

VISVESVARAYA TECHNOLOGICAL UNIVERSITY
“JnanaSangama”, Belgaum -590014, Karnataka.



LAB REPORT
on
Artificial Intelligence (23CS5PCAIN)

Submitted by

AAMAL MOHSIN MAGDUM (1BM23CS002)

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
Sep-2025 to Jan-2026

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



CERTIFICATE

This is to certify that the Lab work entitled “Artificial Intelligence (23CS5PCAIN)” carried out by **Aamal Mohsin Magdum (1BM23CS002)**, 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. Swathi Sridharan Assistant Professor Department of CSE, BMSCE	Dr. Joythi S Nayak Professor & HOD Department of CSE, BMSCE
---	---

Index

Sl. No.	Date	Experiment Title	Page No.
1	30-9-2024	Implement Tic –Tac –Toe Game Implement vacuum cleaner agent	5
2	7-10-2024	Implement 8 puzzle problems using Depth First Search (DFS) Implement Iterative deepening search algorithm	9
3	14-10-2024	Implement A* search algorithm	14
4	21-10-2024	Implement Hill Climbing search algorithm to solve N-Queens problem	17
5	28-10-2024	Simulated Annealing to Solve 8-Queens problem	20
6	11-11-2024	Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.	23
7	2-12-2024	Implement unification in first order logic	26
8	2-12-2024	Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.	29
9	16-12-2024	Create a knowledge base consisting of first order logic statements and prove the given query using Resolution	32
10	16-12-2024	Implement Alpha-Beta Pruning.	37

Course Certificate:



COURSE COMPLETION CERTIFICATE

The certificate is awarded to

Aamal Magdum

for successfully completing the course

Introduction to Artificial Intelligence

on November 18, 2025



Congratulations! You make us proud!

Issued on: Tuesday, November 25, 2025
To verify, scan the QR code at <https://verify.onwingspan.com>

COURSE COMPLETION CERTIFICATE

The certificate is awarded to

Aamal Magdum

for successfully completing the course

Introduction to Natural Language Processing

on November 18, 2025



Congratulations! You make us proud!

Sathesh N.
Sathesha B. Nanjappa
Senior Vice President and Head
Education, Training and Assessment
Infosys Limited



COURSE COMPLETION CERTIFICATE

The certificate is awarded to

Aamal Magdum

for successfully completing the course

Introduction to Deep Learning

on November 18, 2025



Congratulations! You make us proud!

Issued on: Tuesday, November 25, 2025
To verify, scan the QR code at <https://verify.onwingspan.com>

CERTIFICATE OF ACHIEVEMENT

The certificate is awarded to

Aamal Magdum

for successfully completing

Artificial Intelligence Foundation Certification

on November 25, 2025



Congratulations! You make us proud!

Sathesh N.
Sathesha B. Nanjappa
Senior Vice President and Head
Education, Training and Assessment
Infosys Limited

Github Link:

Program 1

Implement Tic – Tac – Toe Game
Implement vacuum cleaner agent

Algorithm:

LAB - 1	
	Date 20/08/25 Page _____
1) Implementing "TIC-TAC-TOE" game board ↪ : () board - array ↪ [] board [1][1] algorithm ↪ 1 + [0][0] entry [1][0] board [1][1] initialized 3x3 matrix according grid values [2][0] board [1][1] = And " " "[1][1] board" entry (2) set player (variable n(1, 2)) ↪ (3) Print (matrix) & display players' choices ↪ do assigning their position P.S.P.T. 2, 1, 3, 4, 5, 6, 7, 8, 9 are marking letters ANS: Anupriya Aishwarya Suganya Nithiyanjan (4) Input player choice & input assign grid (5) Using (backtracking), check for available (position) & make next move (6) Let's conditions (check) ("missed") & specific, same symbol occupies 3 positions along column, row or diagonal → declare winner. : () board - array 3x3 ↪ 1) " " = 1 [] board = [1][1] board = [0][0] board ↪ 2) " " = [2][2] board = [1][1] board = [0][0] board ↪ 3) " " = [0][0] board = [1][1] board = [2][2] board ↪ 4) " " = [1][1] board = [2][2] board = [0][0] board ↪ 5) " " = [0][2] board = [1][1] board = [2][0] board ↪ 6) " " = [2][0] board = [1][1] board = [0][2] board ↪ 7) " " = [0][1] board = [1][0] board = [2][2] board ↪ 8) " " = [2][2] board = [1][0] board = [0][1] board ↪ : board in tac ↪ 3x3 "sit" star number : 3x3 ↪ 9 numbers " 1 2 3 " numbers	

1 - 9A.J Date / / Page /

```

→ board = ["-", "-", "-",
           "-", "-", "-",
           "-", "-", "-"]
def print_board():
    print(board[0] + " | " + board[1] + " | " + board[2])
    print(board[3] + " | " + board[4] + " | " + board[5])
    print(board[6] + " | " + board[7] + " | " + board[8])

def play_game():
    print("X's turn")
    current_player = "X"
    game_over = False
    while not game_over:
        take_turn(current_player)
        if check_game_over():
            game_over = True
            if current_player == "X":
                print("X wins!")
            else:
                print("O wins!")

def take_turn(player):
    position = int(input("Enter a position from 1-9"))
    while position not in [1, 2, 3, 4, 5, 6, 7, 8, 9]:
        position = int(input("Invalid input. Enter a position from 1-9"))
    board[position - 1] = player
    print_board()

def check_game_over():
    if (board[0] == board[1] == board[2] == "X") or
       (board[3] == board[4] == board[5] == "X") or
       (board[6] == board[7] == board[8] == "X") or
       (board[0] == board[3] == board[6] == "X") or
       (board[1] == board[4] == board[7] == "X") or
       (board[2] == board[5] == board[8] == "X") or
       (board[0] == board[4] == board[8] == "X") or
       (board[2] == board[4] == board[6] == "X"):
        return "X"
    elif (board[0] == board[1] == board[2] == "O") or
         (board[3] == board[4] == board[5] == "O") or
         (board[6] == board[7] == board[8] == "O") or
         (board[0] == board[3] == board[6] == "O") or
         (board[1] == board[4] == board[7] == "O") or
         (board[2] == board[5] == board[8] == "O") or
         (board[0] == board[4] == board[8] == "O") or
         (board[2] == board[4] == board[6] == "O"):
        return "O"
    else:
        return "Tie"

print("Play Tic-Tac-Toe!")
play_game()

```

2 - 9A.J Date / / Page /

```

def play_game():
    print("X's turn")
    current_player = "X"
    game_over = False
    while not game_over:
        take_turn(current_player)
        if check_game_over():
            game_over = True
            if current_player == "X":
                print("X wins!")
            else:
                print("O wins!")

def take_turn(player):
    position = int(input("Enter a position from 1-9"))
    while position not in [1, 2, 3, 4, 5, 6, 7, 8, 9]:
        position = int(input("Invalid input. Enter a position from 1-9"))
    board[position - 1] = player
    print_board()

def check_game_over():
    if (board[0] == board[1] == board[2] == "X") or
       (board[3] == board[4] == board[5] == "X") or
       (board[6] == board[7] == board[8] == "X") or
       (board[0] == board[3] == board[6] == "X") or
       (board[1] == board[4] == board[7] == "X") or
       (board[2] == board[5] == board[8] == "X") or
       (board[0] == board[4] == board[8] == "X") or
       (board[2] == board[4] == board[6] == "X"):
        return "X"
    elif (board[0] == board[1] == board[2] == "O") or
         (board[3] == board[4] == board[5] == "O") or
         (board[6] == board[7] == board[8] == "O") or
         (board[0] == board[3] == board[6] == "O") or
         (board[1] == board[4] == board[7] == "O") or
         (board[2] == board[5] == board[8] == "O") or
         (board[0] == board[4] == board[8] == "O") or
         (board[2] == board[4] == board[6] == "O"):
        return "O"
    else:
        return "Tie"

print("Play Tic-Tac-Toe!")
play_game()

```

1 - 9A.J Date / / Page /

```

X's turn
Enter a position from 1-9: 1
X 1 | X 2 | X 3 | X 4 | X 5 | X 6 | X 7 | X 8 | X 9
- - - - - - - - - - -
- 1 - 0 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
- 1 - - - - - - - - - -
O's turn
Enter a position from 1-9: 8
X 1 | X 2 | X 3 | X 4 | X 5 | X 6 | X 7 | O 8 | X 9
- - - - - - - - - - -
- 1 0 - - - - - - - -
X's turn
Enter a position from 1-9: 8
X 1 | X 2 | X 3 | X 4 | X 5 | X 6 | X 7 | O 8 | X 9
- - - - - - - - - - -
- 1 0 - - - - - - - -
X wins!

```

2) Implement vacuum cleaner

```

// Pseudo code: moving
function vacuum_cleaner(room: array):
    for each room in room-array:
        if room is dirty:
            print "room[label] is cleaned"
            room[label] is cleaned
        else:
            print "room[label] is already clean"
            move to next room

```

3) Vacuum cleaner

```

def vacuum_cleaner(room: array):
    label, status = room[0], room[1]
    for room in room:
        if "dirty" in room:
            print(f"Moving to {label} room...")
            print(f"Cleaning {status} room...")
            status = "clean"
            print(f"Room {label} is cleaned")
            room[1] = status
        else:
            print(f"Room {label} is already clean")
            room[1] = status
    print("All rooms are clean")

```

Code:

```

//tic-tac-toe
board = ["-", "-", "-",
           "-", "-", "-",
           "-", "-", "-"]

def print_board():
    print(board[0] + " | " + board[1] + " | " + board[2])
    print(board[3] + " | " + board[4] + " | " + board[5])
    print(board[6] + " | " + board[7] + " | " + board[8])

```

```

def take_turn(player):
    print(player + "'s turn")
    position = input("Enter a position from 1-9: ")
    while position not in ["1", "2", "3", "4", "5", "6", "7", "8", "9"]:
        position = input("Invalid input. Choose a position from 1-9: ")
    position = int(position) - 1
    while board[position] != "-":
        position = int(input("Position already occupied. Choose a different position: ")) - 1
    board[position] = player
    print_board()

def check_game_over():
    if (board[0] == board[1] == board[2] != "-") or \
       (board[3] == board[4] == board[5] != "-") or \
       (board[6] == board[7] == board[8] != "-") or \
       (board[0] == board[3] == board[6] != "-") or \
       (board[1] == board[4] == board[7] != "-") or \
       (board[2] == board[5] == board[8] != "-") or \
       (board[0] == board[4] == board[8] != "-") or \
       (board[2] == board[4] == board[6] != "-"):
        return "win"

    elif '-' not in board:
        return "Tie"

    else:
        return "Play"

def play_game():
    print_board()
    current_player = "X"
    game_over = False
    while not game_over:
        take_turn(current_player)
        game_result = check_game_over()
        if game_result == "win":
            print(current_player + " wins!")
            game_over = True
        elif game_result == "tie":
            print("It's a tie!")
            game_over = True
        else:
            current_player = "O" if current_player == "X" else "X"

    play_game()

```

```
//vacuum cleaner
def vacuum_cleaner(room_array):
    for room in room_array:
        label, status = room[0], room[1].lower()
        print(f"Currently in Room {label}")
        if status == "dirty":
            print(f"Room {label} is cleaned")
        else:
            print(f"Room {label} is already clean")
    print("Moving to next room...\n")

rooms = [('A', 'dirty'), ('B', 'clean'), ('A', 'clean'), ('B', 'dirty')]
vacuum_cleaner(rooms)
```

Program 2

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

Algorithm:

<pre> 1) Implement 8-puzzle problem - puzzle moves: move up, move down, move left, move right - if goal state: [1,2,3,4,5,6,7,8,0] - define goal state: [1,2,3,4,5,6,7,8,0] - define function heuristic(state): h=0 - if reaches state in 'in' set: if t>0: (x1,y1) ← position of zero in current state state[i][x1,y1] ← position of zero in goal-state h ← h + x1-2 + y1-2 - return h as priority - define function f(state): return g(state) + heuristic(state) - create a priority queue insert start state into queue insert priority frontier state into priority - create an empty set explored - while frontier is not empty: state ← remove node from frontier with smallest f value: if state = goal-state: return path from state to start state is a move else: state is expanded: for each neighbour in neighbours(state): if neighbour not in explored: compute f(neighbour) = f(state) + 1 insert neighbour into frontier with priority (f(neighbour)) if frontier becomes empty: return "no solution" </pre>	<pre> 2) Pseudo code initialized using minmax - start -> initial state - function minmax(state): if state is goal state: return 0 if state is terminal state: return eval_fn(state) if state is not terminal state: if max: max_val = -infinity for each action in actions(state): result = apply-action(state,action) val = minmax(result) if val > max_val: max_val = val return max_val else: min_val = infinity for each action in actions(state): result = apply-action(state,action) val = minmax(result) if val < min_val: min_val = val return min_val </pre>
<pre> function greedy-best-first(initial-state,goal-state, heuristic-func) open = a list of tuples (heuristic-score, state, path) visited = set to store visited puzzle states (tuple) initial-score = heuristic-score(initial-state,goal-state) open.append((initial-score,initial-state,initial-path)) initial-state-tuple = tuple of initial-state while open != []: best-node = find tuple in open-list with minimum heuristic-score : remove best-node from open-list current-state = open-state from best-node path = path from best-node if current-state is equal to goal-state: return path else: possibly-actions = get-possible-actions(current-state) for each action in possible-actions: new-state = apply-action(current-state,action) if new-state is not in visited: next-state-tuple = tuple of next-state next-state-path = path + next-state add (heuristic-score(next-state),next-state, next-path) to opened no open-list. open.append() return None = 0.0 </pre>	<pre> function get-possible-actions(state): find position of blank tile (0) if blank tile in top row: add 'up' to actions if blank tile not in bottom row: add 'down' if blank tile not in leftmost column: add 'left' if blank tile not in rightmost column: add 'right' </pre>

Code:

```

//8-puzzle(dfs)
def find_possible_moves(state):
    index = state.index('_')
    moves = {

```

```

0: [1, 3],
1: [0, 2, 4],
2: [1, 5],
3: [0, 4, 6],
4: [1, 3, 5, 7],
5: [2, 4, 8],
6: [3, 7],
7: [6, 8, 4],
8: [5, 7],
}
return moves.get(index, [])

def dfs(initial_state, goal_state, max_depth=50):
    stack = [(initial_state, [], 0)]
    visited = {tuple(initial_state)}
    states_explored = 0
    printed_depths = set()

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

        if depth > max_depth:
            continue

        if depth not in printed_depths:
            print(f"\n--- Depth {depth} ---")
            printed_depths.add(depth)

        states_explored += 1
        print(f"State #{states_explored}: {current_state}")

        if current_state == goal_state:
            print(f"\n Goal reached at depth {depth} after exploring {states_explored} states.\n")
            return path

        possible_moves_indices = find_possible_moves(current_state)

        for move_index in reversed(possible_moves_indices): # Reverse for DFS order
            next_state = list(current_state)
            blank_index = next_state.index('_')
            next_state[blank_index], next_state[move_index] = next_state[move_index],
            next_state[blank_index]

            if tuple(next_state) not in visited:
                visited.add(tuple(next_state))
                stack.append((next_state, path + [next_state], depth + 1))

```

```

print(f"\n Goal state not reachable within depth {max_depth}. Explored {states_explored}\n"
      states.\n")
    return None

initial_state = [1, 2, 3, 4, 8, '_', 7, 6, 5]
goal_state   = [1, 2, 3, 4, 5, 6, 7, 8, '_']

solution_path = dfs(initial_state, goal_state, max_depth=50)

if solution_path is None:
    print("No solution found.")
else:
    print("Solution path:")
    for step, state in enumerate(solution_path, start=1):
        print(f"Step {step}: {state}")

//iterative deepening search
import time

# ----- MOVE GENERATOR -----
def find_possible_moves(state):
    index = state.index('_')

    if index == 0:
        return [1, 3]
    elif index == 1:
        return [0, 2, 4]
    elif index == 2:
        return [1, 5]
    elif index == 3:
        return [0, 4, 6]
    elif index == 4:
        return [1, 3, 5, 7]
    elif index == 5:
        return [2, 4, 8]
    elif index == 6:
        return [3, 7]
    elif index == 7:
        return [4, 6, 8]
    elif index == 8:
        return [5, 7]
    return []

# ----- DEPTH LIMITED SEARCH -----
def depth_limited_dfs(state, goal_state, limit, path, visited):

```

```

if state == goal_state:
    return path

if limit <= 0:
    return None

visited.add(tuple(state))

for move_index in find_possible_moves(state):
    next_state = list(state)
    blank_index = next_state.index('_')
    next_state[blank_index], next_state[move_index] = next_state[move_index],
    next_state[blank_index]

    if tuple(next_state) not in visited:
        result = depth_limited_dfs(next_state, goal_state, limit - 1, path + [next_state], visited)
        if result is not None:
            return result
return None

# ----- ITERATIVE DEEPENING DFS -----
def iddfs(initial_state, goal_state, max_depth=30):
    for depth in range(max_depth):
        print(f"Searching at depth limit = {depth}")
        visited = set()
        result = depth_limited_dfs(initial_state, goal_state, depth, [initial_state], visited)
        if result is not None:
            return result, depth
    return None, max_depth

# ----- TEST -----
initial_state = [1, 2, 3,
                 4, 8, '_',
                 7, 6, 5]

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

# Measure execution time
start_time = time.time()
solution_path, depth_reached = iddfs(initial_state, goal_state, max_depth=30)
end_time = time.time()

if solution_path is None:
    print("Goal state is not reachable within given depth limit.")
else:

```

```
print("\n\nSolution path found:")
for step, state in enumerate(solution_path, start=0):
    print(f"Step {step}: {state}")

print("Execution time: {:.6f} seconds".format(end_time - start_time))
print("Depth reached:", depth_reached)
```

Program 3

Implement A* search algorithm

Algorithm:

LAB-3

Date _____
Page _____

i) Implement 8-puzzle problem using A* search

// pseudocode
function A*search (start-state, goal-state):
 open ← open+ priority queue (ordered by f-value)
 closed ← empty set
 if start-state == goal-state
 return path + (current-state)
 else
 insert (f(start-state), g(start-state)+h(start-state),
 start-state, path + []) into OPEN
 while OPEN is not empty:
 if f(current-state, path) < f(state):
 remove state from with lowest f from OPEN
 if current-state == goal-state:
 return path + (current-state)
 add current-state to closed
 for each neighbour in get-neighbour:
 if neighbour not in closed:
 g(neighbour) = g + 1
 h(neighbour) = heuristic(neighbour)
 f(neighbour) = g(neighbour) +
 h(neighbour)
 insert (f(neighbour), g(neighbour),
 neighbour, path + []) into OPEN

neighbour(path + (current-state))

0. initial state 1. 1 2 3 4 5 6 7 8 -
1. 1 2 3 4 5 6 7 8 -
reaching no solution 2. 1 2 3 4 5 6 7 8 -
3. initial state 3. 1 2 3 4 5 6 7 8 -
4. initial state 4. 1 2 3 4 5 6 7 8 -
5. initial state 5. 1 2 3 4 5 6 7 8 -
6. initial state 6. 1 2 3 4 5 6 7 8 -
7. initial state 7. 1 2 3 4 5 6 7 8 -
8. initial state 8. 1 2 3 4 5 6 7 8 -

start state Goal state
1 2 3 1 2 3
4 5 6 4 5 6
7 8 - 7 8 -

↓
2. initial state 2. 1 2 3 4 5 6 7 8 -
3. 1 2 3 4 5 6 7 8 -
4. 1 2 3 4 5 6 7 8 -
5. 1 2 3 4 5 6 7 8 -

↓
f(n)
g(n)
h(n)

↓
1 2 3 1 2 3
4 5 6 4 5 6
7 8 - 7 8 -

↓
1 2 3 1 2 3
4 - 6 4 5 6
7 5 8 7 - 8

↓

Execution time: 0.000012 seconds
Depth reached: 5

Scanned with CamScanner

Code:

```
import heapq
import time

# Heuristic: Manhattan Distance
def heuristic(state, goal):
    distance = 0
    for i in range(1, 9): # tile numbers 1 to 8
        x1, y1 = divmod(state.index(i), 3)
        x2, y2 = divmod(goal.index(i), 3)
        distance += abs(x1 - x2) + abs(y1 - y2)
    return distance

# Get neighbors by sliding blank (0) up/down/left/right
def get_neighbors(state):
    neighbors = []
    i = state.index(0) # position of blank
    x, y = divmod(i, 3)
    moves = [(-1,0), (1,0), (0,-1), (0,1)]

    for dx, dy in moves:
        new_x, new_y = x + dx, y + dy
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            j = new_x * 3 + new_y
            new_state = list(state)
            new_state[i], new_state[j] = new_state[j], new_state[i]
            neighbors.append(tuple(new_state))
    return neighbors

# A* Search for 8-puzzle
def astar(start, goal):
    open_set = []
    heapq.heappush(open_set, (heuristic(start, goal), 0, start))

    came_from = {}
    g_score = {start: 0}

    while open_set:
        _, cost, current = heapq.heappop(open_set)

        if current == goal:
            # Reconstruct path
            path = []
            while current in came_from:
                path.append(current)
                current = came_from[current]
            return path

    return None
```

```

path.append(start)
return path[::-1]

for neighbor in get_neighbors(current):
    tentative_g = g_score[current] + 1
    if neighbor not in g_score or tentative_g < g_score[neighbor]:
        came_from[neighbor] = current
        g_score[neighbor] = tentative_g
        f_score = tentative_g + heuristic(neighbor, goal)
        heapq.heappush(open_set, (f_score, tentative_g, neighbor))

return None # no solution

# ----- TEST -----
start = (1, 2, 3,
         4, 8, 0,
         7, 6, 5)

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

# Measure execution time
start_time = time.time()
path = astar(start, goal)
end_time = time.time()

if path:
    print("Steps to solve ({} moves):".format(len(path)-1))
    for state in path:
        for i in range(0, 9, 3):
            print(state[i:i+3])
        print()
else:
    print("No solution found")

print("Execution time: {:.6f} seconds".format(end_time - start_time))

```

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:

LAB - 4

Date / /
Page / /

```

1) //Hill Climbing is a local search (1)
   (heuristic search algorithm).
   //Algorithm
   (1) Evaluate initial state; if I=F,
       final state, return success and stop
   (2) Loop : step until solution found
       select and apply operation on state
       to check if neighbouring state is
       more optimal, if not, break it
   (3) Define function heuristic;
       return g(state) + heuristic(state)

// Pseudocode for hill climbing (2)
function hill climbing(problem):
    start = initial solution(problem)
    while True:
        neighbours = generate-neighbours(current)
        best-neighbour = select-best(neighbours)
        if evaluate(best-neighbour) <=
           g(start) + evaluate(Gamma), start (1)
            . . .
        else
            current = best-neighbour (2)
        break
    return current (3)

```

LAB - 4

Date / /
Page / /

```

// Algorithm
(1) Start with an initial/guessing
arbitrarily chosen state (initial)
(2) Create neighbouring candidates //by
making small modifications to the
current state. State 1
(3) Pick the best neighbour that improves
state upon the current state
(4) Stop when no better local neighbour
is found, that is, a local optimum
is reached (Gamma, stopping) (2)
(5) Backward scanning, 5 marks

```

Code:

```
import random
import math
```

```

def compute_cost(state):
    """Count diagonal conflicts for a permutation-state (one queen per row & column)."""
    conflicts = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if abs(state[i] - state[j]) == abs(i - j):
                conflicts += 1
    return conflicts

def random_permutation(n):
    arr = list(range(n))
    random.shuffle(arr)
    return arr

def neighbors_by_swaps(state):
    """All neighbors obtained by swapping two columns (keeps permutation property)."""
    n = len(state)
    for i in range(n - 1):
        for j in range(i + 1, n):
            nb = state.copy()
            nb[i], nb[j] = nb[j], nb[i]
            yield nb

def hill_climb_with_restarts(n, max_restarts=None):
    """Hill climbing on permutations with random restart on plateau (no revisits)."""
    visited = set()
    total_states = math.factorial(n)
    restarts = 0

    while True:
        # pick a random unvisited start permutation
        if len(visited) >= total_states:
            raise RuntimeError("All states visited — giving up (no solution found.)")

        state = random_permutation(n)
        while tuple(state) in visited:
            state = random_permutation(n)
            visited.add(tuple(state))

        # climb from this start
        while True:
            cost = compute_cost(state)
            if cost == 0:
                return state, restarts

        # find best neighbor (swap-based neighbors)

```

```

best_neighbor = None
best_cost = float("inf")
for nb in neighbors_by_swaps(state):
    c = compute_cost(nb)
    if c < best_cost:
        best_cost = c
        best_neighbor = nb

# if strictly better, move; otherwise it's a plateau/local optimum -> restart
if best_cost < cost:
    state = best_neighbor
    visited.add(tuple(state))
else:
    # plateau or local optimum -> restart
    restarts += 1
    if max_restarts is not None and restarts >= max_restarts:
        raise RuntimeError(f"Stopped after {restarts} restarts (no solution found).")
    break # go pick a new unvisited start

def format_board(state):
    n = len(state)
    lines = []
    for r in range(n):
        lines.append(" ".join("Q" if state[c] == r else "-" for c in range(n)))
    return "\n".join(lines)

if __name__ == "__main__":
    n = 4
    solution, restarts = hill_climb_with_restarts(n)
    print("Found solution:", solution)
    print(format_board(solution))

```

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:

```

2) Simulated Annealing
    (Probabilistic Optimization algorithm)
    // Algorithm
    (1) start with an initial guess and set
        a sufficiently high temperature
    (2) propose small random changes to the
        current state
    (3) if the new solution is better, accept
    (otherwise if worse, accept with probability
        =  $e^{-\Delta E/T}$ ) to avoid
    (4) multiply temperature by a cooling
        factor at each iteration.
    (5) stop when temperature is low,
        or a suitable solution is found.

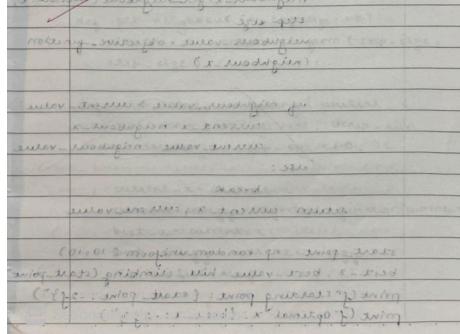
```

Scanned with CamScanner

```

// Pseudocode (written 1/16/2013)
initialise solution x and initial
temperature T to minimum program
while T > T_min and energy(x) > E_threshold:
    generate a neighbour (x_new)
    compute energies: E = energy(x),
    E_new = energy(x_new)
    if  $\Delta E \geq E_{new} - E_{old}$  then
        accept new configuration: ( $x \leftarrow x_{new}$ )
    else:
        accept/reject with probability
             $= \exp(-\Delta E/T)$ 
        reduce temperature:  $T \leftarrow T * \alpha$ 
return best solution found

```



Code:

```
import random
import math
```

```
def cost(state):
```

```

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

def get_neighbor(state):

    neighbor = state[:]
    i, j = random.sample(range(len(state)), 2)
    neighbor[i], neighbor[j] = neighbor[j], neighbor[i]
    return neighbor

def simulated_annealing(n=8, max_iter=10000, temp=100.0, cooling=0.95):

    current = list(range(n))
    random.shuffle(current)
    current_cost = cost(current)

    temperature = temp
    cooling_rate = cooling

    best = current[:]
    best_cost = current_cost

    for _ in range(max_iter):
        if temperature <= 0 or best_cost == 0:
            break

        neighbor = get_neighbor(current)
        neighbor_cost = cost(neighbor)
        delta = current_cost - neighbor_cost

        if delta > 0 or random.random() < math.exp(delta / temperature):
            current, current_cost = neighbor, neighbor_cost
            if neighbor_cost < best_cost:
                best, best_cost = neighbor[:], neighbor_cost

        temperature *= cooling_rate

    return best, best_cost

def print_board(state):

```

```
n = len(state)
for row in range(n):
    line = " ".join("Q" if state[col] == row else "." for col in range(n))
    print(line)
print()

n = 8
solution, cost_val = simulated_annealing(n, max_iter=20000)
print("Best position found:", solution)
print(f"Number of non-attacking pairs: {n*(n-1)//2 - cost_val}")
print("\nBoard:")
print_board(solution)
```

Program 6

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

Algorithm:

LAB-5

①. Create a knowledge base using propositional logic & query entail the knowledge base

→ //Algorithm TT-entail(x,y)-implies

Input:

KB-sentences = list of propositional sentences from user

query = query sentence from user

Process:

Symbol ← all unique propositional symbols appearing in KB-sentences

return TT-check-all(KB, x, symbols, model)

→ function TT-check-all(KB, x, symbols, model)

if symbols is empty then

return true

else if x is a variable? (P-model) then

if P-model then return true (C-model)

else return false

else if x is a conjunction

rest ← symbols - {x}

model_true = model \cup (x=true)

model_false = model \cup (x=false)

return (TT-check-all(KB, x, rest, model_true) AND TT-check-all(KB, x, rest, model_false))

B-3A1

→ function PIsTrue(x, sentence - KBmodel)

for each symbol 's' in sentence do

if s evaluate as false under model

return false

return true

Explanations: P-131 -> 1331949247
 1331949247 → P
 P → P
 (1331949247) 1331949247 → P
 P → P

Propositional Logic Tableau Method

Consider (1) knowledge base KB that contains one following propositional logic sentence

(sentence 1) $P \rightarrow Q$ and (2) query $Q \vee R$

→ Construct a truth-table that shows the (1) truth value of each sentence in KB
 and indicate the models in which the KB is true.

Model	P	Q	R	Value
1	T	F	F	F
2	T	F	T	T
3	F	F	F	F
4	F	F	T	T
5	F	T	F	T
6	F	T	T	T

→ If there is no model in which the KB is true, then the query does not entail the KB

→ If there is at least one model in which the KB is true, then the query entails the KB

P	Q	R	$R \rightarrow P$	$P \rightarrow QR$	$(QR \rightarrow R) \wedge P$
T	T	T	T	T	T
T	T	F	F	F	F
X	T	F	X	X	X
T	F	T	F	T	F
F	T	T	F	T	F
F	T	F	F	T	F
F	F	T	T	T	T
F	F	F	T	F	F
T	T	T	T	T	T
T	T	F	T	F	F
T	F	T	T	F	F
F	T	T	F	T	F
F	T	F	T	F	F
F	F	T	T	F	F
T	T	T	T	T	T
T	T	F	T	F	F
T	F	T	T	F	F
F	T	T	F	T	F
F	T	F	T	F	F
F	F	T	T	F	F

ii) Does KB entail $R \rightarrow P$? T T F

In all models where KB is true, $R \rightarrow P$ is true.
 \therefore KB entails $R \rightarrow P$

iii) Does KB entail $R \rightarrow P$

$P \rightarrow R$ $R \rightarrow P$
 $\left. \begin{array}{l} T P T \\ F P T \\ F P T \end{array} \right\}$ Some place \rightarrow KB does not entail $R \rightarrow P$

iv) Does KB entail $R \rightarrow R$

$P \rightarrow R$ $R \rightarrow R$
 $\left. \begin{array}{l} T T T \\ T F T \\ F F T \end{array} \right\}$ always true \rightarrow KB entails $R \rightarrow R$

Scanned with CamScanner

④					
$\alpha = A \wedge B$	$\neg A \vee \neg B$	$\neg A \vee \neg B$	A	B	T
T	KB = $\neg(A \vee C) \wedge (B \vee \neg C)$	T	T	T	T
T	T	T	T	T	T
T	$\neg A \vee B \vee C$	$\neg A \vee B \vee C$	T	B	T
F	$\neg F \vee F$	F	T	F	F
F	$\neg F \vee T$	T	T	F	F
F	$\neg F \vee F$	F	F	T	F
T	$\neg F \vee T$	T	T	T	T
T	$\neg F \vee F$	F	T	T	T
T	$\neg F \vee T$	T	T	F	T
T	$\neg F \vee F$	F	T	T	T
T	T	T	T	T	T
T	T	T	T	F	T
T	T	T	F	T	T
T	T	F	T	T	T
T	T	F	F	T	T
T	F	T	T	T	T
F	T	T	T	T	T
T	T	T	T	T	T
T	T	T	F	T	T
T	T	F	T	T	T
T	F	T	T	T	T
T	F	F	T	T	T
T	F	F	F	T	T
T	F	F	F	F	T
T	F	F	F	F	F
T	F	F	F	F	F
T	F	F	F	F	F
T	F	F	F	F	F

\therefore KB entails α - justified

Scanned with CamScanner

Code:

```
import itertools
def evaluate_formula(formula, truth_assignment):
    eval_formula = formula
    for symbol, value in truth_assignment.items():
        eval_formula = eval_formula.replace(symbol, str(value))
    return eval(eval_formula)
```

```

def generate_truth_table(variables):
    return list(itertools.product([False, True], repeat=len(variables)))

def is_entailed(KB_formula, alpha_formula, variables):
    truth_combinations = generate_truth_table(variables)
    print(f"{''.join(variables)} | KB Result | Alpha Result")
    print("-" * (len(variables) * 2 + 15))
    for combination in truth_combinations:
        truth_assignment = dict(zip(variables, combination))
        KB_value = evaluate_formula(KB_formula, truth_assignment)
        alpha_value = evaluate_formula(alpha_formula, truth_assignment)
        result_str = " ".join(["T" if value else "F" for value in combination])
        print(f"{{result_str} | {{'T' if KB_value else 'F'}} | {{'T' if alpha_value else 'F'}}}")
    if KB_value and not alpha_value:
        return False
    return True

KB = "(A or C) and (B or not C)"
alpha = "A or B"
variables = ['A', 'B', 'C']

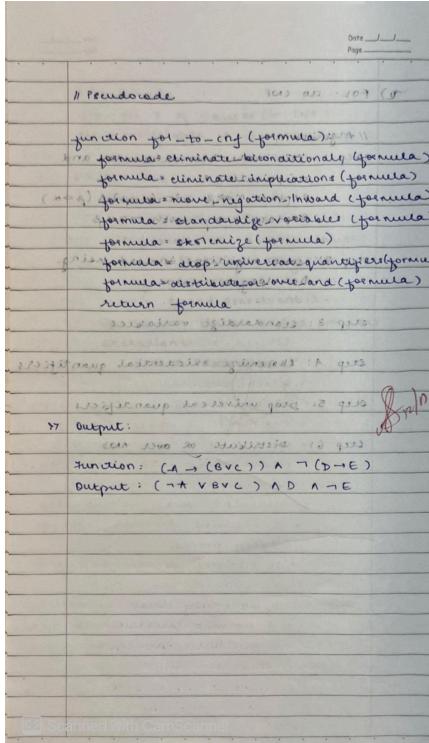
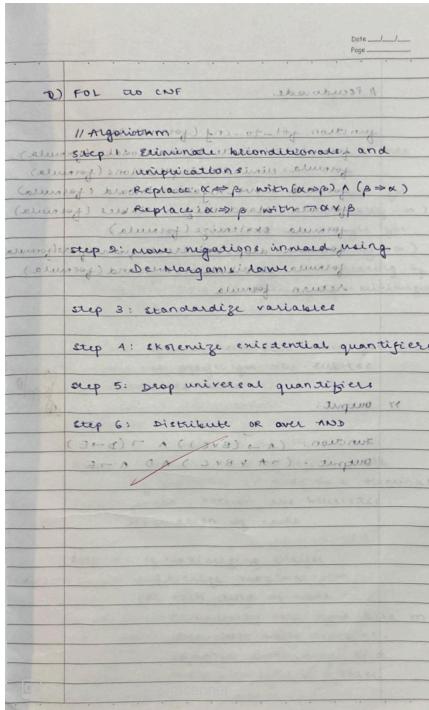
if is_entailed(KB, alpha, variables):
    print("\nThe knowledge base entails alpha.")
else:
    print("\nThe knowledge base does not entail alpha.")

```

Program 7

Implement unification in first order logic

Algorithm:



Scanned with CamScanner

Code:

```
import re

def is_variable(x):
    return x[0].islower() and x.isalpha()

def parse(term):
    term = term.strip()
    if '(' not in term:
        return term
    name, args = term.split('(', 1)
    args = args[:-1] # remove closing parenthesis
    return name.strip(), [parse(a.strip()) for a in args.split(',')]

def occurs_check(var, expr):
    if var == expr:
        return True
    if isinstance(expr, tuple):
        _, args = expr
        return any(occurs_check(var, a) for a in args)
    return False

def substitute(subs, expr):
    if isinstance(expr, str):
        if expr in subs:
            return substitute(subs, subs[expr])
        return expr
    else:
        func, args = expr
        return (func, [substitute(subs, a) for a in args])

def unify(x, y, subs=None):
    if subs is None:
        subs = {}

    x = substitute(subs, x)
    y = substitute(subs, y)

    if x == y:
        return subs

    if isinstance(x, str) and is_variable(x):
        if occurs_check(x, y):
            return None
        subs[x] = y
        return subs
```

```

if isinstance(y, str) and is_variable(y):
    if occurs_check(y, x):
        return None
    subs[y] = x
    return subs

if isinstance(x, tuple) and isinstance(y, tuple):
    if x[0] != y[0] or len(x[1]) != len(y[1]):
        return None
    for a, b in zip(x[1], y[1]):
        subs = unify(a, b, subs)
    if subs is None:
        return None
    return subs

return None

def term_to_str(t):
    if isinstance(t, str):
        return t
    func, args = t
    return f'{func}({", ".join(term_to_str(a) for a in args)})'

def pretty_print(subs):
    return ', '.join(f'{v} : {term_to_str(t)}' for v, t in subs.items())

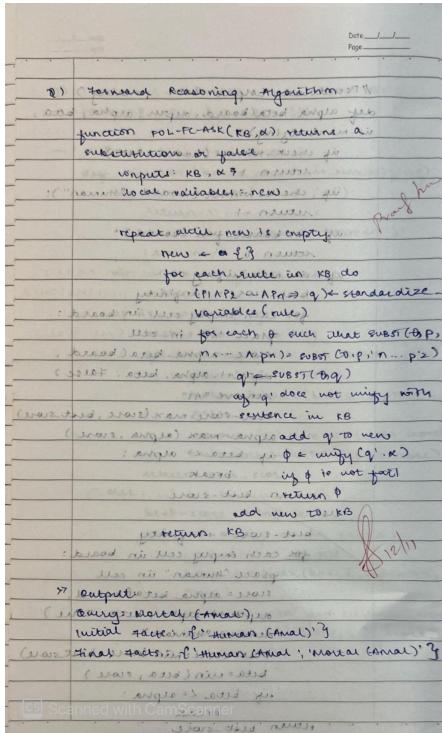
pairs = [
    ("P(f(x),g(y),y)", "P(f(g(z)),g(f(a)),f(a)))",
     ("Q(x,f(x))", "Q(f(y),y)",),
     ("H(x,g(x))", "H(g(y),g(g(z))))")
]
for s1, s2 in pairs:
    print(f"\nUnifying: {s1} and {s2}")
    result = unify(parse(s1), parse(s2))
    if result:
        print("=> Substitution:", pretty_print(result))
    else:
        print("=> Not unifiable.")

```

Program 8

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

Algorithm:



Code:

```
import re
```

```
def match_pattern(pattern, fact):
```

```
    """
```

Checks if a fact matches a rule pattern using regex-style variable substitution.

Variables are lowercase words like p, q, x, r etc.

Returns a dict of substitutions or None if not matched.

```
"""
```

```
# Extract predicate name and arguments
```

```
pattern_pred, pattern_args = re.match(r'(\w+)
```

```
', pattern).groups()
```

```
fact_pred, fact_args = re.match(r'(\w+)
```

```
', fact).groups()
```

```
if pattern_pred != fact_pred:
```

```
    return None # predicate mismatch
```

```
pattern_args = [a.strip() for a in pattern_args.split(",")]
```

```
fact_args = [a.strip() for a in fact_args.split(",")]
```

```

if len(pattern_args) != len(fact_args):
    return None

subst = {}
for p_arg, f_arg in zip(pattern_args, fact_args):
    if re.fullmatch(r'[a-z]\w*', p_arg): # variable
        subst[p_arg] = f_arg
    elif p_arg != f_arg: # constants mismatch
        return None
return subst

def apply_substitution(expr, subst):
    """Replaces all variable names in expr using the given substitution dict."""
    for var, val in subst.items():
        expr = re.sub(rf'\b{var}\b', val, expr)
    return expr

```

----- Knowledge Base -----

```

rules = [
    ("American(p)", "Weapon(q)", "Sells(p,q,r)", "Hostile(r)", "Criminal(p)"),
    ("Missile(x)", "Weapon(x)",),
    ("Enemy(x, America)", "Hostile(x)",),
    ("Missile(x)", "Owns(A, x)", "Sells(Robert, x, A)")
]

```

```

facts = {
    "American(Robert)",
    "Enemy(A, America)",
    "Owns(A, T1)",
    "Missile(T1)"
}

```

goal = "Criminal(Robert)"

```

def forward_chain(rules, facts, goal):
    added = True
    while added:
        added = False
        for premises, conclusion in rules:

```

```

            possible_substs = []
            for p in premises:
                for f in facts:
                    subst = match_pattern(p, f)

```

```

if subst:
    possible_substs.append(subst)
    break
else:
    break
else:

combined = {}
for s in possible_substs:
    combined.update(s)

new_fact = apply_substitution(conclusion, combined)

if new_fact not in facts:
    facts.add(new_fact)
    print(f"Inferred: {new_fact}")
    added = True
if new_fact == goal:
    return True
return goal in facts

print("Goal achieved:", forward_chain(rules, facts, goal))

```

Program 9

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

Algorithm:

LAB-7 Page _____

B) Resolution (10p x 2)

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

→ Knowledge Base :-

(All man : die mortal & x (man(x) → Mortal(x))
 Socrates is a man : man (socrates)
 (query) Is Socrates mortal ? Mortal (socrates)

Conversion :- CNF :- 33 AND 6.0011

$\neg \text{man}(x) \vee \text{mortal}(x)$
 $\text{man}(\text{socrates})$
 $\neg \text{mortal}(\text{socrates})$

Proof :-

Resolve $\text{man}(\text{socrates})$ and $\neg \text{man}(x) \vee \text{mortal}(x)$
 $\text{mortal}(x)$ with $x = \text{socrates}$:-
 $\rightarrow \text{mortal}(\text{socrates})$

Resolve $\text{mortal}(\text{socrates})$ and $\neg \text{mortal}(\text{socrates})$
 $\rightarrow \text{contradiction}$ (empty clause)

This contradiction means the original query $\text{mortal}(\text{socrates})$, is proved true by resolution from the knowledge base.

Scanned with CamScanner

Date _____
 Page _____

//Algorithm

Step 1: Convert all knowledge base sentences and the negation of the query into CNF

Step 2: Initialize clauses set. Collect all clauses from the CNF forms

Step 3: Resolution loop. Repeat until either the empty clause is derived or no new clauses can be added to set

- Select two clauses from set containing complementary literals
- Find MGU for these literals
- Apply MGU to both clauses & resolve them into a new clause by removing complement any literals & combining the rest
- If resolvent is empty, return "Query proved"
- If resolvent is new, add it to set

Step 4: If no empty clause is derived and no new clauses can be added return "Query cannot be proved"

```

class PseudoCNF:
    def solve(self, clauses):
        s1 = set()
        s2 = set()
        resolvent = set()
        for clause in clauses:
            if clause == []:
                return "unsatisfiable"
            if clause.issubset(s1):
                resolvent.add(clause - s1)
            if clause.issubset(s2):
                resolvent.add(clause - s2)
            if len(resolvent) == 0:
                return "satisfiable"
            else:
                s1.add(clause)
                s2.add(resolvent.pop())
        return self.check_if_satisfiable(resolvent)

    def check_if_satisfiable(self, resolvent):
        if len(resolvent) == 0:
            return "satisfiable"
        else:
            return "unsatisfiable"

    def check_if_unsatisfiable(self, resolvent):
        if len(resolvent) == 0:
            return "unsatisfiable"
        else:
            return "satisfiable"

```

Scanned with CamScanner

Code:

from copy import deepcopy

```

def print_step(title, content):
    print(f"\n{'='*45}\n{title}\n{'='*45}")
    if isinstance(content, list):
        for i, c in enumerate(content, 1):
            print(f"{i}. {c}")
    else:
        print(content)

```

```

KB = [
    ["¬Food(x)", "Likes(John,x)"],
    ["Food(Apple)"],
    ["Food(Vegetable)"],
    ["¬Eats(x,y)", "Killed(x)", "Food(y)"],
    ["Eats(Anil,Peanuts)"],
    ["Alive(Anil)"],
    ["¬Alive(x)", "¬Killed(x)"],
    ["Killed(x)", "Alive(x)"]
]

```

QUERY = ["Likes(John,Peanuts)"]

```

def negate(literal):
    if literal.startswith("¬"):

```

```

        return literal[1:]
    return "¬" + literal

def substitute(clause, subs):
    new_clause = []
    for lit in clause:
        for var, val in subs.items():
            lit = lit.replace(var, val)
        new_clause.append(lit)
    return new_clause

def unify(lit1, lit2):
    """Small unifier for patterns like Food(x) and Food(Apple)."""
    if "(" not in lit1 or "(" not in lit2:
        return None
    pred1, args1 = lit1.split("(")
    pred2, args2 = lit2.split("(")
    args1 = args1[:-1].split(",")
    args2 = args2[:-1].split(",")
    if pred1 != pred2 or len(args1) != len(args2):
        return None
    subs = {}
    for a, b in zip(args1, args2):
        if a == b:
            continue
        if a.islower():
            subs[a] = b
        elif b.islower():
            subs[b] = a
        else:
            return None
    return subs

def resolve(ci, cj):
    """Return list of (resolvent, substitution, pair)."""
    resolvents = []
    for li in ci:
        for lj in cj:
            if li == negate(lj):
                new_clause = [x for x in ci if x != li] + [x for x in cj if x != lj]
                resolvents.append((list(set(new_clause)), {}, (li, lj)))
            else:
                # same predicate, opposite sign
                if li.startswith("¬") and not lj.startswith("¬") and li[1:].split("(")[0] == lj.split("(")[0]:
                    subs = unify(li[1:], lj)
                    if subs:
                        new_clause = substitute([x for x in ci if x != li] + [x for x in cj if x != lj], subs)

```

```

        resolvents.append((list(set(new_clause)), subs, (li, lj)))
    elif lj.startswith("¬") and not li.startswith("¬") and lj[1:].split("(")[0] == li.split("(")[0]:
        subs = unify(lj[1:], li)
        if subs:
            new_clause = substitute([x for x in ci if x != li] + [x for x in cj if x != lj], subs)
            resolvents.append((list(set(new_clause)), subs, (li, lj)))
    return resolvents

def resolution(kb, query):
    clauses = deepcopy(kb)
    negated_query = [negate(q) for q in query]
    clauses.append(negated_query)
    print_step("Initial Clauses", clauses)

    steps = []
    new = []
    while True:
        pairs = [(clauses[i], clauses[j]) for i in range(len(clauses))
                  for j in range(i + 1, len(clauses))]
        for (ci, cj) in pairs:
            for r, subs, pair in resolve(ci, cj):
                if not r:
                    steps.append({
                        "parents": (ci, cj),
                        "resolvent": r,
                        "subs": subs
                    })
                    print_tree(steps)
                    print("\n Empty clause derived — query proven.")
                    return True
                if r not in clauses and r not in new:
                    new.append(r)
                    steps.append({
                        "parents": (ci, cj),
                        "resolvent": r,
                        "subs": subs
                    })
        if all(r in clauses for r in new):
            print_step("No New Clauses", "Query cannot be proven ")
            print_tree(steps)
            return False
        clauses.extend(new)

def print_tree(steps):
    print("\n" + "="*45)
    print("Resolution Proof Trace")
    print("="*45)

```

```

for i, s in enumerate(steps, 1):
    p1, p2 = s["parents"]
    r = s["resolvent"]
    subs = s["subs"]
    subs_text = f" Substitution: {subs}" if subs else ""
    print(f" Resolve {p1} and {p2}")
    if subs_text:
        print(subs_text)
    if r:
        print(f" => {r}")
    else:
        print(" => {} (empty clause)")
    print("-"*45)

def main():
    print_step("Knowledge Base in CNF", KB)
    print_step("Negated Query", [negate(q) for q in QUERY])
    proven = resolution(KB, QUERY)
    if proven:
        print("\n Query Proven by Resolution: John likes peanuts.")
    else:
        print("\n Query cannot be proven from KB.")

if __name__ == "__main__":
    main()

```

Program 10

Implement Alpha-Beta Pruning.

Algorithm:

Date _____
Page _____

Ques: Q) Implement Alpha-Beta Pruning.

Alpha = max, Beta = min

Input: root node

Output: leaf node

Variables: depth, maxSearchDepth, alpha, beta

alpha: best value found so far for maximizing player (initially -∞)

beta: best value found so far for minimizing player (initially +∞)

maxSearchDepth: boolean indicating if search for current player is maximizing or minimizing

Optimal heuristic value for all children of current player for one subtree is stored at node.

Algorithm

Step 1: Terminal test or depth limit

If depth == 0 or node is a terminal state, return the heuristic evaluation of node.

Step 2: If maximizing player

Update maxEval = max(maxEval, eval)

for each child of node

eval = alpha-beta-pruning(child, depth-1, alpha, beta)

maxEval = max(maxEval, eval)

alpha = max(alpha, eval)

if eval > beta: break out of loop

Step 3: If minimizing player

Update minEval = min(minEval, eval)

for each child of node

eval = alpha-beta-pruning(child, depth-1, alpha, beta)

minEval = min(minEval, eval)

beta = min(beta, eval)

if eval < alpha: break out of loop

Date _____
Page _____

Update maxEval = max(maxEval, eval)

Update alpha = max(alpha, maxEval)

If beta <= alpha: pruning remaining children by breaking out of loop

maxEval = max(maxEval, eval)

alpha = max(alpha, maxEval)

Step 3: If minimizing player

Update minEval = min(minEval, eval)

for each child of node

eval = alpha-beta-pruning(child, depth-1, alpha, beta)

minEval = min(minEval, eval)

beta = min(beta, eval)

if eval < alpha: break out of loop

return maxEval

Output: Leaf node values: [0, 5, 6, 7, 1, 2, 3, 4]

Optimal value: 5

Root node values: [2, 1, 5, 3, 2, 4, 6, 7]

MaxEval: 7

MinEval: 1

Alpha-Beta Pruning: [2, 1, 5, 3, 2, 4, 6, 7] / [0, 5, 6, 7, 1, 2, 3, 4]

Pseudo code

```

def alphaBetaPruning(node, depth, alpha, beta, maximizingPlayer):
    if depth == maxSearchDepth or isTerminal(node):
        return evaluate(node)
    if maximizingPlayer:
        maxEval = float("-inf")
        for child in getChildren(node):
            eval = alphaBetaPruning(child, depth-1, alpha, beta)
            maxEval = max(maxEval, eval)
            if eval > beta:
                break
        return maxEval
    else:
        minEval = float("inf")
        for child in getChildren(node):
            eval = alphaBetaPruning(child, depth-1, alpha, beta)
            minEval = min(minEval, eval)
            if eval < alpha:
                break
        return minEval

```

Code:

```
def unify(a, b):
    """Very simple unification for small terms like ('line', [X,O,X])"""
    if a == b:
        return {}
    if isinstance(a, str) and a.islower(): # variable
        return {a: b}
    if isinstance(b, str) and b.islower():
        return {b: a}
    if isinstance(a, tuple) and isinstance(b, tuple):
        if a[0] != b[0] or len(a[1]) != len(b[1]):
            return None
        subs = {}
        for x, y in zip(a[1], b[1]):
            s = unify(x, y)
            if s is None:
                return None
            subs.update(s)
        return subs
    return None
```

```
# Winning triples (rows, cols, diagonals)
```

```
WIN_TRIPLES = [(0,1,2),(3,4,5),(6,7,8),(0,3,6),(1,4,7),(2,5,8),(0,4,8),(2,4,6)]
```

```
def winner(board):
    pattern = ('line', ['X','X','X'])
    for i,j,k in WIN_TRIPLES:
        term = ('line', [board[i], board[j], board[k]])
        if unify(term, pattern):
            return 'X'
    if unify(term, ('line',[ 'O','O','O'])):
        return 'O'
    return None
```

```
def is_full(board): return all(c != '_' for c in board)
```

```
def evaluate(board):
    w = winner(board)
    if w == 'X': return 1
    if w == 'O': return -1
    if is_full(board): return 0
    return None
```

```
def alpha_beta(board, player, alpha=-float('inf'), beta=float('inf')):
    val = evaluate(board)
    if val is not None:
```

```

return val, None

moves = [i for i,c in enumerate(board) if c == '_']
best_move = None
if player == 'X':
    max_eval = -float('inf')
    for m in moves:
        new_board = board[:]
        new_board[m] = 'X'
        eval_, _ = alpha_beta(new_board, 'O', alpha, beta)
        if eval_ > max_eval:
            max_eval, best_move = eval_, m
            alpha = max(alpha, eval_)
            if beta <= alpha: break
    return max_eval, best_move
else:
    min_eval = float('inf')
    for m in moves:
        new_board = board[:]
        new_board[m] = 'O'
        eval_, _ = alpha_beta(new_board, 'X', alpha, beta)
        if eval_ < min_eval:
            min_eval, best_move = eval_, m
            beta = min(beta, eval_)
            if beta <= alpha: break
    return min_eval, best_move

def print_board(b):
    for i in range(0,9,3):
        print(' '.join(b[i:i+3]))
    print()

# --- Example usage ---
board = ['_']*9
score, move = alpha_beta(board, 'X')
print("Best first move for X:", move)
board[move] = 'X'
print_board(board)

```