VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT

on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Himika Kakhani (1BM23CS112)

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

Sept-2025 to Jan-2026

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 **Himika Kakhani (1BM23CS112),** who is bonafied 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 Anusha S Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

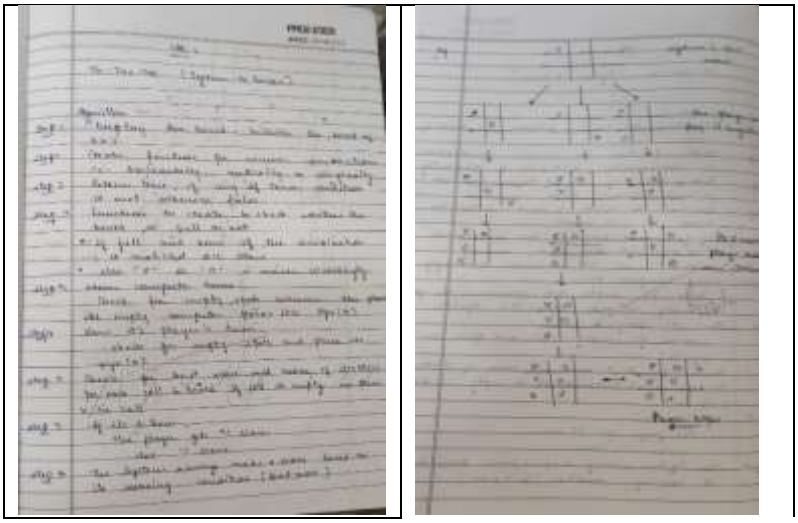
GitHub Link: https://github.com/himika03/AI-Lab

Github Link: https://github.com/Kashvi65/AI-LAB

Program 1

a)Implement Tic –Tac –Toe Game
b)Implement vacuum cleaner agent

Algorithm:



Code:

a)import random

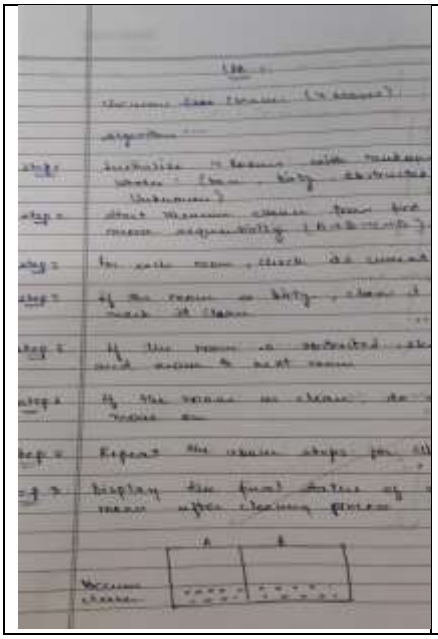
```
def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 9)

def check_winner(board, player):
    # Check rows, columns, and diagonals for a win
    for i in range(3):
        if all([cell == player for cell in board[i]]):
```

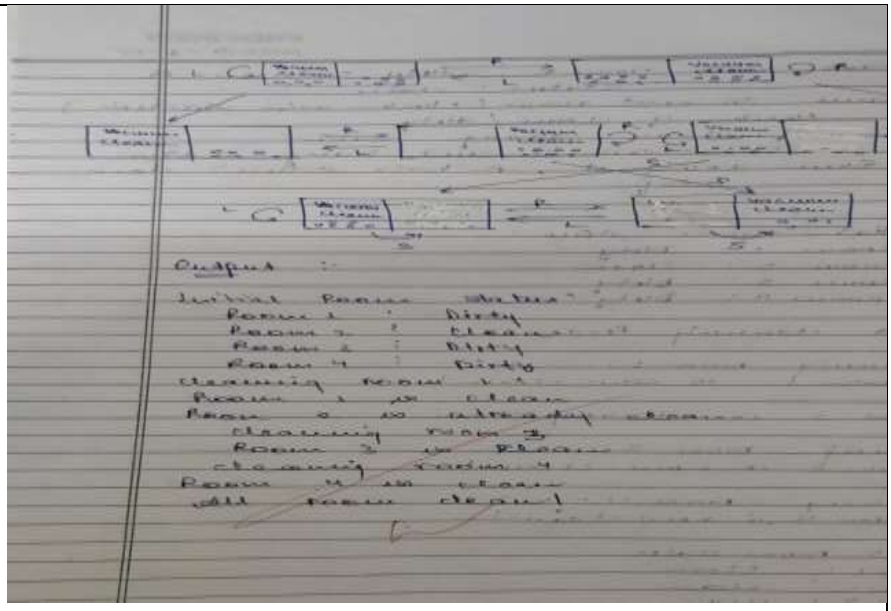
```
            return True
        if all([board[j][i] == player for j in range(3)]):
            return True
    if all([board[i][i] == player for i in range(3)]):
        return True
    if all([board[i][2 - i] == player for i in range(3)]):
        return True
    return False

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

def player_move(board):
    while True:
        try:
            move = int(input("Enter your move (1-9): "))
            if move < 1 or move > 9:
                print("Invalid input. Choose a number from 1 to 9.")
                continue
            row = (move - 1) // 3
            col = (move - 1) % 3
            if board[row][col] != " ":
                print("That spot is taken! Try again.")
            else:
                board[row][col] = "X"
                break
        except ValueError:
            print("Please enter a valid number.")

def computer_move(board):
    empty_cells = [(i, j) for i in range(3) for j in range(3) if board[i][j] == " "]
    if empty_cells:
        row, col = random.choice(empty_cells)
        board[row][col] = "O"

def tic_tac_toe():

    board = [[" " for _ in range(3)] for _ in range(3)]
    print("Welcome to Tic-Tac-Toe!")
    print_board(board)

    while True:
        player_move(board)
        print_board(board)
```

```
        if check_winner(board, "X"):
            print("You win! 🎉")
            break
        if is_board_full(board):
            print("It's a tie!")
            break

        print("Computer's turn:")
        computer_move(board)
        print_board(board)
        if check_winner(board, "O"):
            print("Computer wins! 😞")
            break
        if is_board_full(board):
            print("It's a tie!")
            break

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

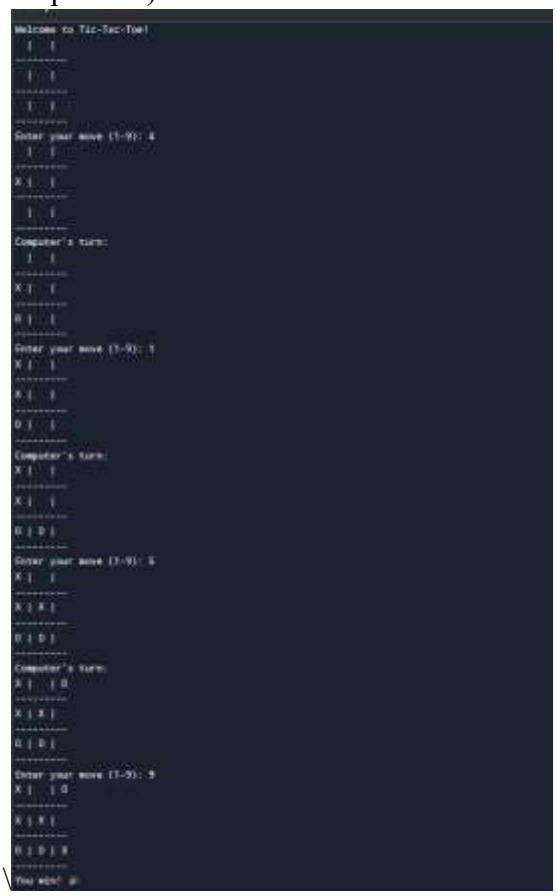
b)
```
def show_rooms_status(rooms):
    for room_number, status in rooms.items():
        print(f"Room {room_number}: {'Clean' if status
else 'Dirty'}")

def clean_room(rooms, room_number):
    if rooms[room_number]:
        print(f"Room {room_number} is already
clean.")
    else:
        print(f"Cleaning room {room_number}...")
        rooms[room_number] = True
        print(f"Room {room_number} is now clean!")

def clean_all_rooms(rooms):
    print("Initial room statuses:")
    show_rooms_status(rooms)
    print("\nStarting cleaning process...\n")
    for room_number in rooms:
        clean_room(rooms, room_number)
        print()
    print("Final room statuses:")
    show_rooms_status(rooms)
```

```
if __name__ == "__main__":
    rooms = {
        1: False,
        2: True,
        3: False,
        4: False
    }
    clean_all_rooms(rooms)
```

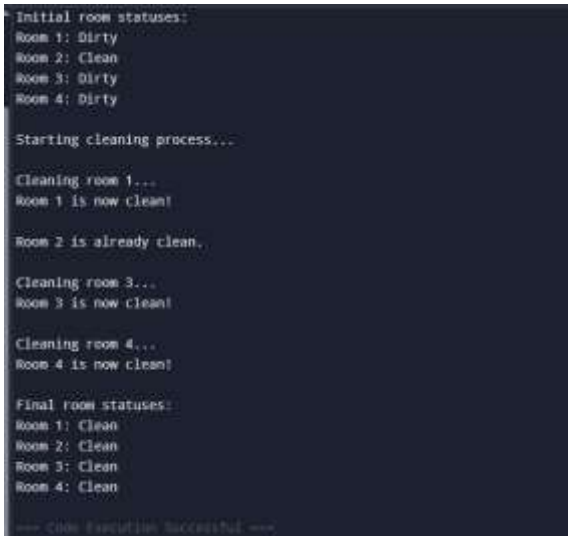




Output: a)


Welcome to Tic-Tac-Toe!
Enter your move (1-9): 4
X | |
Computer's turn:
Enter your move (1-9): 1
Computer's turn:
Enter your move (1-9): 5
Computer's turn:
Enter your move (1-9): 9
You won!

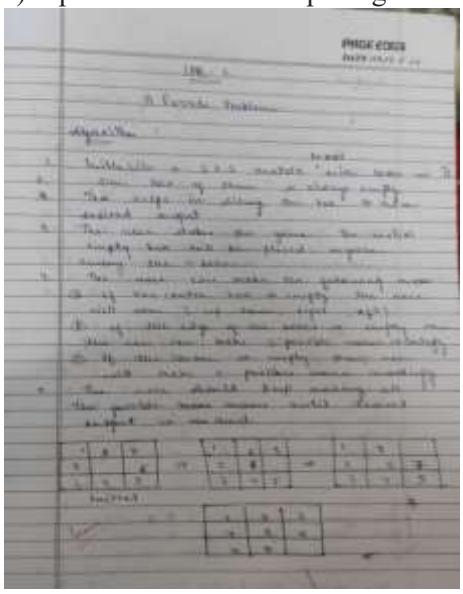
b)

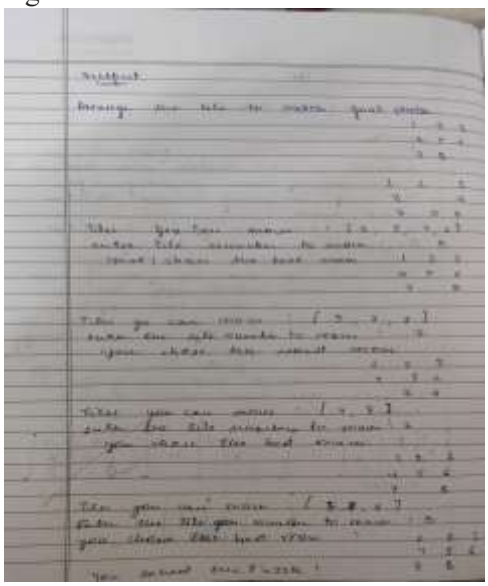

Initial room statuses:
Room 1: Dirty
Room 2: Clean
Room 3: Dirty
Room 4: Dirty
Starting cleaning process...
Cleaning room 1...
Room 1 is now clean!
Room 2 is already clean.
Cleaning room 3...
Room 3 is now clean!
Cleaning room 4...
Room 4 is now clean!
Final room statuses:
Room 1: Clean
Room 2: Clean
Room 3: Clean
Room 4: Clean

Program 2:

a)Implement 8 puzzle problems using Depth First Search (DFS)

b)Implement Iterative deepening search algorithm





Code:

a)

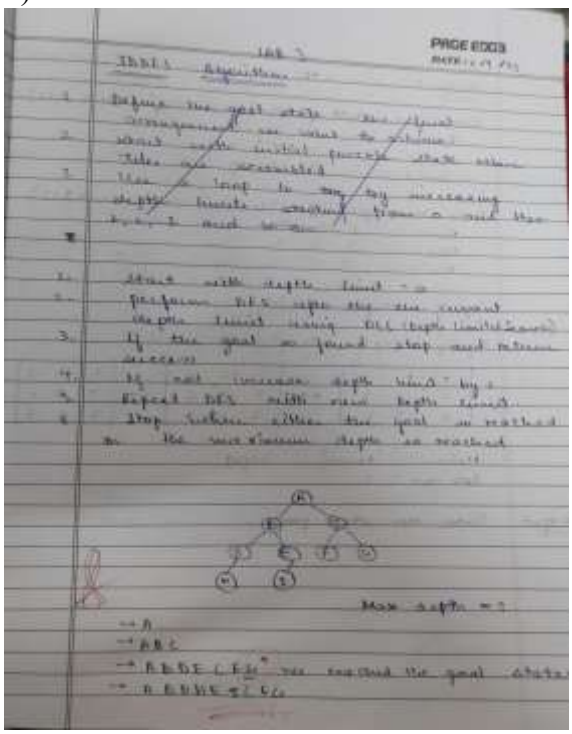
```
import copy

def print_board(board):
    for row in board:
        print(' '.join(str(x) if x != 0 else ' ' for x in row))
    print()

def find_zero(board):
    for i in range(3):
        for j in range(3):
            if board[i][j] == 0:
                return i, j

def is_solved(board):
    solved = [1,2,3,4,5,6,7,8,0]
    flat = [num for row in board for num in row]
    return flat == solved

def valid_moves(zero_pos):
```

```
    i, j = zero_pos
    moves = []
    if i > 0: moves.append((i-1, j))
    if i < 2: moves.append((i+1, j))
    if j > 0: moves.append((i, j-1))
    if j < 2: moves.append((i, j+1))
    return moves

def correct_tiles_count(board):
    """Count how many tiles are in their correct position."""
    count = 0
    goal = [1,2,3,4,5,6,7,8,0]
    flat = [num for row in board for num in row]
    for i in range(9):
        if flat[i] != 0 and flat[i] == goal[i]:
            count += 1
    return count

def get_user_move(board):
    zero_pos = find_zero(board)
    moves = valid_moves(zero_pos)
    movable_tiles = [board[i][j] for (i,j) in moves]

    print(f"Tiles you can move: {movable_tiles}")

    while True:
        try:
            move = int(input("Enter the tile number to move (or 0 to quit): "))
            if move == 0:
                return None
            if move in movable_tiles:
                return move
            else:
                print("Invalid tile. Please choose a tile adjacent to the empty space.")
        except ValueError:
            print("Please enter a valid number.")

def evaluate_move(board, tile):
    """Compare user move to all possible moves and tell if it's best/worst."""
    zero_pos = find_zero(board)
    moves = valid_moves(zero_pos)
    movable_tiles = [board[i][j] for (i,j) in moves]

    scores = {}
```

```
    for t in movable_tiles:
        temp_board = copy.deepcopy(board)
        make_move(temp_board, t)
        scores[t] = correct_tiles_count(temp_board)

    user_score = scores[tile]
    best_score = max(scores.values())
    worst_score = min(scores.values())

    if user_score == best_score and user_score == worst_score:
        # Only one move possible
        return "Your move is the only possible move."
    elif user_score == best_score:
        return "Great! You chose the best move."
    elif user_score == worst_score:
        return "Oops! You chose the worst move."
    else:
        return "Your move is neither the best nor the worst."

def make_move(board, tile):
    zero_i, zero_j = find_zero(board)
    for i, j in valid_moves((zero_i, zero_j)):
        if board[i][j] == tile:
            board[zero_i][zero_j], board[i][j] = board[i][j], board[zero_i][zero_j]
            return

def main():
    board = [
        [1, 2, 3],
        [4, 0, 6],
        [7, 5, 8]
    ]
    print("Welcome to the 8 Puzzle Game!")
    print("Arrange the tiles to match this goal state:")
    print("1 2 3\n4 5 6\n7 8  ")

    while True:
        print_board(board)
        if is_solved(board):
            print("Congratulations! You solved the puzzle!")
            break
        move = get_user_move(board)
        if move is None:
            print("Game exited. Goodbye!")
```

```
            break

        # Evaluate user move
        feedback = evaluate_move(board, move)
        print(feedback)

        make_move(board, move)

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

b)



```
import copy

def get_puzzle(name):
    print(f"\nEnter the {name} puzzle (3x3, use -1 for blank):")
    puzzle = []
    for i in range(3):
        row = list(map(int, input(f'Row {i+1} (space-separated 3 numbers):
").split()))
        puzzle.append(row)
    return puzzle

def move(temp, movement):
```

```
    for i in range(3):
        for j in range(3):
            if temp[i][j] == -1:
                if movement == "up" and i > 0:
                    temp[i][j], temp[i-1][j] = temp[i-1][j], temp[i][j]
                elif movement == "down" and i < 2:
                    temp[i][j], temp[i+1][j] = temp[i+1][j], temp[i][j]
                elif movement == "left" and j > 0:
                    temp[i][j], temp[i][j-1] = temp[i][j-1], temp[i][j]
                elif movement == "right" and j < 2:
                    temp[i][j], temp[i][j+1] = temp[i][j+1], temp[i][j]
                return temp
    return temp

def dls(puzzle, depth, limit, last_move, goal):
    if puzzle == goal:
        return True, [puzzle], []

    if depth >= limit:
        return False, [], []

    for move_dir, opposite in [("up","down"), ("left","right"), ("down","up"),
("right","left")]:
        if last_move == opposite:  # avoid direct backtracking
            continue
        temp = copy.deepcopy(puzzle)
        new_state = move(temp, move_dir)
        if new_state != puzzle:  # valid move
            found, path, moves = dls(new_state, depth+1, limit, move_dir, goal)
            if found:
                return True, [puzzle] + path, [move_dir] + moves
    return False, [], []

def ids(start, goal):
    for limit in range(1, 50):  # reasonable max depth
        print(f"\nTrying depth limit = {limit}")
        found, path, moves = dls(start, 0, limit, None, goal)
        if found:
            print("\Solution found!")
            for step in path:
                print(step)
            print("Moves:", moves)
            print("Path cost =", len(path)-1)
            return
```

```
    print(" Solution not found within depth limit.")

# ---------------- MAIN ----------------
start_puzzle = get_puzzle("start")
goal_puzzle = get_puzzle("goal")

print("\n~~~~~~~~~~~~~~ IDDFS ~~~~~~~~~~~~~~")
ids(start_puzzle, goal_puzzle)
```

Output: a)

Output

```
Trying depth limit: 0
Visited in this iteration: ['A']

Trying depth limit: 1
Visited in this iteration: ['A', 'B', 'C']

Trying depth limit: 2
Visited in this iteration: ['A', 'B', 'D', 'E', 'C', 'F', 'G']

Path to goal:
A
C
G
Solution found in 2 steps.

=== Code Execution Successful ===
```

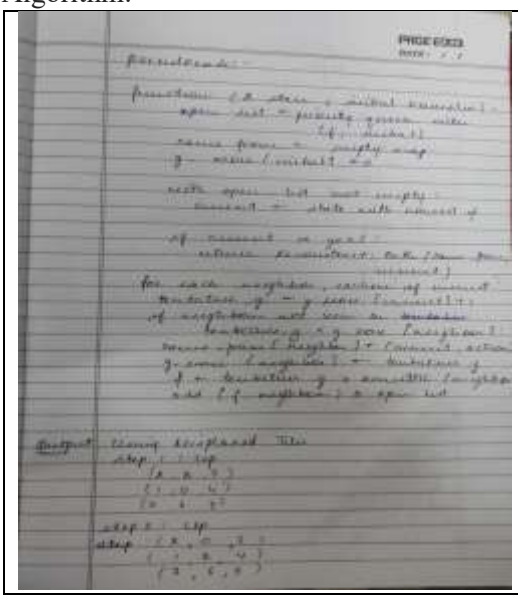
b)

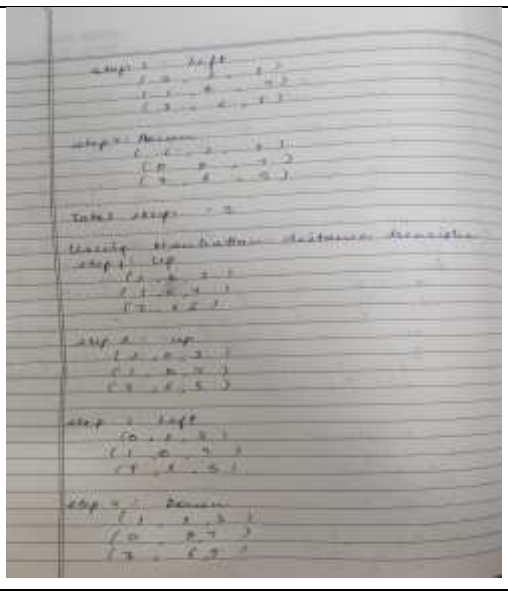
```
<main.py>:49: SyntaxWarning: invalid escape sequence '\S'

Enter the start puzzle (3x3, use -1 for blank):
Row 1 (space-separated 3 numbers): 1 -1 3
Row 2 (space-separated 3 numbers): 4 2 6
Row 3 (space-separated 3 numbers): 7 5 8

Enter the goal puzzle (3x3, use -1 for blank):
Row 1 (space-separated 3 numbers): 1 2 3
Row 2 (space-separated 3 numbers): 4 5 6
Row 3 (space-separated 3 numbers): 7 8 -1

---------- IDDFS ----------

Trying depth limit = 1

Trying depth limit = 2

Trying depth limit = 3
\Solution found!
[[1, -1, 3], [4, 2, 6], [7, 5, 8]]
[[1, 2, 3], [4, -1, 6], [7, 5, 8]]
[[1, 2, 3], [4, 5, 6], [7, -1, 8]]
[[1, 2, 3], [4, 5, 6], [7, 8, -1]]
Moves: ['down', 'down', 'right']
Path cost = 3

=== Code Execution Successful ===
```

Program 3:

Implement A* search algorithm

Algorithm:





Code:

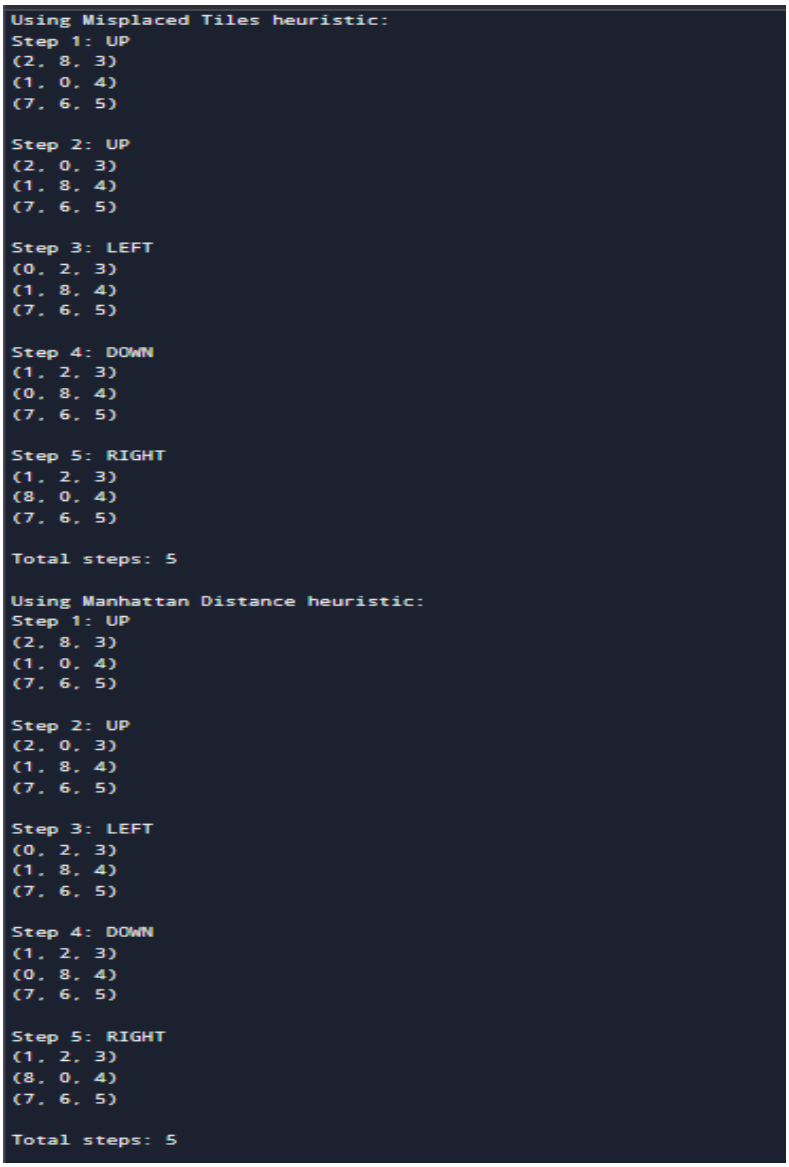
```
from heapq import heappush, heappop


goal_state = [
    [1, 2, 3],
    [8, 0, 4],
    [7, 6, 5]
```

```
]


directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
direction_names = ["UP", "DOWN", "LEFT", "RIGHT"]

def misplaced_tiles(state):
    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]:
                count += 1
    return count

def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            tile = state[i][j]
            if tile != 0:
                goal_x, goal_y = divmod(tile - 1, 3)
                distance += abs(i - goal_x) + abs(j - goal_y)
    return distance

def get_neighbors_with_actions(state):
    neighbors = []
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                x, y = i, j
                break

    for (dx, dy), action in zip(directions, direction_names):
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = [list(row) for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append((new_state, action))
    return neighbors

def state_to_tuple(state):
    return tuple(tuple(row) for row in state)

def reconstruct_path(came_from, current):
```

```
    actions = []
    states = []
    while current in came_from:
        prev_state, action = came_from[current]
        actions.append(action)
        states.append(current)
        current = prev_state
    states.append(current)
    actions.reverse()
    states.reverse()
    return actions, states

def a_star_search_with_steps(initial_state, heuristic_func):
    open_list = []
    closed_set = set()

    g_score = {state_to_tuple(initial_state): 0}
    f_score = {state_to_tuple(initial_state): heuristic_func(initial_state)}
    came_from = {}

    heappush(open_list, (f_score[state_to_tuple(initial_state)], initial_state))

    while open_list:
        _, current_state = heappop(open_list)
        current_t = state_to_tuple(current_state)

        if current_state == goal_state:
            return reconstruct_path(came_from, current_t)

        closed_set.add(current_t)

        for neighbor, action in get_neighbors_with_actions(current_state):
            neighbor_t = state_to_tuple(neighbor)
            if neighbor_t in closed_set:
                continue

            tentative_g = g_score[current_t] + 1
            if neighbor_t not in g_score or tentative_g < g_score[neighbor_t]:
                came_from[neighbor_t] = (current_t, action)
                g_score[neighbor_t] = tentative_g
                f_score[neighbor_t] = tentative_g + heuristic_func(neighbor)
                heappush(open_list, (f_score[neighbor_t], neighbor))

    return None, None
```

```
def print_path(actions, states):
    for i, (action, state) in enumerate(zip(actions, states[1:]), 1):
        print(f"Step {i}: {action}")
        for row in state:
            print(row)
        print()


initial_state = [
    [2, 8, 3],
    [1, 6, 4],
    [7, 0, 5]
]


print("Using Misplaced Tiles heuristic:")
actions, states = a_star_search_with_steps(initial_state, misplaced_tiles)
if actions:
    print_path(actions, states)
    print("Total steps:", len(actions))
else:
    print("No solution found.")


print("\nUsing Manhattan Distance heuristic:")
actions, states = a_star_search_with_steps(initial_state, manhattan_distance)
if actions:
    print_path(actions, states)
    print("Total steps:", len(actions))
else:
    print("No solution found.")
```

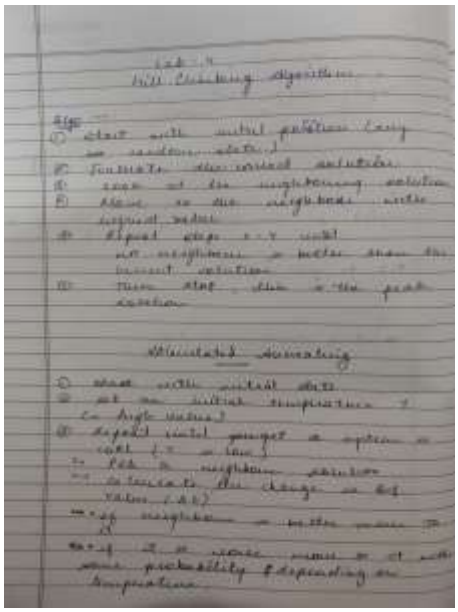
Output:

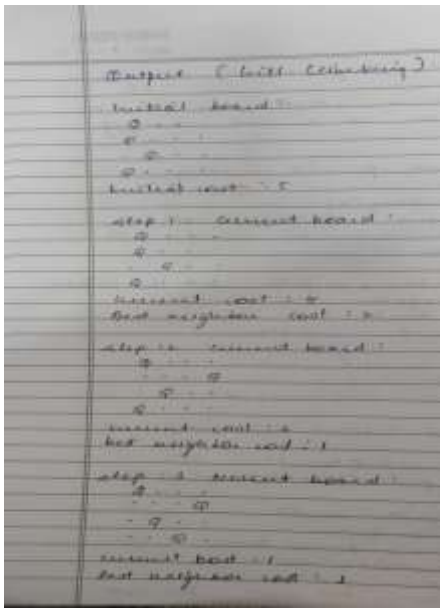


Program 4:

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:





Code:

```
import random

def print_board(board):
    n = len(board)
    for i in range(n):
        row = ["Q" if board[i] == j else "." for j in range(n)]
        print(" ".join(row))
    print()

def calculate_cost(board):
    """Heuristic: number of pairs of queens attacking each other"""
    n = len(board)
    cost = 0
    for i in range(n):
        for j in range(i + 1, n):
            if board[i] == board[j] or abs(board[i] - board[j]) == abs(i - j):
                cost += 1
    return cost

def get_best_neighbor(board):
    n = len(board)
    best_board = list(board)
    best_cost = calculate_cost(board)
```

```
    for row in range(n):
        for col in range(n):
            if board[row] != col:
                neighbor = list(board)
                neighbor[row] = col
                cost = calculate_cost(neighbor)
                if cost < best_cost:
                    best_cost = cost
                    best_board = neighbor
    return best_board, best_cost

def hill_climbing(n):
    current_board = [random.randint(0, n - 1) for _ in range(n)]
    current_cost = calculate_cost(current_board)

    print("Initial Board:")
    print_board(current_board)
    print(f"Initial Cost: {current_cost}\n")

    step = 1
    while True:
        neighbor, neighbor_cost = get_best_neighbor(current_board)
        print(f"Step {step}:")
        print("Current Board:")
        print_board(current_board)
        print(f"Current Cost: {current_cost}")
        print(f"Best Neighbor Cost: {neighbor_cost}\n")

        if neighbor_cost >= current_cost:
            break

        current_board = neighbor
        current_cost = neighbor_cost
        step += 1

    print("Final Board:")
    print_board(current_board)
    print(f"Final Cost: {current_cost}")

    if current_cost == 0:
        print("Goal State Reached!")
    else:
        print("Stuck in Local Minimum!")

# Run for 4-Queens
hill_climbing(4)
```

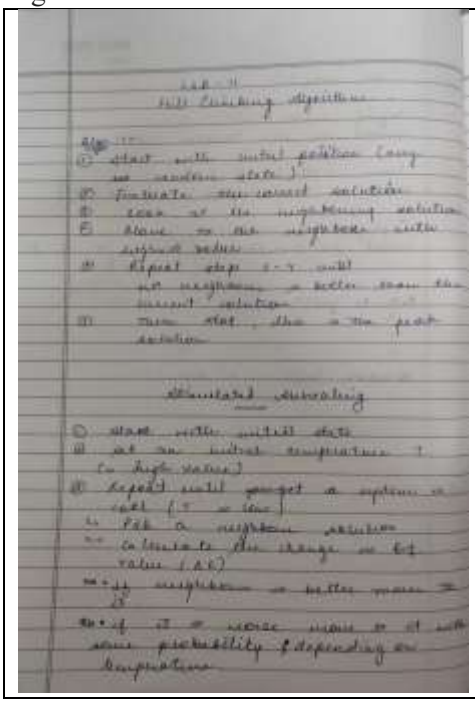
Output:

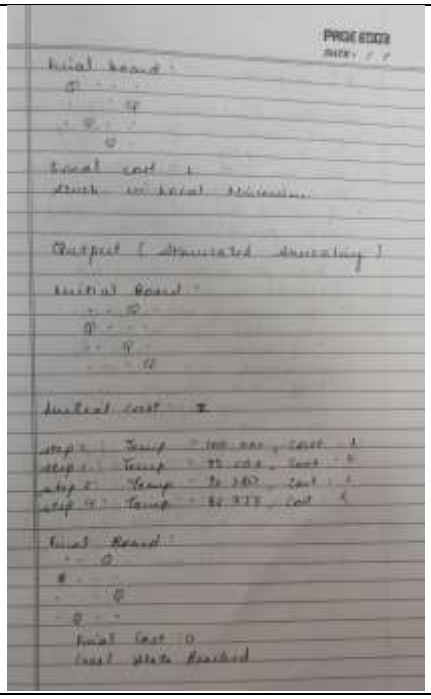
```
Initial Board:
Q . . .
Q . . .
. Q . .
Q . . .

Initial Cost: 5

Step 1:
Current Board:
Q . . .
Q . . .
. Q . .
Q . . .

Current Cost: 5
Best Neighbor Cost: 2

Step 2:
Current Board:
Q . . .
. . . Q
. Q . .
Q . . .

Current Cost: 2
Best Neighbor Cost: 1

Step 3:
Current Board:
Q . . .
. . . Q
. Q . .
. . Q .

Current Cost: 1
Best Neighbor Cost: 1

Final Board:
Q . . .
. . . Q
. Q . .
. . Q .

Final Cost: 1
Stuck in Local Minimum!
```

Program 5:

Simulated Annealing to Solve 8-Queens problem

Algorithm:





Code:

```
import random
import math

def print_board(board):
    n = len(board)
    for i in range(n):
        row = ["Q" if board[i] == j else "." for j in range(n)]
        print(" ".join(row))
    print()

def calculate_cost(board):
    """Heuristic: number of pairs of queens attacking each other"""
    n = len(board)
    cost = 0
    for i in range(n):
        for j in range(i + 1, n):
            if board[i] == board[j] or abs(board[i] - board[j]) == abs(i - j):
                cost += 1
    return cost

def random_neighbor(board):
```

```
    """Generate a random neighboring board by moving one queen"""
    n = len(board)
    neighbor = list(board)
    row = random.randint(0, n - 1)
    col = random.randint(0, n - 1)
    neighbor[row] = col
    return neighbor

def simulated_annealing(n, initial_temp=100, cooling_rate=0.95, stopping_temp=1):
    current_board = [random.randint(0, n - 1) for _ in range(n)]
    current_cost = calculate_cost(current_board)
    temperature = initial_temp
    step = 1

    print("Initial Board:")
    print_board(current_board)
    print(f"Initial Cost: {current_cost}\n")

    while temperature > stopping_temp and current_cost > 0:
        neighbor = random_neighbor(current_board)
        neighbor_cost = calculate_cost(neighbor)
        delta = neighbor_cost - current_cost

        # Acceptance probability
        if delta < 0 or random.random() < math.exp(-delta / temperature):
            current_board = neighbor
            current_cost = neighbor_cost

        print(f"Step {step}: Temp={temperature:.3f}, Cost={current_cost}")
        step += 1
        temperature *= cooling_rate

    print("\nFinal Board:")
    print_board(current_board)
    print(f"Final Cost: {current_cost}")

    if current_cost == 0:
        print("Goal State Reached!")
    else:
        print("Terminated before reaching goal.")

# Run for 8-Queens
simulated_annealing(4)
```

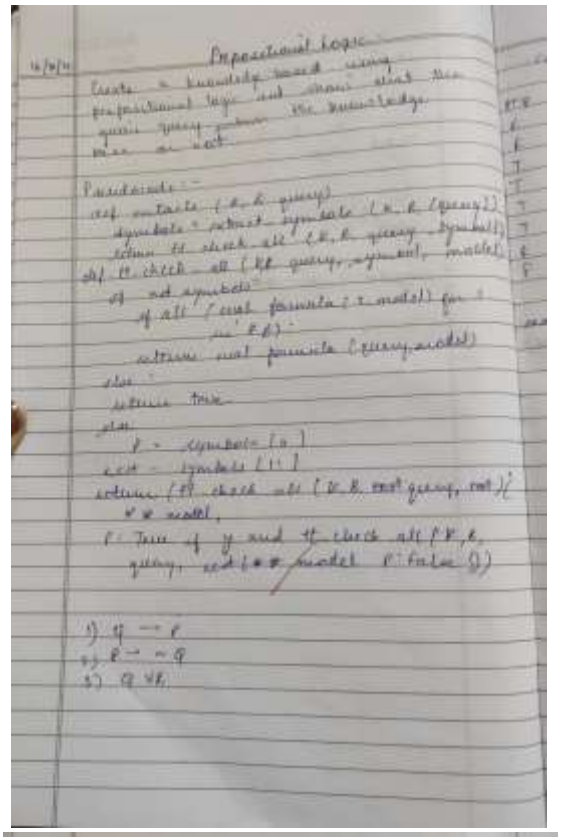
Output:

```
Initial Board:
. . Q .
Q . . .
. . Q .
. . . Q

Initial Cost: 2

Step 1: Temp=100.000, Cost=2
Step 2: Temp=95.000, Cost=2
Step 3: Temp=90.250, Cost=1
Step 4: Temp=85.737, Cost=0

Final Board:
. . Q .
Q . . .
. . . Q
. Q . .

Final Cost: 0
Goal State Reached!

=== Code Execution Successful ===
```

Program 6:

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

Algorithm:



Code:

```
from itertools import product

def extract_symbols(expr):
```

```
    symbols = set()
for c in expr:
if c.isalpha():
            symbols.add(c)
    return sorted(symbols)

def replace_implications(expr):
# If no implication, return as is
if '=>' not in expr:
        return expr
    # Split on first occurrence of =>
left, right = expr.split('=>', 1)     left =
left.strip()     right = right.strip()
    # Replace implication by ((not (left)) or (right))
new_expr = f"((not ({left})) or ({right}))"     return
new_expr

def eval_expr(expr, model):     for
sym, val in model.items():         expr =
expr.replace(sym, str(val))     expr =
expr.replace('~', ' not ')     expr =
expr.replace('&', ' and ')
    expr = expr.replace('|', ' or ')

    # Replace implication once
    expr = replace_implications(expr)

    expr = expr.replace('<=>', '==')

    return eval(expr)

def truth_table(KB_sentences):
    symbols = set()     for s in
KB_sentences:         symbols |=
set(extract_symbols(s))
    symbols = sorted(symbols)

    print("Truth Table:")     header =
symbols + KB_sentences     print(" |
".join(f"{h:8}" for h in header))
    print("-" * (10 * (len(header))))

    models_where_KB_true = []     for values in
product([True, False], repeat=len(symbols)):
        model = dict(zip(symbols, values))
        KB_vals = [str(eval_expr(sentence, model)) for sentence in KB_sentences]
row_vals = [str(model[s]) for s in symbols]         print(" | ".join(f"{v:8}" for v in
(row_vals + KB_vals)))         if all(val == 'True' for val in KB_vals):
```

```
models_where_KB_true.append(model)     print("\nModels where KB is true:")
for m in models_where_KB_true:
        print(m)
    return models_where_KB_true

def tt_entails(KB_sentences, query):
    symbols = set()     for s in
KB_sentences + [query]:         symbols
|= set(extract_symbols(s))
    symbols = sorted(symbols)

    for values in product([True, False], repeat=len(symbols)):
        model = dict(zip(symbols, values))
        if all(eval_expr(sentence, model) for sentence in KB_sentences):
if not eval_expr(query, model):
                return False
return True

# Your KB and queries
KB = [
    "Q => P",
    "P => ~Q",
    "Q | R"
]

queries = [
"R",
    "R => P",
    "Q => R"
]

models = truth_table(KB)

for q in queries:
    print(f"\nDoes KB entail '{q}'? {tt_entails(KB, q)}")
```

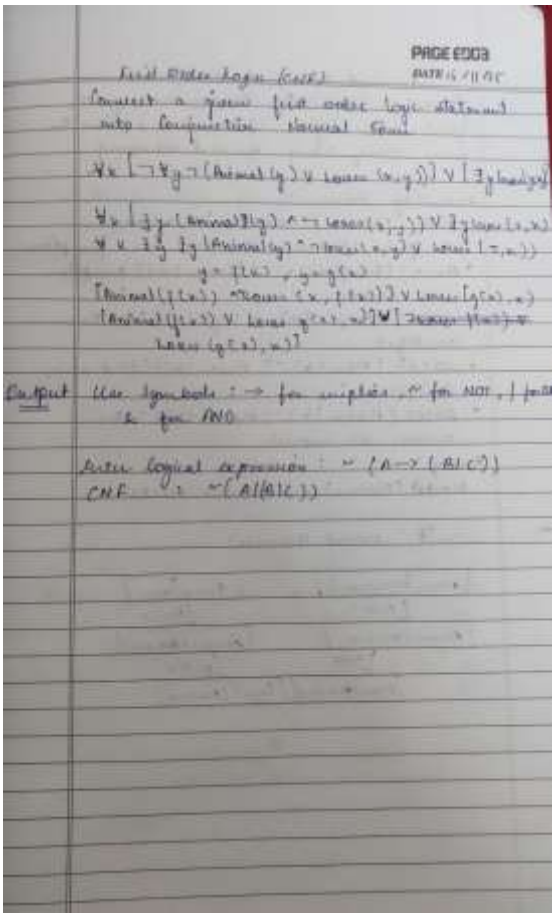
Output

```
------------------ RESTART: C:/Users/student/Desktop/ai_week6.py ---------------
Truth Table:
P        | Q        | R        | Q => P   | P => ~Q  | Q | R
-----------------------------------------------------------
True     | True     | True     | True     | False    | True
True     | True     | False    | True     | False    | True
True     | False    | True     | True     | True     | True
True     | False    | False    | True     | True     | False
False    | True     | True     | False    | True     | True
False    | True     | False    | False    | True     | True
False    | False    | True     | True     | True     | True
False    | False    | False    | True     | True     | False

Models where KB is true:
{'P': True, 'Q': False, 'R': True}
{'P': False, 'Q': False, 'R': True}

Does KB entail 'R'? True

Does KB entail 'R => P'? False

Does KB entail 'Q => R'? True
```

Program 7:

Implement unification in first order logic

Algorithm:



Code: class Var: def __init__(self, name):
    self.name = name

  def __repr__(self):
return self.name

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

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

class Const: def __init__(self, name):
    self.name = name

```
    def __repr__(self):
return self.name

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

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

class Func:     def __init__(self,
name, args):          self.name =
name
        self.args = args  # list of terms

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

    def __eq__(self, other):
        return (isinstance(other, Func) and self.name == other.name and
                len(self.args) == len(other.args) and
                all(x == y for x, y in zip(self.args, other.args)))

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

def is_variable(x):
    return isinstance(x, Var)

def is_compound(x):
    return isinstance(x, Func)

def occurs_check(var, term, subst):     if
var == term:          return True     elif
is_variable(term) and term in subst:
        return occurs_check(var, subst[term], subst)
elif is_compound(term):
        return any(occurs_check(var, arg, subst) for arg in term.args)
else:
        return False

def unify(x, y, subst={}, depth=0):
    indent = "  " * depth  # for pretty printing nested calls
if subst is None:          return None     elif x == y:
        # Terms are already identical
        print(f"{indent}Unify {x} and {y}: terms are identical, no substitution needed")
return subst     elif is_variable(x):
        return unify_var(x, y, subst, depth)
elif is_variable(y):
```

```
        return unify_var(y, x, subst, depth)     elif
is_compound(x) and is_compound(y):         if x.name
!= y.name or len(x.args) != len(y.args):
            print(f"{indent}Cannot unify different function symbols or different arity: {x} and {y}")
return None         for x_arg, y_arg in zip(x.args, y.args):
            subst = unify(x_arg, y_arg, subst, depth + 1)
if subst is None:                 return None         return
subst     else:
        print(f"{indent}Failed to unify {x} and {y}")
return None

def unify_var(var, x, subst, depth):
    indent = "  " * depth
if var in subst:
        print(f"{indent}Variable {var} already substituted with {subst[var]}, try to unify
{subst[var]} and {x}")         return
unify(subst[var], x, subst, depth + 1)     elif
is_variable(x) and x in subst:
        print(f"{indent}Variable {x} already substituted with {subst[x]}, try to unify {var} and
{subst[x]}")         return unify(var, subst[x],
subst, depth + 1)     elif occurs_check(var, x,
subst):
        print(f"{indent}Occurs check failed: {var} occurs in {x}")
return None     else:
        print(f"{indent}Substitute {var} with {x}")
subst_copy = subst.copy()         subst_copy[var] = x
print(f"{indent}Current substitution: {subst_copy}")
return subst_copy


if __name__ == "__main__":
    # Define terms:
    x = Var('x')     y = Var('y')
f1 = Func('f', [x, Const('a')])
    f2 = Func('f', [Const('b'), y])

    print("Start statement:")
print(f"  Term 1: {f1}")     print(f"
Term 2: {f2}")
    print("\nUnification steps:")

    result = unify(f1, f2, {})

    print("\nEnd state:")
if result is None:
        print("  Unification failed.")
    else:
```

```
    print("  Final substitution:")
for var, val in result.items():
print(f"   {var} -> {val}")
```

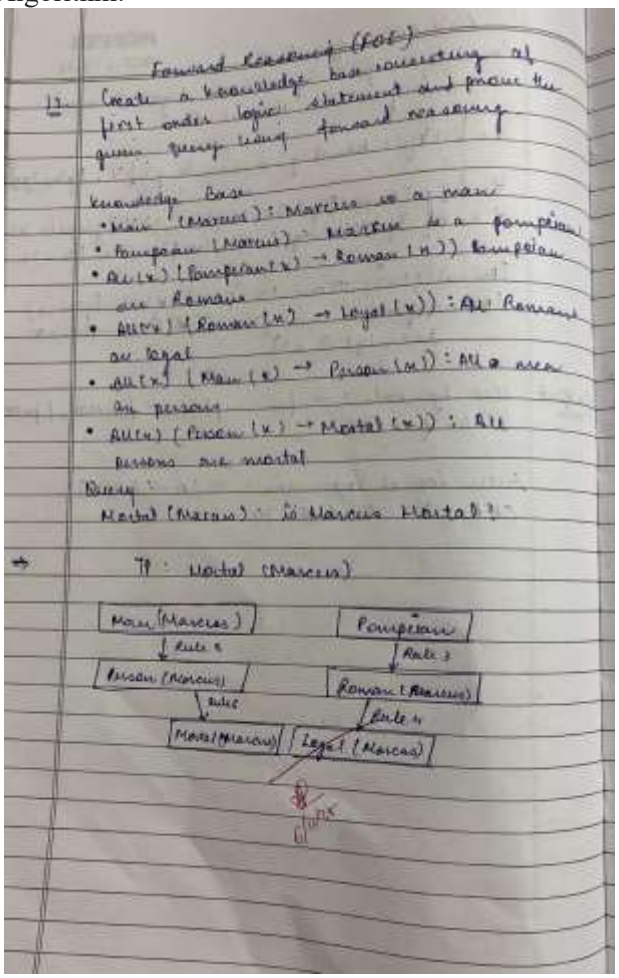
Output:

```
Start statement:
  Term 1: f(x, a)
  Term 2: f(b, y)

Unification steps:
  Substitute x with b
  Current substitution: {x: b}
  Substitute y with a
  Current substitution: {x: b, y: a}

End state:
  Final substitution:
    x -> b
    y -> a
```

Program 8:

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

Algorithm:



Code:

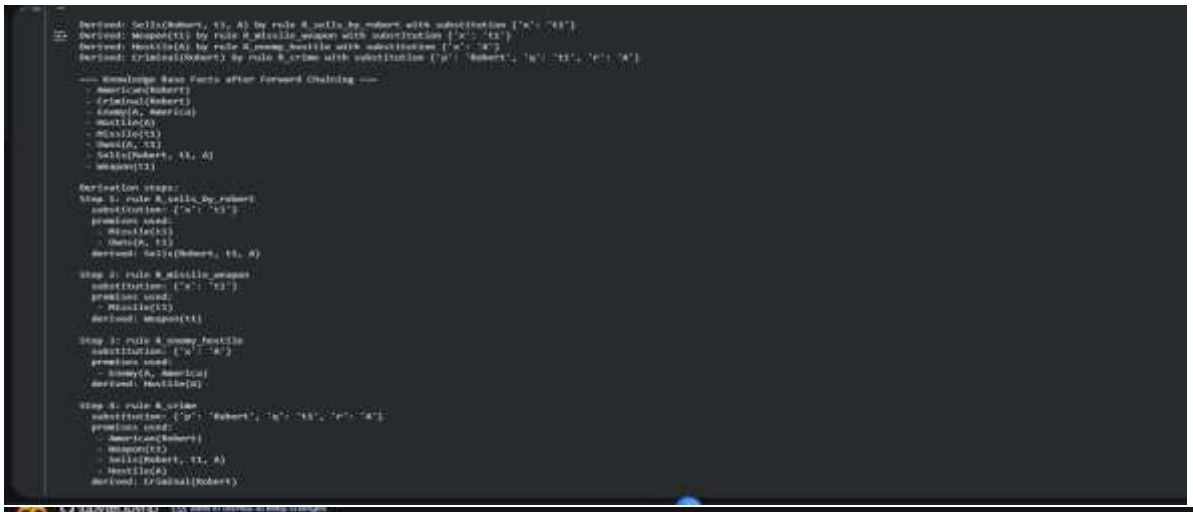
```
class Predicate:     def
__init__(self, name, args):
        self.name = name          self.args = args  # list
of constants or variables

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

    def __eq__(self, other):
        return isinstance(other, Predicate) and self.name == other.name and self.args == other.args
```

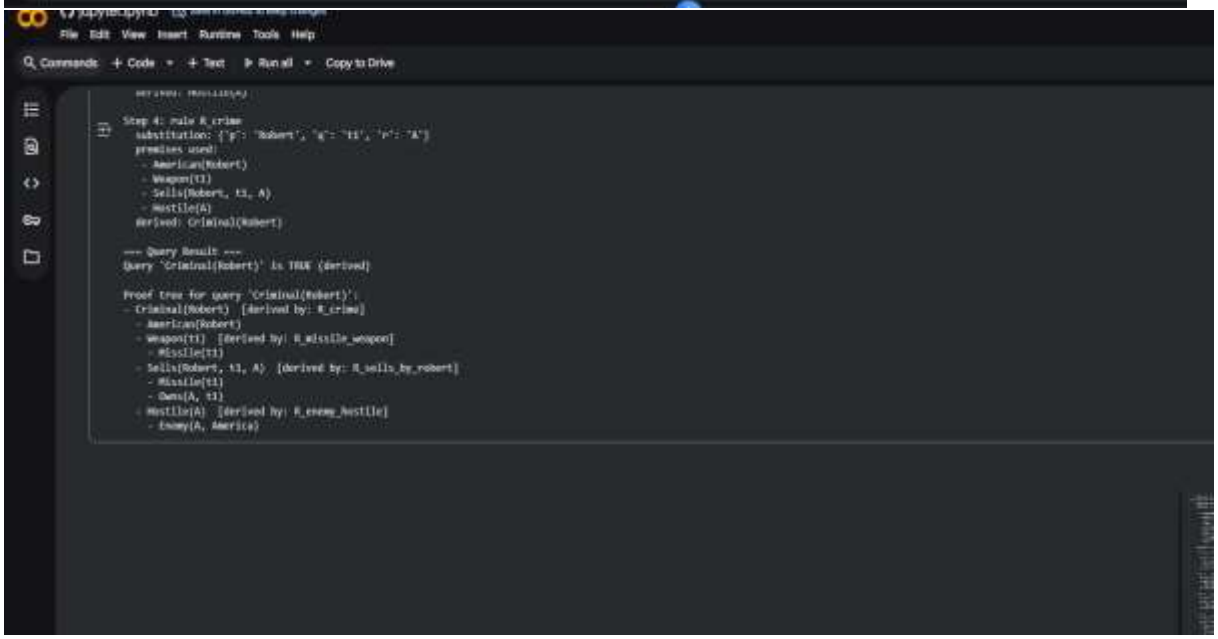
```
    def __hash__(self):
        return hash((self.name, tuple(self.args)))

class Var:     def
__init__(self, name):
        self.name = name

    def __repr__(self):
return self.name

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

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

class Const:     def
__init__(self, name):
        self.name = name

    def __repr__(self):
return self.name

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

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

def is_variable(x):
    return isinstance(x, Var)

def unify(x, y, subst={}):
if subst is None:
return None     elif x ==
y:         return subst
elif is_variable(x): return
unify_var(x, y, subst)
    elif is_variable(y):
        return unify_var(y, x, subst)     elif isinstance(x,
Predicate) and isinstance(y, Predicate):         if x.name !=
y.name or len(x.args) != len(y.args):
            return None         for a, b
in zip(x.args, y.args):
subst = unify(a, b, subst)
if subst is None:                 return
None         return subst     else:
        return None
```

```
def unify_var(var, x, subst):
if var in subst:
        return unify(subst[var], x, subst)
elif is_variable(x) and x in subst:
return unify(var, subst[x], subst)      elif
occurs_check(var, x, subst):
        return None
else:
        subst_copy = subst.copy()
subst_copy[var] = x
        return subst_copy

def occurs_check(var, x, subst):
if var == x:          return True      elif
is_variable(x) and x in subst:
        return occurs_check(var, subst[x], subst)
elif isinstance(x, Predicate):
        return any(occurs_check(var, arg, subst) for arg in x.args)
else:
        return False

def substitute(predicate, subst):
    new_args = []      for arg
in predicate.args:
        val = arg          while is_variable(val)
and val in subst:
            val = subst[val]
new_args.append(val)
    return Predicate(predicate.name, new_args)

class Rule:      def __init__(self, premises,
conclusion):          self.premises = premises  #
list of Predicate
        self.conclusion = conclusion  # Predicate

    def __repr__(self):
        return f"{' ∧ '.join(map(str, self.premises))} => {self.conclusion}" def
forward_chain(kb_facts, kb_rules, query):
    inferred            =             set(kb_facts)
print("Initial Facts:")    for f in inferred:
print(f"     {f}")            print("\nStarting
inference steps:\n")

    new_inferred = True

    while new_inferred:
new_inferred = False          for
rule in kb_rules:
```

```
            possible_substs = [{}]  # substitutions for premises

            for premise in rule.premises:
                temp_substs = []                for
fact in inferred:                    for subst in
possible_substs:
                        subst_try = unify(premise, fact, subst)
if subst_try is not None:
temp_substs.append(subst_try)
possible_substs = temp_substs

            for subst in possible_substs:
                concluded_fact = substitute(rule.conclusion, subst)                if concluded_fact
not in inferred:                    print(f"Inferred: {concluded_fact} from rule {rule} using
substitution {subst}")                    inferred.add(concluded_fact)                    new_inferred
= True                    if unify(concluded_fact, query) is not None:
                        print(f"\nQuery {query} proved!")
return True

    print(f"\nQuery {query} not proved.")
    return False

if __name__ == "__main__":
    a = Const('a')
b = Const('b')
c = Const('c')
x = Var('x')     y
= Var('y')
    z = Var('z')

    kb_facts = {
        Predicate('Parent', [a, b]),
        Predicate('Parent', [b, c]),
    }

    kb_rules = [
        Rule([Predicate('Parent', [x, y])], Predicate('Ancestor', [x, y])),
        Rule([Predicate('Parent', [x, y]), Predicate('Ancestor', [y, z])], Predicate('Ancestor', [x, z])),
    ]

    query = Predicate('Ancestor', [a, c])

    print("Running forward chaining...\n")
    forward_chain(kb_facts, kb_rules, query)
```

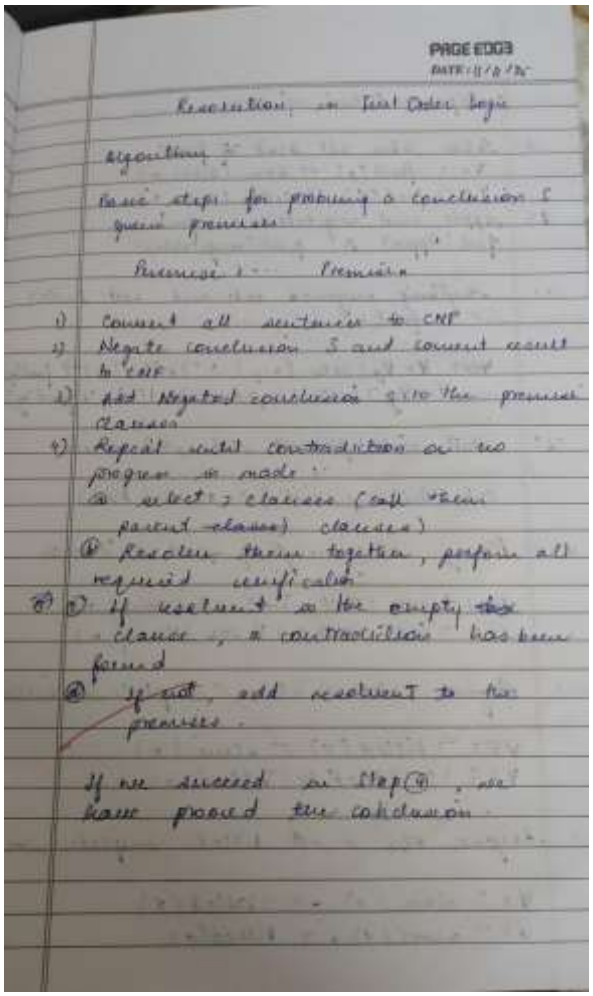
Output:





Program 9:

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



PAGE EDG3
DATE : 11/8/24

Resolution in First Order logic

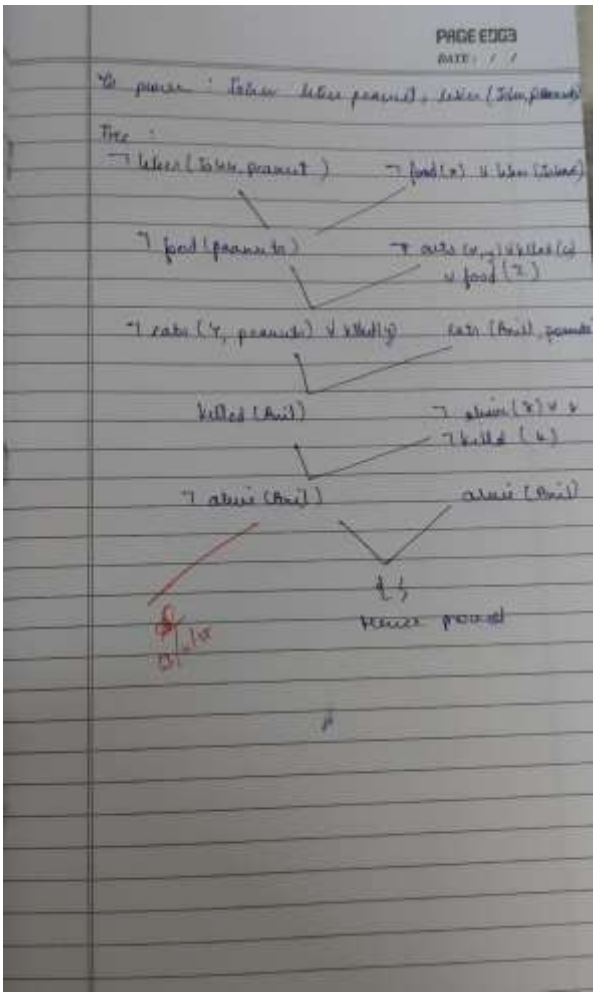
algorithm :

Basic steps for proving a conclusion S given premises

Premise 1 : Premise n

1) Convert all sentences to CNF
2) Negate conclusion S and convert result to CNF
3) Add Negated conclusion cells to the premise clauses
4) Repeat until contradiction or no progress is made :
   a) select 2 clauses (call them parent clauses)
   b) Resolve them together, perform all required unification
   c) If resolvent is the empty clause, a contradiction has been found
   d) If not, add resolvent to the premises.

If we succeed at Step (d), we have proved the conclusion



Code:

```
from copy import deepcopy

# ----------------------------
# Basic utilities
# ----------------------------

def is_variable(x):
    return isinstance(x, str) and x[0].islower()

def substitute(subst, expr):
    """Substitute variables in expr according to subst mapping."""
if isinstance(expr, str):
        return subst.get(expr, expr)
    return (expr[0],) + tuple(substitute(subst, a) for a in expr[1:])
```

```
def unify(x, y, subst=None):
   """Unify two literals, returning substitution if possible."""
if subst is None:         subst = {}      if x == y:
       return subst
if is_variable(x):
       return unify_var(x, y, subst)
if is_variable(y):
       return unify_var(y, x, subst)     if isinstance(x, tuple) and isinstance(y, tuple) and
x[0] == y[0] and len(x) == len(y):
       for a, b in zip(x[1:], y[1:]):
          subst = unify(substitute(subst, a), substitute(subst, b), subst)
if subst is None:                 return None         return subst
   return None

def unify_var(var, x, subst):
if var in subst:
       return unify(subst[var], x, subst)
elif x in subst:
       return unify(var, subst[x], subst)
elif occurs_check(var, x, subst):
       return None
else:
       subst[var] = x
       return subst

def occurs_check(var, x, subst):
"""Prevent circular substitutions."""
if var == x:         return True     elif
isinstance(x, tuple):
       return any(occurs_check(var, arg, subst) for arg in x[1:])
elif is_variable(x) and x in subst:
       return occurs_check(var, subst[x], subst)
return False

def negate(lit):
   return ('~' + lit[0][1:],) + lit[1:] if lit[0].startswith('~') else ('~' + lit[0],) + lit[1:]

# ----------------------------
# Resolution
# ----------------------------

def resolve(ci, cj):
   """Return resolvents between two clauses."""
resolvents = []
   for li in ci:         for lj in cj:             if li[0].startswith('~') != lj[0].startswith('~') and
li[0].lstrip('~') == lj[0].lstrip('~'):
              subst = unify(li, negate(lj))
if subst is not None:
```

```
                new_clause = set(substitute(subst, l) for l in ci.union(cj) if l != li and l != lj)
resolvents.append(frozenset(new_clause))     return resolvents

def fol_resolution(kb, query):
clauses = deepcopy(kb)
clauses.append({negate(query)})
   print("Initial   clauses:")
for c in clauses:       print("
",                              c)
print("\nResolving...\n")

   new = set()
while True:
      pairs = [(clauses[i], clauses[j]) for i in range(len(clauses)) for j in range(i + 1, len(clauses))]
for (ci, cj) in pairs:
          for resolvent in resolve(ci, cj):
if not resolvent:
                print("Derived empty clause -> contradiction found
☑")                  return True                new.add(resolvent)        if
new.issubset(set(map(frozenset, clauses))):            print("No new
clauses can be added ❌")           return False        for c in new:
if c not in clauses:
             clauses.append(set(c))
             print("New clause:", c)

# ----------------------------
# Example KB (CNF)
# ----------------------------

KB = [
   {('~Food', 'x'), ('Likes', 'John', 'x')},
   {('Food', 'Apple')},
   {('Food', 'Vegetable')},
   {('~Eats', 'x', 'y'), ('Killed', 'x'), ('Food', 'y')},
   {('Eats', 'Anil', 'Peanut')},
   {('Alive', 'Anil')},
   {('~Eats', 'Anil', 'y'), ('Eats', 'Harry', 'y')},
   {('~Alive', 'x'), ('~Killed', 'x')},
   {('Killed', 'x'), ('Alive', 'x')}
]

query = ('Likes', 'John', 'Peanut')

# ----------------------------
# Run Resolution
# ----------------------------

proved = fol_resolution(KB, query)
```

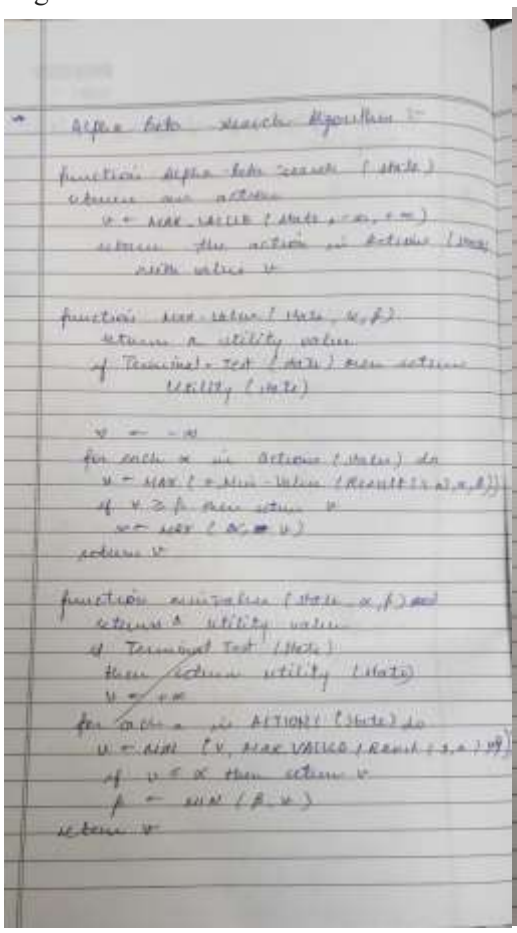
if proved:
    print("\n☑ Proven: John likes peanuts.")
else:
    print("\n❌ Could not prove John likes peanuts.")

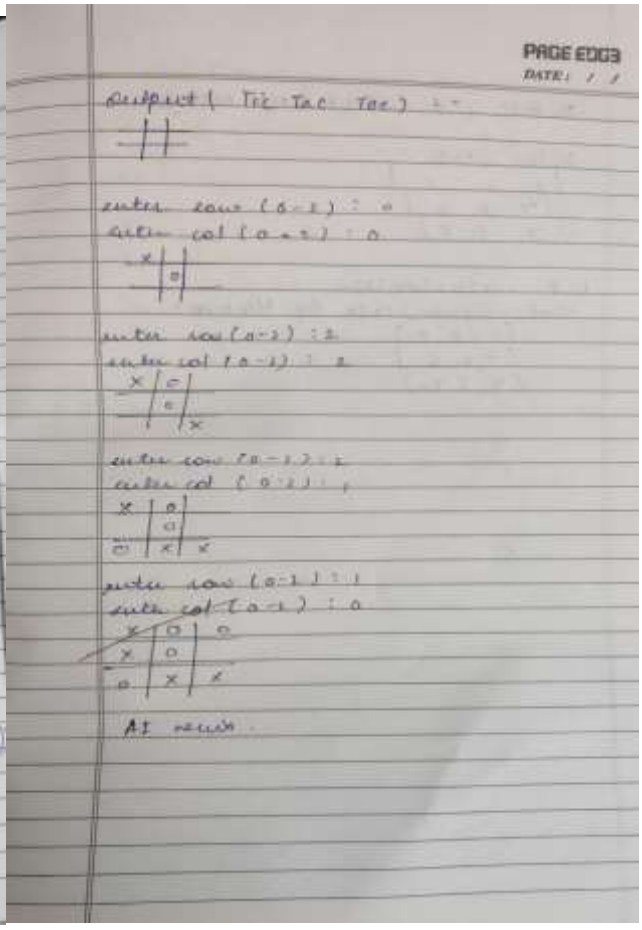
Output:

```
Initial clauses:
   {('Likes', 'John', 'x'), ('~Food', 'x')}
   {('Food', 'Apple')}
   {('Food', 'Vegetable')}
   {('Food', 'y'), ('~Eats', 'x', 'y'), ('Killed', 'x')}
   {('Eats', 'Anil', 'Peanut')}
   {('Alive', 'Anil')}
   {('~Eats', 'Anil', 'y'), ('Eats', 'Harry', 'y')}
   {('~Alive', 'x'), ('~Killed', 'x')}
   {('Alive', 'x'), ('Killed', 'x')}
   {('~Likes', 'John', 'Peanut')}

Resolving...

New clause: frozenset({('Killed', 'Harry'), ('Food', 'y'), ('~Eats', 'Anil', 'y')})
New clause: frozenset({('Killed', 'Anil'), ('Food', 'Peanut')})
New clause: frozenset({('Killed', 'x'), ('~Killed', 'x')})
New clause: frozenset({('Alive', 'x'), ('~Alive', 'x')})
New clause: frozenset({('Likes', 'John', 'Apple')})
New clause: frozenset({('Likes', 'John', 'Vegetable')})
New clause: frozenset({('~Eats', 'y', 'y'), ('Killed', 'y'), ('Likes', 'John', 'y')})
New clause: frozenset({('~Killed', 'x')})
New clause: frozenset({('Likes', 'John', 'Peanut'), ('Killed', 'Anil')})
New clause: frozenset({('Killed', 'Peanut'), ('~Eats', 'Peanut', 'Peanut')})
New clause: frozenset({('~Eats', 'y', 'y'), ('~Alive', 'y'), ('Likes', 'John', 'y')})
New clause: frozenset({('~Alive', 'Anil'), ('Food', 'Peanut')})
New clause: frozenset({('~Eats', 'Anil', 'y'), ('Killed', 'Harry'), ('Likes', 'John', 'y')})
New clause: frozenset({('~Alive', 'x')})
New clause: frozenset({('~Alive', 'Harry'), ('~Eats', 'Anil', 'y'), ('Food', 'y')})
New clause: frozenset({('Likes', 'John', 'Peanut'), ('~Alive', 'Anil')})
New clause: frozenset({('~Eats', 'Anil', 'Peanut'), ('Killed', 'Harry')})
New clause: frozenset({('Likes', 'John', 'Peanut')})
New clause: frozenset({('Killed', 'Anil')})
New clause: frozenset({('~Eats', 'Peanut', 'Peanut')})
New clause: frozenset({('~Eats', 'Anil', 'y'), ('Likes', 'John', 'y')})
New clause: frozenset({('~Alive', 'Peanut'), ('~Eats', 'Peanut', 'Peanut')})
New clause: frozenset({('~Eats', 'Anil', 'y'), ('~Alive', 'Harry'), ('Likes', 'John', 'y')})
Derived empty clause -> contradiction found ☑

☑ Proven: John likes peanuts.
```

Program 10:
Implement Alpha-Beta Pruning.

Algorithm:





Code:

```
import math

# ---------------------------------------------
# Define the game tree structure
# ---------------------------------------------
game_tree = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F', 'G'],
    'D': ['L1', 'L2'],
    'E': ['L3', 'L4'],
    'F': ['L5', 'L6'],
    'G': ['L7', 'L8'],
    'L1': 10,
    'L2': 9,
    'L3': 14,
    'L4': 18,
    'L5': 5,
    'L6': 4,
    'L7': 50,
    'L8': 3
}

# ---------------------------------------------
# Pretty print the game tree as ASCII art
# ---------------------------------------------
def print_tree():
    print("\n================== GAME TREE STRUCTURE ==================\n")
    print("                    A (MAX)")
    print("                  /           \\")
    print("             B (MIN)          C (MIN)")
    print("           /      \\        /      \\")
    print("      D (MAX)     E (MAX)  F (MAX)   G (MAX)")
    print("     /   \\      /   \\    /   \\    /   \\")
```

```
    print("   10    9   14   18  5    4  50    3")
    print("\n==========================================================\n")

# -------------------------------------------
# Alpha-Beta Pruning Implementation (with detailed trace)
# -------------------------------------------
def alphabeta(node, depth, alpha, beta, maximizing_player):
    indent = "  " * depth  # indentation for readability

    # Base case: Leaf node
    if isinstance(game_tree[node], int):
        print(f"{indent}Reached leaf {node} with value = {game_tree[node]}")
        return game_tree[node]

    # MAX Node
    if maximizing_player:
        print(f"{indent}→ MAX node {node} (depth={depth}) | α={alpha:.2f}, β={beta:.2f}")
        max_eval = -math.inf
        for child in game_tree[node]:
            print(f"{indent}  Exploring child {child} of {node} ...")
            eval_value = alphabeta(child, depth + 1, alpha, beta, False)
            max_eval = max(max_eval, eval_value)
            alpha = max(alpha, eval_value)
            print(f"{indent}  Updated {node}: value={max_eval}, α={alpha:.2f}, β={beta:.2f}")
            if beta <= alpha:
                print(f"{indent}  ⚠ Pruning remaining children of {node} (β={beta:.2f} ≤ α={alpha:.2f})")
                break
        return max_eval

    # MIN Node
    else:
        print(f"{indent}→ MIN node {node} (depth={depth}) | α={alpha:.2f}, β={beta:.2f}")
        min_eval = math.inf
        for child in game_tree[node]:
            print(f"{indent}  Exploring child {child} of {node} ...")
            eval_value = alphabeta(child, depth + 1, alpha, beta, True)
            min_eval = min(min_eval, eval_value)
            beta = min(beta, eval_value)
            print(f"{indent}  Updated {node}: value={min_eval}, α={alpha:.2f}, β={beta:.2f}")
            if beta <= alpha:
                print(f"{indent}  ⚠ Pruning remaining children of {node} (β={beta:.2f} ≤ α={alpha:.2f})")
                break
        return min_eval


# -------------------------------------------
# Run the algorithm
```

```
# ----------------------------------------
print_tree()
print("Starting Alpha–Beta Pruning...\n")

best_value = alphabeta('A', 0, -math.inf, math.inf, True)

print("\n============================================================")
print(f"☑ Best achievable value at root (A): {best_value}")
print("============================================================")
```

Output: