2368. Reachable Nodes With Restrictions

Medium Tree Depth-First Search Breadth-First Search Graph Array Hash Table [Leetcode Link]

Problem Description

In this problem, we are given an undirected tree with `n` nodes, where each node is labeled from `0` to `n - 1`. A 2D integer array `edges` is provided that contains `n - 1` pairs of integers, with each pair `[ai, bi]` indicating an edge between the nodes `ai` and `bi`. Alongside this, we have an integer array `restricted` that lists restricted nodes which should not be visited.

Our goal is to find the maximum number of nodes that can be reached from node `0` without visiting any of the restricted nodes. It's important to note that node `0` is not a restricted node.

The result should be the count of accessible nodes, including node `0`, keeping in mind the restricted nodes' constraint.

Intuition

To solve this problem, we use depth-first search (DFS), which is a standard graph traversal algorithm, to explore the graph starting from node `0`. The graph is represented using an adjacency list, which is a common way to represent graphs in memory.

The key steps involve:

1. Constructing an adjacency list from the `edges` array, allowing us to access all connected nodes from a given node easily.
2. Creating a visited array, `vis`, to keep track of both visited and restricted nodes, marking restricted nodes as visited preemptively so that we don't traverse them during DFS.
3. Implementing a recursive DFS function that will traverse the graph. For each node `u` that is not visited and not restricted, the function will:
   - Increase the count of reachable nodes, `ans`, by 1.
   - Mark the node as visited.
   - Recursively call DFS for all adjacent nodes of `u`.

When the DFS function completes, `ans` will contain the total count of reachable nodes from node `0`, ensuring that no restricted nodes are counted. This approach guarantees that we explore all possible paths in a depth-first manner while skipping over the restricted nodes and only counting the valid ones.

Solution Approach

The solution utilizes the Depth-First Search (DFS) algorithm to traverse the tree from node `0` and count the maximum number of reachable nodes without visiting any restricted nodes. Here's how the implementation works:

1. **Data Structure**: We use a `defaultdict` to create an adjacency list `g` that represents the graph. Each key in this dictionary corresponds to a node, and the value is a list of nodes that are connected to it by an edge. This allows us to efficiently access all neighbor nodes of any given node.

2. **Pre-Marking Restricted Nodes**: We create a list `vis` of boolean values to keep track of whether a node has been visited or is restricted. Its length is `n` to cover all nodes. Restricted nodes are preemptively marked as visited (i.e., `True`) since we don't want to include them in our count.

3. **Graph Construction**: For each edge provided in the `edges` array, we add the corresponding nodes to the adjacency list of each other. Since the graph represents an undirected tree, if there is an edge between `a` and `b`, then `b` needs to be in the adjacency list of `a` and vice versa.

4. **DFS (Depth-First Search) Function**: We define a recursive function `dfs` that takes a node `u` as an argument. Within this function, we check if `u` is already visited or not. If it hasn't been visited:

   - We increment the reachable nodes count `ans` by 1.
   - We mark `u` as visited by setting `vis[u]` to `True`.
   - We then call `dfs` recursively for all the unvisited neighbor nodes of `u` found in the adjacency list `g[u]`. The use of recursion inherently follows a depth-first traversal, going as deep as possible along one branch before backtracking.

5. **Initialization and Invocation**: Before invoking the DFS function, we initialize the `ans` variable to `0`, which will be used to keep track of the number of reachable nodes. We then call `dfs(0)` to start the traversal from node `0`.

6. **Result**: After the DFS completes, `ans` gives us the total number of nodes that can be reached from node `0` without traversing any restricted nodes, and this value is returned.

In this way, the DFS algorithm, an efficient graph traversal technique, along with an adjacency list representation of the tree, allows us to solve the problem with a concise and effective approach.

Example Walkthrough

Let's take a small example to illustrate the solution approach described above.

Suppose we have 5 nodes (from `0` to `4`) and `n - 1` edges connecting them as follows: `edges = [[0, 1], [0, 2], [1, 3], [1, 4]]`. The `restricted` list containing the nodes that cannot be visited is `restricted = [3]`.

Based on the edges, the tree structure looks like this:

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

Here, node `3` is restricted and should not be visited.

Now, let's walk through the steps of the solution approach:

1. **Data Structure**: We use a `defaultdict(list)` to construct the adjacency list `g` representing the graph:

   ```
   g = {0: [1, 2], 1: [0, 3, 4], 2: [0], 3: [1], 4: [1]}
   ```

2. **Pre-Marking Restricted Nodes**: We then create a list `vis` initialized with `False` values. Since only node `3` is restricted, `vis` will be:

   ```
   vis = [False, False, False, True, False]
   ```

   Node `3` is marked `True` to indicate it's visited/restricted.

3. **Graph Construction**: Our graph `g` has already been constructed from the edges array in step 1.

4. **DFS (Depth-First Search) Function**: We write a recursive function `dfs(u)` that:

   ```python
   def dfs(u):
       if vis[u]:
           return 0
       vis[u] = True
       count = 1
       for neigh in g[u]:
           count += dfs(neigh)
       return count
   ```

   This function will count reachable nodes not previously counted nor restricted.

5. **Initialization and Invocation**: With `ans = 0`, we call `dfs(0)` and begin the DFS traversal.

6. **Result**: We visit node `0`, which isn't in the `restricted` list nor visited. So `ans` is now `1`. The `dfs` goes on to visit nodes `1` and `2`. Node `1` has two children, `3` and `4`, but since `3` is restricted, it only counts `4`. After the complete DFS call, the `ans` value is `4` (nodes `0`, `1`, `2`, and `4`).

In this example, the `ans` value after running the DFS algorithm implies we can reach 4 nodes from node `0` without visiting any restricted nodes. This is consistent with our tree structure, taking into account the restrictions.

Python Solution

```python
from collections import defaultdict

class Solution:
    def reachableNodes(self, n: int, edges: List[List[int]], restricted: List[int]) -> int:
        # Create a graph represented as an adjacency list
        graph = defaultdict(list)
        # A list to keep track of visited nodes, default to False
        visited = [False] * n

        # Mark restricted nodes as visited so they won't be traversed
        for node in restricted:
            visited[node] = True

        # Construct the graph by adding edges
        for start, end in edges:
            graph[start].append(end)
            graph[end].append(start)

        # Depth-First Search (DFS) function to traverse the graph
        def dfs(node):
            nonlocal count_visited
            # If the node is already visited or restricted, return
            if visited[node]:
                return

            # Increase the count of visited (reachable) nodes
            count_visited += 1
            # Mark the node as visited
            visited[node] = True

            # Recursively visit all adjacent nodes
            for adjacent in graph[node]:
                dfs(adjacent)

        # Initialize the count of reachable nodes
        count_visited = 0
        # Start DFS from the first node
        dfs(0)
        # Return the number of reachable nodes
        return count_visited

```

Java Solution

```java
class Solution {
    // Graph represented as an adjacency list
    private List<Integer>[] graph;
    // Array to keep track of visited nodes
    private boolean[] visited;
    // Counter to keep track of the number of reachable nodes
    private int numberOfReachableNodes;

    // Method to return the number of reachable nodes
    public int reachableNodes(int n, int[][] edges, int[] restricted) {
        // Initialize the graph for each node
        graph = new List[n];
        for (int i = 0; i < n; i++) {
            graph[i] = new ArrayList<>();
        }

        // Mark restricted nodes as visited so they won't be explored in DFS
        visited = new boolean[n];
        for (int restrictedNode : restricted) {
            visited[restrictedNode] = true;
        }

        // Build the graph by adding edges
        for (int[] edge : edges) {
            int from = edge[0], to = edge[1];
            graph[from].add(to);
            graph[to].add(from);
        }

        // Initialize the count of reachable nodes and start DFS from node 0
        numberOfReachableNodes = 0;
        dfs(0);

        // Return the total count of reachable nodes
        return numberOfReachableNodes;
    }

    // DFS method to traverse nodes
    private void dfs(int node) {
        // If the current node is visited/restricted, do not proceed with DFS
        if (visited[node]) {
            return;
        }

        // Mark the node as visited and increment the reachable nodes counter
        numberOfReachableNodes++;
        visited[node] = true;

        // Visit all the neighbors of the current node
        for (int neighbor : graph[node]) {
            dfs(neighbor);
        }
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int nodeCount; // Use a more descriptive name for this variable, which represents the count of reachable nodes.

    int reachableNodes(int n, vector<vector<int>>& edges, vector<int>& restricted) {
        vector<vector<int>> graph(n); // Use 'graph' to store the adjacency list representation of the graph.
        // Construct the graph from the given edge list.
        for (const auto& edge : edges) {
            int from = edge[0], to = edge[1];
            graph[from].push_back(to);
            graph[to].push_back(from);
        }
        vector<bool> visited(n, false); // Use 'visited' to mark the visited nodes, initialized to false.
        // Mark the restricted nodes as visited so they won't be counted.
        for (int node : restricted) visited[node] = true;

        // Initialize the reachable node count to zero before starting the DFS.
        nodeCount = 0;
        // Start depth-first search from node 0.
        dfs(0, graph, visited);
        return nodeCount; // Return the count of reachable nodes after the DFS is complete.
    }

    void dfs(int node, vector<vector<int>>& graph, vector<bool>& visited) {
        if (visited[node]) {
            // If the node has been visited or is restricted, return early.
            return;
        }
        // Mark the current node as visited.
        visited[node] = true;
        // Increment the count of reachable nodes.
        nodeCount++;
        // Iterate over all the neighbors of the current node.
        for (int neighbor : graph[node]) {
            // Perform DFS on all non-visited neighbors.
            dfs(neighbor, graph, visited);
        }
    }
};

```

Typescript Solution

```typescript
function reachableNodes(n: number, edges: number[][], restricted: number[]): number {
    let nodeCount = 0; // Holds the count of nodes that can be reached
    const visited = new Array(n).fill(false); // Tracks visited nodes
    const adjacencyList = new Map<number, number[]>(); // Adjacency list to represent the graph

    // Construct the adjacency list from the edges
    for (const [start, end] of edges) {
        // Append the end node to the adjacency list of the start node
        adjacencyList.set(start, [...(adjacencyList.get(start) ?? []), end]);
        // Append the start node to the adjacency list of the end node
        adjacencyList.set(end, [...(adjacencyList.get(end) ?? []), start]);
    }

    // Depth-First Search function to explore the graph
    const dfs = (currentNode: number) => {
        // If the node has been visited or is restricted, stop the search
        if (restricted.includes(currentNode) || visited[currentNode]) {
            return;
        }
        // Increment node count and mark the node as visited
        nodeCount++;
        visited[currentNode] = true;

        // Iterate over all adjacent nodes and explore them
        for (const neighbor of adjacencyList.get(currentNode) ?? []) {
            dfs(neighbor);
        }
    };

    // Begin the Depth-First Search from node 0
    dfs(0);

    // Return the total number of nodes that can be reached
    return nodeCount;
}

```

Time and Space Complexity

The given Python code defines a `Solution` class with a method `reachableNodes` to count the number of nodes in an undirected graph that can be reached without visiting any of the restricted nodes. It uses a Depth First Search (DFS) approach.

Time Complexity

The time complexity of the code is primarily determined by the DFS traversal of the graph provided by `edges`.

- The adjacency list for the graph is constructed in linear time relative to the number of edges. `for a, b in edges` loop runs $|E|$ times, where $|E|$ is the number of edges.

- The DFS (`dfs` function) visits each vertex only once, because once a vertex has been visited, it is marked as visited (`vis[u] = True`).

- Each edge is traversed exactly twice in an undirected graph (once for each direction), during the entire DFS process.

Therefore, the time complexity of the DFS traversal is $O(|V| + |E|)$, where $|V|$ is the number of vertices and $|E|$ is the number of edges.

Combining the adjacency list construction and the DFS traversal, the total time complexity remains $O(|V| + |E|)$ since both are dependent on the size of the graph.

Space Complexity

The space complexity is determined by:

- The storage of the graph in the adjacency list `g`, which consumes $O(|V| + |E|)$ space.
- The recursive DFS call stack, which, in the worst case, could hold all vertices if the graph is a linked list shape (each vertex connected to only one other), hence $O(|V|)$ in space.
- The `vis` array, which takes $O(|V|)$ space, to keep track of visited nodes, including restricted ones.

Summing this up, the space complexity is $O(|V| + |E|)$ when considering both the adjacency list and the DFS stack.