2316. Count Unreachable Pairs of Nodes in an Undirected Graph

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

Problem Description

In this task, we are presented with an undirected graph defined by `n` nodes numbered from `0` to `n - 1`. The graph's connectivity is provided as an array, `edges`, where each element consists of a pair of integers that represent an undirected edge between two nodes in the graph. Our goal is to determine the number of node pairs that are unreachable from each other. Specifically, we must find all the pairs of different nodes where there is no path from one node to the other within the graph.

To visualize this, you could picture a set of islands (nodes) connected by bridges (edges). We are trying to count how many pairs of islands cannot be traveled between directly or indirectly.

Intuition

The approach to solving this problem involves understanding how connected components in an undirected graph work. A connected component is a subgraph where any two nodes are connected to each other by paths, and which is connected to no additional nodes in the supergraph. Essentially, all nodes within a connected component can reach each other, but they cannot reach nodes in other connected components.

By traversing the graph and determining the size of each connected component, we can calculate the number of unreachable pairs. The idea is that if a connected component has `t` nodes, none of the nodes in this component can reach nodes in the rest of the graph, which we can denote as `s` nodes. The number of unreachable pairs involving nodes from this component would then be the product `s * t`.

For instance, suppose we have a connected component of 4 nodes, and there are 6 nodes not in this component. There can be no paths between any of the 4 nodes and the 6 outside nodes, giving us `4 * 6 = 24` unreachable pairs.

To implement this concept programmatically, depth-first search (DFS) is a fitting choice. DFS can be used to explore the graph from each node, marking visited nodes to avoid counting a connected component more than once.

The algorithm systematically goes through each node. If the node hasn't been visited yet, it gets passed to a depth-first search, which counts all nodes reachable from that starting node (i.e., the size of the connected component). Once we get the size `t` of a connected component, we can calculate the number of unreachable pairs with nodes outside this component (which we have kept track of in `s`), and add it to the answer. We then update `s` to include the nodes from the newly found connected component before moving on to the next unvisited node.

This method ultimately gives us the sum of unreachable pairs for each connected component in the graph, which is the desired result.

Solution Approach

The solution to the problem uses a classical graph traversal method known as Depth-First Search (DFS). DFS is a recursive algorithm that starts at a node and explores as far as possible along each branch before backtracking. This is perfect for exploring and marking all nodes within a connected component.

Here's how the algorithm is implemented:

1. An adjacency list representation of the graph `g` is created, which is a list of lists. For every edge (a, b) in the given list `edges`, we add node `b` to the list of node `a` and vice versa because the graph is undirected.

```
g = [[] for _ in range(n)]
for a, b in edges:
    g[a].append(b)
    g[b].append(a)
```

2. An array `vis` of boolean values is used to keep track of visited nodes. Initially, all nodes are unvisited, so they are set to `False`.

```
vis = [False] * n
```

3. The solution defines a recursive function `dfs` that takes an integer `i` representing the current node. It checks if this node is already visited. If it is, the function returns `0` because it shouldn't be counted again. If not, it sets the current node as visited (`True`) and explores all its neighbors by recursively calling `dfs(j)` for every neighbor `j`.

```
def dfs(i: int) -> int:
    if vis[i]:
        return 0
    vis[i] = True
    return 1 + sum(dfs(j) for j in g[i])
```

The `dfs` function returns `1` (for the current node) plus the sum of nodes that can be reached from it, giving us the total size of the connected component.

4. The main body of the solution maintains two variables, `ans` and `s`. The `ans` variable holds the cumulative count of unreachable pairs, while `s` keeps track of the total number of nodes processed so far across connected components.

5. The solution iterates over all nodes, and for each unvisited node, it calls `dfs` to get the size of its connected component. The product of the current connected component size `t` and the count of nodes processed so far `s` gives us the number of unreachable pairs with respect to the component starting at this node.

```
ans = s = 0
for i in range(n):
    t = dfs(i)
    ans += s * t
    s += t
```

6. Finally, after iterating through all the nodes, `ans` will contain the total number of pairs of nodes that are unreachable from each other. This is returned as the final result.

The algorithm effectively partitions the graph into disconnected "islands" (connected components) and calculates unreachable pairs by considering the complement of nodes for each component encountered.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach.

Suppose we are given a graph with `n = 5` nodes and the following `edges`: `[[0, 1], [1, 2], [3, 4]]`.

This graph consists of two separate connected components:

- Component 1: Nodes `0`, `1`, and `2` are connected (0↔1↔2).
- Component 2: Nodes `3` and `4` are connected (3↔4).

Using the approach described above, we will determine the number of pairs of nodes that are unreachable from each other.

1. We create an adjacency list for our graph:

```
g = [[1], [0, 2], [1], [4], [3]]
```

2. Initialize a visited list with `False` showing none of the nodes is visited:

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

3. We define our DFS function `dfs`. During the DFS process, this function will return the number of connected nodes for each connected component in the graph.

4. We start traversing the nodes and applying DFS:

   - When we apply DFS to node `0`, it will visit nodes `1` and `2` since they are connected. After the DFS call, `visited` becomes `[True, True, True, False, False]`.
   - The size `t` of this component is `3`. The `s` is initialized to `0`, so `ans` becomes `0 * 3 = 0`.
   - We update `s` to `s + t` which becomes `3`.

5. Node `1` and `2` are already visited, so our loop moves on to node `3`. DFS on node `3` will visit node `4`. `visited` becomes `[True, True, True, True, True]`.

   - The size `t` of this second component is `2`. Now `s = 3` (from the previous step), and we update `ans` to `ans + (s * t)` which becomes `0 + (3 * 2) = 6`.
   - Update `s` again to `s + t` which is now `5`.

6. Our `ans` is `6`, which represents the total number of pairs of nodes that can't be reached from each other, which corresponds to the pairs (0,3), (0,4), (1,3), (1,4), (2,3), and (2,4).

After iterating through all nodes, the `ans` variable contains the correct number of unreachable node pairs, which is `6` in this case.

Python Solution

```python
from typing import List

class Solution:
    def countPairs(self, n: int, edges: List[List[int]]) -> int:
        # Depth First Search function to count nodes in a connected component
        def dfs(node: int) -> int:
            if visited[node]:
                return 0
            visited[node] = True
            # Count current node + all nodes reachable from current node
            return 1 + sum(dfs(neighbor) for neighbor in graph[node])

        # Initialize the graph as an adjacency list
        graph = [[] for _ in range(n)]
        for node1, node2 in edges: # Build undirected graph
            graph[node1].append(node2)
            graph[node2].append(node1)

        visited = [False] * n # Track visited nodes

        # Main logic to count pairs
        answer = total_nodes_visited = 0
        for i in range(n):
            component_size = dfs(i) # Size of connected component for node i
            answer += total_nodes_visited * component_size # Multiply with size of previously found components
            total_nodes_visited += component_size # Update total nodes visited after exploring component

        # Return the total number of pairs
        return answer

```

Java Solution

```java
class Solution {

    // Graph represented by an adjacency list
    private List<Integer>[] graph;
    // Visited array to keep track of visited nodes during DFS
    private boolean[] visited;

    // Method to count the number of pairs that can be formed
    public long countPairs(int n, int[][] edges) {
        graph = new List[n];
        visited = new boolean[n];
        // Initialize adjacency lists for each node
        Arrays.setAll(graph, i -> new ArrayList<>());
        // Build the graph by adding edges
        for (int[] edge : edges) {
            int a = edge[0], b = edge[1];
            graph[a].add(b);
            graph[b].add(a);
        }

        long answer = 0;
        // Sum of component sizes found so far
        long sumOfComponentSizes = 0;
        // Traverse each node
        for (int i = 0; i < n; ++i) {
            // Perform a DFS from the node, count the size of the component
            int componentSize = dfs(i);
            // Update the answer with the product of component sizes
            answer += sumOfComponentSizes * componentSize;
            // Add the component size to the sum of component sizes
            sumOfComponentSizes += componentSize;
        }
        return answer;
    }

    // Depth-first search to find component size
    private int dfs(int currentNode) {
        // If node is visited, return 0
        if (visited[currentNode]) {
            return 0;
        }
        // Mark the current node as visited
        visited[currentNode] = true;
        // Start with a count of 1 for the current node
        int count = 1;
        // Recur for all the vertices adjacent to this vertex
        for (int nextNode : graph[currentNode]) {
            count += dfs(nextNode);
        }
        // Return the size of the component
        return count;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    long long countPairs(int n, vector<vector<int>>& edges) {
        // Create an adjacency list for the graph
        vector<int> graph[n];
        for (const auto& edge : edges) {
            int from = edge[0], to = edge[1];
            graph[from].push_back(to);
            graph[to].push_back(from);
        }

        // Create a visited array to keep track of visited nodes
        vector<bool> visited(n, false);

        // Define a depth-first search (DFS) lambda function to count nodes in a component
        function<int(int)> dfs = [&](int node) {
            if (visited[node]) {
                return 0; // If already visited, terminate this path
            }
            visited[node] = true; // Mark this node as visited
            int count = 1; // Start count with the current node itself
            for (int neighbor : graph[node]) {
                count += dfs(neighbor); // Recursively visit neighbors and add to the count
            }
            return count;
        };

        long long answer = 0;   // Initialize the answer to 0
        long long sumOfCounts = 0; // Initialize the running sum of counts to 0

        // Iterate through each node in the graph
        for (int i = 0; i < n; ++i) {
            int componentSize = dfs(i); // Get the size of the component via DFS
            answer += sumOfCounts * componentSize; // Add to the answer the product of current sum of counts and component size
            sumOfCounts += componentSize; // Update the running sum of counts with the size of this component
        }

        // Return the final answer, the total count of pairs
        return answer;
    }
};

```

Typescript Solution

```typescript
// Function to count the number of reachable pairs in the undirected graph,
// where n is the total number of nodes and edges is a list of edges connecting the nodes.
function countPairs(n: number, edges: number[][]): number {
    // Create an adjacency list to represent the graph.
    const graph: number[][] = Array.from({ length: n }, () => []);

    // Populate the adjacency list with bidirectional edges.
    for (const [node1, node2] of edges) {
        graph[node1].push(node2);
        graph[node2].push(node1);
    }

    // Array to track visited nodes to prevent revisiting.
    const visited: boolean[] = Array(n).fill(false);

    // Depth-first search function to count connected nodes.
    const depthFirstSearch = (node: number): number => {
        // If the node is already visited, return 0 to avoid counting it again.
        if (visited[node]) {
            return 0;
        }

        // Mark the current node as visited.
        visited[node] = true;

        // Start with a count of 1 for the current node.
        let count = 1;

        // Recursively visit all connected nodes and increment count.
        for (const connectedNode of graph[node]) {
            count += depthFirstSearch(connectedNode);
        }

        // Return the count of nodes in the connected component.
        return count;
    };

    // Initialize the answer to 0 and sum to keep track of the number of nodes visited so far.
    let answer = 0;
    let sum = 0;

    // Iterate over each node to calculate the number of reachable pairs.
    for (let i = 0; i < n; ++i) {
        // Get the count of nodes in the connected component starting from node i.
        const connectedNodes = depthFirstSearch(i);

        // Update the answer with the number of pairs formed between the current
        // connected component and the previously processed nodes.
        answer += sum * connectedNodes;

        // Update the sum with the number of nodes in the current connected component.
        sum += connectedNodes;
    }

    // Return the final count of reachable pairs in the graph.
    return answer;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is primarily determined by the depth-first search (`dfs`) function and the construction of the graph `g`.

- Constructing the graph `g` involves iterating over all `edges`, which takes `O(m)` time where `m` is the total number of edges.
- The `dfs` function will visit each node exactly once. Since an edge is considered twice (once for each of its endpoints), the `dfs` calls contribute `O(n + m)` time, where `n` is the total number of nodes.
- The main loop (`for i in range(n)`) iterates `n` times and calls `dfs` during its iterations.

Combining these steps, the total time complexity is `O(n + m)` strictly speaking, as it accounts for the time to build the graph and the time to perform the DFS across all nodes and edges.

Space Complexity

The space complexity of the algorithm is influenced by the space needed to store the graph and the `vis` array.

- The graph `g` is an adjacency list representation of the graph, which can consume up to `O(n + m)` space since each edge connects two nodes, and it is stored twice.
- The `vis` array contains one boolean per node, contributing `O(n)` space.

Adding these up, the total space complexity is `O(n + m)` which comes from the adjacency list and the `vis` array.