**1. 8-Puzzle Problem (Using A Algorithm)**

**import heapq**

**goal\_state = [[1,2,3],[4,5,6],[7,8,0]]**

**directions = [(0,1),(1,0),(-1,0),(0,-1)] # Right, Down, Up, Left**

**def heuristic(state):**

**h = 0**

**for i in range(3):**

**for j in range(3):**

**val = state[i][j]**

**if val != 0:**

**goal\_x, goal\_y = divmod(val-1, 3)**

**h += abs(goal\_x - i) + abs(goal\_y - j)**

**return h**

**def a\_star(start):**

**start\_tuple = tuple(map(tuple, start))**

**goal\_tuple = tuple(map(tuple, goal\_state))**

**heap = [(heuristic(start), 0, start)]**

**visited = set()**

**while heap:**

**est, cost, state = heapq.heappop(heap)**

**state\_tuple = tuple(map(tuple, state))**

**if state\_tuple in visited:**

**continue**

**visited.add(state\_tuple)**

**if state\_tuple == goal\_tuple:**

**return state**

**zero\_x, zero\_y = [(ix, iy) for ix, row in enumerate(state)**

**for iy, val in enumerate(row) if val == 0][0]**

**for dx, dy in directions:**

**nx, ny = zero\_x + dx, zero\_y + dy**

**if 0 <= nx < 3 and 0 <= ny < 3:**

**new\_state = [row[:] for row in state]**

**new\_state[zero\_x][zero\_y], new\_state[nx][ny] = new\_state[nx][ny], new\_state[zero\_x][zero\_y]**

**heapq.heappush(heap, (cost + 1 + heuristic(new\_state), cost + 1, new\_state))**

**return None**

**start\_state = [[1, 2, 3], [4, 0, 6], [7, 5, 8]]**

**solution = a\_star(start\_state)**

**print("Solution:")**

**for row in solution:**

**print(row)**

**2. 8-Queens Problem**

**def is\_safe(board, row, col):**

**for i in range(col):**

**if board[i] == row or \**

**board[i] - i == row - col or \**

**board[i] + i == row + col:**

**return False**

**return True**

**def solve\_queens(col, board, n):**

**if col == n:**

**print(board)**

**return**

**for row in range(n):**

**if is\_safe(board, row, col):**

**board[col] = row**

**solve\_queens(col+1, board, n)**

**n = 8**

**solve\_queens(0, [-1]\*n, n)**

**3. Water Jug Problem**

**def water\_jug():**

**visited = set()**

**goal = (2, 0) # Example: get 2 liters in 4-liter jug**

**capacity = (4, 3)**

**q = [(0, 0)]**

**while q:**

**(x, y) = q.pop(0)**

**if (x, y) in visited:**

**continue**

**print(f"{x} liters in Jug1, {y} liters in Jug2")**

**if (x, y) == goal:**

**return**

**visited.add((x, y))**

**q += list(set([**

**(0, y), (x, 0), (capacity[0], y), (x, capacity[1]),**

**(min(x + y, capacity[0]), max(0, x + y - capacity[0])),**

**(max(0, x + y - capacity[1]), min(x + y, capacity[1]))**

**]) - visited)**

**water\_jug()**

**4.**

**Cryptarithmetic Problem (SEND + MORE = MONEY)**

**import itertools**

**def solve\_crypt():**

**letters = 'SENDMORY'**

**for perm in itertools.permutations(range(10), len(letters)):**

**s, e, n, d, m, o, r, y = perm**

**if s == 0 or m == 0:**

**continue**

**send = s\*1000 + e\*100 + n\*10 + d**

**more = m\*1000 + o\*100 + r\*10 + e**

**money = m\*10000 + o\*1000 + n\*100 + e\*10 + y**

**if send + more == money:**

**print(f"SEND: {send}, MORE: {more}, MONEY: {money}")**

**return**

**solve\_crypt()**

**5.**

**Missionaries and Cannibals**

**def is\_valid(m, c):**

**return m == 0 or m >= c**

**def successors(state):**

**moves = [(1,0),(2,0),(0,1),(0,2),(1,1)]**

**successors = []**

**side = -1 if state[2] else 1**

**for m, c in moves:**

**new\_state = [state[0] + m\*side, state[1] + c\*side, 1 - state[2]]**

**if 0 <= new\_state[0] <= 3 and 0 <= new\_state[1] <= 3:**

**if is\_valid(new\_state[0], new\_state[1]) and is\_valid(3 - new\_state[0], 3 - new\_state[1]):**

**successors.append(tuple(new\_state))**

**return successors**

**def solve():**

**from collections import deque**

**start = (3, 3, 1)**

**goal = (0, 0, 0)**

**q = deque([(start, [start])])**

**visited = set()**

**while q:**

**(state, path) = q.popleft()**

**if state == goal:**

**for step in path:**

**print(step)**

**return**

**for s in successors(state):**

**if s not in visited:**

**visited.add(s)**

**q.append((s, path + [s]))**

**solve()**

**6.**

**Vacuum Cleaner Problem**

**def vacuum\_cleaner(states):**

**for i, state in enumerate(states):**

**print(f"Room {i + 1} is {'Dirty' if state else 'Clean'}")**

**if state:**

**print(f"Cleaning room {i + 1}...")**

**print("All rooms are clean!")**

**vacuum\_cleaner([1, 0, 1]) # 1 = Dirty, 0 = Clean**

**7.**

**Breadth-First Search (BFS)**

**from collections import deque**

**def bfs(graph, start):**

**visited = set()**

**queue = deque([start])**

**while queue:**

**node = queue.popleft()**

**if node not in visited:**

**print(node)**

**visited.add(node)**

**queue.extend(graph[node] - visited)**

**graph = {**

**'A': {'B', 'C'},**

**'B': {'D', 'E'},**

**'C': {'F'},**

**'D': set(),**

**'E': {'F'},**

**'F': set()**

**}**

**bfs(graph, 'A')**

**8.**

**Depth-First Search (DFS)**

**def dfs(graph, start, visited=None):**

**if visited is None:**

**visited = set()**

**print(start)**

**visited.add(start)**

**for neighbor in graph[start]:**

**if neighbor not in visited:**

**dfs(graph, neighbor, visited)**

**dfs(graph, 'A')**

**9.**

**Travelling Salesman Problem (Brute Force)**

**import itertools**

**def tsp(graph, start):**

**vertices = list(graph.keys())**

**vertices.remove(start)**

**min\_path = float('inf')**

**best\_path = []**

**for perm in itertools.permutations(vertices):**

**current\_weight = 0**

**k = start**

**for j in perm:**

**current\_weight += graph[k][j]**

**k = j**

**current\_weight += graph[k][start]**

**if current\_weight < min\_path:**

**min\_path = current\_weight**

**best\_path = [start] + list(perm) + [start]**

**print("Path:", best\_path, "Cost:", min\_path)**

**graph = {**

**'A': {'A': 0, 'B': 10, 'C': 15, 'D': 20},**

**'B': {'A': 10, 'B': 0, 'C': 35, 'D': 25},**

**'C': {'A': 15, 'B': 35, 'C': 0, 'D': 30},**

**'D': {'A': 20, 'B': 25, 'C': 30, 'D': 0}**

**}**

**tsp(graph, 'A')**

**10.**

**A Algorithm (General Form)**

**\***

**from queue import PriorityQueue**

**def a\_star\_general(graph, start, goal, h):**

**open\_set = PriorityQueue()**

**open\_set.put((0, start))**

**came\_from = {}**

**g\_score = {node: float("inf") for node in graph}**

**g\_score[start] = 0**

**while not open\_set.empty():**

**current = open\_set.get()[1]**

**if current == goal:**

**path = []**

**while current in came\_from:**

**path.append(current)**

**current = came\_from[current]**

**path.append(start)**

**return path[::-1]**

**for neighbor, cost in graph[current].items():**

**tentative = g\_score[current] + cost**

**if tentative < g\_score[neighbor]:**

**came\_from[neighbor] = current**

**g\_score[neighbor] = tentative**

**open\_set.put((tentative + h[neighbor], neighbor))**

**graph = {**

**'A': {'B': 1, 'C': 4},**

**'B': {'C': 2, 'D': 5},**

**'C': {'D': 1},**

**'D': {}**

**}**

**heuristic = {'A': 7, 'B': 6, 'C': 2, 'D': 0}**

**print("Path:", a\_star\_general(graph, 'A', 'D', heuristic))**

**11. Map Coloring using CSP (Python)**

**from constraint import Problem**

**problem = Problem()**

**colors = ["Red", "Green", "Blue"]**

**regions = ["WA", "NT", "SA", "Q", "NSW", "V", "T"]**

**for region in regions:**

**problem.addVariable(region, colors)**

**problem.addConstraint(lambda a, b: a != b, ("WA", "NT"))**

**problem.addConstraint(lambda a, b: a != b, ("WA", "SA"))**

**problem.addConstraint(lambda a, b: a != b, ("NT", "SA"))**

**problem.addConstraint(lambda a, b: a != b, ("NT", "Q"))**

**problem.addConstraint(lambda a, b: a != b, ("SA", "Q"))**

**problem.addConstraint(lambda a, b: a != b, ("SA", "NSW"))**

**problem.addConstraint(lambda a, b: a != b, ("SA", "V"))**

**problem.addConstraint(lambda a, b: a != b, ("Q", "NSW"))**

**problem.addConstraint(lambda a, b: a != b, ("NSW", "V"))**

**solutions = problem.getSolutions()**

**for sol in solutions:**

**print(sol)**

**break**

**12. Tic Tac Toe Game (Python)**

**def print\_board(board):**

**for row in board:**

**print(" | ".join(row))**

**print("-"\*5)**

**def check\_winner(board, player):**

**for row in board:**

**if all(cell == player for cell in row):**

**return True**

**for col in range(3):**

**if all(board[r][col] == player for r in range(3)):**

**return True**

**if all(board[i][i] == player for i in range(3)) or \**

**all(board[i][2-i] == player for i in range(3)):**

**return True**

**return False**

**def tic\_tac\_toe():**

**board = [[" "]\*3 for \_ in range(3)]**

**player = "X"**

**for \_ in range(9):**

**print\_board(board)**

**x, y = map(int, input(f"Player {player}, enter row and col: ").split())**

**if board[x][y] != " ":**

**print("Cell occupied!")**

**continue**

**board[x][y] = player**

**if check\_winner(board, player):**

**print\_board(board)**

**print(f"Player {player} wins!")**

**return**

**player = "O" if player == "X" else "X"**

**print("It's a draw!")**

**# tic\_tac\_toe()**

**13. Minimax Algorithm (Python)**

**def minimax(depth, node, is\_max, values, alpha, beta):**

**if depth == 3:**

**return values[node]**

**if is\_max:**

**best = float('-inf')**

**for i in range(2):**

**val = minimax(depth+1, node\*2+i, False, values, alpha, beta)**

**best = max(best, val)**

**return best**

**else:**

**best = float('inf')**

**for i in range(2):**

**val = minimax(depth+1, node\*2+i, True, values, alpha, beta)**

**best = min(best, val)**

**return best**

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

**print("Best value:", minimax(0, 0, True, values, -999, 999))**

**14. Alpha-Beta Pruning (Python)**

**def minimax(depth, node, is\_max, values, alpha, beta):**

**if depth == 3:**

**return values[node]**

**if is\_max:**

**best = float('-inf')**

**for i in range(2):**

**val = minimax(depth+1, node\*2+i, False, values, alpha, beta)**

**best = max(best, val)**

**alpha = max(alpha, best)**

**if beta <= alpha:**

**break**

**return best**

**else:**

**best = float('inf')**

**for i in range(2):**

**val = minimax(depth+1, node\*2+i, True, values, alpha, beta)**

**best = min(best, val)**

**beta = min(beta, best)**

**if beta <= alpha:**

**break**

**return best**

**15. Decision Tree (Python using sklearn)**

**from sklearn import tree**

**features = [[1, 1], [1, 0], [0, 1], [0, 0]]**

**labels = [1, 1, 0, 0]**

**clf = tree.DecisionTreeClassifier()**

**clf = clf.fit(features, labels)**

**print(clf.predict([[1, 1], [0, 0]]))**

**16. Feed Forward Neural Network (Python using keras)**

**from keras.models import Sequential**

**from keras.layers import Dense**

**import numpy as np**

**X = np.array([[0,0],[0,1],[1,0],[1,1]])**

**y = np.array([[0],[1],[1],[0]])**

**model = Sequential()**

**model.add(Dense(4, input\_dim=2, activation='relu'))**

**model.add(Dense(1, activation='sigmoid'))**

**model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy'])**

**model.fit(X, y, epochs=500, verbose=0)**

**print(model.predict(X))**