2858. Minimum Edge Reversals So Every Node Is Reachable

Hard · Depth-First Search · Breadth-First Search · Graph · Dynamic Programming · Leetcode Link

Problem Description

In this LeetCode problem, we are given a directed graph that has `n` nodes numbered from `0` to `n - 1`. The interesting property of this graph is that it would form a tree if the edges were bi-directional, meaning that there are no cycles and the graph is connected when considering the edges as undirected. However, as the edges are actually directed, not all nodes may be reachable from a given node.

We are provided with a 2D integer array `edges`, which represents the directed edges of the graph, where each element `edges[i] = [u_i, v_i]` denotes an edge from node `u_i` to node `v_i`.

An edge reversal is an operation that changes the direction of a directed edge, so an edge from `u` to `v` would be reversed to point from `v` to `u` instead.

The task is to calculate, for every node `i`, the minimum number of edge reversals that must be performed so that starting from node `i`, it would be possible to reach every other node in the graph. The solution should be returned as an integer array `answer`, where `answer[i]` gives the minimum number of edge reversals starting from node `i`.

Intuition

The solution to this problem involves two main steps. First, we construct a graph representation that can help us determine the number of reversals needed to make all nodes accessible from a starting node. This is done by capturing not just the connection between nodes, but also the "directionality" with a value that indicates whether an edge needs to be reversed or not (signified by 1 for a properly directed edge, and -1 for an edge that requires reversal).

Once the graph is constructed, we commence a Depth First Search (DFS) to traverse the tree starting from an arbitrary root node (in this case, chosen to be node `0`). During this traversal, we accumulate the number of reversals needed - which is when we encounter an edge where value `k` is negative.

After the initial pass, we do a second DFS to calculate the minimum number of reversals for all nodes based on the previously obtained information stored in `ans[0]`. This second DFS leverages the property that a reversal affects the reachability of all the nodes in the subtree of the reversed edge. Thus, by adjusting `ans[i]` for each node `i` by the direction of the incoming edge `k`, we propagate the required count of edge reversals throughout the tree.

This approach intuitively understands that in order to make all nodes reachable from one point, the direction of some edges need to align appropriately - which is essentially forming an outward spanning tree from the starting node, with edges oriented away from it.

Solution Approach

The implemented solution for the problem uses the Depth First Search (DFS) algorithm twice on the graph. The graph is given a special structure where every edge is represented by a tuple containing the node it leads to and an integer indicating whether the edge points towards the node (`1`) or it needs to be reversed (`-1`).

Here are the detailed steps of the solution implementation:

1. The graph `g` is initialized as a list of lists, to hold the adjacency list representation of the graph. Each list within `g` corresponds to a node and contains tuples of its adjacent nodes and the directionality of the edges.
2. The `dfs` function is defined to traverse the graph starting from the root node `0`. During the traversal, if a reversal is needed (indicated by a negative value), the number of necessary edge reversals `ans[0]` is incremented. The `dfs` function is called with the parameters `i` (the current node being visited) and `fa` (the node from which the current node was reached).
   - The function iterates over all the nodes `j` adjacent to the current node `i`.
   - If `j` is not the parent node `fa`, it inspects the direction `k` of the edge between `i` and `j`.
   - If `k` is negative, it means the edge is directed towards `i` and not away from it, and one reversal is counted.
3. After the initial DFS which counts reversals starting from the root, the `dfs2` function is then called in a similar manner. It's called initially with the root node `0` and its parent `-1`.
   - This function also iterates over each adjacent node `j` of the current node `i` and its edge direction `k`.
   - Instead of just counting reversals, `dfs2` propagates the reversal count to each node. It sets `ans[j]` to `ans[i] + k` for each node `j` that is not the parent.
   - The value of `k` is added because if `k` is `1`, no reversal is needed from `i` to `j`, and if `k` is `-1`, an extra reversal is needed to reach `j` from `i`.

Through this propagation, `dfs2` calculates the optimal number of reversals needed to reach all other nodes from a given node `i` in the tree. Since `ans[i]` already contains the minimum reversals to reach `i` from the root (calculated in the initial DFS), adding `k` adjusts this number based on the direction of the edge leading to the child node `j`.

In the end, we return the `ans` array that contains the minimum number of reversals needed for each node to reach all other nodes. The usage of DFS is crucial as it allows us to visit each node and its subtree, accumulating and propagating the count of necessary reversals across the graph. The logic applied in the second DFS makes use of information collected during the first DFS to solve the problem efficiently.

Example Walkthrough

To illustrate the solution approach, consider a small directed graph with `n = 3` nodes and the following edges: `edges = [[0, 1], [1, 2], [2, 0]]`. The graph forms a cycle if we ignore the direction of the edges. Here's how we apply the solution approach to determine the number of reversals needed for each node to reach all other nodes:

1. We initialize the graph representation `g` with the adjacency list for the directed graph:

```
g = [
  [(1, 1)],       // Node 0 has a directed edge towards Node 1
  [(2, 1)],       // Node 1 has a directed edge towards Node 2
  [(0, -1)]       // Node 2 has an edge pointing towards Node 0, which needs reversal
]
```

2. We define a `dfs` function to traverse the graph and begin by calling `dfs(0, -1)`, which starts the traversal from Node 0 with -1 signifying no parent.
   - The `dfs` function will find no reversals are needed from Node 0 to Node 1. The traversal continues to Node 2, where the `dfs` detects that an edge reversal is needed (from Node 2 to Node 0). This increments the number of necessary edge reversals for the root, `ans[0]`, by 1.
3. We then implement the `dfs2` function and first call it with `dfs2(0, -1)` to set the number of reversals for each node. It propagates the reversals needed from Node 0 to Node 1 (which remains 0 since the edge from Node 0 to Node 1 does not require reversal). Then it sets the number of reversals for Node 2:
   - Since no reversal is needed to go from Node 1 to Node 2 (`k = 1`), `ans[2]` is set equal to `ans[1]`.
   - However, remember that starting from Node 2, we would need to reverse the edge to Node 0 to create a path from Node 2 to Node 0, adding 1 to the reversal count for Node 2.

By applying these steps, we obtain the final array `ans`, which gives us `ans = [1, 0, 0]`. This means that starting from Node 0, one reversal is required for all nodes to be reachable (reversing the edge from Node 2 to Node 0). For Nodes 1 and 2, no reversals are needed because there is a direct path from these nodes to every other node in the graph.

This example shows the basic thinking behind the solution: construct a graph that captures directionality, perform an initial DFS to count reversals from the root, and then propagate that count to all nodes using a second DFS, adjusting the count based on the direction of the incoming edges to each node.

Python Solution

```python
from typing import List

class Solution:
    def minEdgeReversals(self, n: int, edges: List[List[int]]) -> List[int]:
        # Initialize the answer list to keep the minimum edge reversals needed to reach each node
        min_reversals = [0] * n
        # Initialize the graph as an adjacency list with additional info about edge direction
        graph = [[] for _ in range(n)]
        for start, end in edges:
            # Positive direction for original edge, negative for reversed
            graph[start].append((end, 1))
            graph[end].append((start, -1))

        # Depth-First Search function to calculate minimum reversals from the root to other nodes
        def dfs(from_node: int, parent_node: int):
            for next_node, edge_type in graph[from_node]:
                if next_node != parent_node:
                    # If we encounter a reversed edge, increment the reversal count
                    min_reversals[0] += int(edge_type < 0)
                    dfs(next_node, from_node)

        # First DFS to calculate reversals on path from root (node 0) to all other nodes
        dfs(0, -1)

        # Depth-First Search function to update the minimum reversals needed to reach each node
        def dfs_update(node: int, parent_node: int):
            for next_node, edge_type in graph[node]:
                if next_node != parent_node:
                    # Update the reversal count depending on edge direction
                    min_reversals[next_node] = min_reversals[node] + edge_type
                    dfs_update(next_node, node)

        # Second DFS to update the min_reversals list with the correct counts for each node
        dfs_update(0, -1)

        # Return the list of minimum edge reversals needed to reach each node from the root
        return min_reversals

```

Java Solution

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

class Solution {
    // Adjacency list representation of the graph, where each array element is a pair (neighbor, edge type)
    // Edge type is 1 for normal edge, -1 for reversed edge
    private List<int[]>[] graph;

    // Array to store the minimum number of edge reversals needed to reach each node from the root (node 0)
    private int[] minEdgeReversals;

    public int[] minEdgeReversals(int n, int[][] edges) {
        // Initialize the minEdgeReversals array and graph adjacency list
        minEdgeReversals = new int[n];
        graph = new List[n];
        Arrays.setAll(graph, i -> new ArrayList<>());

        // Populate the graph with the edges, marking each edge with a type (normal or reversed)
        for (int[] edge : edges) {
            int from = edge[0], to = edge[1];
            graph[from].add(new int[] {to, 1}); // Add normal edge
            graph[to].add(new int[] {from, -1}); // Add reversed edge
        }

        // Perform first DFS to calculate minimum reversals to reach each node from the root
        dfs(0, -1);

        // Perform second DFS to propagate the number of reversals down the tree
        propagateEdgeReversals(0, -1);

        // Return the minimum number of edge reversals needed for each node
        return minEdgeReversals;
    }

    // Helper method to perform DFS traversal and calculate the edge reversals for root node
    private void dfs(int node, int parent) {
        for (int[] neighbor : graph[node]) {
            int nextNode = neighbor[0], edgeType = neighbor[1];
            if (nextNode != parent) {
                minEdgeReversals[0] += edgeType < 0 ? 1 : 0; // Reverse the edge if necessary
                dfs(nextNode, node);
            }
        }
    }

    // Helper method to propagate the calculated edge reversals down the tree
    private void propagateEdgeReversals(int node, int parent) {
        for (int[] neighbor : graph[node]) {
            int nextNode = neighbor[0], edgeType = neighbor[1];
            if (nextNode != parent) {
                // Add the edgeType to the current node's reversals for child node
                // If edgeType is -1 (reversed edge), we subtract 1, otherwise no change
                minEdgeReversals[nextNode] = minEdgeReversals[node] + edgeType;
                propagateEdgeReversals(nextNode, node);
            }
        }
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <functional> // To use std::function for the DFS lambda functions
using namespace std;

class Solution {
public:
    // Finds the minimum number of edge reversals in a graph to make all edges directed
    // from node 0 to any other node. Assumes the graph is connected.
    // Parameters:
    // - n: number of nodes in the graph
    // - edges: a list of edges represented by pairs of nodes
    //
    // Returns a vector<int> with the minimum number of reversals for each node
    vector<int> minEdgeReversals(int nodeCount, vector<vector<int>>& edges) {
        // Create an adjacency list where each edge is represented by a pair
        // consisting of the adjacent node and the edge type (1 for original direction, -1 for reversed)
        vector<pair<int, int>> adjacencyList[nodeCount];
        // Create a vector to hold the number of reversals needed for each node
        vector<int> reversals(nodeCount, 0);

        // Convert edge list to adjacency list with edge types
        for (auto& edge : edges) {
            int source = edge[0], destination = edge[1];
            adjacencyList[source].emplace_back(destination, 1);
            adjacencyList[destination].emplace_back(source, -1);
        }

        // Depth-First Search (DFS) to determine the reversals starting from node 0
        function<void(int, int)> dfs1 = [&](int currentNode, int parent) {
            for (auto& [adjacentNode, edgeType] : adjacencyList[currentNode]) {
                if (adjacentNode != parent) {
                    reversals[0] += edgeType < 0; // Increment if the edge is reversed
                    dfs1(adjacentNode, currentNode);
                }
            }
        };

        // Secondary DFS to calculate the correct number of reversals for each node
        function<void(int, int)> dfs2 = [&](int currentNode, int parent) {
            for (auto& [adjacentNode, edgeType] : adjacencyList[currentNode]) {
                if (adjacentNode != parent) {
                    reversals[adjacentNode] = reversals[currentNode] + edgeType;
                    dfs2(adjacentNode, currentNode);
                }
            }
        };

        // Perform the two DFS traversals to compute the edge reversals
        dfs1(0, -1); // Start at node 0 with no parent
        dfs2(0, -1); // Start at node 0 with no parent to calculate actual reversals

        return reversals;
    }
};

```

Typescript Solution

```typescript
// Define the function to calculate the minimum edge reversals
// n: number of nodes in the graph
// edges: list of edges represented by pairs of nodes
function minEdgeReversals(n: number, edges: number[][]): number[] {
    // Construct the graph where each node points to a list of [neighbor node, edge type]
    // Edge type: 1 for forward edge, -1 for reversed edge
    const graph: number[][][] = Array.from({ length: n }, () => []);
    // Populate the graph adjacency list with bidirectional edges
    for (const [from, to] of edges) {
        graph[from].push([to, 1]);   // Forward edge
        graph[to].push([from, -1]);  // Reversed edge
    }

    // Initialize an array to hold the minimum reversal count for each node
    const minReversals: number[] = Array(n).fill(0);

    // Depth-first search to count reversals needed to reach each node from node 0
    const depthFirstSearch = (node: number, parent: number) => {
        for (const [nextNode, edgeType] of graph[node]) {
            if (nextNode !== parent) { // Exclude parent to prevent cycling back
                // If edge is reversed, add 1 to the reversal count
                minReversals[0] += edgeType < 0 ? 1 : 0;
                // Recursive DFS call for child nodes
                depthFirstSearch(nextNode, node);
            }
        }
    };

    // Second DFS to update each node's minimum reversal count based on the path from root
    const depthFirstSearchUpdate = (node: number, parent: number) => {
        for (const [nextNode, edgeType] of graph[node]) {
            if (nextNode !== parent) {
                // Update the reversal count for child node based on current node's count
                minReversals[nextNode] = minReversals[node] + edgeType;
                // Recursive call to continue updating child nodes
                depthFirstSearchUpdate(nextNode, node);
            }
        }
    };

    // Call the DFS function starting from node 0 with no parent (-1)
    depthFirstSearch(0, -1);
    // After initial count, update the reversal count for each node
    depthFirstSearchUpdate(0, -1);

    // Return the array containing minimum reversal counts for each node
    return minReversals;
}

```

Time and Space Complexity

Time Complexity

The given code consists of two Depth First Search (DFS) functions, `dfs` and `dfs2`, which are used to traverse the graph represented by the list of edges. Every vertex and every edge is visited exactly once in each DFS operation.

- The first DFS (`dfs`) traverses the graph to count the minimum number of edge reversals required. It does so by incrementing the answer only if an edge is directed towards the parent (`k < 0`), which requires a reversal.
- The second DFS (`dfs2`) computes the final answer array for each node, by updating the answer for each child based on the edge direction (`k`) with respect to its parent.

The time complexity of each DFS call is `O(V + E)` where `V` is the number of vertices (nodes) and `E` is the number of edges, since it traverses each vertex and edge exactly once. Since we are running two DFS operations sequentially, and each has the same complexity, the total time complexity of the algorithm is `O(V + E + V + E)` which simplifies to `O(V + E)` because constants are dropped in Big O notation.

Therefore, the overall time complexity is `O(V + E)`.

Space Complexity

The space complexity of the code involves the storage required for:

- The adjacency list `g` which has `E` entries (each edge stored twice, once for each direction, with an additional bit to indicate the original direction).
- The recursion stack for the DFS calls, which in the worst case, if the graph is implemented as a linked list, could go up to `O(V)`.
- The `ans` array, which is an array of length `V`.

Combining the above, the space complexity is `O(V + 2E) + O(V) + O(V)`.

Since `E` can be at most `V(V - 1)/2` for a complete graph, the term `O(2E)` can dominate the space complexity. However, we typically express space complexity in terms of just `O(E)` or `O(V)`, depending on which is larger. Therefore, the space complexity is generally represented as `O(E + V)`.

In conclusion, the space complexity of the algorithm is `O(E + V)`.