DAA DAY 8

1. Implement Floyd's Algorithm to find the shortest path between all pairs of cities. Display the distance matrix before and after applying the algorithm. Identify and print the shortest path

**import numpy as np**

**graph = [**

**[0, 3, np.inf, 5],**

**[2, 0, np.inf, 4],**

**[np.inf, 1, 0, np.inf],**

**[np.inf, np.inf, 2, 0]**

**]**

**print("Original Distance Matrix:")**

**for row in graph:**

**print(" ".join(f"{x:7}" if x != np.inf else "inf" for x in row))**

**n = len(graph)**

**dist = np.array(graph)**

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

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

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

**dist[i, j] = min(dist[i, j], dist[i, k] + dist[k, j])**

**print("\nDistance Matrix after applying Floyd-Warshall:")**

**for row in dist:**

**print(" ".join(f"{x:7}" if x != np.inf else "inf" for x in row))**

**start, end = 0, 2**

**path = [start]**

**while start != end:**

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

**if dist[start, end] == dist[start, k] + graph[k][end]:**

**path.append(k)**

**start = k**

**break**

**print(f"\nThe shortest path from city {path[0]} to city {end} is: {path}")**

2. Write a Program to implement Floyd's Algorithm to calculate the shortest paths between all pairs of routers. Simulate a change where the link between Router B and Router D fails. Update the distance matrix according

**import numpy as np**

**graph = [**

**[0, 3, np.inf, 5],**

**[3, 0, 1, 4],**

**[np.inf, 1, 0, 2],**

**[5, 4, 2, 0]**

**]**

**dist = np.array(graph)**

**n = len(dist)**

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

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

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

**dist[i, j] = min(dist[i, j], dist[i, k] + dist[k, j])**

**print("Distance Matrix after applying Floyd-Warshall:")**

**print(dist)**

**dist[1, 3] = np.inf**

**dist[3, 1] = np.inf**

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

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

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

**dist[i, j] = min(dist[i, j], dist[i, k] + dist[k, j])**

**print("\nUpdated Distance Matrix after link failure between Router B and Router D:")**

**print(dist)**

3. Implement Floyd's Algorithm to find the shortest path between all pairs of cities. Display the distance matrix before and after applying the algorithm. Identify and print the shortest path

**import numpy as np**

**graph = [**

**[0, 3, np.inf, 5],**

**[2, 0, np.inf, 4],**

**[np.inf, 1, 0, np.inf],**

**[np.inf, np.inf, 2, 0]**

**]**

**n = len(graph)**

**dist = np.array(graph)**

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

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

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

**dist[i, j] = min(dist[i, j], dist[i, k] + dist[k, j])**

**print("Distance Matrix after applying Floyd-Warshall:")**

**print(dist)**

**def reconstruct\_path(start, end):**

**path = []**

**if dist[start, end] == np.inf:**

**return None # No path**

**while start != end:**

**path.append(start)**

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

**if dist[start, end] == dist[start, k] + graph[k][end]:**

**start = k**

**break**

**path.append(end)**

**return path**

**start\_city = 0**

**end\_city = 2**

**path = reconstruct\_path(start\_city, end\_city)**

**print(f"\nThe shortest path from city {start\_city} to city {end\_city} is: {path}")**

4. Implement the Optimal Binary Search Tree algorithm for the keys A,B,C,D with frequencies 0.1,0.2,0.4,0.3 Write the code using any programming language to construct the OBST for the given keys and frequencies. Execute your code and display the resulting OBST and its cost. Print the cost and root matrix.

**import numpy as np**

**keys = ['A', 'B', 'C', 'D']**

**freq = [0.1, 0.2, 0.4, 0.3]**

**n = len(keys)**

**cost = np.zeros((n, n))**

**root = np.zeros((n, n), dtype=int)**

**for length in range(1, n + 1): # Length of the subproblem**

**for i in range(n - length + 1):**

**j = i + length - 1**

**if length == 1:**

**cost[i, j] = freq[i]**

**root[i, j] = i**

**else:**

**min\_cost = np.inf**

**for r in range(i, j + 1):**

**left\_cost = cost[i, r - 1] if r > i else 0**

**right\_cost = cost[r + 1, j] if r < j else 0**

**total\_cost = left\_cost + right\_cost + sum(freq[i:j + 1])**

**if total\_cost < min\_cost:**

**min\_cost = total\_cost**

**root[i, j] = r**

**cost[i, j] = min\_cost**

**print("Cost Matrix:")**

**print(cost)**

**print("\nRoot Matrix:")**

**print(root)**

**def print\_obst(root, keys, i, j):**

**if i > j:**

**return**

**r = root[i, j]**

**print(f"Key {keys[r]} is at root of subtree with keys {keys[i]} to {keys[j]}")**

**print\_obst(root, keys, i, r - 1)**

**print\_obst(root, keys, r + 1, j)**

**print("\nOptimal Binary Search Tree:")**

**print\_obst(root, keys, 0, n - 1)**

**print(f"\nTotal Cost of OBST: {cost[0, n - 1]}")**

5. Consider a set of keys 10,12,16,21 with frequencies 4,2,6,3 and the respective probabilities. Write a Program to construct an OBST in a programming language of your choice. Execute your code and display the resulting OBST, its cost and root matrix.

**import numpy as np**

**keys = [10, 12, 16, 21]**

**freq = [4, 2, 6, 3]**

**n = len(keys)**

**cost = np.zeros((n, n))**

**root = np.zeros((n, n), dtype=int)**

**for length in range(1, n + 1): # Length of the subproblem**

**for i in range(n - length + 1):**

**j = i + length - 1**

**if length == 1:**

**cost[i, j] = freq[i]**

**root[i, j] = i**

**else:**

**min\_cost = np.inf**

**for r in range(i, j + 1):**

**left\_cost = cost[i, r - 1] if r > i else 0**

**right\_cost = cost[r + 1, j] if r < j else 0**

**total\_cost = left\_cost + right\_cost + sum(freq[i:j + 1])**

**if total\_cost < min\_cost:**

**min\_cost = total\_cost**

**root[i, j] = r**

**cost[i, j] = min\_cost**

**print("Cost Matrix:")**

**print(cost)**

**print("\nRoot Matrix:")**

**print(root)**

**def print\_obst(root, keys, i, j):**

**if i > j:**

**return**

**r = root[i, j]**

**print(f"Key {keys[r]} is at root of subtree with keys {keys[i]} to {keys[j]}")**

**print\_obst(root, keys, i, r - 1)**

**print\_obst(root, keys, r + 1, j)**

**print("\nOptimal Binary Search Tree:")**

**print\_obst(root, keys, 0, n - 1)**

**print(f"\nTotal Cost of OBST: {cost[0, n - 1]}")**

6. A game on an undirected graph is played by two players, Mouse and Cat, who alternate turns. The graph is given as follows: graph[a] is a list of all nodes b such that ab is an edge of the graph. The mouse starts at node 1 and goes first, the cat starts at node 2 and goes second, and there is a hole at node 0. During each player's turn, they must travel along one edge of the graph that meets where they are. For example, if the Mouse is at node 1, it must travel to any node in graph[1]. Additionally, it is not allowed for the Cat to travel to the Hole (node 0).Then, the game can end in three ways: If ever the Cat occupies the same node as the Mouse, the Cat wins. If ever the Mouse reaches the Hole, the Mouse wins. If ever a position is repeated (i.e., the players are in the same position as a previous turn, and it is the same player's turn to move), the game is a draw. Given a graph, and assuming both players play optimally, return 1 if the mouse wins the game, 2 if the cat wins the game, or 0 if the game is a draw.

**from collections import deque**

**graph = {**

**0: [1, 2], # Hole**

**1: [0, 2, 3], # Mouse's start**

**2: [0, 1, 3], # Cat's start**

**3: [1, 2] # Other nodes**

**}**

**queue = deque([(1, 2, 1)])**

**visited = set()**

**while queue:**

**mouse, cat, turn = queue.popleft()**

**if mouse == 0:**

**print(1)**

**break**

**if mouse == cat:**

**print(2)**

**break**

**state = (mouse, cat, turn)**

**if state in visited:**

**print(0)**

**break**

**visited.add(state)**

**if turn == 1: # Mouse's turn**

**for next\_mouse in graph[mouse]:**

**queue.append((next\_mouse, cat, 2)) # Mouse moves, Cat's turn next**

**else:**

**for next\_cat in graph[cat]:**

**if next\_cat != 0: # Cat cannot move to the Hole**

**queue.append((mouse, next\_cat, 1)) # Cat moves, Mouse's turn next**

7. You are given an undirected weighted graph of n nodes (0-indexed), represented by an edge list where edges[i] = [a, b] is an undirected edge connecting the nodes a and b with a probability of success of traversing that edge succProb[i]. Given two nodes start and end, find the path with the maximum probability of success to go from start to end and return its success probability. If there is no path from start to end, return 0. Your answer will be accepted if it differs from the correct answer by at most 1e-5.

**import heapq**

**from collections import defaultdict**

**def maxProbability(n, edges, succProb, start, end):**

**graph = defaultdict(list)**

**for (a, b), prob in zip(edges, succProb):**

**graph[a].append((b, prob))**

**graph[b].append((a, prob))**

**max\_heap = [(-1.0, start)]**

**max\_prob = [0] \* n**

**max\_prob[start] = 1.0**

**while max\_heap:**

**prob, u = heapq.heappop(max\_heap)**

**prob = -prob**

**if u == end:**

**return prob**

**for v, p in graph[u]:**

**new\_prob = prob \* p**

**if new\_prob > max\_prob[v]:**

**max\_prob[v] = new\_prob**

**heapq.heappush(max\_heap, (-new\_prob, v))**

**return 0**

**n = 3**

**edges = [[0, 1], [1, 2]]**

**succProb = [0.5, 0.5]**

**start = 0**

**end = 2**

**print(maxProbability(n, edges, succProb, start, end))**

8. There is a robot on an m x n grid. The robot is initially located at the top-left corner (i.e., grid[0][0]). The robot tries to move to the bottom-right corner (i.e., grid[m - 1][n - 1]). The robot can only move either down or right at any point in time. Given the two integers m and n, return the number of possible unique paths that the robot can take to reach the bottom-right corner. The test cases are generated so that the answer will be less than or equal to 2 \* 10 9.

**import math**

**m = 3 # Number of rows**

**n = 4 # Number of columns**

**total\_paths = math.comb(m + n - 2, m - 1)**

**print(total\_paths)**

9. Given an array of integers nums, return the number of good pairs. A pair (i, j) is called good if nums[i] == nums[j] and i < j.

**from collections import Counter**

**nums = [1, 2, 3, 1, 1, 3]**

**count = Counter(nums)**

**good\_pairs = sum(v \* (v - 1) // 2 for v in count.values())**

**print(good\_pairs)**

10. There are n cities numbered from 0 to n-1. Given the array edges where edges[i] = [fromi, toi, weighti] represents a bidirectional and weighted edge between cities fromi and toi, and given the integer distanceThreshold. Return the city with the smallest number of cities that are reachable through some path and whose distance is at most distanceThreshold, If there are multiple such cities, return the city with the greatest number. Notice that the distance of a path connecting cities i and j is equal to the sum of the edges' weights along that path.

**import heapq**

**import sys**

**n = 4 # Number of cities**

**edges = [[0, 1, 1], [1, 2, 2], [2, 3, 1], [0, 2, 2], [1, 3, 4]]**

**distanceThreshold = 2**

**graph = [[] for \_ in range(n)]**

**for u, v, w in edges:**

**graph[u].append((v, w))**

**graph[v].append((u, w))**

**def dijkstra(start):**

**distances = [float('inf')] \* n**

**distances[start] = 0**

**min\_heap = [(0, start)] # (distance, node)**

**while min\_heap:**

**current\_dist, u = heapq.heappop(min\_heap)**

**if current\_dist > distances[u]:**

**continue**

**for v, weight in graph[u]:**

**distance = current\_dist + weight**

**if distance < distances[v]:**

**distances[v] = distance**

**heapq.heappush(min\_heap, (distance, v))**

**return distances**

**best\_city = -1**

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

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

**distances = dijkstra(i)**

**reachable\_count = sum(1 for d in distances if d <= distanceThreshold)**

**if reachable\_count < min\_reachable or (reachable\_count == min\_reachable and i > best\_city):**

**min\_reachable = reachable\_count**

**best\_city = i**

**print(best\_city)**