is None:
        print("No solution found.")
        return
    n = len(state)
    for row in range(n):
        line = ""
        for col in range(n):
            line += " Q " if state[row] == col else " . "
        print(line)

# Get input from the user
```

```
initial_state_str = input(
    "Enter the initial state as a comma-separated list of column positions "
    "(e.g., 1,3,0,2 for 4 queens): "
)
max_iterations_str = input("Enter the number of iterations to run: ")

try:
    initial_state_list = [int(x.strip()) for x in initial_state_str.split(',')]
    max_iterations = int(max_iterations_str)
    n = len(initial_state_list)

    # Validate: permutation of 0..n-1
    if set(initial_state_list) != set(range(n)):
        raise ValueError("Initial state must be a permutation of 0..n-1 (each value exactly once).")

    print(
        f"\nSolving {n}-Queens Problem with Hill Climbing from initial state "
        f"{initial_state_list} for {max_iterations} iterations:"
    )
    final_state = hill_climbing(initial_state_list, max_iterations)

    print("\nFinal Board State:")
    print_board(final_state)

except ValueError as e:
    print(f"Invalid input: {e}")
    print("Please enter the initial state as a comma-separated permutation of integers 0..n-1, "
          "and the numb
```

OUTPUT:

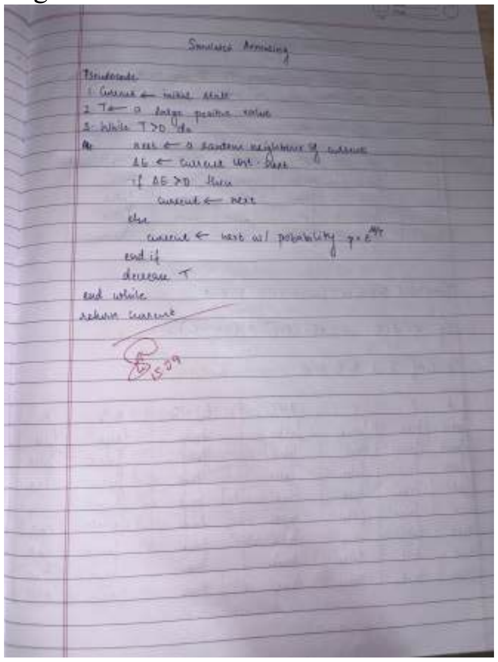


```
Enter the initial state as a comma-separated list of column positions (e.g., 1,3,0,2 for 4 queens): 3,1,2,0
Enter the number of iterations to run: 6

Solving 4-Queens Problem with Hill Climbing from initial state [3, 1, 2, 0] for 6 iterations:
Initial State: [3, 1, 2, 0], Conflicts: 2
Iteration 1: [1, 3, 2, 0], Conflicts: 1
Iteration 2: [1, 3, 0, 2], Conflicts: 0
Solution found at iteration 2: [1, 3, 0, 2], Conflicts: 0
Perturbing state after solution at iteration 2: [3, 1, 0, 2], Conflicts: 1
Iteration 4: [1, 3, 0, 2], Conflicts: 0
Solution found at iteration 4: [1, 3, 0, 2], Conflicts: 0
Perturbing state after solution at iteration 4: [2, 3, 0, 1], Conflicts: 4
Iteration 6: [1, 3, 0, 2], Conflicts: 0
Max iterations reached: [1, 3, 0, 2], Conflicts: 0

Final Board State:
 . Q . .
 . . . Q
 Q . . .
 . . Q .
```

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:



Simulated Annealing

Pseudocode

```
1. current ← initial state
2. T ← a large positive value
3. While T > 0 do
4.      next ← a random neighbour of current
        ΔE ← current cost - next
        if ΔE > 0 then
            current ← next
        else
            current ← next w/ probability $p = e^{\Delta E/T}$
        end if
        decrease T
   end while
   return current
```

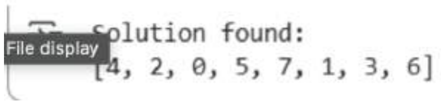
15/09

Code:

```
import random
import math


def cost_function(state):
    conflicts = 0
    for i in range(len(state)):
        for j in range(i + 1, len(state)):
            if state[i] == state[j] or state[i] - i == state[j] - j or state[i] + i == state[j] + j:
                conflicts += 1
    return conflicts


def get_neighbors(state):
    neighbors = []
    for i in range(len(state)):
        for j in range(len(state)):
            if i != j:
                neighbor = list(state)
                neighbor[i] = j
            neighbors.append(neighbor)
    return neighbors

def simulated_annealing(initial_state, temperature, cooling_rate):
    current_state = initial_state
    current_cost = cost_function(current_state)
    best_state = current_state
    best_cost = current_cost
    while temperature > 0.1:
        neighbors = get_neighbors(current_state)
        if not neighbors:
            break
        next_state = random.choice(neighbors)
        next_cost = cost_function(next_state)
        delta_cost = next_cost - current_cost
        if delta_cost < 0 or random.random() < math.exp(-delta_cost / temperature):
            current_state = next_state
            current_cost = next_cost
        if current_cost < best_cost:
            best_state = current_state
            best_cost = current_cost
        temperature *= cooling_rate
    return best_state, best_cost

def solve_8_queens_simulated_annealing():
    n = 8
    initial_state = [random.randint(0, n - 1) for _ in range(n)]
    temperature = 1000
    cooling_rate = 0.95
    solution, cost = simulated_annealing(initial_state, temperature, cooling_rate)
    if cost == 0:
        print("Solution found:")
        print(solution)
    else:
        print("No perfect solution found, best found with conflicts:", cost)
        print(solution)
```

solve_8_queens_simulated_annealing()

OUTPUT:

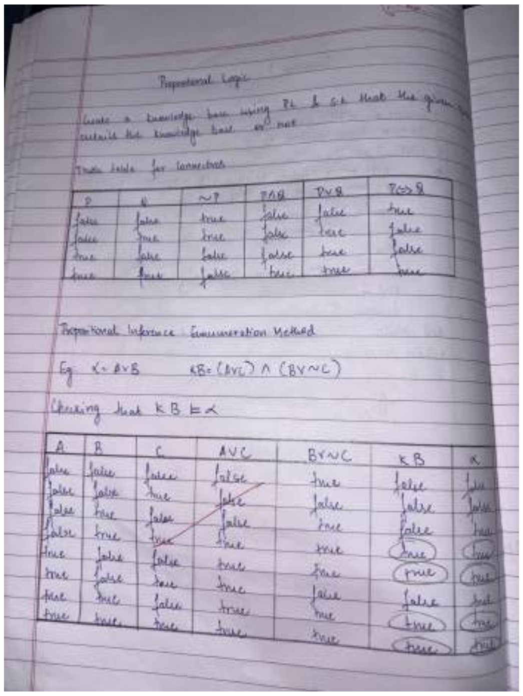


```
Solution found:
[4, 2, 0, 5, 7, 1, 3, 6]
```

Program 6

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

Algorithm:



Propositional Logic

Create a knowledge base using PL & s.t. that the given query entails the knowledge base or not

Truth table for connectives

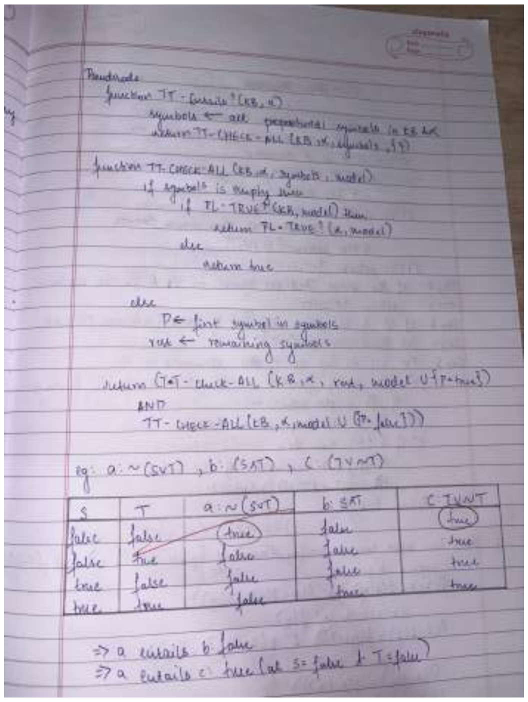
P	Q	~P	P∧Q	P∨Q	P⇔Q
false	false	true	false	false	true
false	true	true	false	true	false
true	false	false	false	true	false
true	true	false	true	true	true

Propositional Inference : Enumeration Method

Eg. α = A∨B   KB = (A∨C) ∧ (B∨~C)

Checking that KB ⊨ α

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



```
Pseudocode
 function TT-Entails?(KB, α)
      symbols ← all propositional symbols in KB & α
      return TT-CHECK-ALL (KB, α, symbols, {})

 function TT-CHECK-ALL (KB, α, symbols, model)
      if symbols is empty then
           if PL-TRUE?(KB, model) then
                return PL-TRUE?(α, model)
           else
                return true

      else
           P ← first symbol in symbols
           rest ← remaining symbols

 return (TT-Check-ALL (KB, α, rest, model ∪ {P=true})
      AND
      TT-CHECK-ALL (KB, α, model ∪ {P=false}))
```

Eg: a: ~(S∨T) , b: (S∧T) , c: (T∨~T)

S	T	a: ~(S∨T)	b: S∧T	c: T∨~T
false	false	true	false	true
false	true	false	false	true
true	false	false	false	true
true	true	false	true	true

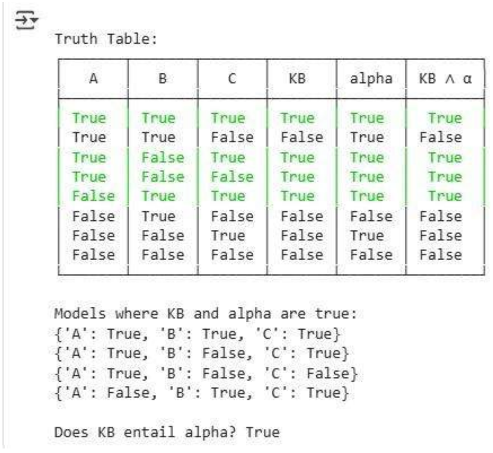
⇒ a entails b false

⇒ a entails c true (at S = false & T = false)

Code:

```
def is_variable(x):
    return isinstance(x, str) and x.islower()

def is_constant(x):
    return isinstance(x, str) and (x.isupper() or x.isdigit() or (x.isalpha() and not x.islower()))

def occurs_in(var, expr):
    if var == expr:
        return True
    if isinstance(expr, dict):
        return any(occurs_in(var, arg) for arg in expr.get('args', []))
    return False

def predicate_symbol(expr):
    if isinstance(expr, dict) and 'pred' in expr:
        return expr['pred']
    return None

def apply_substitution(subst, expr):
    if isinstance(expr, str):
        return subst.get(expr, expr)
    elif isinstance(expr, dict):
        return {
            'pred': expr['pred'],
            'args': [apply_substitution(subst, arg) for arg in expr.get('args', [])]
        }
    return expr

def compose_subst(s1, s2):
    result = {}
    for v, val in s1.items():
        result[v] = apply_substitution(s2, val)
    for v, val in s2.items():
        result[v] = val
    return result

def unify(x, y):
    if x == y:
        return {}
    if isinstance(x, str) and is_variable(x):
        if occurs_in(x, y):
            return "FAILURE"
        return {x: y}
    if isinstance(y, str) and is_variable(y):
        if occurs_in(y, x):
            return "FAILURE"
        return {y: x}
    if isinstance(x, str) and isinstance(y, str):
        return "FAILURE"
    if isinstance(x, dict) and isinstance(y, dict):
        if predicate_symbol(x) != predicate_symbol(y):
            return "FAILURE"
        if len(x.get('args', [])) != len(y.get('args', [])):
            return "FAILURE"
        SUBST = {}
        for a, b in zip(x['args'], y['args']):
            a_ap = apply_substitution(SUBST, a)
            b_ap = apply_substitution(SUBST, b)
            S = unify(a_ap, b_ap)
            if S == "FAILURE":
                return "FAILURE"
            if S:
                SUBST = compose_subst(SUBST, S)
        return SUBST
    return "FAILURE"

def sentence_to_str(sentence):
    if isinstance(sentence, str):
        return sentence
    elif isinstance(sentence, dict):
        args_str = ",".join(sentence_to_str(arg) for arg in sentence.get('args', []))
        return f"{sentence['pred']}({args_str})"
    return str(sentence)

def str_to_sentence(s):
    s = s.strip()
    pred_end = s.find("(")
    if pred_end == -1:
        return s
    pred = s[:pred_end]
    args_str = s[pred_end+1:-1]
    args = [a.strip() for a in args_str.split(",")] if args_str else []
    return {'pred': pred, 'args': args}

def find_substitutions_for_premises(premises, known_facts):
    results = []
    def backtrack(i, subst):
        if i == len(premises):
            results.append(subst.copy())
            return
        prem = apply_substitution(subst, premises[i])
        for fact in known_facts:
            S = unify(prem, fact)
            if S == "FAILURE":
                continue
            new_subst = compose_subst(subst, S)
            backtrack(i + 1, new_subst)
    backtrack(0, {})
    return results

def sentence_in_list(sentence, lst):
    s_str = sentence_to_str(sentence)
    return any(sentence_to_str(s) == s_str for s in lst)

def fol_fc_ask(KB, alpha):
    query = alpha
    known_facts = []
    agenda = []
    for premises, concl in KB:
        if not premises:
            fact = concl
            if not sentence_in_list(fact, known_facts):
                known_facts.append(fact)
                agenda.append(fact)
    while agenda:
        fact = agenda.pop(0)
        if unify(fact, query) != "FAILURE":
            return True
        for premises, concl in KB:
            subs = find_substitutions_for_premises(premises, known_facts)
            for s in subs:
                new_fact = apply_substitution(s, concl)
                if not sentence_in_list(new_fact, known_facts):
                    known_facts.append(new_fact)
                    agenda.append(new_fact)
    return False

KB = [
    ([], {'pred': 'prime', 'args': ['11']}),
    ([{'pred': 'prime', 'args': ['x']}], {'pred': 'odd', 'args': ['x']})
]

alpha = {'pred': 'odd', 'args': ['11']}
result = fol_fc_ask(KB, alpha)
print("Result:", result)
```

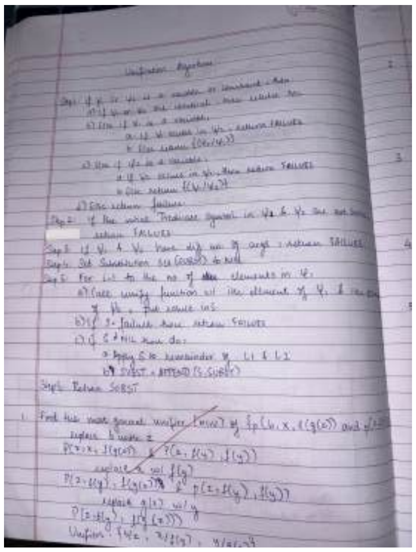
OUTPUT:



```
Truth Table:
+-------+-------+-------+-------+-------+--------+
|   A   |   B   |   C   |  KB   | alpha | KB ∧ α |
+-------+-------+-------+-------+-------+--------+
| True  | True  | True  | True  | True  |  True  |
| True  | True  | False | False | True  | False  |
| True  | False | True  | True  | True  |  True  |
| True  | False | False | True  | True  |  True  |
| False | True  | True  | True  | True  |  True  |
| False | True  | False | False | False | False  |
| False | False | True  | False | True  | False  |
| False | False | False | False | False | False  |
+-------+-------+-------+-------+-------+--------+

Models where KB and alpha are true:
{'A': True, 'B': True, 'C': True}
{'A': True, 'B': False, 'C': True}
{'A': True, 'B': False, 'C': False}
{'A': False, 'B': True, 'C': True}

Does KB entail alpha? True
```

Program 7
Implement unification in first order logic

Algorithm:



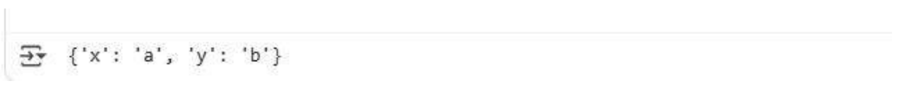
Unification Algorithm

Step 1: If Ψ1 or Ψ2 is a variable or constant, then:
a) If Ψ1 or Ψ2 are identical, then return NIL.
b) Else if Ψ1 is a variable,
a. If Ψ1 occurs in Ψ2, return FAILURE
b. Else return {(Ψ2/Ψ1)}
c) Else if Ψ2 is a variable,
a. If Ψ2 occurs in Ψ1 then return FAILURE
b. Else return {(Ψ1/Ψ2)}
d) Else return failure.
Step 2: If the initial Predicate symbol in Ψ1 & Ψ2 are not same, return FAILURE
Step 3: If Ψ1 & Ψ2 have diff no of args, return FAILURE
Step 4: Set Substitution set (SUBST) to NIL
Step 5: For i=1 to the no of elements in Ψ1
a) Call unify function with the element of Ψ1 & [illegible] of Ψ2, put the result in S
b) If S = failure then return FAILURE
c) If S ≠ NIL then do:
a. Apply S to remainder of L1 & L2
b) SUBST = APPEND(S, SUBST)
Step 6: Return SUBST

1. Find the most general unifier (MGU) of {p(b, x, f(g(z))) and [illegible]
replace b with z
p(z, x, f(g(z))) & p(z, f(y), f(y))
replace x with f(y)
p(z, f(y), f(g(z))) & p(z, f(y), f(y))
replace g(z) with y
p(z, f(y), f(g(z)))
Unifier: {b/z, x/f(y), y/g(z)}

Code:
```
def unify(psi1, psi2):
    if is_variable_or_constant(psi1) or is_variable_or_constant(psi2):
        if psi1 == psi2:
            return {}
        elif is_variable(psi1):
            if occurs_in(psi1, psi2):
                return "FAILURE"
            else:
                return {psi1: psi2}
        elif is_variable(psi2):
            if occurs_in(psi2, psi1):
                return "FAILURE"
            else:
                return {psi2: psi1}
        else:
            return "FAILURE"

    if predicate_symbol(psi1) != predicate_symbol(psi2):
        return "FAILURE"

    if len(psi1['args']) != len(psi2['args']):
        return "FAILURE"

    SUBST = {}
    for i in range(len(psi1['args'])):
        S = unify(psi1['args'][i], psi2['args'][i])
        if S == "FAILURE":
            return "FAILURE"
        if S:
            psi1 = apply_substitution(S, psi1)
            psi2 = apply_substitution(S, psi2)
            SUBST.update(S)
    return SUBST


def is_variable_or_constant(x):
    return isinstance(x, str) and (x.islower() or x.isalpha())


def is_variable(x):
    return isinstance(x, str) and x.islower()


def occurs_in(var, expr):
    if var == expr:
        return True
    if isinstance(expr, dict):
        return any(occurs_in(var, arg) for arg in expr.get('args', []))
    return False


def predicate_symbol(expr):
    if isinstance(expr, dict) and 'pred' in expr:
        return expr['pred']
    return None


def apply_substitution(subst, expr):
    if isinstance(expr, str):
        return subst.get(expr, expr)
    elif isinstance(expr, dict):
        return {
            'pred': expr['pred'],
            'args': [apply_substitution(subst, arg) for arg in expr.get('args', [])]
        }
    return expr


# Example Usage
psi1 = {'pred': 'P', 'args': ['x', 'y']}
psi2 = {'pred': 'P', 'args': ['a', 'b']}
result = unify(psi1, psi2)
print("Unification Result:", result)
```

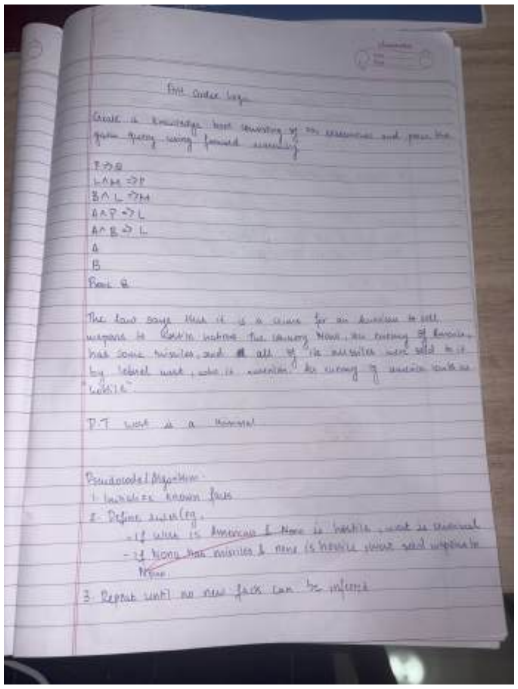
Output:



```
{'x': 'a', 'y': 'b'}
```

Program 8

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



First Order Logic

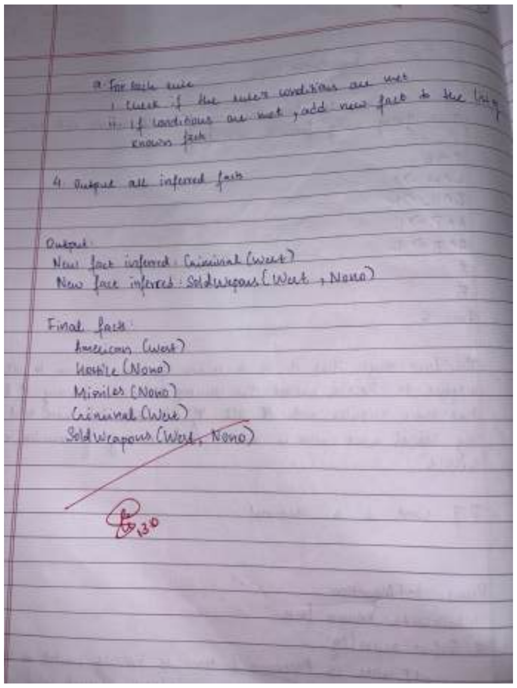
Create a knowledge base consisting of [illegible] statements and prove the given query using forward reasoning.

$P \Rightarrow Q$
$L \wedge M \Rightarrow P$
$B \wedge L \Rightarrow M$
$A \wedge P \Rightarrow L$
$A \wedge B \Rightarrow L$
A
B
Prove Q

The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American. An enemy of America counts as "hostile".

P.T West is a criminal

```
Pseudocode/Algorithm:
1. Initialize known facts
2. Define rules (eg.
      - If West is American & Nono is hostile, West is criminal
      - If Nono has missiles & Nono is hostile, West sold weapons to Nono.
3. Repeat until no new facts can be inferred
```



```
   a. For each rule
      i. check if the rule's conditions are met
      ii. If conditions are met, add new fact to the list of
          known facts
4. Output all inferred facts
```

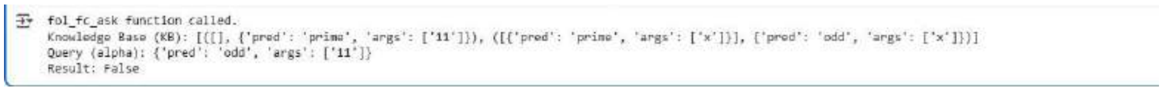
```
Output:
New fact inferred: Criminal(West)
New fact inferred: SoldWeapons(West, Nono)

Final facts:
    American(West)
    Hostile(Nono)
    Missiles(Nono)
    Criminal(West)
    SoldWeapons(West, Nono)
```

[illegible] 13/0

Code:

```
def is_variable(x):
    return isinstance(x, str) and x.islower()

def is_constant(x):
    return isinstance(x, str) and (x.isupper() or x.isdigit() or (x.isalpha() and not x.islower()))

def occurs_in(var, expr):
    if var == expr:
        return True
    if isinstance(expr, dict):
        return any(occurs_in(var, arg) for arg in expr.get('args', []))
    return False

def predicate_symbol(expr):
    if isinstance(expr, dict) and 'pred' in expr:
        return expr['pred']
    return None

def apply_substitution(subst, expr):
    if isinstance(expr, str):
        return subst.get(expr, expr)
    elif isinstance(expr, dict):
        return {
            'pred': expr['pred'],
            'args': [apply_substitution(subst, arg) for arg in expr.get('args', [])]
        }
    return expr

def compose_subst(s1, s2):
    result = {}
    for v, val in s1.items():
        result[v] = apply_substitution(s2, val)
    for v, val in s2.items():
        result[v] = val
    return result

def unify(x, y):
    if x == y:
        return {}
    if isinstance(x, str) and is_variable(x):
        if occurs_in(x, y):
            return "FAILURE"
        return {x: y}
    if isinstance(y, str) and is_variable(y):
        if occurs_in(y, x):
            return "FAILURE"
        return {y: x}
    if isinstance(x, str) and isinstance(y, str):
        return "FAILURE"
    if isinstance(x, dict) and isinstance(y, dict):
        if predicate_symbol(x) != predicate_symbol(y):
            return "FAILURE"
        if len(x.get('args', [])) != len(y.get('args', [])):
            return "FAILURE"
        SUBST = {}
        for a, b in zip(x['args'], y['args']):
            a_ap = apply_substitution(SUBST, a)
            b_ap = apply_substitution(SUBST, b)
            S = unify(a_ap, b_ap)
            if S == "FAILURE":
                return "FAILURE"
            if S:
                SUBST = compose_subst(SUBST, S)
        return SUBST
    return "FAILURE"

def sentence_to_str(sentence):
    if isinstance(sentence, str):
        return sentence
    elif isinstance(sentence, dict):
        args_str = ",".join(sentence_to_str(arg) for arg in sentence.get('args', []))
        return f"{sentence['pred']}({args_str})"
    return str(sentence)

def str_to_sentence(s):
    s = s.strip()
    pred_end = s.find("(")
    if pred_end == -1:
        return s
    pred = s[:pred_end]
    args_str = s[pred_end+1:-1]
    args = [a.strip() for a in args_str.split(",")] if args_str else []
    return {'pred': pred, 'args': args}

def find_substitutions_for_premises(premises, known_facts):
    results = []
    def backtrack(i, subst):
        if i == len(premises):
            results.append(subst.copy())
            return
        prem = apply_substitution(subst, premises[i])
        for fact in known_facts:
            S = unify(prem, fact)
            if S == "FAILURE":
                continue
            new_subst = compose_subst(subst, S)
            backtrack(i + 1, new_subst)
    backtrack(0, {})
    return results

def sentence_in_list(sentence, lst):
    s_str = sentence_to_str(sentence)
    return any(sentence_to_str(s) == s_str for s in lst)

def fol_fc_ask(KB, alpha):
    query = alpha
    known_facts = []
    agenda = []
    for premises, concl in KB:
        if not premises:
            fact = concl
            if not sentence_in_list(fact, known_facts):
                known_facts.append(fact)
                agenda.append(fact)
    while agenda:
        fact = agenda.pop(0)
        if unify(fact, query) != "FAILURE":
            return True
        for premises, concl in KB:
            subs = find_substitutions_for_premises(premises, known_facts)
            for s in subs:
                new_fact = apply_substitution(s, concl)
                if not sentence_in_list(new_fact, known_facts):
                    known_facts.append(new_fact)
                    agenda.append(new_fact)
    return False

KB = [
    ([], {'pred': 'prime', 'args': ['11']}),
    ([{'pred': 'prime', 'args': ['x']}], {'pred': 'odd', 'args': ['x']})
]

alpha = {'pred': 'odd', 'args': ['11']}
result = fol_fc_ask(KB, alpha)
print("Result:", result)
```

OUTPUT:

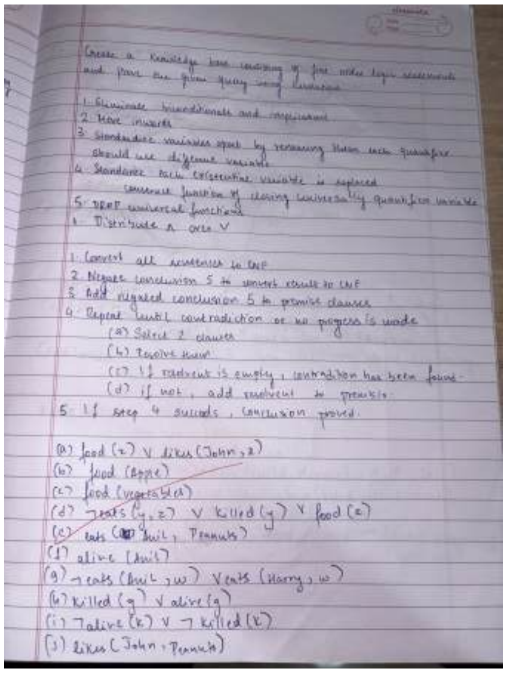


```
fol_fc_ask function called.
Knowledge Base (KB): [([], {'pred': 'prime', 'args': ['11']}), ([{'pred': 'prime', 'args': ['x']}], {'pred': 'odd', 'args': ['x']})]
Query (alpha): {'pred': 'odd', 'args': ['11']}
Result: False
```

Program 9

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

Algorithm:

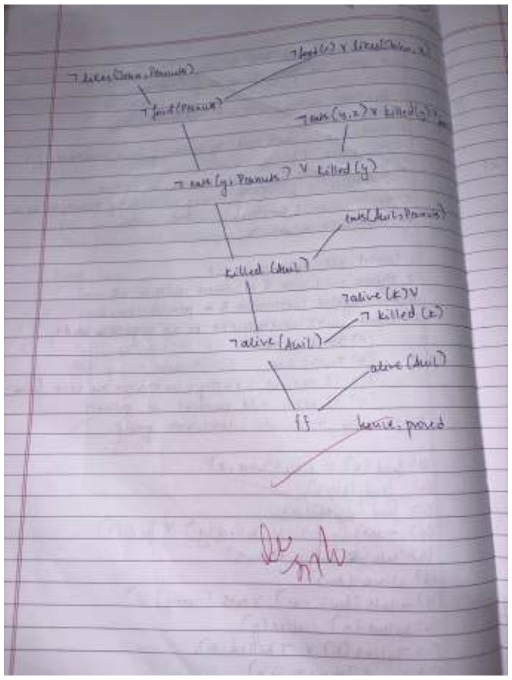


Create a knowledge base consisting of first order logic statements and prove the given query using Resolution

1. Eliminate biconditionals and implications
2. Move negations
3. Standardize variables apart by renaming them each quantifier should use different variable
4. Skolemize each existential variable is replaced skolem function of closing universally quantified variable
5. Drop universal quantifiers
1. Distribute ∧ over ∨

1. Convert all sentences to CNF
2. Negate conclusion S to convert result to CNF
3. Add negated conclusion S to premise clauses
4. Repeat until contradiction or no progress is made
   (a) Select 2 clauses
   (b) resolve them
   (c) If resolvent is empty, contradiction has been found.
   (d) if not, add resolvent to premises.
5. If step 4 succeeds, conclusion proved.

(a) food (x) ∨ likes (John, x)
(b) food (Apple)
(c) food (vegetables)
(d) ¬eats (y, z) ∨ killed (y) ∨ food (z)
(e) eats (Anil, Peanuts)
(f) alive (Anil)
(g) ¬eats (Anil, w) ∨ eats (Harry, w)
(h) killed (g) ∨ alive (g)
(i) ¬alive (k) ∨ ¬killed (k)
(j) likes (John, Peanuts)

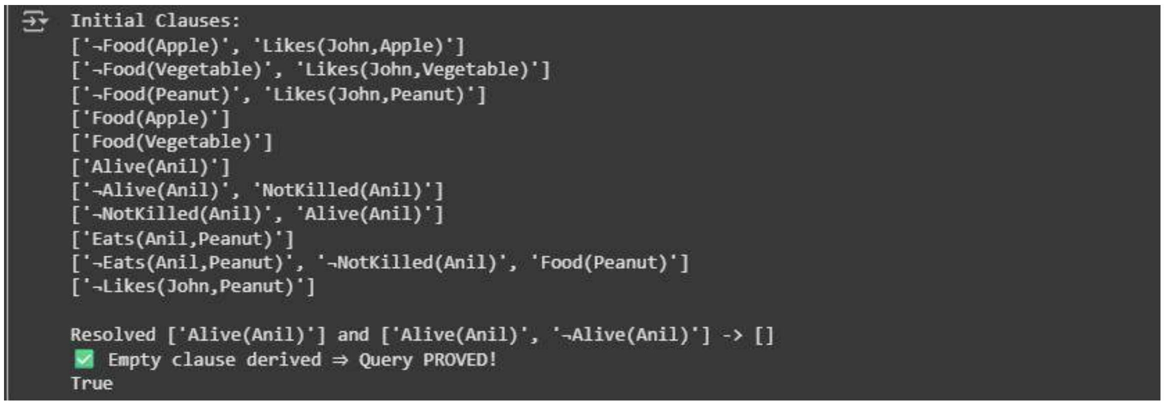


Code:

```
from copy import deepcopy

def resolve_pair(ci, cj):
    resolvents = []
    for lit_i in ci:
        for lit_j in cj:
            if lit_i.startswith('¬') and lit_i[1:] == lit_j:
                new_clause = list(set(ci + cj))
                new_clause.remove(lit_i)
                new_clause.remove(lit_j)
                resolvents.append(new_clause)
            elif lit_j.startswith('¬') and lit_j[1:] == lit_i:
                new_clause = list(set(ci + cj))
                new_clause.remove(lit_j)
                new_clause.remove(lit_i)
                resolvents.append(new_clause)
    return resolvents

def resolution(KB, query):
    clauses = deepcopy(KB)
    clauses.append(['¬' + query])
    print("=== INITIAL CLAUSES ===")
    for c in clauses:
        print(c)

    new = set()

    while True:
        n = len(clauses)
        pairs = [(clauses[i], clauses[j]) for i in range(n) for j in range(i + 1, n)]

        for (ci, cj) in pairs:
            resolvents = resolve_pair(ci, cj)
            for res in resolvents:
                if not res:
                    print(f"\nResolved {ci} and {cj} -> []")
                    print("✅ Empty clause derived ⇒ Query PROVED!")
                    return True
                new.add(tuple(sorted(res)))

        new_clauses = [list(x) for x in new if list(x) not in clauses]

        if not new_clauses:
            print("\nNo new clauses ⇒ Query cannot be proved.")
            return False

        for c in new_clauses:
            clauses.append(c)
            print("Added new clause:", c)

KB = [
    ['¬Food(Apple)', 'Likes(John,Apple)'],
    ['¬Food(Vegetable)', 'Likes(John,Vegetable)'],
    ['¬Food(Peanut)', 'Likes(John,Peanut)'],
    ['Food(Apple)'],
    ['Food(Vegetable)'],
    ['Alive(Anil)'],
    ['¬Alive(Anil)', 'NotKilled(Anil)'],
    ['¬NotKilled(Anil)', 'Alive(Anil)'],
    ['Eats(Anil,Peanut)'],
    ['¬Eats(Anil,Peanut)', '¬NotKilled(Anil)', 'Food(Peanut)']
]

query = 'Likes(John,Peanut)'

if __name__ == "__main__":
    print(f"Proving query: {query}\n")
    result = resolution(KB, query)

    print("\n=== RESULT ===")
    if result:
        print("✅ The query is PROVED using resolution.")
```

```
else:
    print("❌ The query CANNOT be proved from the KB.")
```

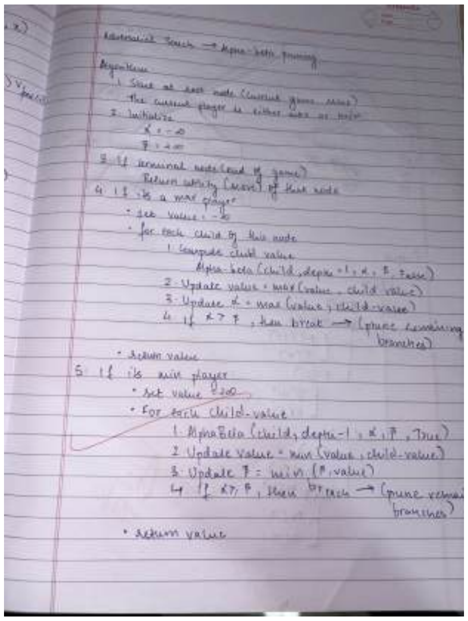
OUTPUT:



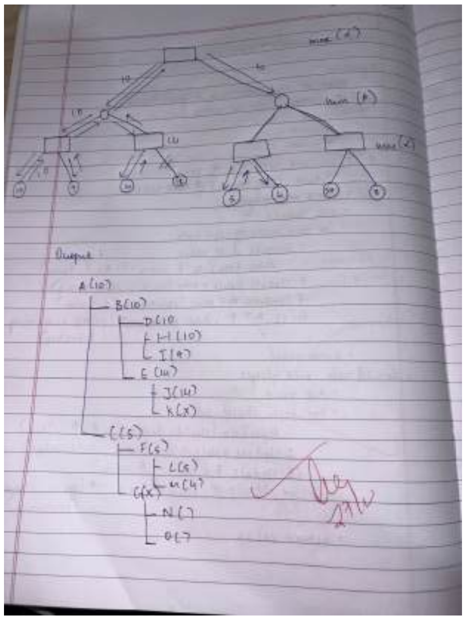
```
Initial Clauses:
['¬Food(Apple)', 'Likes(John,Apple)']
['¬Food(Vegetable)', 'Likes(John,Vegetable)']
['¬Food(Peanut)', 'Likes(John,Peanut)']
['Food(Apple)']
['Food(Vegetable)']
['Alive(Anil)']
['¬Alive(Anil)', 'NotKilled(Anil)']
['¬NotKilled(Anil)', 'Alive(Anil)']
['Eats(Anil,Peanut)']
['¬Eats(Anil,Peanut)', '¬NotKilled(Anil)', 'Food(Peanut)']
['¬Likes(John,Peanut)']

Resolved ['Alive(Anil)'] and ['Alive(Anil)', '¬Alive(Anil)'] -> []
✅ Empty clause derived ⇒ Query PROVED!
True
```

Program 10
Implement Alpha-Beta Pruning.

Algorithm:



```
Adversarial Search → Alpha-beta pruning
Algorithm
  1. Start at root node (current game state)
     The current player is either max or min
  2. Initialize
       α = -∞
       β = +∞
  3. If terminal node (end of game)
       Return utility (score) of that node
  4. If it's a max player
       • set value = -∞
       • for each child of this node
           1. Compute child value
                Alpha-beta (child, depth-1, α, β, False)
           2. Update value = max (value, child value)
           3. Update α = max (value, child-value)
           4. if α ≥ β, then break → (prune remaining
                                       branches)
     • return value
  5. If it's min player
       • set value = +∞
       • For each child-value
           1. AlphaBeta (child, depth-1, α, β, True)
           2. Update value = min (value, child-value)
           3. Update β = min (β, value)
           4. If α ≥ β, then break → (prune remaining
                                       branches)
       • return value
```



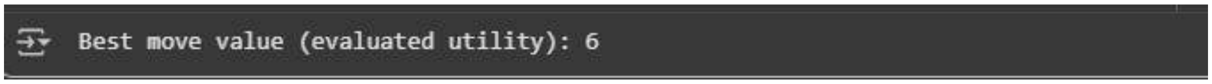
```
Output
A (10)
  ├─ B(10)
  │   ├─ D(10)
  │   │   ├ H (10)
  │   │   └ I (9)
  │   └─ E (14)
  │       ├ J(14)
  │       └ K(X)
  └─ C(5)
      ├─ F(5)
      │   ├ L(5)
      │   └ M(4)
      └ G(X)
          ├─ N()
          └─ O()
```

Code:

```python
from typing import Any, Dict, List, Optional, Tuple


State = Dict[str, Any]
Action = Dict[str, Any]


def alpha_beta_search(state: State) -> Tuple[float, Optional[Action]]:
    value, move = max_value(state, float('-inf'), float('inf'))
    return value, move


def max_value(state: State, alpha: float, beta: float) -> Tuple[float, Optional[Action]]:
    if terminal_test(state):
        return utility(state), None
    value = float('-inf')
    best_move: Optional[Action] = None
    for action in actions(state):
        v, _ = min_value(result(state, action), alpha, beta)
        if v > value:
            value = v
            best_move = action
        if value >= beta:
            return value, best_move
        alpha = max(alpha, value)
    return value, best_move

def min_value(state: State, alpha: float, beta: float) -> Tuple[float, Optional[Action]]:
    if terminal_test(state):
        return utility(state), None
    value = float('inf')
    best_move: Optional[Action] = None
    for action in actions(state):
        v, _ = max_value(result(state, action), alpha, beta)
        if v < value:
            value = v
            best_move = action
        if value <= alpha:
            return value, best_move
        beta = min(beta, value)
    return value, best_move

def actions(state: State) -> List[Action]:
    return state.get('actions', [])

def result(state: State, action: Action) -> State:
    return action['next']

def terminal_test(state: State) -> bool:
    return state.get('terminal', False)

def utility(state: State) -> float:
    return float(state.get('utility', 0))
```

```
if __name__ == "__main__":
    leaf1 = {'terminal': True, 'utility': 3}
    leaf2 = {'terminal': True, 'utility': 5}
    leaf3 = {'terminal': True, 'utility': 6}
    leaf4 = {'terminal': True, 'utility': 9}

    B = {'actions': [
        {'name': 'B->leaf1', 'next': leaf1},
        {'name': 'B->leaf2', 'next': leaf2}
    ]}

    C = {'actions': [
        {'name': 'C->leaf3', 'next': leaf3},
        {'name': 'C->leaf4', 'next': leaf4}
    ]}

    root = {'actions': [
        {'name': 'Root->B', 'next': B},
        {'name': 'Root->C', 'next': C}
    ]}

    best_value, best_move = alpha_beta_search(root)

    print("Best move value (evaluated utility):", best_value)
    if best_move is not None:
        print("Best move chosen:", best_move.get('name', best_move))
    else:
        print("No move (terminal state).")
```

OUTPUT:



```
Best move value (evaluated utility): 6
```