**POAI LAB PROGRAMS:**

1. N = 8def print\_solution(board): for row in board: line = "" for col in row: line += "Q " if col else ". " print(line) print()def is\_safe(board, row, col): # Check this row on the left for i in range(col): if board[row][i]: return False # Check upper diagonal on the left for i, j in zip(range(row, -1, -1), range(col, -1, -1)): if board[i][j]: return False # Check lower diagonal on the left for i, j in zip(range(row, N, 1), range(col, -1, -1)): if board[i][j]: return False return Truedef solve\_n\_queens(board, col): if col >= N: print\_solution(board) return True res = False for i in range(N): if is\_safe(board, i, col): board[i][col] = True res = solve\_n\_queens(board, col + 1) or res board[i][col] = False # backtrack return resdef solve(): board = [[False] \* N for \_ in range(N)] if not solve\_n\_queens(board, 0): print("No solution exists")solve()2 : def dfs(graph, start, goal, path=None, visited=None): if visited is None: visited = set() if path is None: path = [] visited.add(start) path.append(start) if start == goal: return path for neighbor in graph.get(start, []): if neighbor not in visited: result = dfs(graph, neighbor, goal, path.copy(), visited.copy()) if result: return result return None# Example graph (as an adjacency list)graph = { 'A': ['B', 'C'], 'B': ['D', 'E'], 'C': ['F'], 'D': [], 'E': ['F'], 'F': []}# Call the DFS functionstart\_node = 'A'goal\_node = 'F'path = dfs(graph, start\_node, goal\_node)# Outputif path: print(f"Path from {start\_node} to {goal\_node}: {' -> '.join(path)}")else: print("No path found.") 3: import math# ConstantsPLAYER = 'X' # HumanAI = 'O' # AIEMPTY = ' '# Initialize boardboard = [EMPTY] \* 9def print\_board(board): for i in range(3): print(board[i\*3:(i+1)\*3]) print()def is\_winner(board, player): win\_conditions = [ [0,1,2], [3,4,5], [6,7,8], # rows [0,3,6], [1,4,7], [2,5,8], # columns [0,4,8], [2,4,6] # diagonals ] return any(all(board[i] == player for i in combo) for combo in win\_conditions)def is\_draw(board): return EMPTY not in boarddef minimax(board, is\_maximizing): if is\_winner(board, AI): return 1 if is\_winner(board, PLAYER): return -1 if is\_draw(board): return 0 if is\_maximizing: best\_score = -math.inf for i in range(9): if board[i] == EMPTY: board[i] = AI score = minimax(board, False) board[i] = EMPTY best\_score = max(score, best\_score) return best\_score else: best\_score = math.inf for i in range(9): if board[i] == EMPTY: board[i] = PLAYER score = minimax(board, True) board[i] = EMPTY best\_score = min(score, best\_score) return best\_scoredef best\_move(board): best\_score = -math.inf move = None for i in range(9): if board[i] == EMPTY: board[i] = AI score = minimax(board, False) board[i] = EMPTY if score > best\_score: best\_score = score move = i return move# Play demoprint("Initial board:")print\_board(board)# Simulate one AI movemove = best\_move(board)board[move] = AIprint("Board after AI move:")print\_board(board)4: def heuristic(a, b): return abs(a[0] - b[0]) + abs(a[1] - b[1])def a\_star(grid, start, goal): open\_list = [start] came\_from = {} g\_score = {start: 0} f\_score = {start: heuristic(start, goal)} while open\_list: # Get node with lowest f\_score current = min(open\_list, key=lambda x: f\_score.get(x, float('inf'))) if current == goal: # Reconstruct path path = [] while current in came\_from: path.append(current) current = came\_from[current] path.append(start) return path[::-1] open\_list.remove(current) x, y = current for dx, dy in [(-1,0),(1,0),(0,-1),(0,1)]: neighbor = (x+dx, y+dy) if 0 <= neighbor[0] < len(grid) and 0 <= neighbor[1] < len(grid[0]): if grid[neighbor[0]][neighbor[1]] == 1: continue # wall tentative\_g = g\_score[current] + 1 if tentative\_g < g\_score.get(neighbor, float('inf')): came\_from[neighbor] = current g\_score[neighbor] = tentative\_g f\_score[neighbor] = tentative\_g + heuristic(neighbor, goal) if neighbor not in open\_list: open\_list.append(neighbor) return None# Grid: 0 = free, 1 = wallgrid = [ [0, 0, 0, 0], [1, 1, 0, 1], [0, 0, 0, 0], [0, 1, 1, 0]]start = (0, 0)goal = (3, 3)path = a\_star(grid, start, goal)print("Path:", path)5: def unify(x, y, subs={}): if x == y: return subs if is\_var(x): return unify\_var(x, y, subs) if is\_var(y): return unify\_var(y, x, subs) return Nonedef unify\_var(var, x, subs): if var in subs: return unify(subs[var], x, subs) subs[var] = x return subsdef is\_var(x): return isinstance(x, str) and x.islower()# Example:print(unify('x', 'John')) # {'x': 'John'}print(unify('x', 'y')) # {'x': 'y'}print(unify('x', 'x')) # {}print(unify('x', 'Mary')) # {'x': 'Mary'}def negate(lit): return lit[1:] if lit.startswith('~') else '~' + litdef resolve(c1, c2): for l in c1: if negate(l) in c2: return (c1 - {l}) | (c2 - {negate(l)}) return None# Example:c1 = {'A', 'B'}c2 = {'~B', 'C'}resolvent = resolve(c1, c2)print(resolvent) # {'A', 'C'}6: B CHAIN# Knowledge base: rules and factsrules = { 'wet\_grass': ['rain', 'sprinkler\_on'], 'rain': [], 'sprinkler\_on': []}facts = {'rain'} # Known factsdef backward\_chain(goal): # If goal is a known fact if goal in facts: return True # If goal is not in rules, can't prove if goal not in rules: return False # Try to prove all subgoals in the rule for the goal for subgoal in rules[goal]: if not backward\_chain(subgoal): return False return True# Test the systemgoal = 'wet\_grass'result = backward\_chain(goal)print(f"Is '{goal}' true? -> {result}")7: F Chain# Knowledge base: rules and factsrules = { 'wet\_grass': ['rain', 'sprinkler\_on'], 'rain': [], 'sprinkler\_on': []}facts = {'rain'} # Known facts initiallydef forward\_chain(rules, facts): inferred = set(facts) added = True while added: added = False for conclusion, premises in rules.items(): if conclusion not in inferred and all(p in inferred for p in premises): inferred.add(conclusion) added = True return inferred# Run forward chaininginferred\_facts = forward\_chain(rules, facts)print("Inferred facts:", inferred\_facts)