VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Shashank U (1BM23CS314)

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 **Shashank U (1BM23CS314),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

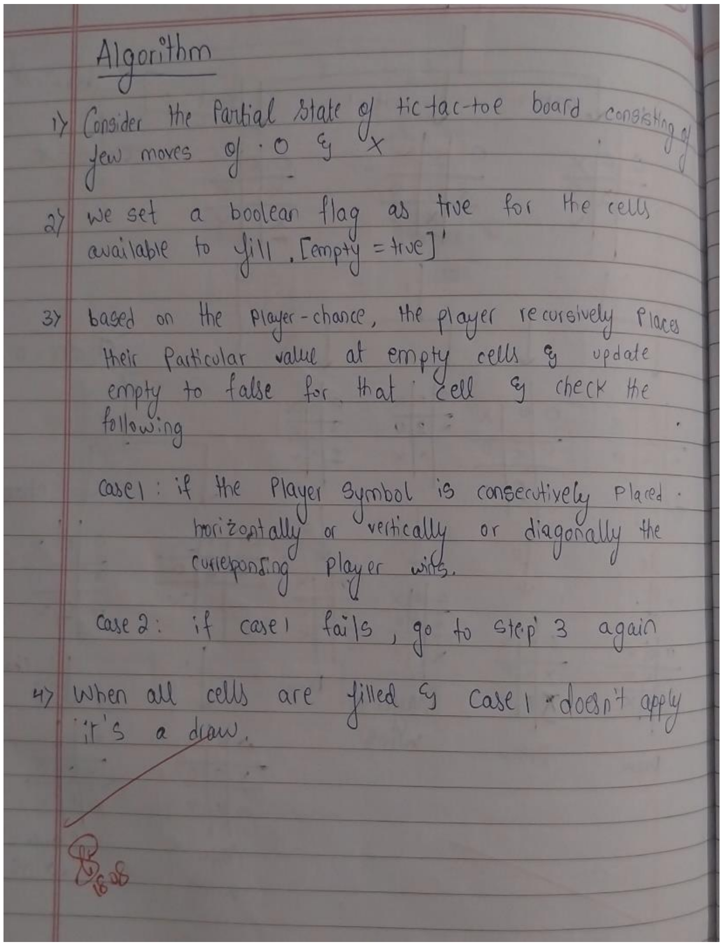
Prof. Surabhi S Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/Shashank-u803/AI-Lab

Program 1
Implement Tic –Tac –Toe Game
Implement vacuum cleaner agent

Algorithm:

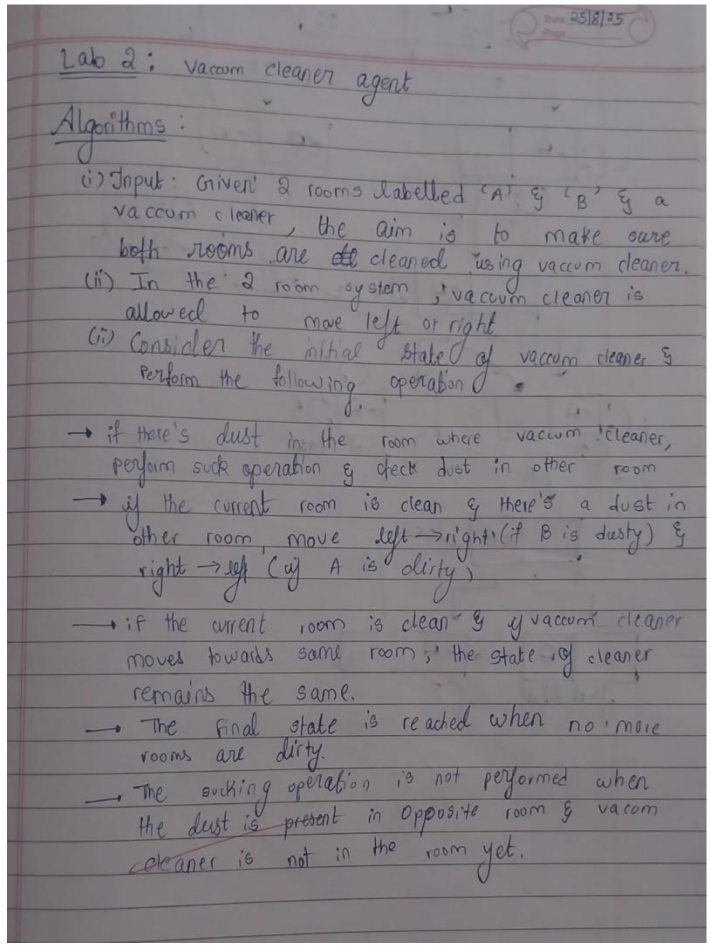

Algorithm

1) Consider the partial state of tic-tac-toe board consisting of few moves of O & X
2) we set a boolean flag as true for the cells available to fill. [empty = true]
3) based on the player-chance, the player recursively places their particular value at empty cells & update empty to false for that cell & check the following

case 1: if the player symbol is consecutively placed horizontally or vertically or diagonally the corresponding player wins.

case 2: if case 1 fails, go to step 3 again

4) When all cells are filled & case 1 doesn't apply it's a draw.


25/8/25

Lab 2: Vaccum cleaner agent

Algorithms:

(i) Input: Given 2 rooms labelled 'A' & 'B' & a vaccum cleaner, the aim is to make sure both rooms are ~~cl~~ cleaned using vaccum cleaner.
(ii) In the 2 room system, vaccum cleaner is allowed to move left or right
(i) Consider the initial state of vaccum cleaner & perform the following operation.

→ if there's dust in the room where vaccum cleaner, perform suck operation & check dust in other room
→ if the current room is clean & there's a dust in other room, move left → right (if B is dusty) & right → left (if A is dirty)
→ if the current room is clean & if vaccum cleaner moves towards same room, the state of cleaner remains the same.
→ The final state is reached when no more rooms are dirty.
→ The sucking operation is not performed when the dust is present in opposite room & vacom cleaner is not in the room yet.

Code:
TIC TAC TOE Game

```
def print_board(board):
    print()
    print("-------------")
    for i in range(3):
        print("|", board[i][0], "|", board[i][1], "|", board[i][2], "|")
        print("-------------")
    print()

def is_win(board,symbol):
    #same row
```

```
    for i in range(3):
        if board[i][0] == symbol and board[i][1] == symbol and board[i][2] ==
symbol:
            return True

    #same Column
    for j in range(3):
        if board[0][j] == symbol and board[1][j] == symbol and board[2][j] ==
symbol:
            return True

    #same diagonal
    if board[0][0] == symbol and board[1][1] == symbol and board[2][2] == symbol:
        return True

    if board[0][2] == symbol and board[1][1] == symbol and board[2][0] == symbol:
        return True

    #if all case fails
    return False


def is_draw(board):
    for i in range(3):
        for j in range(3):
            if board[i][j] == " ":
                return False
    return True



def get_input(mark,id):
    while True:
        try:
            move = input(f"Player{id}, Enter the position to place {mark}->  ")
            ip = move.strip().split()

            if len(ip) != 2:
                print("Enter exactly 2 coordinates!")
                continue

            row,col = int(ip[0]),int(ip[1])

            if (row>=1 and row<= 3) and (col>=1 and col<=3):
                return [row-1,col-1]
            else:
                print("Enter values between 1 and 3 only")
        except ValueError:
            print("Invalid inputs, enter numbers only")


board = []
for i in range(3):
    board.append([" "," "," "])

mark1 = "X"
mark2 = "O"
p1 = 1
p2 = 2
```

```
curr_p = 1
curr_m = "X"
while True:
    print_board(board)
    row,col = get_input(curr_m,curr_p)

    if board[row][col] != " ":
        print("Move already taken!,Try again")
        continue

    board[row][col] = curr_m

    if is_win(board,curr_m):
        print_board(board)
        print(f"Congrats🎉, Player{curr_p} wins!!!")
        break

    if is_draw(board):
        print_board(board)
        print("Its a draw,🤓!")
        break

    if curr_p == p1:
        curr_p = p2
        curr_m = "O"
    else:
        curr_p = p1
        curr_m = "X"
```

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

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

-------------
| X |   |   |
-------------
|   |   |   |
-------------
|   |   |   |
-------------







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

-------------
```

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

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

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

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

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

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

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

Congrats🎉, Player1 wins!!!
```

Vacuum Cleaner Agent

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


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

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

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

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

    return cost


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

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

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

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

```python
    else:
        print("Invalid action choice")
```

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

Program 2
Implement 8 puzzle problems using Depth First Search (DFS)
Implement Iterative deepening search algorithm

Algorithm:

Algorithm (using BFS)

1] Consider a Scrambled 8 puzzle setup with given initial & Goal state.

2] Match i.state with final state
if (matched):
return answer with Backtracking steps
else
explore all possible ways by utilizing empty space until matched

3] Compute the depth at each state & return finally

Code:

8 Puzzle using DFS

```
from collections import deque

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

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

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

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

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

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

        if state == goal:
            return path

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

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

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

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

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

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

Solving puzzle with DFS...

Solution found using DFS! (may not be optimal)

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

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

```
(7, 5, 0)

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

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

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

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

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

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

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

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

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

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

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

```
(4, 0, 6)

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

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

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

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

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

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

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

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

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

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

Move: Down
(2, 8, 3)
(6, 4, 5)
```

```
(0, 1, 7)

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

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

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

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

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

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

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

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

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

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

Move: Left
(2, 8, 3)
(7, 4, 5)
```

```
(6, 0, 1)

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

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

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

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

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

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

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

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

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

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

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

```
(7, 6, 5)

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

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

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

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

8 Puzzle using Iterative Deepening DFS

```
from collections import deque

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

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

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

# Depth-Limited Search (helper for IDDFS)
def dls(state, goal, depth, path, explored):
```

```python
    if state == goal:
        return path
    if depth == 0:
        return None
    explored.add(state)
    for neighbor, action in get_neighbors(state):
        if neighbor not in explored:
            result = dls(neighbor, goal, depth-1, path + [(action, neighbor)],
explored)
            if result is not None:
                return result
    return None

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

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

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

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

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

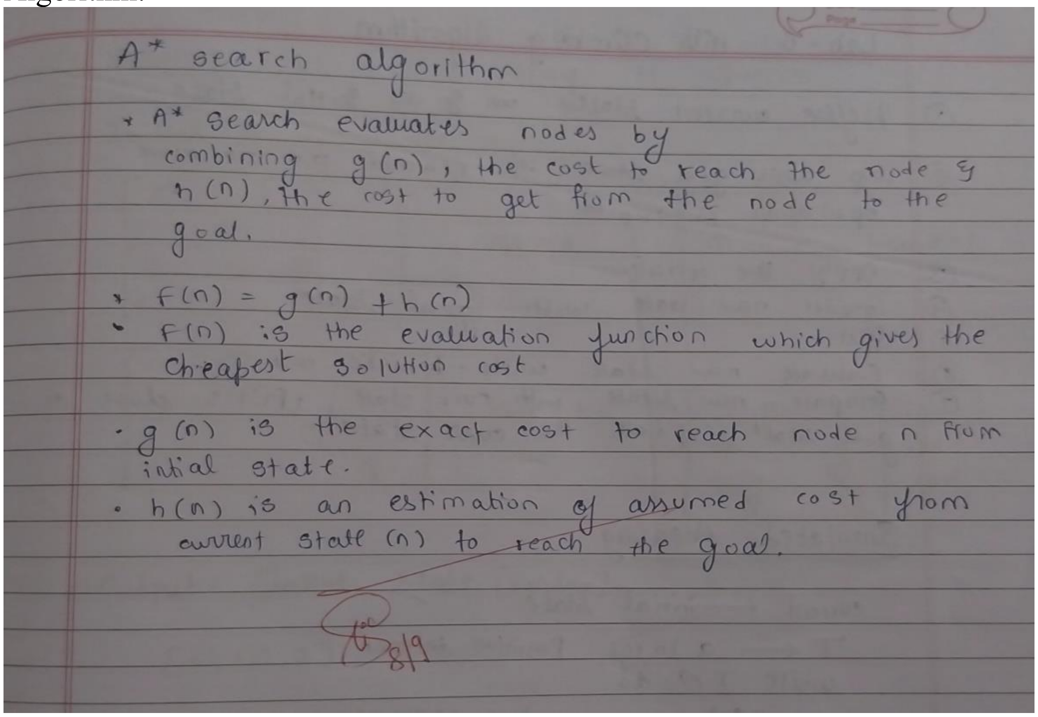
```
Enter the initial state (0 for blank, space-separated, 9 numbers):
2 8 3 1 6 4 7 0 5
Enter the goal state (0 for blank, space-separated, 9 numbers):
1 2 3 8 0 4 7 6 5

Solving puzzle with Iterative Deepening DFS...

Optimal solution found using IDDFS!

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

```
(7, 0, 5)

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

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

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

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

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

Program 3

Implement A* search algorithm

Algorithm:



A* search algorithm

* A* search evaluates nodes by combining g(n), the cost to reach the node & h(n), the cost to get from the node to the goal.
* f(n) = g(n) + h(n)
* f(n) is the evaluation function which gives the cheapest solution cost
* g(n) is the exact cost to reach node n from inital state.
* h(n) is an estimation of assumed cost from current state (n) to reach the goal.

8/9

Code:

A* using misplaced tiles for 8 puzzle

```
from heapq import heappush, heappop

MOVES = {
    'up': -3,
    'down': 3,
    'left': -1,
    'right': 1
}

def is_valid_move(zero_pos, move):
    if move == 'left' and zero_pos % 3 == 0:
        return False
    if move == 'right' and zero_pos % 3 == 2:
        return False
    if move == 'up' and zero_pos < 3:
        return False
    if move == 'down' and zero_pos > 5:
        return False
    return True

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

def print_state_formatted(state):
    for i in range(0, 9, 3):
```

```
        row = state[i:i+3]
        print(" ".join(str(x) if x != 0 else " " for x in row))
    print()

def a_star_misplaced(start, goal):
    open_set = []
    closed_set = set()
    g = 0
    h = misplaced_tiles(start, goal)
    f = g + h
    heappush(open_set, (f, g, start, [], None))  # last element: move that got here

    while open_set:
        f, g, current, path, move_made = heappop(open_set)

        # Print detailed info with matrix format and move made
        if move_made is None:
            print(f"Expanding node with f={f}, g={g} - Start state")
        else:
            print(f"Expanding node with f={f}, g={g} - Move: {move_made}")
        print_state_formatted(current)

        if current == goal:
            print("Goal reached!")
            print("Solution path (moves):", path)
            print(f"Final depth (number of moves): {g}")
            return path, g

        closed_set.add(current)

        zero_pos = current.index(0)

        for move, shift in MOVES.items():
            if is_valid_move(zero_pos, move):
                new_zero_pos = zero_pos + shift
                new_state = list(current)
                new_state[zero_pos], new_state[new_zero_pos] =
new_state[new_zero_pos], new_state[zero_pos]
                new_state = tuple(new_state)

                if new_state in closed_set:
                    continue

                new_g = g + 1
                new_h = misplaced_tiles(new_state, goal)
                new_f = new_g + new_h
                heappush(open_set, (new_f, new_g, new_state, path + [move], move))

    print("No solution found.")
    return None, None

def get_state_matrix(prompt):
    print(prompt)
    matrix = []
    for i in range(3):
        while True:
            try:
                row = list(map(int, input().strip().split()))
                if len(row) != 3:
```

```
                    raise ValueError("Each row must have exactly 3 numbers.")
                matrix.extend(row)
                break
            except Exception as e:
                print("Invalid input:", e)
    if set(matrix) != set(range(9)):
        print("Error: The numbers must be from 0 to 8 without repetition.")
        return get_state_matrix(prompt)
    return tuple(matrix)

if __name__ == "__main__":
    start_state = get_state_matrix("Enter the initial state (3 rows, each with 3
numbers separated by spaces):")
    goal_state = get_state_matrix("Enter the goal state (3 rows, each with 3 numbers
separated by spaces):")

    print("\nInitial State:")
    print_state_formatted(start_state)
    print("Goal State:")
    print_state_formatted(goal_state)

    a_star_misplaced(start_state, goal_state)
```

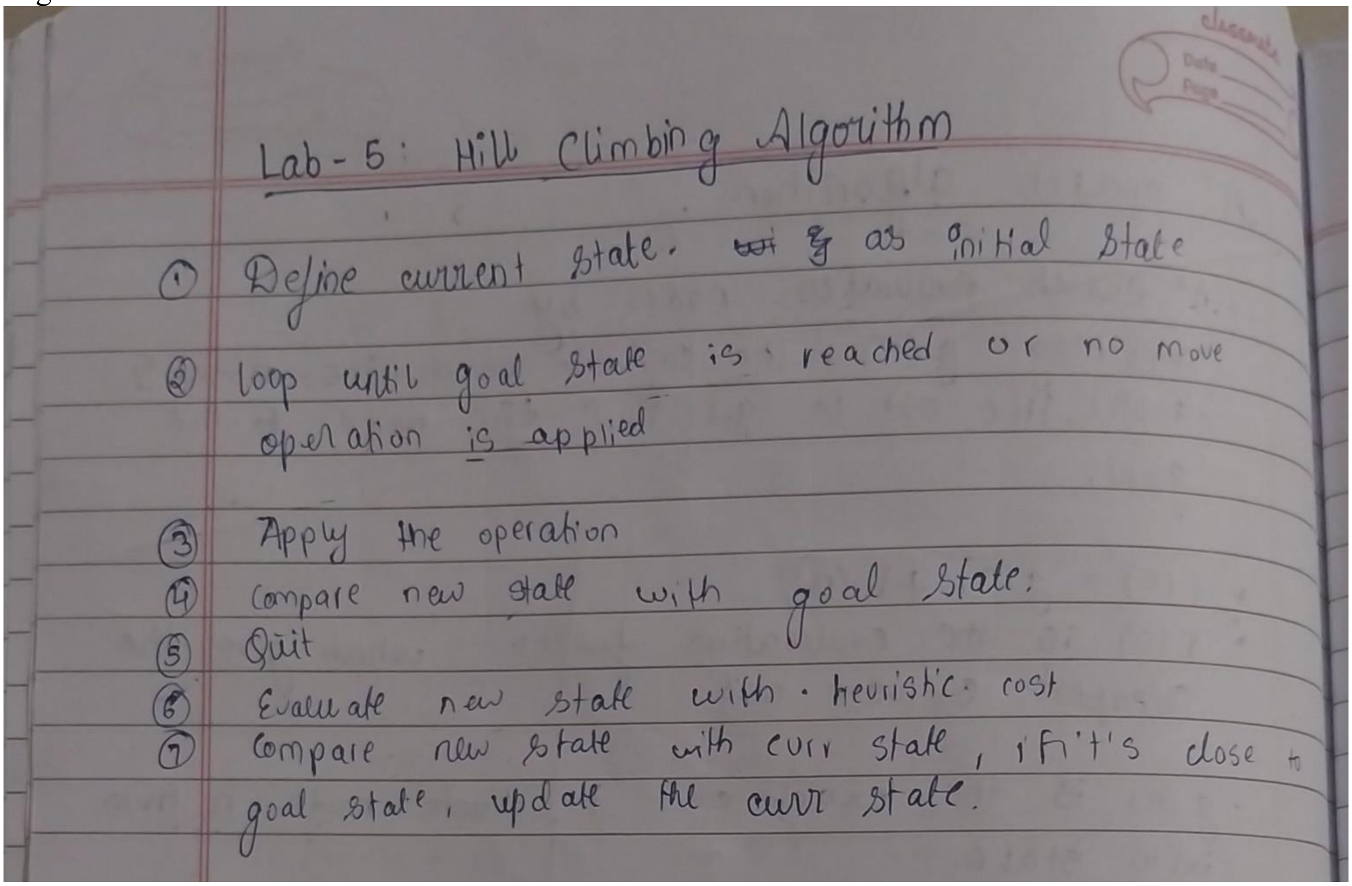
```
Enter the initial state (3 rows, each with 3 numbers separated by spaces):
2 8 3
1 6 4
7 0 5
Enter the goal state (3 rows, each with 3 numbers separated by spaces):
1 2 3
8 0 4
7 6 5

Initial State:
2 8 3
1 6 4
7   5

Goal State:
1 2 3
8   4
7 6 5

Expanding node with f=4, g=0 - Start state
2 8 3
1 6 4
7   5

Expanding node with f=4, g=1 - Move: up
2 8 3
1   4
7 6 5

Expanding node with f=5, g=2 - Move: up
2   3
1 8 4
```

```
7 6 5

Expanding node with f=5, g=2 - Move: left
2 8 3
  1 4
7 6 5

Expanding node with f=5, g=3 - Move: left
  2 3
1 8 4
7 6 5



Expanding node with f=5, g=4 - Move: down
1 2 3
  8 4
7 6 5

Expanding node with f=5, g=5 - Move: right
1 2 3
8   4
7 6 5

Goal reached!
Solution path (moves): ['up', 'up', 'left', 'down', 'right']
Final depth (number of moves): 5
```

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:



Lab - 5: Hill Climbing Algorithm

1. Define current state. & as initial state
2. loop until goal state is reached or no move operation is applied
3. Apply the operation
4. Compare new state with goal state.
5. Quit
6. Evaluate new state with heuristic cost
7. Compare new state with curr state, if it's close to goal state, update the curr state.

Code:

```
def get_user_input(n):
    board = []
    print(f"Enter the row positions (0 to {n-1}) of the queens for each column:")
    for col in range(n):
        while True:
            try:
                row = int(input(f"Column {col}: "))
                if 0 <= row < n:
                    board.append(row)
                    break
                else:
                    print(f"Invalid input. Please enter a number between 0 and {n-
1}.")
            except ValueError:
                print("Invalid input. Please enter an integer.")
    return board

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

```python
            else:
                line += " . "
        print(line)
    print()

def heuristic(board):
    n = len(board)
    attacks = 0
    for i in range(n):
        for j in range(i+1, n):
            # Check same row
            if board[i] == board[j]:
                attacks += 1
            # Check same diagonal
            if abs(board[i] - board[j]) == abs(i - j):
                attacks += 1
    return attacks

def get_best_neighbor(board):
    n = len(board)
    current_heuristic = heuristic(board)
    best_board = list(board)
    best_heuristic = current_heuristic

    for col in range(n):
        for row in range(n):
            if board[col] != row:
                new_board = list(board)
                new_board[col] = row
                new_heuristic = heuristic(new_board)
                if new_heuristic < best_heuristic:
                    best_heuristic = new_heuristic
                    best_board = new_board
    return best_board, best_heuristic

def hill_climbing_with_user_input(n):
    board = get_user_input(n)
    current_heuristic = heuristic(board)

    steps = 0
    while True:
        print(f"Step {steps}: Heuristic = {current_heuristic}")
        print_board(board)

        if current_heuristic == 0:
            print("Solution found!")
            return board

        neighbor, neighbor_heuristic = get_best_neighbor(board)

        # If no improvement, stuck at local minimum
        if neighbor_heuristic >= current_heuristic:
            print("Reached local minimum (no better neighbors).")
            return board

        board = neighbor
        current_heuristic = neighbor_heuristic
        steps += 1
```

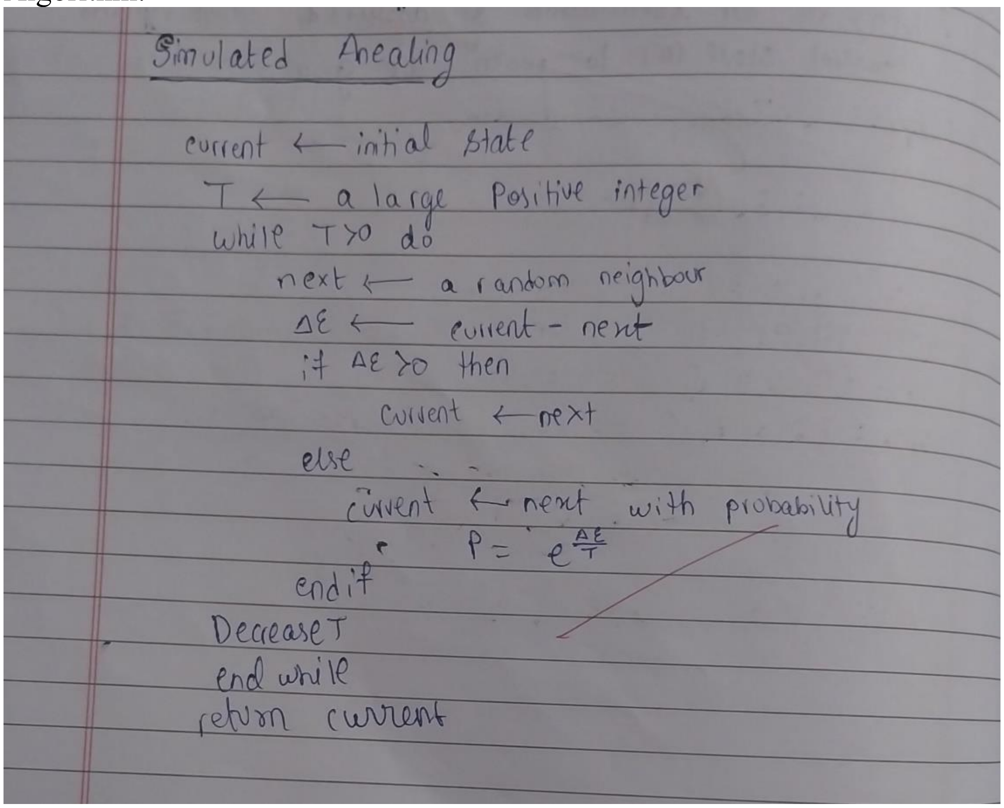
```python
# Run the algorithm
if __name__ == "__main__":
    n = 4
    solution = hill_climbing_with_user_input(n)
    print("Final solution:")
    print_board(solution)
```

```
Enter the row positions (0 to 3) of the queens for each column:
Column 0: 3
Column 1: 1
Column 2: 2
Column 3: 0
```

```
Step 0: Heuristic = 2
 .  .  .  Q
 .  Q  .  .
 .  .  Q  .
 Q  .  .  .

Reached local minimum (no better neighbors).
Final solution:
 .  .  .  Q
 .  Q  .  .
 .  .  Q  .
 Q  .  .  .
```

Program 5
Simulated Annealing to Solve 8-Queens problem

Algorithm:



Simulated Anealing

current ← initial state
T ← a large Positive integer
while T>0 do
    next ← a random neighbour
    ΔE ← current - next
    if $\Delta E > 0$ then
        Current ← next
    else
        current ← next with probability
        $P = e^{\frac{\Delta E}{T}}$
    endif
Decrease T
end while
return current

Code:

```
import random
import math

def get_user_input(n):
    board = []
    print(f"Enter the row positions (0 to {n-1}) of the queens for each column:")
    for col in range(n):
        while True:
            try:
                row = int(input(f"Column {col}: "))
                if 0 <= row < n:
                    board.append(row)
                    break
                else:
                    print(f"Invalid input. Please enter a number between 0 and {n-
1}.")
            except ValueError:
                print("Invalid input. Please enter an integer.")
    return board

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

```
    print()

def heuristic(board):
    n = len(board)
    attacks = 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):
                attacks += 1
    return attacks

def random_neighbor(board):
    n = len(board)
    neighbor = list(board)
    col = random.randint(0, n-1)
    row = random.randint(0, n-1)
    while row == neighbor[col]:
        row = random.randint(0, n-1)
    neighbor[col] = row
    return neighbor

def simulated_annealing(n=8, max_iter=100000, initial_temp=100, cooling_rate=0.995):
    current_board = get_user_input(n)
    current_heuristic = heuristic(current_board)
    temperature = initial_temp
    iteration = 0

    print(f"Initial heuristic: {current_heuristic}")
    print_board(current_board)

    while temperature > 0.1 and current_heuristic > 0 and iteration < max_iter:
        neighbor = random_neighbor(current_board)
        neighbor_heuristic = heuristic(neighbor)
        delta_e = current_heuristic - neighbor_heuristic

        if delta_e > 0:
            current_board = neighbor
            current_heuristic = neighbor_heuristic
        else:
            probability = math.exp(delta_e / temperature)
            if random.random() < probability:
                current_board = neighbor
                current_heuristic = neighbor_heuristic

        temperature *= cooling_rate
        iteration += 1

        if iteration % 1000 == 0:
            print(f"Iteration {iteration}, Temperature: {temperature:.2f},
Heuristic: {current_heuristic}")
            print_board(current_board)

    if current_heuristic == 0:
        print("Solution found!")
    else:
        print("Stopped without full solution. Best board found:")
    print(f"Final heuristic: {current_heuristic}")
    print_board(current_board)
```

```python
if __name__ == "__main__":
    simulated_annealing()
```

```
Enter the row positions (0 to 7) of the queens for each column:
Column 0: 4
Column 1: 6
Column 2: 1
Column 3: 5
Column 4: 2
Column 5: 0
Column 6: 3
Column 7: 7
```

```
Initial heuristic: 0
 .  .  .  .  .  Q  .  .
 .  .  Q  .  .  .  .  .
 .  .  .  .  Q  .  .  .
 .  .  .  .  .  .  Q  .
 Q  .  .  .  .  .  .  .
 .  .  .  Q  .  .  .  .
 .  Q  .  .  .  .  .  .
 .  .  .  .  .  .  .  Q

Solution found!
Final heuristic: 0
 .  .  .  .  .  Q  .  .
 .  .  Q  .  .  .  .  .
 .  .  .  .  Q  .  .  .
 .  .  .  .  .  .  Q  .
 Q  .  .  .  .  .  .  .
 .  .  .  Q  .  .  .  .
 .  Q  .  .  .  .  .  .
 .  .  .  .  .  .  .  Q
```

Program 6

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

Algorithm:

Algorithm

function TT-ENTAILS? (KB, α) returns true or false
  inputs: KB, the knowledge base, a sentence in propositional logic α, the query, a sentence in propositional logic

  symbols ← a list of proposition symbols in KB & α
  return ← TT-CHECK-All (KB, α, symbols, { })

function TT-CHECK-ALL (KB, α, symbols, model) returns true or false
  if PL-True? (KB, model) then return PL-True? (α, model)
  else return true // when KB is false, always return true

else do
  P ← First (symbols)
  rest ← REST (symbols)
  return (TT-check-all (KB, α, rest, model ∪ {P = true})
  and
  TT-CHECK-all (KB, α, rest, model ∪ {P = false}))

Code:

```
import itertools
import pandas as pd
import re

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

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

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

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

    return eval(expr)

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

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

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

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

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

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

    df = pd.DataFrame(rows)

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

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

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

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

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

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

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

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

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

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

```
Enter Knowledge Base (KB) sentences, separated by commas.
Use symbols like A, B, C and operators: and, or, not, =>, <=>
KB: not (S or T)
Enter query (alpha): T or (not T)

==================================================
                TRUTH TABLE
==================================================
┌───────┬───────┬───────┬───────┐
│   S   │   T   │  KB   │ alpha │
├───────┼───────┼───────┼───────┤
│ True  │ True  │ False │ True  │
│ True  │ False │ False │ True  │
│ False │ True  │ False │ True  │
│ False │ False │ True  │ True  │
└───────────────────────────────┘

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

Program 7

Implement unification in first order logic

Algorithm:

* Unification Algorithm *

Step 1: If Ψ_1 or Ψ_2 is a variable or constant, then:

a) if Ψ_1 or Ψ_2 are identical, return NIL

b) Else if Ψ_1 is a variable,

  a. then if Ψ_1 occurs in Ψ_2, then return FAILURE

  b. Else return $\{(\Psi_2/\Psi_1)\}$

c) Else if Ψ_2 is a variable:

  a. if Ψ_2 occurs in Ψ_1 then return FAILURE

  b. Else return $\{(\Psi_1/\Psi_2)\}$

d) Else return Failure.

Step 2: if the initial Predicate symbol in Ψ_1 and Ψ_2 are not same, then return Failure

Step 3: if Ψ_1 & Ψ_2 have a different no. of args, then return FAILURE

Step 4: Set Substitution set (SUBST) to NIL

Code:

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

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

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

    def __repr__(self):
        return self.name

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

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

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

    def __repr__(self):
        return self.name

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

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

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

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

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

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

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

    # Apply existing substitutions
    term1 = apply_substitution(term1, subst)
    term2 = apply_substitution(term2, subst)

    # Step 1: If term1 or term2 is a variable or constant
```

```python
    # Step 1a: If both are identical
    if term1 == term2:
        return subst

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

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

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

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

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

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

        return current_subst

    else:
        return None  # FAILURE

def print_substitution(subst):
    """Pretty print substitution set"""
    if subst is None:
        print("FAILURE: Unification failed")
    elif not subst:
        print("NIL: Terms are already unified")
    else:
        print("Substitution:")
        for var, term in subst.items():
            print(f"  {var} -> {term}")

def parse_term(term_str):
```

```python
    """Parse a string representation of a term into Term objects"""
    term_str = term_str.strip()

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

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

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

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

        return Predicate(pred_name, args)

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

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

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

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

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

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

        if term2_str.lower() == 'quit':
```

```python
            print("Exiting...")
            break

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

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

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

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

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

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

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

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

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

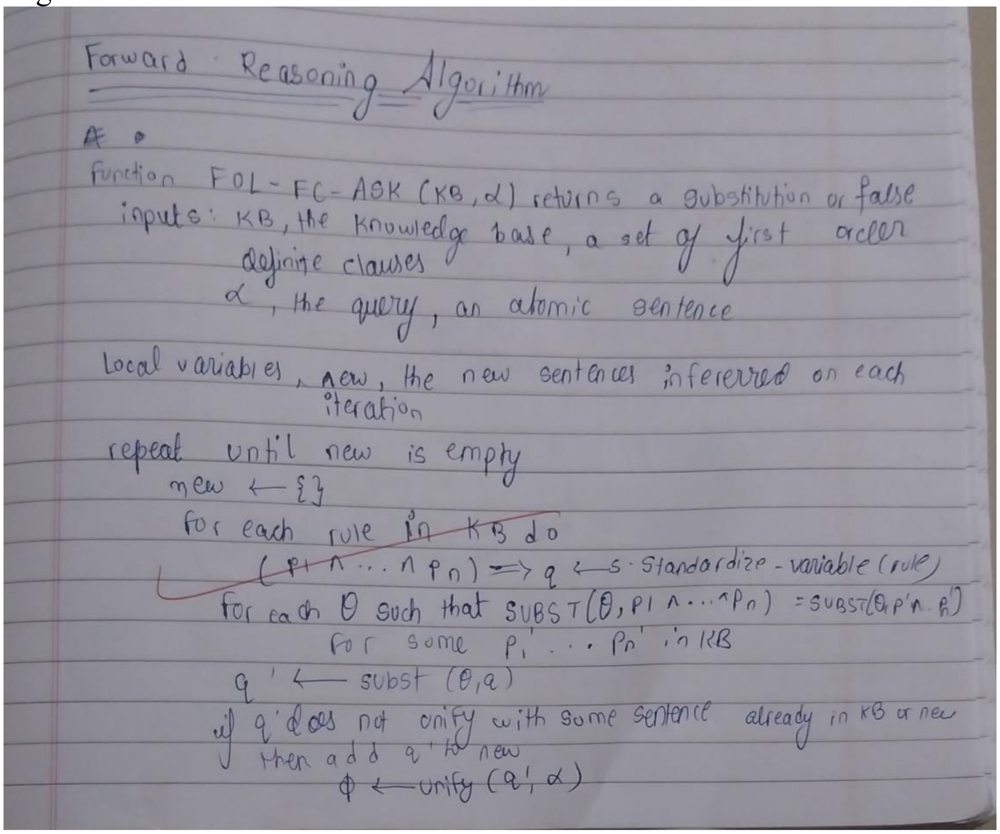
```python
    # Example 6: Failure case - different predicates
    print("Example 6: Unify(P(x), Q(x)) - Different Predicates")
    p1 = Predicate('P', [Variable('x')])
    p2 = Predicate('Q', [Variable('x')])
    result = unify(p1, p2)
    print_substitution(result)
    print()

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

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

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

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

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

Enter choice (1 or 2): 1

=== Unification Algorithm Examples ===

Example 1: Unify(x, y)
Substitution:
  x -> y

Example 2: Unify(x, John)
Substitution:
  x -> John

Example 3: Unify(P(x, y), P(John, z))
Substitution:
  x -> John
  y -> z
```

```
Example 4: Unify(P(x, f(y)), P(a, f(b)))
Substitution:
  x -> a
  y -> b

Example 5: Unify(x, f(x)) - Occurs Check
FAILURE: Unification failed

Example 6: Unify(P(x), Q(x)) - Different Predicates
FAILURE: Unification failed

Example 7: Unify(John, Mary) - Different Constants
FAILURE: Unification failed
```

Program 8
Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.

Algorithm:



Forward Reasoning Algorithm

function FOL-FC-ASK(KB, α) returns a substitution or false
inputs: KB, the knowledge base, a set of first order definite clauses
α, the query, an atomic sentence

Local variables, new, the new sentences inferred on each iteration

repeat until new is empty
new ← {}
for each rule in KB do
$(p_1 \wedge \ldots \wedge p_n) \Rightarrow q$ ← Standardize-variable(rule)
for each θ such that SUBST(θ, $p_1 \wedge \ldots \wedge p_n$) = SUBST(θ, $p'_1 \wedge \ldots p'_n$)
for some $p'_1 \ldots p'_n$ in KB
q' ← subst(θ, q)
if q' does not unify with some sentence already in KB or new
then add q' to new
φ ← unify(q', α)

Code:

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

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

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

    def __repr__(self):
        return self.name

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

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

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

    def __repr__(self):
        return self.name

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

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

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

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

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

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

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

# Variable counter for standardization
_var_counter = 0

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

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

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

```
            return term

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

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

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

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

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

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

        current_subst = subst.copy()
```

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

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

    if not premises:
        return [subst]

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

    all_substitutions = []

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

    return all_substitutions

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

    Args:
        kb_facts: List of atomic sentences (facts) in KB
        kb_rules: List of implication rules in KB
        query: The query to prove (atomic sentence)
        max_iterations: Maximum number of iterations to prevent infinite loops

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

    iteration = 0

    while iteration < max_iterations:
```

```
        iteration += 1
        new = []

        print(f"Iteration {iteration}:")

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

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

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

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

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

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

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

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

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

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

        args = []
        depth = 0
        current_arg = ""
        for char in args_str:
            if char == ',' and depth == 0:
```

```
                args.append(parse_term(current_arg))
                current_arg = ""
            else:
                if char == '(':
                    depth += 1
                elif char == ')':
                    depth -= 1
                current_arg += char

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

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

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

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

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

        return Rule(premises, conclusion)
    else:
        # It's just a fact
        return parse_term(rule_str)

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

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

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

        # Rules
        kb_rules = [
            # Animal(x) ∧ Loves(y, x) => Loves(x, y)
```

```python
        Rule([Predicate('Animal', [Variable('x')]),
              Predicate('Loves', [Variable('y'), Variable('x')])],
             Predicate('Loves', [Variable('x'), Variable('y')])),

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

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

    result = fol_fc_ask(kb_facts, kb_rules, query)

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

    kb_facts = []
    kb_rules = []

    # Input facts
    print("Enter facts (one per line, empty line to finish):")
    print("Example: Animal(Dog), Loves(John, Dog)")
    while True:
        fact_str = input("Fact: ").strip()
        if not fact_str:
            break
        try:
            fact = parse_term(fact_str)
            kb_facts.append(fact)
        except Exception as e:
            print(f"Error parsing fact: {e}")

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

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

else:
```

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

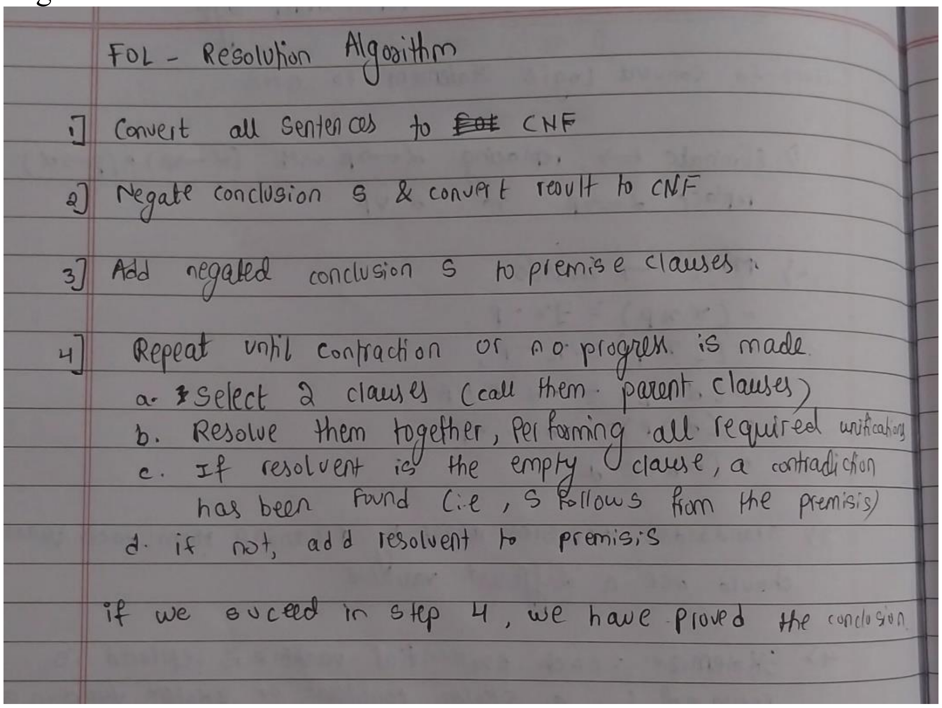
```
Choose mode:
1. Run example (Animal reasoning)
2. Interactive mode

Enter choice (1 or 2): 2

=== Interactive Forward Chaining ===
Enter facts and rules for the knowledge base.

Enter facts (one per line, empty line to finish):
Example: Animal(Dog), Loves(John, Dog)
Fact: Owns(A, T1)
Fact: Missile(T1)
Fact: American(Robert)
Fact: Enemy(A, America)
Fact:

Enter rules (one per line, empty line to finish):
Example: Animal(x) ∧ Loves(y,x) => Loves(x,y)
You can also use 'AND' instead of ∧
Rule: American(p) ∧ Weapon(q) ∧ Sells(p,q,r) ∧ Hostile(r) => Criminal(p)
Rule: Missile(x) ∧ Owns(A,x) => Sells(Robert,x,A)
Rule: Missile(x) => Weapon(x)
Rule: Enemy(x,America) => Hostile(x)
Rule:

Enter query:
Query: Criminal(Robert)
=== Forward Chaining Algorithm ===

Query: Criminal(Robert)

Initial KB Facts:
  Owns(A, T1)
  Missile(T1)
  American(Robert)
  Enemy(A, America)

KB Rules:
  American(p) ∧ Weapon(q) ∧ Sells(p, q, r) ∧ Hostile(r) => Criminal(p)
  Missile(x) ∧ Owns(A, x) => Sells(Robert, x, A)
  Missile(x) => Weapon(x)
  Enemy(x, America) => Hostile(x)

==================================================
```

```
Iteration 1:
  Inferred: Sells(Robert, T1, A)
    From rule: Missile(v4) ∧ Owns(A, v4) => Sells(Robert, v4, A)
    With substitution: {v4: T1}
  Inferred: Weapon(T1)
    From rule: Missile(v5) => Weapon(v5)
    With substitution: {v5: T1}
  Inferred: Hostile(A)
    From rule: Enemy(v6, America) => Hostile(v6)
    With substitution: {v6: A}

Iteration 2:
  Inferred: Criminal(Robert)
    From rule: American(v7) ∧ Weapon(v8) ∧ Sells(v7, v8, v9) ∧ Hostile(v9) => Crimin
al(v7)
    With substitution: {v7: Robert, v8: T1, v9: A}

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

Program 9

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

Algorithm:



FOL - Resolution Algorithm

1] Convert all sentences to ~~FOL~~ CNF

2] Negate conclusion S & convert result to CNF

3] Add negated conclusion S to premise clauses.

4] Repeat until contraction or no progress is made

a. Select 2 clauses (call them parent clauses)

b. Resolve them together, performing all required unifications

c. If resolvent is the empty clause, a contradiction has been found (i.e, S follows from the premisis)

d. if not, add resolvent to premisis

if we suceed in step 4, we have proved the conclusion

Code:

```
import re
import itertools

VAR_RE = re.compile(r'^[a-z](_\d+)?$')    # single-letter variable optionally
standardized (x_0, y_3)

def is_variable(token: str) -> bool:
    return bool(VAR_RE.fullmatch(token))

def parse_literal(text):
    text = text.strip()
    neg = False
    if text.startswith('¬') or text.startswith('~'):
        neg = True
        text = text[1:].strip()
    if '(' in text:
        pred = text[:text.index('(')].strip()
        args = [a.strip() for a in text[text.index('(')+1:-1].split(',')]
    else:
        pred = text
        args = []
    return {'neg': neg, 'pred': pred, 'args': args}

def clause_to_str(clause):
    if clause == []:
        return '⊥'
    parts = []
    for lit in clause:
```

```
        s = ('¬' if lit['neg'] else '') + (lit['pred'] + '(' + ',
'.join(lit['args']) + ')' if lit['args'] else lit['pred'])
        parts.append(s)
    return ' ∨ '.join(parts)

def standardize_apart_clause(clause, idx):
    # only rename variables (single-letter) to var_index form
    mapping = {}
    new_clause = []
    for lit in clause:
        new_args = []
        for a in lit['args']:
            if is_variable(a):
                if a not in mapping:
                    mapping[a] = f"{a}_{idx}"
                new_args.append(mapping[a])
            else:
                new_args.append(a)
        new_clause.append({'neg': lit['neg'], 'pred': lit['pred'], 'args':
new_args})
    return new_clause

# ----- Unification for flat args (no nested function terms) -----
def occurs_check(var, val, subs):
    # var and val are token strings
    if var == val:
        return True
    if is_variable(val) and val in subs:
        return occurs_check(var, subs[val], subs)
    return False

def apply_subs_token(tok, subs):
    if is_variable(tok):
        while tok in subs:
            tok = subs[tok]
        return tok
    return tok

def apply_subs_literal(lit, subs):
    new_args = [apply_subs_token(a, subs) for a in lit['args']]
    return {'neg': lit['neg'], 'pred': lit['pred'], 'args': new_args}

def unify_tokens(x, y, subs):
    # x,y are token strings (variables or constants)
    if x == y:
        return subs
    if is_variable(x):
        if x in subs:
            return unify_tokens(subs[x], y, subs)
        if occurs_check(x, y, subs):
            return None
        new = subs.copy()
        new[x] = y
        return new
    if is_variable(y):
        return unify_tokens(y, x, subs)
    # both constants and different => fail
    return None
```

```python
def unify_arg_lists(a_list, b_list):
    if len(a_list) != len(b_list):
        return None
    subs = {}
    for a, b in zip(a_list, b_list):
        a_ap = a if not is_variable(a) else a
        b_ap = b if not is_variable(b) else b
        subs = unify_tokens(apply_subs_token(a_ap, subs), apply_subs_token(b_ap,
subs), subs)
        if subs is None:
            return None
    return subs

# ----- Resolution -----
def is_tautology_clause(clause):
    # clause is a list of literals (after substitution). If it contains A and ¬A
same args -> tautology
    seen = {}
    for lit in clause:
        key = (lit['pred'], tuple(lit['args']))
        if key in seen:
            if seen[key] != lit['neg']:
                return True
        else:
            seen[key] = lit['neg']
    return False

def resolve_pair(c1, c2):
    # c1, c2 are lists of literals (each literal dict)
    for i, l1 in enumerate(c1):
        for j, l2 in enumerate(c2):
            if l1['pred'] == l2['pred'] and l1['neg'] != l2['neg']:
                # try to unify their args
                subs = unify_arg_lists(l1['args'], l2['args'])
                if subs is None:
                    continue
                # apply substitution to the remainder of both clauses
                new_clause = []
                for k, lit in enumerate(c1):
                    if k == i: continue
                    new_clause.append(apply_subs_literal(lit, subs))
                for k, lit in enumerate(c2):
                    if k == j: continue
                    new_clause.append(apply_subs_literal(lit, subs))
                # remove duplicates (syntactic)
                uniq = []
                for lit in new_clause:
                    if not any(lit['pred']==u['pred'] and lit['neg']==u['neg'] and
lit['args']==u['args'] for u in uniq):
                        uniq.append(lit)
                if is_tautology_clause(uniq):
                    continue
                return uniq, subs, (i, j)
    return None, None, None

# ----- Build derivation tree nodes -----
class Node:
    def __init__(self, clause, parents=None, label=None):
        self.clause = clause
```

```
        self.parents = parents if parents else []
        self.label = label

def resolution_with_tree(initial_clauses, goal_clause):
    # standardize apart initial clauses
    clauses_nodes = []
    for idx, c in enumerate(initial_clauses):
        std = standardize_apart_clause(c, idx)
        clauses_nodes.append(Node(std, parents=[] , label=f"C{idx}"))

    # add negated goal as a fresh clause (standardize apart too)
    neg_goal = []
    # goal_clause is a clause list (we take its first literal if single-literal
goal)
    for lit in goal_clause:
        # negate each literal in goal clause (if goal is a single positive literal
user passed)
        neg_goal.append({'neg': not lit['neg'], 'pred': lit['pred'], 'args':
lit['args'][:]})
    neg_goal_std = standardize_apart_clause(neg_goal, len(clauses_nodes))
    goal_node = Node(neg_goal_std, parents=[], label="¬Goal")
    clauses_nodes.append(goal_node)

    # mapping from index -> Node
    idx = len(clauses_nodes)
    seen_clauses = {clause_to_str(n.clause): i for i, n in enumerate(clauses_nodes)}

    # perform breadth-first-ish resolution (pairwise), record parents as indices
    for a_index in range(len(clauses_nodes)):
        pass  # placeholder, we'll use dynamic loop below

    frontier_changed = True
    while True:
        new_added = False
        # iterate pairs over current clauses
        n = len(clauses_nodes)
        pairs = [(i,j) for i in range(n) for j in range(i+1, n)]
        for i,j in pairs:
            c1 = clauses_nodes[i].clause
            c2 = clauses_nodes[j].clause
            resolvent, subs, which = resolve_pair(c1, c2)
            if resolvent is None:
                continue
            s = clause_to_str(resolvent)
            if s in seen_clauses:
                continue
            # add node
            new_node = Node(resolvent, parents=[i, j], label=f"R{idx}")
            clauses_nodes.append(new_node)
            seen_clauses[s] = idx
            new_added = True
            idx += 1
            if resolvent == []:
                # build bottom-up tree node for ⊥
                root = new_node
                return clauses_nodes, seen_clauses, idx-1  # return nodes, map,
index of empty clause node
        if not new_added:
            return clauses_nodes, seen_clauses, None
```

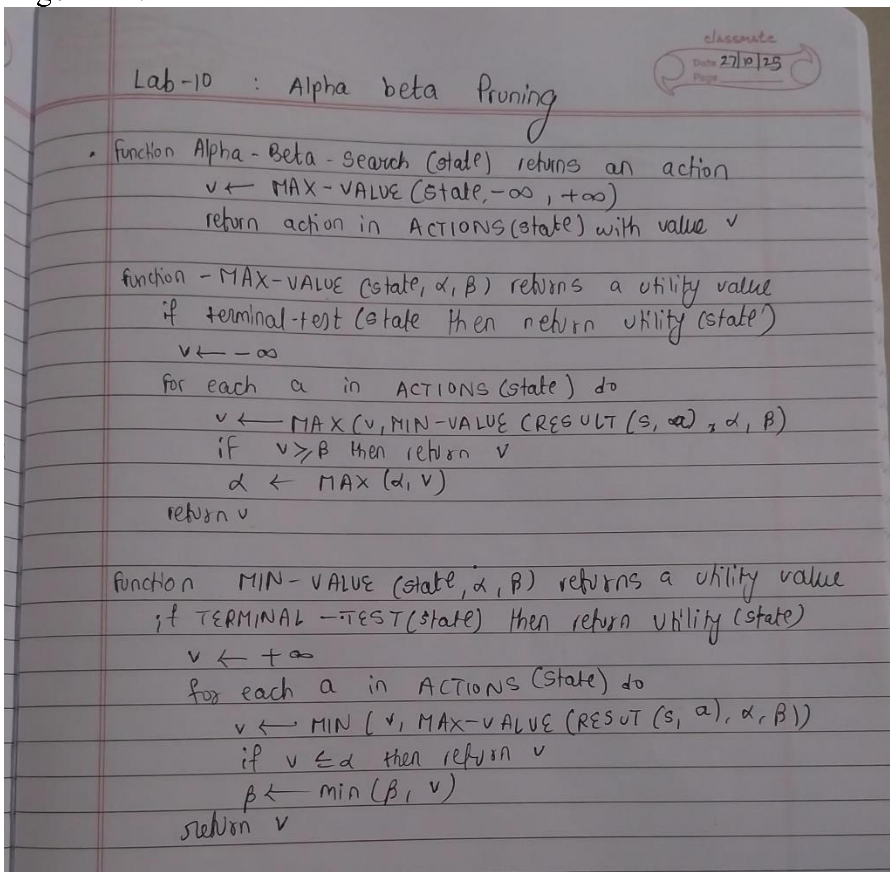
```python
# ----- ASCII print bottom-up (root bottom) -----
def print_bottom_up_tree(nodes, root_index):
    # recursively print node; ensure parents printed above
    def recurse(node_index, prefix="", is_last=True):
        node = nodes[node_index]
        connector = "└── " if is_last else "├── "
        print(prefix + connector + clause_to_str(node.clause))
        # if this node has parents, print them above (parents as children in recursion so they appear above)
        parents = node.parents
        for k, pidx in enumerate(parents):
            recurse(pidx, prefix + ("    " if is_last else "│   "), k == len(parents)-1)
    recurse(root_index, "", True)

# ----- Runner -----
if __name__ == "__main__":
    print("="*70)
    print("FIRST-ORDER LOGIC RESOLUTION SYSTEM (FIXED)")
    print("="*70)
    print("Enter CNF clauses (one per line). End with a blank line.")
    raw = []
    while True:
        try:
            line = input().strip()
        except EOFError:
            break
        if line == "":
            break
        raw.append(line)
    clauses = [ [parse_literal(tok.strip()) for tok in re.split(r"∨", line) ] for line in raw ]

    # read goal
    goal_line = input("\nEnter GOAL clause (single literal form): ").strip()
    goal_clause = [parse_literal(goal_line)]

    nodes, seen_map, root_idx = resolution_with_tree(clauses, goal_clause)
    if root_idx is None:
        print("\nNo empty clause could be derived — goal not entailed by KB.")
    else:
        print("\nDERIVATION TREE (bottom-up):")
        print_bottom_up_tree(nodes, root_idx)
        print("\nResolution complete — ⊥ derived.")
```

```
================================================================================
FIRST-ORDER LOGIC RESOLUTION SYSTEM
================================================================================
Enter CNF clauses (one per line). End with a blank line.
¬human(x) ∨ mortal(x)
human(Socrates)


Enter GOAL clause (single literal form): mortal(Socrates)

DERIVATION TREE (bottom-up):
└── ⊥
    ├── human(Socrates)
    └── ¬human(Socrates)
        ├── ¬human(x_0) ∨ mortal(x_0)
        └── ¬mortal(Socrates)

Resolution complete — ⊥ derived.
```

Program 10

Implement Alpha-Beta Pruning.

Algorithm:


classmate

Date 27/10/25

Lab-10 : Alpha beta Pruning

• function Alpha-Beta-Search (state) returns an action
  v ← MAX-VALUE (state, −∞, +∞)
  return action in ACTIONS(state) with value v

function - MAX-VALUE (state, α, β) returns a utility value
  if terminal-test (state then return utility (state)
  v ← −∞
  for each a in ACTIONS (state) do
    v ← MAX (v, MIN-VALUE (RESULT (s, a), α, β)
    if v ≥ β then return v
    α ← MAX (α, v)
  return v

function MIN-VALUE (state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return utility (state)
  v ← +∞
  for each a in ACTIONS (State) do
    v ← MIN (v, MAX-VALUE (RESUT (s, a), α, β))
    if v ≤ α then return v
    β ← min (β, v)
  return v

Code:

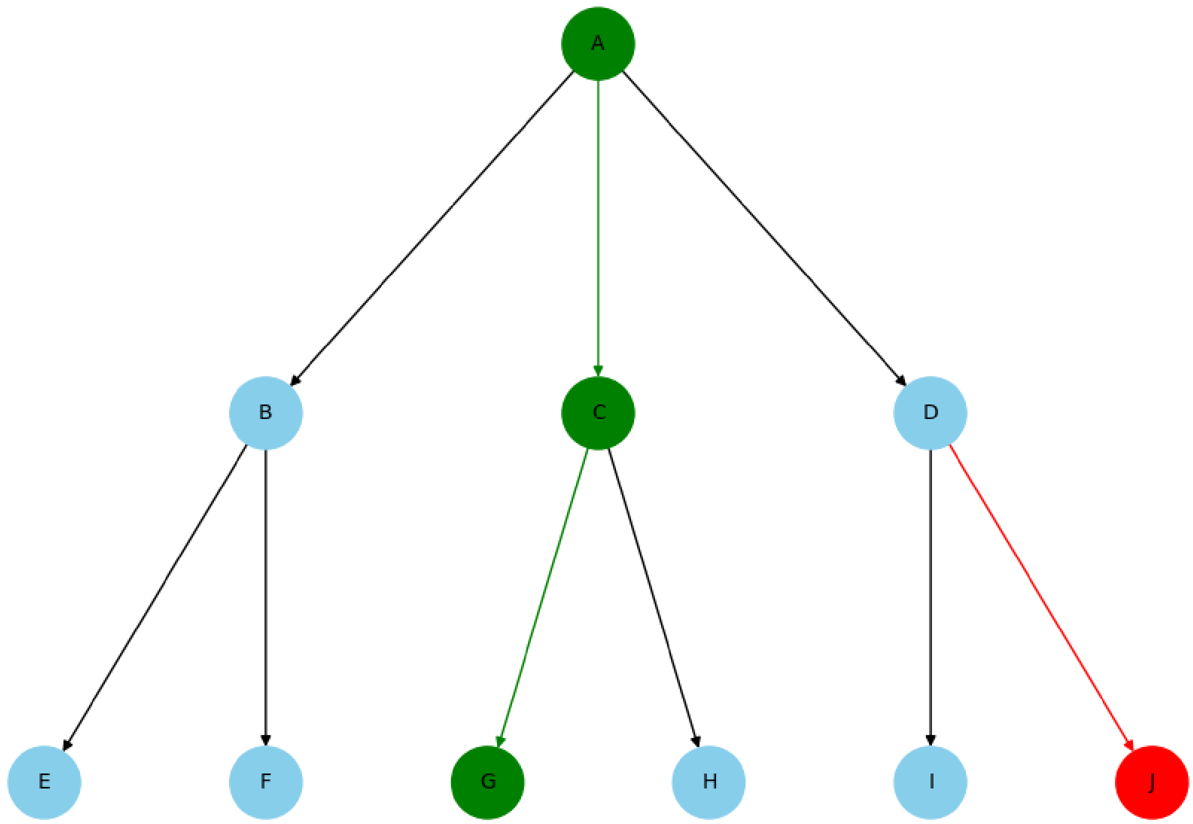
```
import networkx as nx
import matplotlib.pyplot as plt
import math

# --- Alpha-Beta Pruning ---
def alpha_beta(node, depth, alpha, beta, maximizing, tree, values, pruned_nodes,
path):
    # Leaf node
    if depth == 0 or node not in tree:
        return values.get(node, None)

    if maximizing:
        value = -math.inf
        for child in tree[node]:
            val = alpha_beta(child, depth - 1, alpha, beta, False, tree, values,
pruned_nodes, path)
            if val is None:
                continue
            value = max(value, val)
            alpha = max(alpha, value)
            if beta <= alpha:
                # Prune remaining children
                prune_index = tree[node].index(child) + 1
                for c in tree[node][prune_index:]:
```

```
                    pruned_nodes.append(c)
                break
        values[node] = value
        return value
    else:
        value = math.inf
        for child in tree[node]:
            val = alpha_beta(child, depth - 1, alpha, beta, True, tree, values,
pruned_nodes, path)
            if val is None:
                continue
            value = min(value, val)
            beta = min(beta, value)
            if beta <= alpha:
                prune_index = tree[node].index(child) + 1
                for c in tree[node][prune_index:]:
                    pruned_nodes.append(c)
                break
        values[node] = value
        return value


# --- Draw Game Tree ---
def draw_game_tree(G, path, pruned):
    pos = nx.nx_agraph.graphviz_layout(G, prog="dot")
    plt.figure(figsize=(9, 6))

    edge_colors = []
    for (u, v) in G.edges():
        if u in path and v in path:
            edge_colors.append('green')
        elif v in pruned:
            edge_colors.append('red')
        else:
            edge_colors.append('black')

    node_colors = []
    for node in G.nodes():
        if node in path:
            node_colors.append('green')
        elif node in pruned:
            node_colors.append('red')
        else:
            node_colors.append('skyblue')

    nx.draw(
        G, pos, with_labels=True,
        node_color=node_colors,
        edge_color=edge_colors,
        node_size=1200,
        font_size=10
    )

    plt.title("Alpha-Beta Pruning Game Tree\nGreen = Optimal Path | Red = Pruned
Nodes")
    plt.show()
```

```python
# --- Main Program ---
def main():
    tree = {}
    G = nx.DiGraph()

    n = int(input("Enter number of non-leaf nodes: "))
    for _ in range(n):
        parent = input("\nEnter parent node: ").strip()
        children = input("Enter children of " + parent + " (space separated): 
").split()
        tree[parent] = children
        for c in children:
            G.add_edge(parent, c)

    leaf_count = int(input("\nEnter number of leaf nodes: "))
    values = {}
    for _ in range(leaf_count):
        leaf, val = input("Enter leaf node and its value (e.g. E 3): ").split()
        values[leaf] = int(val)

    root = input("\nEnter root node: ").strip()
    depth = int(input("Enter total depth of tree: "))

    pruned_nodes = []
    path = []

    print("\n--------------------------------")
    result = alpha_beta(root, depth, -math.inf, math.inf, True, tree, values, 
pruned_nodes, path)
    print(f"Final Optimal Value: {result}")
    print(f"Pruned Nodes: {pruned_nodes}")
    print("--------------------------------")

    draw_game_tree(G, path=[root, 'C', 'G'], pruned=pruned_nodes)


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

```
Enter number of non-leaf nodes: 4

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

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

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

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

Enter number of leaf nodes: 6
Enter leaf node and its value (e.g. E 3): E 3
Enter leaf node and its value (e.g. E 3): F 5
Enter leaf node and its value (e.g. E 3): G 6
```

Alpha-Beta Pruning Game Tree
Green = Optimal Path | Red = Pruned Nodes




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

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

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