**1. 8-Puzzle problem**

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

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

**queue = [[initial, []]]**

**visited = []**

**while queue:**

**state, path = queue.pop(0)**

**visited.append(state)**

**if state == goal:**

**print("Solution:", path)**

**break**

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

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

**if state[i][j] == 0:**

**x, y = i, j**

**for dx, dy, move in [(-1, 0, 'Up'), (1, 0, 'Down'), (0, -1, 'Left'), (0, 1, 'Right')]:**

**nx, ny = x + dx, y + dy**

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

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

**new\_state[x][y], new\_state[nx][ny] = new\_state[nx][ny], new\_state[x][y]**

**if new\_state not in visited: queue.append([new\_state, path + [move]])**

**2. 8-Queen**

**def solve(n, board=[], solutions=[]):**

**if len(board) == n: solutions.append(board)**

**return**

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

**if all(col != c and abs(len(board) - r) != abs(col - c) for r, c in enumerate(board)):**

**solve(n, board + [col], solutions)**

**n = 8**

**solutions = []**

**solve(n, solutions=solutions)**

**for solution in solutions[:1]:**

**for row in solution:**

**print(''.join('Q' if i == row else '.' for i in range(n)))**

**print()**

**3.water jug**

**x, y = 0, 0**

**capacity1, capacity2 = 4, 3**

**goal = 2**

**while True:**

**print(f"Jug 1: {x}, Jug 2: {y}")**

**if x == goal or y == goal:**

**print("Solution found!")**

**break**

**if x == 0:**

**x = capacity1**

**elif y == capacity2:**

**y = 0**

**else:**

**transfer = min(x, capacity2 - y)**

**x -= transfer**

**y += transfer**

**4.Cript-Arithmetic**

**from itertools import permutations**

**letters = 'SENDMORY'**

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

**mapping = dict(zip(letters, perm))**

**S, E, N, D = mapping['S'], mapping['E'], mapping['N'], mapping['D']**

**M, O, R, Y = mapping['M'], mapping['O'], mapping['R'], mapping['Y']**

**if S == 0 or M == 0:**

**continue**

**send = 1000 \* S + 100 \* E + 10 \* N + D**

**more = 1000 \* M + 100 \* O + 10 \* R + E**

**money = 10000 \* M + 1000 \* O + 100 \* N + 10 \* E + Y**

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

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

**break**

**5.Missionaries Cannibal**

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

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

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

**def is\_valid(state):**

**m, c, \_ = state**

**return (0 <= m <= 3 and 0 <= c <= 3 and**

**(m == 0 or m >= c) and (3 - m == 0 or 3 - m >= 3 - c))**

**def next\_states(state):**

**m, c, b = state**

**for dm, dc in moves:**

**if b == 1:**

**new\_state = (m - dm, c - dc, 0)**

**else:**

**new\_state = (m + dm, c + dc, 1)**

**if is\_valid(new\_state):**

**yield new\_state**

**visited = set()**

**path = [(start, [])]**

**while path:**

**state, steps = path.pop(0)**

**if state in visited:**

**continue**

**visited.add(state)**

**if state == goal:**

**print("Solution:", steps + [goal])**

**break**

**for next\_state in next\_states(state): path.append((next\_state, steps + [state]))**

**6.Vacuum Cleaner**

**left, right, pos = 1, 1, 0**

**visited = []**

**path = [(left, right, pos)]**

**while path:**

**state = path.pop(0)**

**left, right, pos = state**

**if left == 0 and right == 0:**

**print("Solution:", path)**

**break**

**if state in visited:**

**continue**

**visited.append(state)**

**if pos == 0:**

**if left == 1:**

**new\_state = (0, right, 0) path.append(new\_state)**

**if right != 0:**

**new\_state = (left, right, 1) path.append(new\_state)**

**else:**

**if right == 1:**

**new\_state = (left, 0, 1) path.append(new\_state)**

**if left != 0:**

**new\_state = (left, right, 0) path.append(new\_state)**

**7.BFS**

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

**visited = set()**

**queue = [start]**

**while queue:**

**node = queue.pop(0)**

**if node not in visited:**

**print(node, end=" ")**

**visited.add(node)**

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

**if neighbor not in visited:**

**queue.append(neighbor)**

**graph = {**

**'A': ['B', 'C'],**

**'B': ['A', 'D', 'E'],**

**'C': ['A', 'F'],**

**'D': ['B'],**

**'E': ['B', 'F'],**

**'F': ['C', 'E']**

**}**

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

**8.DFS**

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

**if visited is None:**

**visited = set()**

**visited.add(start)**

**print(start, end=" ")**

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

**if neighbor not in visited:**

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

**graph = {**

**'A': ['B', 'C'],**

**'B': ['A', 'D', 'E'],**

**'C': ['A', 'F'],**

**'D': ['B'],**

**'E': ['B', 'F'],**

**'F': ['C', 'E']**

**}**

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

**9.Travelling Salesman**

**import itertools**

**def calculate\_distance(city1, city2):**

**return ((city1[0] - city2[0]) \* 2 + (city1[1] - city2[1]) \* 2) \*\* 0.5**

**def tsp(cities):**

**best\_route = None**

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

**for route in itertools.permutations(cities):**

**dist = sum(calculate\_distance(route[i], route[i + 1]) for i in range(len(route) - 1))**

**dist += calculate\_distance(route[-1], route[0])**

**if dist < min\_distance:**

**min\_distance = dist**

**best\_route = route**

**return best\_route, min\_distance**

**cities = [(0, 0), (2, 2), (2, 4), (4, 0)]**

**best\_route, min\_distance = tsp(cities)**

**print("Best route:", best\_route)**

**print("Minimum distance:", min\_distance)**

**10.A\*algorithm**

**import heapq**

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

**open\_list = [(0 + heuristic[start], 0, start)]**

**closed\_list = set()**

**while open\_list:**

**\_, g, current = heapq.heappop(open\_list)**

**if current == goal:**

**return g closed\_list.add(current)**

**for neighbor, cost in graph[current]:**

**if neighbor not in closed\_list:**

**f = g + cost + heuristic.get(neighbor, 0)**

**heapq.heappush(open\_list, (f, g + cost, neighbor))**

**return None**

**graph = {**

**'A': [('B', 1), ('C', 4)],**

**'B': [('A', 1), ('C', 2), ('D', 5)],**

**'C': [('A', 4), ('B', 2), ('D', 1)],**

**'D': [('B', 5), ('C', 1)]**

**}**

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

**cost = a\_star(graph, 'A', 'D', heuristic)**

**print("Cost:", cost)**

**11.Map Coloring CPS**

**def is\_valid(graph, coloring, region, color):**

**return all(coloring.get(neighbor) != color for neighbor in graph[region])**

**def map\_coloring(graph, colors, coloring={}):**

**if len(coloring) == len(graph):**

**return coloring**

**for region in graph:**

**if region not in coloring:**

**for color in colors:**

**if is\_valid(graph, coloring, region, color): coloring[region] = color**

**result = map\_coloring(graph, colors, coloring)**

**if result:**

**return result**

**del coloring[region]**

**return None**

**graph = {**

**'A': ['B', 'C'],**

**'B': ['A', 'D', 'E'],**

**'C': ['A', 'E'],**

**'D': ['B'],**

**'E': ['B', 'C']**

**}**

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

**solution = map\_coloring(graph, colors)**

**print(solution)**

**12.Tic Tac Toe**

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

**for row in board:**

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

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

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

**return any(all(cell == player for cell in row) for row in board) or \**

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

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

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

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

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

**players = ['X', 'O']**

**turn = 0**

**while True:**

**print\_board(board)**

**row, col = map(int, input(f"Player {players[turn % 2]}, enter row and column: ").split())**

**if board[row][col] == ' ':**

**board[row][col] = players[turn % 2]**

**if check\_win(board, players[turn % 2]): print\_board(board)**

**print(f"Player {players[turn % 2]} wins!")**

**break**

**if all(cell != ' ' for row in board for cell in row): print\_board(board)**

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

**break**

**turn += 1**

**else:**

**print("Invalid move. Try again.")**

**tic\_tac\_toe()**

**13. Minimax algorithm**

**def minimax(depth, node\_index, maximizing\_player, values, alpha, beta):**

**if depth == 0:**

**return values[node\_index]**

**if maximizing\_player:**

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

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

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

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

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

**if beta <= alpha:**

**break**

**return best**

**def best\_move(values, depth):**

**return minimax(depth, 0, True, values, -float('inf'), float('inf'))**

**values = [3, 5, 2, 9, 12, 4, 6, 8]**

**depth = 3**

**best\_value = best\_move(values, depth)**

**print("Best Value for the Root Node: ", best\_value)**

**14. Apha & Beta pruning algorithm**

**def alpha\_beta(depth, node\_index, maximizing\_player, values, alpha, beta):**

**if depth == 0:**

**return values[node\_index]**

**if maximizing\_player:**

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

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

**best = max(best, alpha\_beta(depth - 1, node\_index \* 2 + i, False, values, alpha, beta))**

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

**if beta <= alpha:**

**break**

**return best**

**else:**

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

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

**best = min(best, alpha\_beta(depth - 1, node\_index \* 2 + i, True, values, alpha, beta))**

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

**if beta <= alpha:**

**break**

**return best**

**def best\_move(values, depth):**

**return alpha\_beta(depth, 0, True, values, -float('inf'), float('inf'))**

**values = [3, 5, 2, 9, 12, 4, 6,**

**8]**

**depth = 3**

**print("Best Value for the Root Node:", best\_move(values, depth))**

**15. Decision Tree**

**import math**

**def entropy(data):**

**labels = [row[-1] for row in data]**

**return -sum(labels.count(label)/len(labels) \* math.log2(labels.count(label)/len(labels)) for label in set(labels))**

**def information\_gain(data, feature\_index):**

**total\_entropy = entropy(data)**

**values = set(row[feature\_index] for row in data)**

**weighted\_entropy = sum((len([row for row in data if row[feature\_index] == value]) / len(data)) \* entropy([row for row in data if row[feature\_index] == value]) for value in values)**

**return total\_entropy - weighted\_entropy**

**def build\_tree(data, features):**

**labels = [row[-1] for row in data]**

**if len(set(labels)) == 1:**

**return labels[0]**

**if not features:**

**return max(set(labels), key=labels.count)**

**best\_feature = max(features, key=lambda f: information\_gain(data, f))**

**tree = {best\_feature: {}}**

**for value in set(row[best\_feature] for row in data):**

**subset = [row for row in data if row[best\_feature] == value] tree[best\_feature][value] = build\_tree(subset, [f for f in features if f != best\_feature])**

**return tree**

**data = [ [1, 1, 'Yes'], [1, 0, 'No'],[0, 1, 'Yes'],[0, 0, 'No']]**

**features = [0, 1]**

**tree = build\_tree(data, features)print(tree)**

**16.Feed forward neural Network**

**import numpy as np**

**def sigmoid(x):**

**return 1 / (1 + np.exp(-x))**

**def sigmoid\_derivative(x):**

**return x \* (1 - x)**

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

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

**np.random.seed(1)**

**input\_layer\_size = 2**

**hidden\_layer\_size = 4**

**output\_layer\_size = 1**

**weights\_input\_hidden = np.random.rand(input\_layer\_size, hidden\_layer\_size)**

**weights\_hidden\_output = np.random.rand(hidden\_layer\_size, output\_layer\_size)**

**bias\_hidden = np.random.rand(1, hidden\_layer\_size)**

**bias\_output = np.random.rand(1, output\_layer\_size)**

**learning\_rate = 0.1**

**epochs = 10000**

**for epoch in range(epochs):**

**hidden\_layer\_input = np.dot(X, weights\_input\_hidden) + bias\_hidden**

**hidden\_layer\_output = sigmoid(hidden\_layer\_input)**

**output\_layer\_input = np.dot(hidden\_layer\_output, weights\_hidden\_output) + bias\_output**

**output\_layer\_output = sigmoid(output\_layer\_input)**

**error = y - output\_layer\_output**

**d\_output = error \* sigmoid\_derivative(output\_layer\_output)**

**d\_hidden = d\_output.dot(weights\_hidden\_output.T) \* sigmoid\_derivative(hidden\_layer\_output)**

**weights\_hidden\_output += hidden\_layer\_output.T.dot(d\_output) \* learning\_rate**

**weights\_input\_hidden += X.T.dot(d\_hidden) \* learning\_rate**

**bias\_output += np.sum(d\_output, axis=0, keepdims=True) \* learning\_rate**

**bias\_hidden += np.sum(d\_hidden, axis=0, keepdims=True) \* learning\_rate**

**if epoch % 1000 == 0:**

**print(f"Epoch {epoch}, Error: {np.mean(np.abs(error))}")**

**print("\nFinal Output After Training:")**

**print(output\_layer\_outpu)**

**17.Sum the Integers from 1 to n.**

**sum(0, 0). % Base case: sum of 0 is 0**

**sum(N, Sum) :-**

**N > 0,**

**N1 is N - 1,**

**sum(N1, Sum1),**

**Sum is Sum1 + N.**

**18.DOB**

**% Facts: Name and Date of Birth (DOB)**

**dob(john, '12-May-1990').**

**dob(mary, '22-Aug-1985').**

**dob(sam, '15-Mar-2000').**

**dob(susan, '05-Nov-1995').**

**dob(kate, '30-Jun-2010').**

**dob(tom, '14-Sep-2012').**

**% Query Examples**

**% ?- dob(john, Date). % What is John's DOB?% ?- dob(Name, '22-Aug-1985'). % Who was born on 22-Aug-1985?**

**19.Teacher, Subject, Subject Code, and Student**

**% Facts: Teacher, Subject, Subject Code, and Student**

**teaches(mr\_smith, math, 101).**

**teaches(ms\_jones, physics, 102).**

**studies(john, 101).**

**studies(mary, 102).**

**% Rule: Find the teacher of a student**

**teacher\_of\_student(Student, Teacher) :-**

**studies(Student, Code),**

**teaches(Teacher, \_, Code)**

**20.PLANETS DB**

**% Facts: Planet(Name, Type, Distance\_from\_Sun\_in\_million\_km)**

**planet(mercury, terrestrial, 58).**

**planet(venus, terrestrial, 108).**

**planet(earth, terrestrial, 150).**

**planet(mars, terrestrial, 228).**

**planet(jupiter, gas\_giant, 778).**

**planet(saturn, gas\_giant, 1427).**

**planet(uranus, ice\_giant, 2871).**

**planet(neptune, ice\_giant, 4495).**

**% Rule: Find planets of a given type**

**planet\_type(Type, Name) :-**

**planet(Name, Type, \_)**

**21.owers of Hanoi.**

**% Base case: No move needed for 0 disks**

**hanoi(0, \_, \_, \_) :- !.**

**% Recursive case: Solve for N disks**

**hanoi(N, Source, Target, Auxiliary) :-**

**N > 0,**

**M is N - 1,**

**hanoi(M, Source, Auxiliary, Target), % Move top N-1 disks to Auxiliary**

**format('Move disk from ~w to ~w~n', [Source, Target]), % Move the Nth disk to Target**

**hanoi(M, Auxiliary, Target, Source). % Move the N-1 disks from Auxiliary to Target**

**22.BIRD PROGRAM**

**can\_fly(eagle).**

**can\_fly(sparrow).**

**can\_fly(hummingbird).\**

**cannot\_fly(ostrich).**

**cannot\_fly(penguin).**

**check\_fly(Bird) :-**

**can\_fly(Bird),**

**write(Bird), write(' can fly.'), nl.**

**check\_fly(Bird) :-**

**cannot\_fly(Bird),**

**write(Bird), write(' cannot fly.'), nl.**

**check\_fly(eagle).**

**check\_fly(ostrich).**

**23.FAMILY TREE**

**parent(john, mary).**

**parent(john, james).**

**parent(mary, linda).**

**parent(james, alex).**

**parent(linda, sophia).**

**male(john).**

**male(james).**

**male(alex).**

**female(mary).**

**female(linda).**

**female(sophia).**

**father(X, Y) :- male(X), parent(X, Y).**

**mother(X, Y) :- female(X), parent(X, Y).**

**sibling(X, Y) :- parent(Z, X), parent(Z, Y), X \= Y.**

**grandparent(X, Y) :- parent(X, Z), parent(Z, Y).**

**aunt\_or\_uncle(X, Y) :- sibling(X, Z), parent(Z, Y).**

**Query examples:**

**father(john, mary).**

**sibling(mary, james).**

**grandparent(john, alex).**

**aunt\_or\_uncle(mary, alex).**

**24.DIETING DOR DISEASE**

**diet(diabetes, low\_sugar).**

**diet(heart\_disease, low\_fat).**

**diet(obesity, low\_calorie).**

**diet(high\_blood\_pressure, low\_sodium).**

**suggest\_diet(Disease) :-**

**diet(Disease, Diet),**

**write('For ', Disease), write(', follow a ', Diet), write(' diet.').**

**diet(diabetes).**

**diet(heart\_disease).**

**25.monkey banana**

**monkey banana**

**at(monkey, ground).**

**at(box, floor).**

**at(banana, ceiling).**

**can\_reach\_banana :-**

**at(monkey, ground),**

**at(box, floor),**

**write('Monkey moves box, climbs, and reaches banana.').**

**can\_reach\_banana.**

**26.FRUIT AND COLOR**

**fruit(apple, red).**

**fruit(banana, yellow).**

**fruit(grape, purple).**

**fruit(orange, orange).**

**fruit\_color(Fruit, Color) :-**

**fruit(Fruit, Color).**

**fruit\_color(apple, Color).**

**27.BEST FIRST SEARCH**

**edge(a, b, 1).**

**edge(a, c, 4).**

**edge(b, d, 2).**

**edge(c, d, 5).**

**edge(d, e, 3).**

**heuristic(a, 6).**

**heuristic(b, 2).**

**heuristic(c, 5).**

**heuristic(d, 1).**

**heuristic(e, 0).**

**Best First Search rule**

**best\_first\_search(Start, Goal, Path) :-**

**best\_first([[Start]], Goal, Path).**

**best\_first\_search([[Goal|Path]|\_], Goal, [Goal|Path]).**

**best\_first\_search([Path|Paths], Goal, FinalPath) :-**

**extend(Path, NewPaths),**

**append(Paths, NewPaths, Paths1),**

**sort\_paths(Paths1, SortedPaths), best\_first\_search(SortedPaths, Goal, FinalPath).**

**extend([Node|Path], NewPaths) :-**

**findall([Next, Node|Path],**

**(edge(Node, Next, \_), \+ member(Next, [Node|Path])),**

**NewPaths).**

**sort\_paths(Paths, SortedPaths) :-**

**map\_list\_to\_pairs(path\_heuristic, Paths, Pairs),**

**keysort(Pairs, SortedPairs), pairs\_values(SortedPairs, SortedPaths).**

**path\_heuristic([Node|\_], H) :- heuristic(Node, H).**

**best\_first\_search(a, e, Path).**

**28.MEDICAL DIAGONISIS**

**disease(flu, [fever, cough, tiredness]).**

**disease(cold, [cough, sore\_throat, runny\_nose]).**

**disease(covid, [fever, cough, tiredness, loss\_of\_taste, difficulty\_breathing]).**

**Diagnose disease based on symptoms**

**diagnose(Disease) :-**

**write('Enter symptoms: '), nl,**

**read(Symptoms),**

**disease(Disease, DiseaseSymptoms), subset(DiseaseSymptoms, Symptoms),**

**write('You may have '), write(Disease), nl.**

**diagnose(Disease).**

**29.FORWARD CHAINIG**

**human(socrates).**

**mortal(X) :- human(X).**

**check\_mortal :-**

**human(X),**

**write(X), write(' is a mortal.'), nl,**

**assert(mortal(X)).**

**forward\_chain :-**

**check\_mortal,**

**fail.**

**forward\_chain. % to stop after all facts are inferred**

**forward\_chain.**

**30.BACKWARD CHAINING**

**human(socrates).**

**mortal(X) :- human(X).**

**% Backward Chaining Rule to check if someone is mortal**

**is\_mortal(X) :- mortal(X).**

**% Query example: is\_mortal(socrates).**

**32. pattern matching**

**% Pattern matching: Checks if Sublist is a part of the List**

**match\_pattern([], \_). % An empty list is always a pattern in any list**

**match\_pattern([H|T], [H|T2]) :- % If the head matches, continue with the rest**

**match\_pattern(T, T2).**

**match\_pattern([H|T], [\_|T2]) :- % If heads don't match, skip the first element**

**match\_pattern([H|T],T2).**

**33. number of vowels**

**count\_vowels([], 0).**

**count\_vowels([H|T], Count) :-**

**is\_vowel(H),**

**count\_vowels(T, TailCount),**

**Count is TailCount + 1.**

**count\_vowels([H|T], Count) :-**

**\+ is\_vowel(H),**

**count\_vowels(T, Count).**

**is\_vowel(A) :- member(A, ['a', 'e', 'i', 'o', 'u', 'A', 'E', 'I', 'O', 'U'])**