1245. Tree Diameter

Medium Tree Depth-First Search Breadth-First Search Graph Topological Sort Leetcode Link

Problem Description

The task is to find the diameter of a tree. The diameter is defined as the number of edges in the longest path between any two nodes in the tree. Given that the tree is undirected and consists of `n` nodes labeled from `0` to `n - 1`, and an array of edges indicating the connections between the nodes, the objective is to calculate and return the tree's diameter.

Intuition

Finding the diameter of a tree involves discovering the longest path between two nodes. A common technique to solve this problem is to use a depth-first search (DFS) twice. The intuition behind this approach is based on the property that the longest path in the tree will have as its endpoints two of the tree's leaves.

The first DFS begins at an arbitrary node (usually the first node provided) and traverses the tree to find the leaf node that lies at the greatest distance from the starting node. This leaf node will be one end of the longest path. The second DFS then begins at this leaf node and searches for the farthest leaf node from it, thus uncovering the other end of the longest path.

By keeping track of the number of edges traversed during the second DFS (which is essentially the length or the height of the tree from the perspective of the starting leaf node), we can directly calculate the diameter of the tree.

Solution Approach

The provided Python code implements a solution to the problem using a depth-first search (DFS) algorithm. Here's how it works:

1. A `dfs` function is defined, which takes a node `u` and a counter `t` that tracks the total number of edges traversed from the starting node. The `dfs` function recursively visits nodes, incrementing the counter as it goes deeper into the tree.
2. Inside this DFS function, a base condition checks if the current node has already been visited to prevent loops. If it hasn't been visited, it is marked as visited.
3. The recursive step involves iterating over all the adjacent nodes (`v`) of the current node (`u`). For each adjacent node, the `dfs` function is called again with `v` as the current node and the counter `t` incremented by 1.
4. While returning from each call of `dfs`, the code compares the current value of `t` with a variable `ans`, which keeps track of the maximum distance found so far. If `t` is greater, it means a longer path has been found, and thus `ans` is updated with the current value of `t`, and `next` is updated with the current node `u`.

The main function `treeDiameter` utilizes the DFS as follows:

- A defaultdict `d` is created to store the tree in an adjacency list representation. This makes it easy to access all the nodes connected to a given node in the tree. The `edges` are used to populate this adjacency list.
- An array `vis` is used to keep track of which nodes have been visited to prevent repeating the same nodes during the DFS.
- The first DFS is initiated from the first node in the `edges` list. After this DFS, we have a candidate leaf node for the longest path in the variable `next`.
- The `vis` array is reset to prepare for the second DFS call.
- The second DFS is initiated from the `next` node found from the first DFS call.
- At the end of the second DFS, the variable `ans` holds the diameter of the tree, which is the number of edges in the longest path. This is returned as the function's result.

In summary, the algorithm finds the longest path (or diameter) of the tree by first locating one end of the longest path using a DFS and then executing another DFS from that endpoint to find the actual longest path.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Suppose we have a tree with `n=6` nodes and the following edges:

```
0 - 1
1 - 2
1 - 3
2 - 4
2 - 5
```

The tree structure would look like this:

```
  0
   \
    1
   / \
  2   3
 / \
4   5
```

Given this tree, we want to find its diameter. Following the solution approach:

1. We first use DFS starting from an arbitrary node, say node `0`.
2. We traverse the tree from node `0`, and we may reach a leaf node, such as node `4`. Let's assume the DFS process went along the path `0->1->2->4`. During this process, each time DFS calls itself, it increments the counter `t`.
3. Since node `4` is a leaf, we backtrack, but in doing so, we keep track of the node that gave us the maximum depth. Let's assume this maximum depth was first encountered at node `4`.
4. With node `4` as our furthest leaf from node `0`, we now reset the `vis` array for the second DFS.
5. The second DFS starts at node `4` and proceeds to find the furthest node from it. Node `4` will branch only to `2`, so we go to `2`, then to `1`, and we have a choice to go to node `0` or node `3`.
6. If we go to `3`, which is another leaf, we've found the longest path in the tree: `4->2->1->3`.

During the second DFS, we keep a variable `ans` that tracks the max length. Since the path `4->2->1->3` is the longest path in the tree, `ans` would be `3` in the end, which means there are three edges in the path. Thus, the diameter of this tree is `3`.

That concludes the example walkthrough using the solution approach. We've illustrated how the algorithm recursively searches for the furthest possible nodes step by step and eventually finds the longest path, which defines the diameter of the tree.

Python Solution

```python
from collections import defaultdict

class Solution:
    # Function to find the diameter of a tree given its edges
    def treeDiameter(self, edges: List[List[int]]) -> int:
        # Helper function to perform a depth-first search (DFS)
        def dfs(node, distance):
            nonlocal max_diameter, visited, graph, next_node
            if visited[node]:  # If this node has already been visited, skip
                return
            visited[node] = True
            for neighbor in graph[node]:
                dfs(neighbor, distance + 1)  # DFS on connected nodes

            # Update the diameter and the next node if we found a longer path
            if max_diameter < distance:
                max_diameter = distance
                next_node = node

        # Convert the edge-list to an adjacency list representation for the graph
        graph = defaultdict(set)
        # Initialize a visited list to keep track of visited nodes during DFS
        visited = [False] * (len(edges) + 1)
        # Establish the adjacency list from the edges
        for u, v in edges:
            graph[u].add(v)
            graph[v].add(u)

        max_diameter = 0  # Initialize the diameter of the tree
        next_node = 0  # This variable will hold one end of the maximum diameter

        # Perform the first DFS to find one end of the maximum diameter path
        dfs(edges[0][0], 0)

        # Reset the visited list for the next DFS
        visited = [False] * (len(edges) + 1)

        # Perform the second DFS from the farthest node found in the first DFS
        dfs(next_node, 0)

        # The maximum distance found during the second DFS is the tree diameter
        return max_diameter
```

Java Solution

```java
class Solution {
    private Map<Integer, Set<Integer>> graph; // Represents the adjacency list for the graph
    private boolean[] visited;                 // Array to keep track of visited nodes
    private int farthestNode;                  // Tracks the farthest node found during DFS
    private int diameter;                      // Stores the current diameter of the tree

    public int treeDiameter(int[][] edges) {
        int n = edges.length;       // Number of nodes will be #edges + 1 in a tree
        diameter = 0;               // Initialize diameter as zero
        graph = new HashMap<>();    // Initialize the graph

        // Building an undirected graph using the provided edges
        for (int[] edge : edges) {
            graph.computeIfAbsent(edge[0], k -> new HashSet<>()).add(edge[1]);
            graph.computeIfAbsent(edge[1], k -> new HashSet<>()).add(edge[0]);
        }

        // Initialize the visited array for the first DFS traversal
        visited = new boolean[n + 1]; // Index 0 is ignored since node numbering starts from 1
        farthestNode = edges[0][0];   // Start from the first node
        dfs(farthestNode, 0);         // Perform DFS to find the farthest node

        // Reset the visited array for the second DFS traversal
        visited = new boolean[n + 1];
        dfs(farthestNode, 0);         // Perform DFS from the farthest node to find the diameter
        return diameter;              // Return the diameter
    }

    private void dfs(int node, int distance) {
        if (visited[node]) {
            return;
        }
        visited[node] = true;
        // Update the diameter and farthest node if the current distance is greater
        if (diameter < distance) {
            diameter = distance;
            farthestNode = node;
        }
        // Visit all the connected nodes of the current node using DFS
        for (int adjacent : graph.get(node)) {
            dfs(adjacent, distance + 1);
        }
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <unordered_map>
#include <unordered_set>

class Solution {
public:
    // Graph representation using an adjacency list
    unordered_map<int, unordered_set<int>> graph;
    // Visited vector to keep track of visited nodes during DFS
    vector<bool> visited;
    // Variable to store the length of the diameter of the tree
    int diameter;
    // Variable to store the node that will be the next candidate for DFS
    int furthestNode;

    // Function to calculate the diameter of the tree
    int treeDiameter(vector<vector<int>>& edges) {
        // Construct the graph
        for (auto& edge : edges) {
            graph[edge[0]].insert(edge[1]);
            graph[edge[1]].insert(edge[0]);
        }
        int numNodes = edges.size();
        diameter = 0;
        // Initialize visited vector of size numNodes+1 (since it's a tree, it has numNodes+1 vertices)
        visited.resize(numNodes + 1, false);
        // Initialize the furthestNode with any node (e.g., the first node of the first edge)
        furthestNode = edges[0][0];

        // Perform the first DFS to find the furthest node from the starting node
        dfs(furthestNode, 0);
        // Reset the visited vector for the next DFS
        visited.assign(visited.size(), false);
        // Perform the DFS starting from the furthest node to determine the diameter
        dfs(furthestNode, 0);

        // Diameter is the maximum number of edges between any two nodes in the tree
        return diameter;
    }

    // Depth-First Search (DFS) function to traverse the tree
    void dfs(int currentNode, int currentLength) {
        // If the current node has been visited, return
        if (visited[currentNode]) return;
        // Mark the current node as visited
        visited[currentNode] = true;
        // If the current length is greater than the diameter, update the diameter and furthestNode
        if (diameter < currentLength) {
            diameter = currentLength;
            furthestNode = currentNode;
        }
        // Visit all adjacent nodes
        for (int adjacentNode : graph[currentNode]) {
            dfs(adjacentNode, currentLength + 1);
        }
    }
};
```

Typescript Solution

```typescript
// Importing necessary functionalities from 'collections' module in TypeScript
import { Set, Map } from "typescript-collections";

// Global graph representation using an adjacency list
const graph: Map<number, Set<number>> = new Map();
// Global visited Map to keep track of visited nodes during DFS
const visited: Map<number, boolean> = new Map();
// Variable to store the length of the diameter of the tree
let diameter: number = 0;
// Variable to store the node that will be the next candidate for DFS
let furthestNode: number;

// Function to calculate the diameter of the tree
function treeDiameter(edges: number[][]): number {
    // Construct the graph
    edges.forEach(edge => {
        if (!graph.containsKey(edge[0])) {
            graph.setValue(edge[0], new Set());
        }
        if (!graph.containsKey(edge[1])) {
            graph.setValue(edge[1], new Set());
        }
        graph.getValue(edge[0]).add(edge[1]);
        graph.getValue(edge[1]).add(edge[0]);
    });

    const numNodes: number = edges.length;
    diameter = 0;
    // Initialize visited Map for all nodes as false
    for (let i = 0; i < numNodes + 1; i++) {
        visited.setValue(i, false);
    }
    // Initialize the furthestNode with any node (e.g., the first node of the first edge)
    furthestNode = edges[0][0];

    // Perform the first DFS to find the furthest node from the starting node
    dfs(furthestNode, 0);
    // Reset the visited Map for the next DFS
    visited.forEach((value, key) => {
        visited.setValue(key, false);
    });
    // Perform the DFS starting from the furthest node to determine the diameter
    dfs(furthestNode, 0);

    // Diameter is the maximum number of edges between any two nodes in the tree
    return diameter;
}

// Depth-First Search (DFS) function to traverse the tree
function dfs(currentNode: number, currentLength: number): void {
    // If the current node has been visited, return
    if (visited.getValue(currentNode)) {
        return;
    }
    // Mark the current node as visited
    visited.setValue(currentNode, true);
    // If the current length is greater than the diameter, update the diameter and furthestNode
    if (currentLength > diameter) {
        diameter = currentLength;
        furthestNode = currentNode;
    }
    // Visit all adjacent nodes
    graph.getValue(currentNode).forEach(adjacentNode => {
        dfs(adjacentNode, currentLength + 1);
    });
}
```

Time and Space Complexity

Time Complexity

The given Python code snippet defines a `Solution` class with a method `treeDiameter` to find the diameter of an undirected tree represented by a list of edges. The algorithm uses a depth-first search (`dfs`) starting from an arbitrary node to find the farthest node from it, and then performs a second depth-first search from this newly discovered node to find the diameter of the tree.

Let `n` be the number of nodes in the tree, which is one more than the number of edges (`len(edges)`), as it is a connected, undirected tree with `n-1` edges.

The time complexity for constructing the adjacency list `d` is `O(n)`, since each edge `(u, v)` is processed exactly twice, once for each direction.

Each `dfs` call traverses all edges in the tree exactly once. Since there are `n-1` edges and each edge contributes to two entries in the adjacency list (one for each end of the edge), the total number of adjacency entries to traverse is `2(n-1)`, which is `O(n)`. There are two `dfs` calls, meaning that the total time for traversing the tree is `2*O(n)`, which simplifies to `O(n)`.

Therefore, the overall time complexity of the code is `O(n)`.

Space Complexity

The space complexity consists of the storage for the adjacency list `d`, the visited flags array `vis`, and the auxiliary space used by the `dfs` recursion call stack.

The adjacency list `d` has `2(n-1)` entries as explained earlier; thus, its space complexity is `O(n)`.

The `vis` array is of length `n`, so it also occupies `O(n)` space.

The depth of the recursive `dfs` call stack will be at most `n` in the worst case (a path graph), contributing an additional `O(n)` to the space complexity.

Combining these factors, the overall space complexity of the algorithm is `O(n)`.