2492. Minimum Score of a Path Between Two Cities

Medium | Depth-First Search | Breadth-First Search | Union Find | Graph | Leetcode Link

Problem Description

In this problem, we are working with a graph that represents a system of cities and bidirectional roads connecting these cities. The graph is composed of `n` cities, which are numbered from 1 to `n`. The roads are represented by a 2D array `roads`, where each element in the array is a trio `[a, b, distance]`. This trio indicates that there is a road connecting city `a` and city `b` with a distance of `distance`. It's important to note that the graph of cities may not be fully connected, meaning some cities may not have a direct path to others.

The "score" of a path in this graph is a little unique; rather than summing up the total distance or considering average distance, the score is defined as the minimum distance of any road in that particular path. Put another way, out of all the roads used in a path, the score is the distance of the shortest one among them.

Our goal is to determine the minimum possible score of a path between city 1 and city `n`. A few rules apply to paths:

- A path is simply a sequence of roads connecting two cities; it may pass through intermediate cities.
- It is permissible for a path to include the same road more than once, if necessary.
- You can go through cities 1 and `n` multiple times to find the path with the optimal score.
- There is at least one path between city 1 and city `n` in all test cases.

Ultimately, we want to compute the highest "weakest" road in the best path from city 1 to city `n`.

Intuition

To solve this problem, we can utilize Depth-First Search (DFS) to explore all possible paths from city 1 to city `n`. As we perform DFS, we track the minimum distance encountered along the current path. Before starting DFS, we initialize the answer `ans` with a value representing infinity (`inf`), because we will be minimizing this score as we go along.

To apply DFS, we first create a `defaultdict` of lists called `g`, which will be our graph where the keys are city numbers and the values are lists of tuples representing connected cities alongside the respective distances. We also create a list `vis` of boolean values, initialized to `False`, that keeps track of the cities that have been visited to prevent revisiting them in the current path exploration.

Our DFS process named `dfs()` takes a city `i` as an argument and iterates through all of its connections. For each adjacent city `j` with distance `d` to city `i`, we minimize `ans` with `d` to find the smallest distance seen so far along this path segment. We also mark `j` as visited and perform DFS from `j`. The process recursively continues until we have exhausted all possible paths.

After initializing our data structures, we simply call `dfs(1)` to start from the first city and allow the DFS to explore paths until city `n`. Once DFS is done, the `ans` variable will hold the minimum score among all paths from city 1 to city `n`.

The solution is neat because it elegantly uses DFS to explore the space of paths, and it cleverly updates the minimum score on-the-fly. It relies on the fact that, since the score is the minimum distance along a path, we just need to track the smallest road distance encountered at any point in our DFS exploration.

Solution Approach

The implementation of the solution utilizes a Depth-First Search (DFS) algorithm which is a common strategy for exploring all the vertices of a graph. Let us walk through the important points of the implementation:

1. **Graph Representation**: A `defaultdict` of lists is used to represent the graph (`g`). Each key in the defaultdict represents a city, and the value is a list of tuples. Each tuple consists of a destination city and the distance to that city. This allows us to easily iterate over all neighbors of a city.

2. **Visited Set**: A list called `vis` is created with a size of `n + 1`, where `n` is the number of cities. The list is initialized with `False` to indicate that no city has been visited at the start. This ensures that we do not enter into an infinite loop by revisiting cities during our DFS traversal.

3. **Distance Initialization**: A variable `ans` is initialized with `inf` (infinity). This variable is used to keep track of the minimum distance encountered on any path between city 1 and city `n`.

4. **DFS Algorithm**: A recursive function `dfs(i)` is defined, which is responsible for traversing the graph:
   - Inside the DFS function, we iterate over each neighbor `j` and its associated distance `d` of the current city `i`.
   - We update `ans` with the minimum of its current value and `d` each time we encounter a smaller distance.
   - If the neighbor `j` has not yet been visited (`vis[j]` is `False`), we mark it as visited by setting `vis[j]` to `True` and then recursively call `dfs(j)` to continue exploring the graph from city `j`.
5. **Starting the DFS Traversal**: We call `dfs(1)` to begin our DFS traversal from city 1. This will explore different paths to reach city `n`.

6. **Result**: Given that there is at least one path from city 1 to city `n`, by the end of DFS traversal, `ans` will contain the minimum possible score (which is the minimum distance of the weakest road) in some path between city 1 and city `n`. This `ans` is then returned as a result.

The clever use of DFS allows for an efficient search through all the paths while keeping track of the minimum distance encountered, therefore the weakest link in terms of road distance, which effectively determines the score of the path.

Example Walkthrough

Let's illustrate the solution approach with a small example where `n = 4` representing four cities and `roads` as `[[1, 2, 4], [2, 3, 3], [3, 4, 2], [1, 3, 6]]`. This setup implies the following connections:

- City 1 and city 2 are connected by a road with a distance of 4.
- City 2 and city 3 are connected by a road with a distance of 3.
- City 3 and city 4 are connected by a road with a distance of 2.
- City 1 and city 3 are directly connected as well with a distance of 6.

Following our solution approach:

1. **Graph Representation**: We first convert the `roads` array to a graph represented as a `defaultdict` of lists, `g`, so that we have:

   ```
   g = {
       1: [(2, 4), (3, 6)],
       2: [(1, 4), (3, 3)],
       3: [(2, 3), (4, 2), (1, 6)],
       4: [(3, 2)]
   }
   ```

   Each list contains tuples that represent connections to neighboring cities and the distance to them.

2. **Visited Set**: We create `vis = [False, False, False, False, False]`.

3. **Distance Initialization**: Set `ans = inf`.

4. **DFS Algorithm**: For the `dfs` function, starting with `dfs(1)`:
   - Look at neighbors of city 1: (2, 4) and (3, 6).
   - For neighbor (2, 4), since city 2 is not visited, mark it visited and compare `ans` with 4. `ans` becomes min(`inf`, 4) = 4.
   - Continue DFS by calling `dfs(2)`.
     - In `dfs(2)`, iterate over neighbors: (1, 4) and (3, 3).
     - Skip city 1 since it's visited; for city 3 with distance 3, since city 3 is not visited, mark it visited and update `ans` to min(4, 3) = 3.
     - Continue DFS by calling `dfs(3)`.
       - In `dfs(3)`, check neighbors (2, 3), (4, 2), and (1, 6).
       - City 2 is visited, skip. Update `ans` with min(3, 2) when visiting city 4, so `ans` becomes 2.
       - City 4 has no unvisited neighbors left, so DFS goes back up.

   This process continues until all possible paths between city 1 and city 4 have been explored through recursion.

5. **Result**: After the DFS is finished, `ans` holds the minimum distance encountered on any path, which in this case is 2. That means the highest "weakest" road on the best path from city 1 to city 4 has a distance of 2. This is the score of our path.

Hence, the minimum possible score for a path between city 1 and city 4 is 2, given this set of roads and cities.

Python Solution

```python
from collections import defaultdict
from math import inf  # Represents positive infinity

class Solution:
    def minScore(self, num_nodes: int, roads: List[List[int]]) -> int:
        # Depth-First Search, where `current_node` is the current node being visited.
        def dfs(current_node):
            nonlocal minimum_road_score
            for neighbor, road_score in graph[current_node]:
                minimum_road_score = min(minimum_road_score, road_score)
                if not visited[neighbor]:
                    visited[neighbor] = True
                    dfs(neighbor)

        # Build graph representation from roads input.
        graph = defaultdict(list)
        for src, dest, score in roads:
            graph[src].append((dest, score))
            graph[dest].append((src, score))

        # Initialize visited nodes list.
        visited = [False] * (num_nodes + 1)

        # Initialize the answer with infinity.
        minimum_road_score = inf

        # Start DFS from node 1 (assuming nodes are 1-indexed).
        visited[1] = True
        dfs(1)

        # Return the minimum road score found during DFS.
        return minimum_road_score
```

Java Solution

```java
class Solution {
    private List<int[]>[] graph; // Graph represented as adjacency list
    private boolean[] visited; // Visited array to keep track of visited nodes
    private int minimumScore = Integer.MAX_VALUE; // Initialize minimum score to maximum possible value

    // Function to find minimum score in the graph
    public int minScore(int n, int[][] roads) {
        graph = new List[n]; // Create graph with 'n' nodes
        visited = new boolean[n]; // Initialize 'visited' array for 'n' nodes
        // Fill the graph with empty lists for each node
        Arrays.setAll(graph, k -> new ArrayList<>());
        // Build the graph from the given roads information
        for (int[] road : roads) {
            int from = road[0] - 1; // Convert to 0-indexed
            int to = road[1] - 1; // Convert to 0-indexed
            int weight = road[2];
            // Add edge to the undirected graph
            graph[from].add(new int[] {to, weight});
            graph[to].add(new int[] {from, weight});
        }
        // Start depth-first search traversal from node 0
        dfs(0);
        return minimumScore; // Return the minimum score found during DFS
    }

    // Helper function to perform depth-first search
    private void dfs(int currentNode) {
        // Go through all the connected nodes
        for (int[] edge : graph[currentNode]) {
            int nextNode = edge[0]; // Destination node
            int weight = edge[1]; // Weight of the edge
            // Update the minimum score encountered so far
            minimumScore = Math.min(minimumScore, weight);
            // If the next node is not visited, continue DFS traversal
            if (!visited[nextNode]) {
                visited[nextNode] = true; // Mark this node as visited
                dfs(nextNode); // Recursively call dfs for the next node
            }
        }
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <climits>
#include <cstring>
#include <functional>
using namespace std;

class Solution {
public:
    int minScore(int n, vector<vector<int>>& roads) {
        // Create an adjacency list to represent the graph
        vector<vector<pair<int, int>>> graph(n);
        // Visited array to keep track of visited nodes
        bool visited[n];
        // Initialize the visited array to false for all nodes
        memset(visited, 0, sizeof visited);

        // Populate the adjacency list with road data
        for (auto& road : roads) {
            int from = road[0] - 1;  // Convert to 0-based index
            int to = road[1] - 1;    // Convert to 0-based index
            int distance = road[2];
            graph[from].emplace_back(to, distance);
            graph[to].emplace_back(from, distance);
        }

        // Initialize answer to maximum possible value
        int answer = INT_MAX;

        // Define the depth-first search (DFS) lambda function
        function<void(int)> dfs = [&](int node) {
            // Go through all edges connected to the current node
            for (auto [adj_node, adj_distance] : graph[node]) {
                // Update answer with the minimum distance so far
                answer = min(answer, adj_distance);
                // If adjacent node has not been visited
                if (!visited[adj_node]) {
                    // Mark as visited
                    visited[adj_node] = true;
                    // Continue DFS from the adjacent node
                    dfs(adj_node);
                }
            }
        };

        // Start DFS from node 0 (converted to 0-based index previously)
        visited[0] = true; // Mark node 0 as visited
        dfs(0);

        // Return the minimum score found by DFS
        return answer;
    }
};
```

Typescript Solution

```typescript
function minScore(nodeCount: number, edges: number[][]): number {
    // visited array to keep track of visited nodes
    const visited = new Array(nodeCount + 1).fill(false);
    // graph represented by an adjacency list
    const graph = Array.from({ length: nodeCount + 1 }, () => []);

    // Construct the graph from the given edges
    for (const [nodeFrom, nodeTo, value] of edges) {
        graph[nodeFrom].push([nodeTo, value]);
        graph[nodeTo].push([nodeFrom, value]);
    }

    // Initialize answer with Infinity to find the minimum value later
    let minimumScore = Infinity;

    // Depth-first search to traverse the graph and find the minimum edge value
    const depthFirstSearch = (currentNode: number) => {
        // If the current node is already visited, skip it
        if (visited[currentNode]) {
            return;
        }
        // Mark the current node as visited
        visited[currentNode] = true;
        // Iterate over all neighbor nodes
        for (const [nextNode, edgeValue] of graph[currentNode]) {
            // Update the minimum score with the minimum edge value found so far
            minimumScore = Math.min(minimumScore, edgeValue);
            // Continue the search with the next node
            depthFirstSearch(nextNode);
        }
    };

    // Start the DFS from the first node (assuming nodes are labeled starting from 1)
    depthFirstSearch(1);

    // Return the minimum edge score found during the DFS
    return minimumScore === Infinity ? -1 : minimumScore; // Return -1 if no edges were found
}

// Example usage:
// const score = minScore(4, [[1, 2, 3], [2, 3, 1], [1, 3, 4]]);
// console.log(score); // Output will be 1 which is the minimum edge value
```

Time and Space Complexity

Time Complexity

The given Python code implements a depth-first search (DFS) on a graph represented as a adjacency list. The `dfs` function is called recursively to traverse the graph.

The time complexity of the algorithm depends on the number of nodes `n` and the number of edges in the `roads` list. The DFS will visit each node exactly once, due to the `vis[j] = True` guard before each recursive call.

The complexity can have two major components: the time it takes to set up the adjacency list and the time it takes to do the DFS traversal.

- **Adjacency List Creation:** The adjacency list is created by iterating through each road in the `roads` list. This operation takes `O(E)` time where `E` is the number of edges as each edge is visited once.

- **DFS Traversal:** DFS traversal typically has a time complexity of `O(V + E)` for visiting each node and edge at most once.

Therefore the total time complexity is `O(V + E)` where `V` is the number of vertices and `E` is the number of edges.

Space Complexity

The space complexity includes the storage for the graph (adjacency list), the visited array, and the stack space used by the recursive calls of DFS.

- **Graph Storage:** Each node stores a list of its edges, so in total this is proportional to `O(E)`, where `E` is the number of edges.

- **Visited Array:** This is an array of length `n + 1`, thus taking up `O(V)` space where `V` is the number of vertices.

- **DFS Stack space:** In the worst case, the recursive DFS could go as deep as the number of nodes in the graph (imagine a linked-list shaped graph), so the stack space in the worst case can be `O(V)`.

Consequently, the total space complexity is `O(E + V)` with consideration for the adjacency list, visited array, and recursive stack space.