VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Sthavir Kiran Kulkarni (1BM23CS342)

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 **Sthavir Kiran Kulkarni(1BM23CS342),** 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.

Dr. Seema Patil Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Sl. No.	Date	Experiment Title	Page No.
1	18-08-2025	Implement Tic –Tac –Toe Game Implement vacuum cleaner agent	
2	25-08-2025	Implement 8 puzzle problems using Depth First Search (DFS) Implement Iterative deepening search algorithm	
3	01-09-2025	Implement A* search algorithm	
4	08-09-2025	Implement Hill Climbing search algorithm to solve N-Queens problem	
5	15-09-2025	Simulated Annealing to Solve 8-Queens problem	
6	22-09-2025	Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.	
7	13-10-2025	Implement unification in first order logic	
8	13-10-2025	Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.	
9	27-10-2025	Create a knowledge base consisting of first order logic statements and prove the given query using Resolution	
10	27-10-2025	Implement Alpha-Beta Pruning.	

Date __/__/__
Page _____

INDEX

Name: Sthavir. Kiran. Kulkarni
USN: 1BM23CS342
Section: (AI observation) 5F
Subject: AI

Sl No.	Date	Title	Page	Teachers Sign
01	18/08/25	Implement Tic-Tac-Toe Game	01 - 03	10
02	25/08/25	Implement Vacuum Cleaner Problem	04	
03	01/09/25	Implement of BFS & DFS	05 - 08	8/9/25
04	02/09/25	Implement of A*	09 - 12	
05	15/09/25	Implement Hill Climbing Search (N-Queens Problem)	13 - 15	22/9/25
06	22/09/25	Week-6: Propositional Logic	16 - 19	
07	13/10/25	Unification Algorithm	20 - 23	13/10/25
08	13/10/25	First Order Logic	24 - 26	
09	27/10/25	Resolution in FOL	27 (29)	
10	27/10/25	Alpha-Beta Pruning	28	

Github Link : https://github.com/sthabirrr/AI-1BM23CS342

<u>Program 1.1</u>

Implement Tic –Tac –Toe Game

Algorithm:

Date 18/08/25
Page 01

0

X	O	O
O	X	X
X		O

↓0

X	O	O
O	X	X
X	O	O

(Draw)

(r) = 2

Date 18/08/25
Page 02

→ **Algorithm:**

- This game is played between two people.
- One chooses X and the other player choose O
- The game starts with one person / player and then turn by turn they play at their chance and if the player got three consecutive X or O in row / column / diagnol wins.
- If they do not get three consecutive X or O then its a draw.

→ **Output:**

Current board
[, ,]
[, ,]
[, ,]

Player X, enter row (0-2) : 0
enter column (0-2) : 2

[, , X]
[, ,]
[, ,]

Player O, enter row (0-2) : 2
enter column (0-2) : 1

[, , X]
[, ,]
[, O ,]

Player X, enter row (0-2) : 0
enter column (0-2) : 0

[X , , X]
[, ,]
[, O ,]

Player O, enter row (0-2) : 0
enter column (0-2) : 1

[X , O , X]
[, ,]
[, O ,]

Player X, enter row (0-2) : 2
enter column (0-2) : 0

[X , O , X]
[, ,]
[X , O ,]

Player O, enter row (0-2) : 1
enter column (0-2) : 1

[X , O , X]
[, O ,]
[X , O ,]

Player O wins 1
Total moves (cost) : 6

Code:

```
import math

NAME = "Sthavir K"
USN = "1BM23CS342"

# Function to print the board
def print_board(board):
    for row in board:
        print("|".join(row))
    print()

# Function to check if moves are left
def is_moves_left(board):
    for row in board:
        if " " in row:
            return True
    return False

# Function to evaluate board
def evaluate(board):
    # Check rows
    for row in board:
        if row.count(row[0]) == 3 and row[0] != " ":
            return 10 if row[0] == "O" else -10
    # Check cols
    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col] != " ":
            return 10 if board[0][col] == "O" else -10
    # Check diagonals
    if board[0][0] == board[1][1] == board[2][2] != " ":
        return 10 if board[0][0] == "O" else -10
    if board[0][2] == board[1][1] == board[2][0] != " ":
        return 10 if board[0][2] == "O" else -10
    return 0

# Minimax function
def minimax(board, depth, is_max):
    score = evaluate(board)
    if score == 10: return score - depth
    if score == -10: return score + depth
    if not is_moves_left(board): return 0

    if is_max:
        best = -math.inf
        for i in range(3):
```

```
            for j in range(3):
                if board[i][j] == " ":
                    board[i][j] = "O"
                    best = max(best, minimax(board, depth + 1, False))
                    board[i][j] = " "
        return best
    else:
        best = math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == " ":
                    board[i][j] = "X"
                    best = min(best, minimax(board, depth + 1, True))
                    board[i][j] = " "
        return best

# Find best move for computer
def find_best_move(board):
    best_val = -math.inf
    best_move = (-1, -1)
    for i in range(3):
        for j in range(3):
            if board[i][j] == " ":
                board[i][j] = "O"
                move_val = minimax(board, 0, False)
                board[i][j] = " "
                if move_val > best_val:
                    best_move = (i, j)
                    best_val = move_val
    return best_move

# Play game
def play_game():
    board = [[" "]*3 for _ in range(3)]
    print("Tic-Tac-Toe Game! You are X, Computer is O")
    print_board(board)

    for turn in range(9):
        if turn % 2 == 0:  # Human move
            row = int(input("Enter row (0-2): "))
            col = int(input("Enter col (0-2): "))
            if board[row][col] != " ":
                print("Invalid move, try again.")
                continue
            board[row][col] = "X"
        else:  # Computer move
            move = find_best_move(board)
```

```
            board[move[0]][move[1]] = "O"
            print(f"Computer places O at {move}")

        print_board(board)
        score = evaluate(board)
        if score == 10:
            print("Computer Wins!")
            print(f"\nSubmitted by: {NAME}, USN: {USN}")
            return
        elif score == -10:
            print("You Win!")
            print(f"\nSubmitted by: {NAME}, USN: {USN}")
            return

    print("It's a Draw!")
    print(f"\nSubmitted by: {NAME}, USN: {USN}")

play_game()
```

Output:

```
Tic-Tac-Toe Game! You are X, Computer is O
 | |
 | |
 | |

Enter row (0-2): 1
Enter col (0-2): 1
 | |
 |X|
 | |

Computer places O at (0, 0)
O| |
 |X|
 | |

Enter row (0-2): 0
Enter col (0-2): 2
O| |X
 |X|
 | |

Computer places O at (2, 0)
O| |X
 |X|
O| |

Enter row (0-2): 2
Enter col (0-2): 2
O| |X
 |X|
O| |X

Computer places O at (1, 0)
O| |X
O|X|
O| |X

Computer Wins!

Submitted by: Sthavir Kiran Kulkarni, USN: 1BM23CS342
```

Program 1.2

Implement Vacuum Cleaner Agent

Algorithm:

Date 25/05/25
Page 04

LAB - 02

Q) Implement vacuum cleaner agent:

→ Algorithm:

- Check current location and its status
- If current state is dirty, clean it, else move right or left.
- Repeat until both rooms are clean.

→ Output:

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

Code:

```
import random

NAME = "Sthavir K"
USN = "1BM23CS342"

def reflex_vacuum_agent(location, status):
    if status == "Dirty":
        return "Suck"
    elif location == "A":
        return "Right"
    elif location == "B":
        return "Left"

def vacuum_world():
    locations = {"A": random.choice(["Clean", "Dirty"]),
                 "B": random.choice(["Clean", "Dirty"])}
    location = random.choice(["A", "B"])

    print("Initial State:", locations, "| Vacuum at:", location)

    steps = 5
    for _ in range(steps):
        status = locations[location]
        action = reflex_vacuum_agent(location, status)
        print(f"Vacuum at {location} | Status: {status} -> Action: {action}")

        if action == "Suck":
            locations[location] = "Clean"
        elif action == "Right":
            location = "B"
        elif action == "Left":

        print("World State:", locations)

    print("\nFinal State:", locations)
    print(f"\nSubmitted by: {NAME}, USN: {USN}")

vacuum_world()
```

Output:

```
•••  Initial State: {'A': 'Dirty', 'B': 'Dirty'} | Vacuum at: A
     Vacuum at A | Status: Dirty -> Action: Suck
     World State: {'A': 'Clean', 'B': 'Dirty'}
     Vacuum at A | Status: Clean -> Action: Right
     World State: {'A': 'Clean', 'B': 'Dirty'}
     Vacuum at B | Status: Dirty -> Action: Suck
     World State: {'A': 'Clean', 'B': 'Clean'}
     Vacuum at B | Status: Clean -> Action: Left
     World State: {'A': 'Clean', 'B': 'Clean'}
     Vacuum at A | Status: Clean -> Action: Right
     World State: {'A': 'Clean', 'B': 'Clean'}

     Final State: {'A': 'Clean', 'B': 'Clean'}

     Submitted by: Sthavir K, USN: 1BM23CS342
```

Program 2

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

Date 01/09/25
Page 05

LAB - 03

Q1) BFS without Heuristic Approach:

→ Algorithm:

- Put the initial board into a queue
- Note the position of the blank space
- From the blank find all possible moves (up, down, left, right)
- Create new boards by making the moves
- Repeat until reaching goal board

→ Output:

Enter initial state: 1 2 3 _ 4 6 7 5 8
Enter goal state: 1 2 3 4 5 6 7 8 _

Minimum cost: 3

Steps:

1	2	3	→	1	2	3		
_	4	6		4	_	6		
7	5	8		7	5	8		
				↓				
				1	2	3	1	2 3
				4	5	6	4	5 6
				7	_	8	7	8 _
				↓				

Q1) Using BFS

→ Solution:

2	8	3
1		4
7	6	5

2		
1	8	
7	6	

	2	3
1	8	4
7	6	5

1	2	
	8	
7	6	

Q) Using BFS solve 8 puzzle without heuristic approach:

2	8	3
1	6	4
7		5

Initial

1	2	3
8		4
7	6	5

Goal

→ Solution:

1	2	3
8		4
7	6	5

✓

1	2	3
7	8	4
	6	5

→ **Algorithm (DFS):**

- Start with the initial puzzle state
- Put this state into a stack
- Keep a set of visited states to avoid repeating the same board
- While the stack is not empty:
  - → Take the top state from the stack
  - → If it matches the goal state, stop.

→ **Output:**

For BFS:

Number of steps : 5
Move empty space from (2,1) to (1,1)
Move empty space from (1,1) to (0,1)
" (0,1) to (0,0)
" (0,0) to (1,0)
Move empty space from (1,0) to (1,1)
Total no. of visited states : 62

For DFS :

→

Number of steps: 15
Total no. of visits: 40419

Code (BFS):

```
from collections import deque


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


MOVES = {
    'Up':    (-1, 0),
    'Down':  (1, 0),
    'Left':  (0, -1),
    'Right': (0, 1)
}

def index_to_pos(index):
    return index // 3, index % 3

def pos_to_index(row, col):
    return row * 3 + col

def get_neighbors(state):
    neighbors = []
    zero_index = state.index(0)
    zero_row, zero_col = index_to_pos(zero_index)

    for move, (dr, dc) in MOVES.items():
        new_row, new_col = zero_row + dr, zero_col + dc
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_zero_index = pos_to_index(new_row, new_col)
            new_state = list(state)

            new_state[zero_index], new_state[new_zero_index] = new_state[new_zero_index],
new_state[zero_index]
            neighbors.append((tuple(new_state), move))
    return neighbors

def bfs(start_state, max_depth=50):
    queue = deque([(start_state, [], [start_state])])
    visited = set([start_state])

    while queue:
        state, path_moves, path_states = queue.popleft()

        if state == GOAL_STATE:
            return path_moves, path_states

        if len(path_moves) < max_depth:
            for neighbor, move in get_neighbors(state):
                if neighbor not in visited:
```

```
                visited.add(neighbor)
                queue.append((neighbor, path_moves + [move], path_states + [neighbor]))
    return None, None

def print_puzzle(state):
    for i in range(0, 9, 3):
        print(state[i:i+3])
    print()




name = input("Enter your name: ")
usn = input("Enter your USN: ")


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

print(f"\nSolver started for {name} (USN: {usn})")
print("\nStart State:")
print_puzzle(start_state)

moves, states = bfs(start_state, max_depth=50)

if moves is not None:
    print(f"Solution found in {len(moves)} moves!\n")

    print("States passed through from start to goal:")
    for i, state in enumerate(states):
        print(f"Step {i}:")
        print_puzzle(state)

    print("Moves to solve the puzzle:")
    print(moves)
else:
    print("No solution found within the depth limit.")
```

Code (DFS):

```
from collections import deque

GOAL_STATE = (1, 2, 3,
              8, 0, 4,
              7, 6, 5)  # 0 represents the blank tile


MOVES = {
    'Up':    (-1, 0),
    'Down':  (1, 0),
```

```
    'Left':  (0, -1),
    'Right': (0, 1)
}

def index_to_pos(index):
    return index // 3, index % 3

def pos_to_index(row, col):
    return row * 3 + col

def get_neighbors(state):
    neighbors = []
    zero_index = state.index(0)
    zero_row, zero_col = index_to_pos(zero_index)

    for move, (dr, dc) in MOVES.items():
        new_row, new_col = zero_row + dr, zero_col + dc
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_zero_index = pos_to_index(new_row, new_col)
            new_state = list(state)

            new_state[zero_index], new_state[new_zero_index] = new_state[new_zero_index],
new_state[zero_index]
            neighbors.append((tuple(new_state), move))
    return neighbors

def dfs(start_state, max_depth=50):
    stack = [(start_state, [], [start_state], 0)]
    visited = set()

    while stack:
        state, path_moves, path_states, depth = stack.pop()

        if state == GOAL_STATE:
            return path_moves, path_states

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

        if depth < max_depth:
            for neighbor, move in get_neighbors(state):
                if neighbor not in visited:
                    stack.append((neighbor, path_moves + [move], path_states + [neighbor], depth + 1))
    return None, None

def print_puzzle(state):
    for i in range(0, 9, 3):
        print(state[i:i+3])
    print()
```

```
name = input("Enter your name: ")
usn = input("Enter your USN: ")


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

print(f"\nSolver started for {name} (USN: {usn})")
print("\nStart State:")
print_puzzle(start_state)

moves, states = dfs(start_state, max_depth=30)

if moves is not None:
    print(f"Solution found in {len(moves)} moves!\n")

    print("States passed through from start to goal:")
    for i, state in enumerate(states):
        print(f"Step {i}:")
        print_puzzle(state)

    print("Moves to solve the puzzle:")
    print(moves)
else:
    print("No solution found within the depth limit.")
```

Output (BFS):

```
...  Enter your name: Sthavir K
     Enter your USN: 1BM23CS342

     Solver started for Sthavir K (USN: 1BM23CS342)

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

     Solution found in 5 moves!

     States passed through from start to goal:
     Step 0:
     (2, 8, 3)
     (1, 6, 4)
     (7, 0, 5)

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

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

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

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

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

     Moves to solve the puzzle:
     ['Up', 'Up', 'Left', 'Down', 'Right']
```

Output (DFS):

```
...  Enter your name: Sthavir K
     Enter your USN: 1BM23CS342

     Solver started for Sthavir K (USN: 1BM23CS342)

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

     Solution found in 29 moves!

     States passed through from start to goal:
     Step 0:
     (2, 8, 3)
     (1, 6, 4)
     (7, 0, 5)

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

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

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

     Step 4:
```

Program 3

Implement A* search algorithm

	2	3
1	8	4
7	6	5

$g = 3$

$m = 3$

$f(n) = 3 + 3 = 6$

1	2	3
	8	4
7	6	5

$g = 4$

$m = 2$

$f(n) = 4 + 2 = 6$

1	2	3
8		4
7	6	5

$g = 5$

$m = 0$

$f(n) = 5 + 0 = 5$

$\therefore f(n) = 5 + 0 = 5$

→ **Algorithm:**

Step 1: Start with a jumbled puzzle and input the goal state

Step 2: Score each state: no. of misplaced tiles + levels

Step 3: Always pick the lowest score to explore next

Step 4: Make all possible moves from it

Step 5: Stop when you reach the goal state.

→ **Manhattan Distance:**

1	5	8
3	2	
4	6	7

initial

	1	2
3	4	5
6	7	8

Goal

1	5	8
3	2	
4	6	7

1	5	
3	2	8
4	6	7

1 2 3 4 5 6 7 8

$0+1+3+1+1+2+2+2$

$= 12$

$f(n) = 12+1 = 13$

1	5	8
3		2
4	6	7

1 2 3 4 5 6 7 8

$0+2+3+1+1+2+2+3$

$= 14$

$f(n) = 14+1 = 15$

1	5	8
3	2	7
4	6	

1 2 3 4 5 6 7 8

$0+1+3+1+1+2+3+3$

$= 14$

$f(n) = 14+1 = 15$

1		5
3	2	8
4	6	9

1 2 3 4 5 6 7 8

$0+1+3+1+2+2+2+2$

$= 13$

$f(n) = 13+2 = 15$

1	2	5
3		8
4	6	7

1 2 3 4 5 6 7 8

$0+0+3+1+2+2+2+2$

$= 12$

$f(n) = 12+3 = 15$

→ **Algorithm:**

- Initialize the open list with the start node and set all scores.
- Select the node from the open list with the lowest score.
- Check if the selected node is the goal, if yes, return the path.
- Generate valid neighbours, calculate their scores using Manhattan distance and update the open list.
- Repeat steps 2-4 until a goal is found or the open list is empty.

Code (Misplaced Tiles):

```
import copy
import heapq


NAME = "Sthavir K"
USN = "1BM23CS342"

print(f"Name: {NAME}")
print(f"USN : {USN}\n")


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

def heuristic_misplaced(current_state):
    misplaced = 0
    for i in range(3):
        for j in range(3):
            if current_state[i][j] != 0 and current_state[i][j] != goal_state[i][j]:
                misplaced += 1
    return misplaced

def is_goal(state):
    return state == goal_state

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

def get_neighbors(state):
    neighbors = []
    x, y = find_blank(state)
    directions = [(-1,0), (1,0), (0,-1), (0,1)]
    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny <3:
            new_state = copy.deepcopy(state)
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(new_state)
    return neighbors

def a_star_misplaced(initial_state):
```

```
    open_list = []
    heapq.heappush(open_list, (heuristic_misplaced(initial_state), 0, initial_state, []))
    visited = set()

    while open_list:
        heuristic, g, current_state, path = heapq.heappop(open_list)
        state_id = str(current_state)
        if state_id in visited:
            continue
        visited.add(state_id)

        if is_goal(current_state):
            return path + [current_state]

        for neighbor in get_neighbors(current_state):
            heapq.heappush(open_list, (heuristic_misplaced(neighbor) + g + 1, g + 1, neighbor, path +
[current_state]))
    return None

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

solution = a_star_misplaced(initial_state)

print("\nSolution Steps:\n")
for step in solution:
    for row in step:
        print(row)
    print()
```

Code (Manhattan Distance):

```
import copy
import heapq

NAME = "Sthavir K"
USN = "1BM23CS342"

print(f"Name: {NAME}")
print(f"USN : {USN}\n")

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

def heuristic_manhattan(current_state):
```

```
    distance = 0
    for i in range(3):
        for j in range(3):
            value = current_state[i][j]
            if value != 0:
                goal_x, goal_y = divmod(value - 1, 3)
                distance += abs(i - goal_x) + abs(j - goal_y)
    return distance

def is_goal(state):
    return state == goal_state

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

def get_neighbors(state):
    neighbors = []
    x, y = find_blank(state)
    directions = [(-1,0), (1,0), (0,-1), (0,1)]
    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny <3:
            new_state = copy.deepcopy(state)
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(new_state)
    return neighbors

def a_star_manhattan(initial_state):
    open_list = []
    heapq.heappush(open_list, (heuristic_manhattan(initial_state), 0, initial_state, []))
    visited = set()

    while open_list:
        heuristic, g, current_state, path = heapq.heappop(open_list)
        state_id = str(current_state)
        if state_id in visited:
            continue
        visited.add(state_id)

        if is_goal(current_state):
            return path + [current_state]

        for neighbor in get_neighbors(current_state):
            heapq.heappush(open_list, (heuristic_manhattan(neighbor) + g + 1, g + 1, neighbor, path +
```

```
[current_state]))
    return None

# Initial state example
initial_state = [[2, 8, 3],
                 [1, 6, 4],
                 [7, 0, 5]]

solution = a_star_manhattan(initial_state)

print("\nSolution Steps:\n")
for step in solution:
    for row in step:
        print(row)
    print()
```

Output (Misplaced Tiles):

```
Name: Sthavir K
USN : 1BM23CS342


Solution Steps:

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

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

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

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

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

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

Output (Manhattan Distance):

```
Name: Sthavir K
USN : 1BM23CS342


Solution Steps:

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

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

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

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

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

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

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Date 15/09/25
Page 13

LAB - 05

Q) Implement Hill Climbing search algorithm to solve N-Queens problem:

→ Algorithm:

S1 - Start

S2 - Define current state as initial state

S3 - Loop until goal state is reached, or no more operation is applied

- Apply an operand

- Compare new state with goal state

- Quit

- Evaluate newstate within its hierarchy

- Compare

- If newstate is close to goal state then update current state

S4 - Display

S5 - End

→ 4-Q

→ Outp

Total

State :

State :

State :

→ **4 - Queens Problem:**

h = 0

		Q	
Q			
			Q
	Q		

Initial State

	Q		
Q			
			Q
	Q		

h = 3

	Q		
		Q	
			Q
	Q		

h = 5

	Q		
			Q
Q			
		Q	

h = 0

		Q	
Q			
			Q
Q			

h = 2

Q			
		Q	
	Q		
			Q

h = 4

→ **Output:**

Total possible states: 256

State: (0,0,0,0), h = 6
State: (0,0,0,1), h = 4
State: (0,0,0,2), h = 4

Code:

```
import random


def calculate_cost(state):
    cost = 0
    n = len(state)
    for i in range(n):
        for j in range(i+1, n):
            if state[i] == state[j]:
                cost += 1
            if abs(state[i] - state[j]) == abs(i - j):
                cost += 1
    return cost

def random_state(n):
    return [random.randint(0, n-1) for _ in range(n)]


def get_best_neighbor(state):
    n = len(state)
    best_state = state[:]
    best_cost = calculate_cost(state)

    for col in range(n):
        for row in range(n):
            if state[col] != row:
                new_state = state[:]
                new_state[col] = row
                new_cost = calculate_cost(new_state)
                if new_cost < best_cost:
                    best_cost = new_cost
                    best_state = new_state
    return best_state, best_cost


def hill_climbing(n):
    current = random_state(n)
    current_cost = calculate_cost(current)

    while True:
        neighbor, neighbor_cost = get_best_neighbor(current)

        if neighbor_cost < current_cost:
            current, current_cost = neighbor, neighbor_cost
        else:
```

```
            break

    return current, current_cost

solution, cost = hill_climbing(4)

print("Name: Sthavir K")
print("USN : 1BM23CS342")
print("\nHill Climbing for 4-Queens Problem")
print("Final State (row positions per column):", solution)
print("Final Cost (0 means solved):", cost)
if cost == 0:
    print("Solution Found: Queens are safe!")
else:
    print(" Stuck in Local Optimum (Not a solution)")
```

Output:

```
•••  Name: Sthavir K
     USN : 1BM23CS342

     Hill Climbing for 4-Queens Problem
     Final State (row positions per column): [0, 2, 3, 1]
     Final Cost (0 means solved): 1
      Stuck in Local Optimum (Not a solution)
```

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:

Date 15/09/25
Page 15

→ Simulated Annealing:

- Algorithm:

1) Current ← initial state
2) T ← a large positive value
3) while T > 0 do
4) next ← a random neighbour of current
5) ΔE ← current.cost - next.cost
6) if ΔE > 0 then
7) current ← next
8) else current ← next with probability $P = e^{\Delta E / T}$
9) end if decrease T
10) end while.
11) return current

- Output:

The best position found is: [5, 1, 1, 4, 7, 5, 0, 2]
The number of queens that are not attracting each other: 5

15/9/25

Code:

```
import random
import math


def calculate_cost(state):
    cost = 0
    n = len(state)
    for i in range(n):
        for j in range(i+1, n):
            if state[i] == state[j]:
                cost += 1
            if abs(state[i] - state[j]) == abs(i - j):
                cost += 1
    return cost


def random_state(n):
    return [random.randint(0, n-1) for _ in range(n)]


def get_neighbor(state):
    n = len(state)
    neighbor = state[:]
    col = random.randint(0, n-1)
    row = random.randint(0, n-1)
    neighbor[col] = row
    return neighbor


def simulated_annealing(n, max_steps=100000, start_temp=100, cooling_rate=0.99):
    current = random_state(n)
    current_cost = calculate_cost(current)
    T = start_temp

    for step in range(max_steps):
        if current_cost == 0:
            break
        neighbor = get_neighbor(current)
        neighbor_cost = calculate_cost(neighbor)
        delta = neighbor_cost - current_cost

        if delta < 0 or random.uniform(0, 1) < math.exp(-delta / T):
            current, current_cost = neighbor, neighbor_cost

        T = T * cooling_rate
```

```
        if T < 1e-5:
            break

    return current, current_cost


solution, cost = simulated_annealing(8)

print("Name: Sthavir K")
print("USN : 1BM23CS342")
print("\nSimulated Annealing for 8-Queens Problem")
print("Final State (row positions per column):", solution)
print("Final Cost (0 means solved):", cost)
if cost == 0:
    print("Solution Found: Queens are safe!")
else:
    print("Stuck in Local Optimum (Not a solution)")
```

Output:

```
•••  Name: Sthavir K
     USN : 1BM23CS342

     Simulated Annealing for 8-Queens Problem
     Final State (row positions per column): [4, 2, 0, 5, 7, 1, 3, 6]
     Final Cost (0 means solved): 0
     Solution Found: Queens are safe!
```

Program 6

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

Algorithm:

Date 22/09/25
Page 16

LAB - 06

Propositional Logic

→ Implementation of Truth-Table enumeration algorithm for deciding propositional entailment.
i.e. Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.

- Propositional logic : Semantics

- Truth Tables for connectives :

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

- Implication Operator :

P	Q	P⇒Q
false	false	true
false	true	true
true	false	false
true	true	true

- Propositional Inference : Enumeration Method

- Example :

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

- Checking that KB ⊨ α :

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

→ **Algorithm:**

Step 1: Extract Symbols:

- Collect all propositional symbols (variables like P, Q, R, ...) from both KB and Query.
- Example: If KB = (P → Q) ∧ P and Query = Q → symbols = {P, Q}

Step 2: Generate Models:

- A model is a truth assignment (True / False) for all symbols
- For n symbols, there are 2^n models.
- Example: For {P, Q}, models =
  - {P = True, Q = True}
  - {P = True, Q = False}
  - {P = False, Q = True}
  - {P = False, Q = False}

Step 3: Check each model:

- For each model:
  - Evaluate KB under this model.
  - If KB is True:
    - Check if Query (a) is also True.
    - If Query is False → Entailment Fails.

Step 4: Final Decision:

- If all models where KB is True also make Query True, then KB ⊨ α (KB entails query)
- Otherwise, KB does not entail α.

→ <u>Output:</u>

Symbols found: ['Q', 'P']

Generating all possible models...

Model: {'Q': True, 'P': True}
KB = True, Query = True
Model: {'Q': True, 'P': False}
KB = False, Query = True
Model: {'Q': False, 'P': True}
KB = False, Query = False
Model: {'Q': False, 'P': False}
KB = False, Query = False

✓ All models checked. KB entails Query.

Final Result: Does KB entail Query? → True

Code:

```
import itertools

def pl_true(expr, model):
    if isinstance(expr, str):
        return model[expr]
    elif isinstance(expr, tuple):
        op = expr[0]
        if op == "not":
            return not pl_true(expr[1], model)
        elif op == "and":
            return pl_true(expr[1], model) and pl_true(expr[2], model)
        elif op == "or":
            return pl_true(expr[1], model) or pl_true(expr[2], model)
        elif op == "implies":
            return (not pl_true(expr[1], model)) or pl_true(expr[2], model)
    return False

def get_symbols(expr):
    if isinstance(expr, str):
        return {expr}
    elif isinstance(expr, tuple):
        return get_symbols(expr[1]) | (get_symbols(expr[2]) if len(expr) > 2 else set())
    return set()

def tt_entails_print(KB, query):
    symbols = list(get_symbols(KB) | get_symbols(query))
    all_models = list(itertools.product([True, False], repeat=len(symbols)))

    entailment = True
    print("\nTruth Table Evaluation:")
    print("-" * 50)
    print("Model".ljust(20), "KB".ljust(10), "Query".ljust(10))
    print("-" * 50)

    for values in all_models:
        model = dict(zip(symbols, values))
        kb_val = pl_true(KB, model)
        q_val = pl_true(query, model)

        print(str(model).ljust(20), str(kb_val).ljust(10), str(q_val).ljust(10))

        if kb_val and not q_val:
            entailment = False

    print("-" * 50)
```

```
    return entailment

KB = ("and", ("implies", "P", "Q"), "P")
query = "Q"

result = tt_entails_print(KB, query)

print("\nName: Sthavir K")
print("USN : 1BM23CS342")
print("\nKnowledge Base: (P → Q) and (P)")
print("Query: Q")
print("Does KB entail Query? :", "YES " if result else "NO ")
```

Output:

```
...
    Truth Table Evaluation:
    ------------------------------------------------
    Model                KB         Query
    ------------------------------------------------
    {'Q': True, 'P': True} True       True
    {'Q': True, 'P': False} False      True
    {'Q': False, 'P': True} False      False
    {'Q': False, 'P': False} False      False
    ------------------------------------------------

    Name: Sthavir K
    USN : 1BM23CS342

    Knowledge Base: (P → Q) and (P)
    Query: Q
    Does KB entail Query? : YES
```

Program 7

Implement unification in first order logic:

Algorithm:

Date 13/10/25
Page 20

Lab - 07

Unification Algorithm

→ Algorithm: Unify (Ψ_1, Ψ_2)

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

  a) If Ψ_1 or Ψ_2 are identical, the 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 { (Ψ_2 / Ψ_1) }.
  c) Else if Ψ_2 is a variable,
     a. If Ψ_2 occurs in Ψ_1 then return failure
     b. Else return { (Ψ_1 / Ψ_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 and Ψ_2 have a different number of arguments, the return FAILURE.

- Step 4: Set Substitute set (SUBST) to NIL.

- Step 5: For i = 1 to the number of elements in Ψ_1.

  a) Call Unify function with the ith element of Ψ_1 and ith element of Ψ_2, and put the result into S.
  b) If S = failure then returns Failure
  c) If S ≠ NIL then do,
     a. Apply S to the remainder of both L1 & L2
     b. SUBST = APPEND (S, SUBST)

- Step 6 : Return SUBST

→ Solve the following in observation book :

- Find Most General Unifier (MGU) of { p(b, X, f(g(Z)) and p(z, f(Y), f(Y))

→ We are given two terms :
1) (p(b, X, f(g(Z)))
2) (p(z, f(Y), f(Y)))

First argument : b and z (as they are not the same, we need to unify b with z, which gives the substitution b → z)

Second argument: X and f(Y) (we need to unify X with f(Y), so we get the substitution X → f(Y))

Third argument : f(g(Z)) and f(Y) (to unify f(g(Z) with f(Y), we need to unify g(Z) with Y, which gives the substitution Y → g(Z))

Final Substitution:

{ b → z, X → f(g(Z)), Y → g(Z) }

- Find Most General Unifier (MGU) of { Q(a, g(x, a), f(y)) and Q(a, g(f(b), a), x) }

→ First argument : a and a → no change
Second Argument: x → f(b)
Third Argument: y → f(b)

Result: { x →

- Find MGU of

→ First argument
Second argument

Result : Unifica

- Unify { prin

→ Constant 11

Result : Unifi

- Unify { know

→ First Argument
Second Argum

Result: { y →

Result: $\{x \to f(b),\ y \to f(b)\}$

- Find MGU of $\{p(f(a), g(Y)),\ p(X,X)\}$

→ First argument : $f(a)$ and $X \to X \to f(a)$
Second argument : $g(Y)$ and $f(a)$ → unification fails (no solution)

Result : Unification Fails.

- Unify {prime(11) and prime(y)}

→ Constant 11 and variable y cannot unify.

Result : Unification fails.

- Unify {knows(John, x), knows(y, mother(y))}

→ First Argument: John and $y \to y \to$ John
Second Argument: x and mother(y) → x → mother(John)

Result: $\{y \to$ John, $x \to$ mother(John)$\}$

Code:

```
import json

# --- Helper Functions for Term Manipulation ---

def is_variable(term):
    """Checks if a term is a variable (a single capital letter string)."""
    return isinstance(term, str) and len(term) == 1 and 'A' <= term[0] <= 'Z'

def occurs_check(variable, term, sigma):
    """
    Checks if 'variable' occurs anywhere in 'term' under the current substitution 'sigma'.
    This prevents infinite recursion (e.g., unifying X with f(X)).
    """
    term = apply_substitution(term, sigma) # Check the substituted term

    if term == variable:
        return True

    # If the term is a list (function/predicate), check its arguments recursively
    if isinstance(term, list):
        for arg in term[1:]:
            if occurs_check(variable, arg, sigma):
                return True

    return False

def apply_substitution(term, sigma):
    """
    Applies the current substitution 'sigma' to a 'term' recursively.
    """
    if is_variable(term):
        # If the variable is bound in sigma, apply the binding
        if term in sigma:
            # Recursively apply the rest of the substitutions to the binding's value
            # This is critical for chains like X/f(Y), Y/a -> X/f(a)
            return apply_substitution(sigma[term], sigma)
        return term

    if isinstance(term, list):
        # Apply substitution to the arguments of the function/predicate
        new_term = [term[0]] # Keep the function/predicate symbol
        for arg in term[1:]:
            new_term.append(apply_substitution(arg, sigma))
        return new_term
```

```
    # Term is a constant or an unhandled type, return as is
    return term

def term_to_string(term):
    """
    Converts the internal list representation of a term into standard logic notation string.
    e.g., ['f', 'Y'] -> "f(Y)"
    """
    if isinstance(term, str):
        return term

    if isinstance(term, list):
        # Term is a function or predicate
        symbol = term[0]
        args = [term_to_string(arg) for arg in term[1:]]
        return f"{symbol}({', '.join(args)})"

    return str(term)


# --- Main Unification Function ---

def unify(term1, term2):
    """
    Implements the Unification Algorithm to find the MGU for term1 and term2.
    Returns the MGU as a dictionary or None if unification fails.
    """
    # Initialize the substitution set (MGU)
    sigma = {}

    # Initialize the list of pairs to resolve (the difference set)
    diff_set = [[term1, term2]]

    print(f"--- Unification Process Started ---")
    print(f"Initial Terms:")
    print(f"L1: {term_to_string(term1)}")
    print(f"L2: {term_to_string(term2)}")
    print("-" * 35)

    while diff_set:
        # Pop the current pair of terms to unify
        t1, t2 = diff_set.pop(0)

        # 1. Apply the current MGU to the terms before comparison
        t1_prime = apply_substitution(t1, sigma)
        t2_prime = apply_substitution(t2, sigma)
```

```
    print(f"Attempting to unify: {term_to_string(t1_prime)} vs {term_to_string(t2_prime)}")


    # 2. Check if terms are identical
    if t1_prime == t2_prime:
        print(f"  -> Identical. Current MGU: {term_to_string(sigma)}")
        continue

    # 3. Handle Variable-Term unification
    if is_variable(t1_prime):
        var, term = t1_prime, t2_prime
    elif is_variable(t2_prime):
        var, term = t2_prime, t1_prime
    else:
        var, term = None, None

    if var:
        # Check if term is a variable, and if so, don't bind V/V
        if is_variable(term):
            print(f"  -> Both are variables. Skipping {var} / {term}")
            # Ensure they are added back if not identical (which is caught by step 2).
            # If V1 != V2, we add V1/V2 or V2/V1 to sigma. Since step 2 handles V/V, this means V1
!= V2 here.
            if var != term:
                sigma[var] = term
                print(f"  -> Variable binding added: {var} / {term_to_string(term)}. New MGU:
{term_to_string(sigma)}")
        # Occurs Check: Fail if the variable occurs in the term it's being bound to
        elif occurs_check(var, term, sigma):
            print(f"  -> OCCURS CHECK FAILURE: Variable {var} occurs in
{term_to_string(term)}")
            return None

        # Create a new substitution {var / term}
        else:
            sigma[var] = term
            print(f"  -> Variable binding added: {var} / {term_to_string(term)}. New MGU:
{term_to_string(sigma)}")

    # 4. Handle Complex Term (Function/Predicate) unification
    elif isinstance(t1_prime, list) and isinstance(t2_prime, list):
        # Check functor/predicate symbol and arity (number of arguments)
        if t1_prime[0] != t2_prime[0] or len(t1_prime) != len(t2_prime):
            print(f"  -> FUNCTOR/ARITY MISMATCH: {t1_prime[0]} != {t2_prime[0]} or arity
mismatch.")
            return None
```

```
            # Add corresponding arguments to the difference set
            # Start from index 1 (after the symbol)
            for arg1, arg2 in zip(t1_prime[1:], t2_prime[1:]):
                diff_set.append([arg1, arg2])
            print(f"  -> Complex terms matched. Adding arguments to difference set.")

        # 5. Handle Constant-Constant or other mismatches (Fail)
        else:
            print(f"  -> TYPE/CONSTANT MISMATCH: {term_to_string(t1_prime)} and
{term_to_string(t2_prime)} cannot be unified.")
            return None

    print("-" * 35)
    print("--- Unification Successful ---")

    # Final cleanup to ensure all bindings are fully resolved
    final_mgu = {k: apply_substitution(v, sigma) for k, v in sigma.items()}
    return final_mgu

# --- Define the Input Terms ---

# L1 = Q(a, g(X, a), f(Y))
literal1 = ['Q', 'a', ['g', 'X', 'a'], ['f', 'Y']]

# L2 = Q(a, g(f(b), a), X)
literal2 = ['Q', 'a', ['g', ['f', 'b'], 'a'], 'X']

# --- Run the Unification ---

mgu_result = unify(literal1, literal2)

if mgu_result is not None:
    print("\n[ Final MGU Result ]")

    # Format the final MGU for display using the new helper function
    clean_mgu = {k: term_to_string(v) for k, v in mgu_result.items()}
    final_output = ', '.join([f"{k} / {v}" for k, v in clean_mgu.items()])
    print(f"Final MGU: {{ {final_output} }}")

    # --- Verification ---
    print("\n[ Verification ]")
    unified_l1 = apply_substitution(literal1, mgu_result)
    unified_l2 = apply_substitution(literal2, mgu_result)

    print(f"L1 after MGU: {term_to_string(unified_l1)}")
    print(f"L2 after MGU: {term_to_string(unified_l2)}")
```

```
    if unified_l1 == unified_l2:
        print("-> SUCCESS: L1 and L2 are identical after applying the MGU.")
    else:
        print("-> ERROR: Unification failed verification.")
else:
    print("\nUnification FAILED.")

print("Sthavir K 1BM23CS342")
```

Output:

```
--- Unification Process Started ---
Initial Terms:
L1: Q(a, g(X, a), f(Y))
L2: Q(a, g(f(b), a), X)
----------------------------------
Attempting to unify: Q(a, g(X, a), f(Y)) vs Q(a, g(f(b), a), X)
  -> Complex terms matched. Adding arguments to difference set.
Attempting to unify: a vs a
  -> Identical. Current MGU: {}
Attempting to unify: g(X, a) vs g(f(b), a)
  -> Complex terms matched. Adding arguments to difference set.
Attempting to unify: f(Y) vs X
  -> Variable binding added: X / f(Y). New MGU: {'X': ['f', 'Y']}
Attempting to unify: f(Y) vs f(b)
  -> Complex terms matched. Adding arguments to difference set.
Attempting to unify: a vs a
  -> Identical. Current MGU: {'X': ['f', 'Y']}
Attempting to unify: Y vs b
  -> Variable binding added: Y / b. New MGU: {'X': ['f', 'Y'], 'Y': 'b'}
----------------------------------
--- Unification Successful ---

[ Final MGU Result ]
Final MGU: { X / f(b), Y / b }

[ Verification ]
L1 after MGU: Q(a, g(f(b), a), f(b))
L2 after MGU: Q(a, g(f(b), a), f(b))
-> SUCCESS: L1 and L2 are identical after applying the MGU.
Sthavir K 1BM23CS342
```

Program 8

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

Algorithm:

Date 13/10/25
Page 24

LAB - 08

First Order Logic

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

I

Premises → Conclusion

$P \Rightarrow Q$

$L \wedge M \Rightarrow P$

$B \wedge L \Rightarrow M$

$A \wedge P \Rightarrow L$

$A \wedge B \Rightarrow L$

} rules

A

B

} facts

Prove Q

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

Prove that "West is Criminal".

Rules & Facts:

1) $\forall x, y, z$ American$(x) \wedge$ weapon$(y) \wedge$ sells$(x, y, z) \wedge$ Hostile(z) = criminal(x)
2) $\forall x$ missile$(x) \wedge$ owns$(Nono, x)$ = sells$(West, x, Nono)$
3) $\forall x$ Enemy$(x, America)$ = Hostile(x)
4) $\forall x$ missile(x) = weapon(x)
5) American$(west)$
6) Enemy$(Nono, America)$
7) Owns$(Nono, M1)$ and
8) Missile$(M1)$

} Facts

→ **Forward Reasoning Algorithm:**

A conceptually straightforward, but very inefficient, forward-chaining algorithm. On each iteration, it adds to KB all the atomic sentences that can be inferred in one step from the implication sentences and the atomic sentences already in KB. The function STANDARDIZE-VARIABLES replaces all variables in its arguments with new ones that have not been used before.

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-VARIABLES(rule)
    for each θ such that SUBST (θ, $p_1 \wedge \ldots \wedge p_n$) =
    SUBST (θ, $p'_1 \wedge \ldots \wedge p'_n$) for some $p'_1, \ldots, p'_n$ in KB

q' ←
if q
or ne
φ ←
if c
add new
return false.

→ **Output:**

Inferred :
Inferred :
Inferred :
Inferred :
Result : R

13/10/25

Code :

```
def FOL_FC_ASK(KB, alpha):
    # Initialize the new sentences to be inferred in this iteration
    new = set()

    while new:  # Repeat until new is empty
        new = set()  # Clear new sentences on each iteration

        # For each rule in KB
        for rule in KB:
            # Standardize the rule variables to avoid conflicts
            standardized_rule = standardize_variables(rule)
            p1_to_pn, q = standardized_rule  # Premises (p1, ..., pn) and conclusion (q)

            # For each substitution θ such that Subst(θ, p1, ..., pn) matches the premises
            for theta in get_matching_substitutions(p1_to_pn, KB):
                q_prime = apply_substitution(theta, q)

                # If q_prime does not unify with some sentence already in KB or new
                if not any(unify(q_prime, sentence) != 'FAILURE' for sentence in KB.union(new)):
                    new.add(q_prime)  # Add q_prime to new

                    # Unify q_prime with the query (alpha)
                    phi = unify(q_prime, alpha)
                    if phi != 'FAILURE':
                        return phi  # Return the substitution if it unifies

        # Add newly inferred sentences to the knowledge base
        KB.update(new)

    return False  # Return false if no substitution is found


def standardize_variables(rule):
    """
    Standardize variables in the rule to avoid variable conflicts.
    Rule is assumed to be a tuple (premises, conclusion).
    """
    premises, conclusion = rule
    # Apply standardization logic here (for simplicity, assume no conflict in this case)
    return (premises, conclusion)


def get_matching_substitutions(premises, KB):
    """
    Get matching substitutions for the premises in the KB.
```

```
    This is a placeholder to represent how substitutions would be found.
    """
    # Implement substitution matching here
    return []  # This should return a list of valid substitutions


def apply_substitution(theta, expression):
    """
    Apply a substitution θ to an expression.
    This function will replace variables in expression with their corresponding terms from θ.
    """
    if isinstance(expression, str) and expression.startswith('?'):
        return theta.get(expression, expression)  # Apply substitution to variable
    elif isinstance(expression, tuple):
        return tuple(apply_substitution(theta, arg) for arg in expression)
    return expression


def unify(psi1, psi2, subst=None):
    """Unification algorithm (simplified)"""
    if subst is None:
        subst = {}

    def apply_subst(s_map, expr):
        if isinstance(expr, str) and expr.startswith('?'):
            return s_map.get(expr, expr)
        elif isinstance(expr, tuple):
            return tuple(apply_subst(s_map, arg) for arg in expr)
        return expr

    def is_variable(expr):
        return isinstance(expr, str) and expr.startswith('?')

    _psi1 = apply_subst(subst, psi1)
    _psi2 = apply_subst(subst, psi2)

    if is_variable(_psi1) or is_variable(_psi2) or not isinstance(_psi1, tuple) or not isinstance(_psi2,
tuple):
        if _psi1 == _psi2:
            return subst
        elif is_variable(_psi1):
            if _psi1 in str(_psi2):
                return 'FAILURE'
            return {**subst, _psi1: _psi2}
        elif is_variable(_psi2):
            if _psi2 in str(_psi1):
                return 'FAILURE'
```

```
        return {**subst, _psi2: _psi1}
    else:
        return 'FAILURE'

    if _psi1[0] != _psi2[0] or len(_psi1) != len(_psi2):
        return 'FAILURE'

    for arg1, arg2 in zip(_psi1[1:], _psi2[1:]):
        s = unify(arg1, arg2, subst)
        if s == 'FAILURE':
            return 'FAILURE'
        subst = s

    return subst

# Knowledge Base (KB)
KB = set()

# Adding facts to KB:
KB.add(('american', 'Robert'))  # Robert is an American
KB.add(('hostile_nation', 'Country_A'))  # Country A is a hostile nation
KB.add(('sell_weapons', 'Robert', 'Country_A'))  # Robert sold weapons to Country A

# Adding the rule (the law):
KB.add((('american(x)', 'hostile_nation(y)', 'sell_weapons(x, y)'),
        'criminal(x)'))

# Goal: Prove that Robert is a criminal
goal = 'criminal(Robert)'

# Calling FOL_FC_ASK to prove the goal
result = FOL_FC_ASK(KB, goal)

if result != 'FAILURE':
    print("Robert is a criminal!")
else:
    print("Robert is not a criminal.")
```

Output:

```
...  Robert is a criminal!
```

Program 9

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

Algorithm:

Date 27/10/25
Page 27

LAB - 09

Resolution in First Order Logic

Algorithm:

1) Convert all sentences to CNF
2) Negate, conclusion S and convert result to CNF.
3) Add negated conclusion S to premise clause.
4) Repeat while contradiction or no progress is made:
   → Select 2 clauses (call them parent clauses)
   → Resolve them together together, performing all required unification.
   → If resolvent is the empty clause then a contradiction is found.
   → If not, add resolvent to the premises
5) If we succeed in step 4, we have proved the ~~contrad~~ conclusion.

Output:

Proof by resolution: John likes peanuts

Q. Given the K.B, Prove: $\exists V$

John likes all kinds of food
Apple + vegetables are food
Anything anyone eats + not killed is food
Anil eats peanuts and still alive
Rany eats everything that Anil eats.
Anyone who is alive implies not be killed
Anyone who is not killed implies alive

Prove by resolution: John likes peanuts

Soln. $\forall x$: food $(x) \rightarrow$ likes (John, x)
food (Apple) $\wedge$ food (vegetables)
$\forall x \forall y$: eats$(x, y) \wedge \neg$ killed $(x) \rightarrow$ food (y)
eats (Anil, peanuts) $\wedge$ alive (Anil)
$\forall x$: eats (Anil, x) $\rightarrow$ eats (Rany, x)
$\forall x$: $\neg$ killed $(x) \rightarrow$ alive (x)
likes (John, Peanuts)

Code:

```
from itertools import combinations

def unify(x, y, theta=None):
    if theta is None:
        theta = {}
    if x == y:
        return theta
    elif isinstance(x, str) and x.islower():
        return unify_var(x, y, theta)
    elif isinstance(y, str) and y.islower():
        return unify_var(y, x, theta)
    elif isinstance(x, tuple) and isinstance(y, tuple) and len(x) == len(y):
        return unify(x[1:], y[1:], unify(x[0], y[0], theta))
    else:
        return None

def unify_var(var, x, theta):
    if var in theta:
        return unify(theta[var], x, theta)
    elif x in theta:
        return unify(var, theta[x], theta)
    else:
        theta[var] = x
        return theta

def negate(predicate):
    if isinstance(predicate, tuple) and predicate[0] == 'not':
        return predicate[1]
    else:
        return ('not', predicate)

def substitute_predicate(predicate, theta):
    if isinstance(predicate, str):
        return theta.get(predicate, predicate)
    elif isinstance(predicate, tuple):
        return (predicate[0],) + tuple(theta.get(arg, arg) for arg in predicate[1:])
    return predicate

def substitute(clause, theta):
    return {substitute_predicate(p, theta) for p in clause}

def resolve(clause1, clause2):
    resolvents = []
```

```
    for p1 in clause1:
        for p2 in clause2:
            theta = unify(p1, negate(p2))
            if theta is not None:
                new_clause = (substitute(clause1, theta) | substitute(clause2, theta)) - {p1, p2}
                resolvents.append(frozenset(new_clause))
    return resolvents

def resolution(kb, query):
    negated_query = frozenset({negate(query)})
    clauses = kb | {negated_query}
    new = set()

    while True:
        pairs = list(combinations(clauses, 2))
        for (ci, cj) in pairs:
            resolvents = resolve(ci, cj)
            if frozenset() in resolvents:
                return True
            new |= set(resolvents)
        if new.issubset(clauses):
            return False
        clauses |= new

# Knowledge Base
kb = {
    frozenset({('not', ('Food', 'x')), ('Likes', 'John', 'x')}),  # a
    frozenset({('Food', 'Apple')}),  # b
    frozenset({('Food', 'Vegetables')}),  # b
    frozenset({('not', ('Eats', 'x', 'y')), ('Killed', 'x'), ('Food', 'y')}),  # c
    frozenset({('Eats', 'Anil', 'Peanuts')}),  # d
    frozenset({('Alive', 'Anil')}),  # d
    frozenset({('not', ('Eats', 'Anil', 'x')), ('Eats', 'Harry', 'x')}),  # e
    frozenset({('not', ('Alive', 'x')), ('not', ('Killed', 'x'))}),  # f
    frozenset({('Killed', 'x'), ('Alive', 'x')}),  # g
}
query = ('Likes', 'John', 'Peanuts')

# Run resolution
result = resolution(kb, query)

if result:
    print("Proved by resolution: John likes peanuts.")
else:
    print("Cannot prove that John likes peanuts.")
```

Output:

```
•••  Proved by resolution: John likes peanuts.
```

Program 10

Implement Alpha-Beta Pruning

Date 27/10/25
Page 28

LAB - 10
Alpha - Beta Pruning

Algorithm:

```
function AB-SEARCH returns an action
    v ← MAX-VALUE (state, -∞, -∞)
    returns the action on 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 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
```

Output:

Best value for maximizer : 3

Code:

```
import math

def alpha_beta_search(state):
    return max_value(state, -math.inf, math.inf)

def max_value(state, alpha, beta):
    if terminal_test(state):
        return utility(state)
    v = -math.inf
    for a in actions(state):
        v = max(v, min_value(result(state, a), alpha, beta))
        if v >= beta:
            return v
        alpha = max(alpha, v)
    return v

def min_value(state, alpha, beta):
    if terminal_test(state):
        return utility(state)
    v = math.inf
    for a in actions(state):
        v = min(v, max_value(result(state, a), alpha, beta))
        if v <= alpha:
            return v
        beta = min(beta, v)
    return v

values = [3, 5, 6, 9, 1, 2, 0, -1]
max_depth = 3

def terminal_test(state):
    return state >= len(values) // 2**(max_depth - depth[state])

def utility(state):
    return values[state]

def actions(state):
    if depth[state] == max_depth:
        return []
    return [0, 1]
```

```
def result(state, action):
    child = state * 2 + 1 + action
    depth[child] = depth[state] + 1
    return child

depth = {0: 0}
print("Best value for maximizer:", alpha_beta_search(0))
print("Sthavir / 1BM23CS342")
```

Output :

```
...  Best value for maximizer: 6
     Sthavir / 1BM23CS342
```