2685. Count the Number of Complete Components

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

Problem Description

You are tasked with finding the number of complete connected components in an undirected graph. The graph has `n` vertices, each identified by a unique number from `0` to `n - 1`. An array `edges` contains pairs of integer indices representing the edges that connect vertices in the graph.

A *connected component* is a set of vertices in which each pair of vertices is connected by some path. Importantly, no vertex in a connected component connects to any vertex outside that component in the graph.

A *complete connected component* (also known as a clique) is a special kind of connected component where there is a direct edge between every pair of nodes within the component.

So, the objective is to count how many complete connected components there are in the given graph.

Intuition

To solve this problem, we can use **Depth-First Search (DFS)**. DFS allows us to explore each vertex and its neighbors recursively. The solution employs DFS to count vertices (`x`) and edges (`y`) in each connected component. Since a complete graph with `n` vertices should have exactly `n * (n - 1) / 2` edges, this fact can be used to verify if a connected component is complete.

Here's how the algorithm works step by step:

1. Create a graph representation using a list where each index represents a node and the values are lists of connected nodes.
2. Initialize a list of boolean values to track visited nodes during DFS to ensure nodes are not counted more than once.
3. Iterate through each node. If a node is unvisited, perform DFS starting from that node to determine the size of the connected component.
4. During DFS, count the number of nodes (`x`) and the number of edges (`y`). Note that since the graph is undirected, each edge will be counted twice - once for each endpoint.
5. After each DFS call, check if the number of edges matches the formula `x * (x - 1)` (which is double `x * (x - 1) / 2` since each edge is counted twice). If it does, then we have found a complete connected component.
6. Increment a counter for complete connected components any time the check in step 5 passes.
7. After completing the iteration through all nodes, return the counter value.

This solution is efficient because it only requires a single pass through the graph, using DFS to explore the connected components and immediately checking which ones are complete.

Solution Approach

The solution uses the following algorithms, data structures, and patterns:

Depth-First Search (DFS): This algorithm allows us to explore all nodes in a connected component of the graph. We define a recursive `dfs` function that takes a starting vertex `i` and explores every connected vertex, returning the total count of vertices and edges in the connected component.

Adjacency List: The graph is represented as an adjacency list using a `defaultdict(list)` from Python's collections library. This data structure is efficient for storing sparse graphs and allows us to quickly access all neighbors of a graph node.

Visited List: To keep track of which nodes have been visited during the DFS exploration, we maintain a list `vis` of boolean values. this avoids counting the same node more than once and ensures we don't enter into an infinite loop by revisiting the same nodes.

Here's how the implementation follows the algorithm:

1. Initialize the graph as an adjacency list (`g`) from the `edges` array, where for each edge `[a, b]` we add `b` to the list of adjacent vertices of `a` and vice versa since the graph is undirected.

2. Create a visited list `vis` of boolean values, initialized to `False`, to mark vertices as visited.

3. For each vertex `i` from `0` to `n-1`, check if it has not been visited:

   - If not visited, call the `dfs` function on vertex `i`. This will traverse all vertices in the same connected component as `i`, marking them as visited and counting the vertices and edges.
   - The `dfs` function works as follows:
     - Mark the current vertex `i` as visited.
     - Initialize local variables `x` (vertex count) to `1` and `y` (edge count) to the length of the adjacency list at `i` since every adjacent vertex represents an edge.
     - Iterate over each neighbor `j` of vertex `i`. If `j` is not visited, call `dfs` recursively and update `x` and `y` with the counts from the component connected through `j`.
     - Return the count of vertices `x` and the double count of edges `y`.
4. After the recursive DFS, compare the number of vertices `a` and the double count of edges `b` using the formula `a * (a - 1)`. A complete component must have exactly `a * (a - 1) / 2` edges, but since each edge is counted twice, we check for `a * (a - 1)` without dividing by 2. If the component is complete, increment the `ans`.

5. Finally, when all vertices are processed, return `ans`, which represents the count of complete connected components.

The solution effectively utilizes the `dfs` function for graph traversal and the combinatorial property of complete graphs to detect complete connected components in a single scan.

Example Walkthrough

Let's assume we have `n = 5` vertices, and we are given the following `edges` array, which defines an undirected graph:

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

The graph can be visualized as two separate connected components:

```
0 -- 1 -- 2

3 -- 4
```

Here's a step-by-step illustration of applying the solution approach:

1. First, we create an adjacency list `g` for the given graph. This list will look like:

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

2. We initialize a visited list `vis = [False, False, False, False, False]` to keep track of visited nodes.

3. We start iterating through each vertex. When iterating over vertex `0`, since it's not visited, we call the `dfs` function.

4. The `dfs` function now starts on `0`, marking it as visited (`vis[0] = True`) and setting `x = 1` and `y = len(g[0]) = 1`.

5. Within DFS on vertex `0`, we move to its neighbor `1` which is unvisited, and call DFS on `1`, marking it visited and adding its own vertices and edges. We get `x = 2` and `y = 3` (since it's connected to 0 and 2).

6. Continuing the DFS, we visit vertex `2`. No new vertices are discovered, but this adds to the edge count: `y += len(g[2]) = 1`.

7. The DFS call on vertex `0` ends, returning `x = 3` and `y = 4`.

8. We then check if `x * (x - 1) == y` which would mean it's a complete connected component. `3 * (3 - 1) = 6` which does not equal `4`, so this component is not complete.

9. We update the visited list for all vertices in the connected component containing `0`, `1`, and `2`.

10. We continue the iteration to vertex `3`, calling DFS since it's not visited.

11. Similar to the previous DFS, we mark `3` as visited, setting `x = 1` and `y = len(g[3]) = 1` since it's connected to `4`.

12. On visiting neighbor `4`, we update `x = 2` and `y = 2`.

13. After the DFS on `3`, we check if `x * (x - 1) == y`. `2 * (2 - 1) = 2` which equals `y`, so this is indeed a complete connected component.

14. We add `1` to our complete connected component count `ans`.

15. Finally, after checking all vertices, we find that there is `1` complete connected component in the graph.

In conclusion, after traversing this example graph, our algorithm would correctly report that there is 1 complete connected component.

Python Solution

```python
from collections import defaultdict

class Solution:
    def countCompleteComponents(self, n: int, edges: List[List[int]]) -> int:
        # dfs function to traverse the graph and return the number
        # of nodes (vertex_count) and number of edges (edge_count) in the component.
        def dfs(vertex: int) -> (int, int):
            visited[vertex] = True
            vertex_count, edge_count = 1, len(graph[vertex])  # Initialize counts with current node counts
            for neighbor in graph[vertex]:
                if not visited[neighbor]:
                    additional_vertices, additional_edges = dfs(neighbor)
                    vertex_count += additional_vertices
                    edge_count += additional_edges
            return vertex_count, edge_count

        # build the graph from the edges list
        graph = defaultdict(list)
        for a, b in edges:
            graph[a].append(b)
            graph[b].append(a)

        visited = [False] * n  # keep track of visited nodes
        complete_components_count = 0  # counter for complete components

        # check each node; if it's not visited, perform dfs from that node
        for i in range(n):
            if not visited[i]:
                vertex_count, edge_count = dfs(i)
                # In a complete graph AKA clique, the number of edges is vertex_count * (vertex_count - 1) / 2
                # We multiply by 2 to compare with the undirected edge count (each edge counted twice)
                if vertex_count * (vertex_count - 1) == edge_count:
                    complete_components_count += 1

        return complete_components_count

```

Java Solution

```java
class Solution {
    // Graph represented as an adjacency list and a visited array.
    private List<Integer>[] graph;
    private boolean[] visited;

    // Method to count the number of complete components in the graph.
    public int countCompleteComponents(int n, int[][] edges) {
        // Initialize the graph with empty lists for each node.
        graph = new List[n];
        visited = new boolean[n];
        Arrays.setAll(graph, k -> new ArrayList<>());

        // Populate the adjacency list with the edges.
        for (int[] edge : edges) {
            int a = edge[0], b = edge[1];
            graph[a].add(b);
            graph[b].add(a);
        }

        int count = 0;  // Counter for complete components.

        // Traverse each node to check if it forms a complete component.
        for (int i = 0; i < n; ++i) {
            // If the node has not been visited, perform DFS.
            if (!visited[i]) {
                int[] result = dfs(i);
                // A complete component has (n*(n-1))/2 edges. Here, result[0] is the number of nodes (n)
                // and result[1] is the number of edges in this component.
                if (result[0] * (result[0] - 1) == result[1]) {
                    count++;
                }
            }
        }
        return count; // Return the total count of complete components.
    }

    // Depth First Search (DFS) helper method to compute the total number of nodes and edges in a component.
    private int[] dfs(int index) {
        visited[index] = true;
        int nodesCount = 1;  // Start with one node, the one we are currently at.
        int edgesCount = graph[index].size();  // Counts edges connected to the current node.

        // Recursively visit all the neighbors that have not been visited.
        for (int neighbor : graph[index]) {
            if (!visited[neighbor]) {
                int[] result = dfs(neighbor);
                // Increment the count of nodes and edges with the counts from the neighbor.
                nodesCount += result[0];
                edgesCount += result[1];
            }
        }

        // Return the total count of nodes and edges in this component.
        return new int[] {nodesCount, edgesCount};
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <cstring>
#include <functional>

using namespace std;

class Solution {
public:
    int countCompleteComponents(int n, vector<vector<int>>& edges) {
        vector<vector<int>> graph(n); // Using 'graph' for clarity.
        vector<bool> visited(n, false);// Using vector of bool for visited to replace the C-style array.

        // Building the undirected graph
        for (auto& edge : edges) {
            int a = edge[0], b = edge[1];
            graph[a].push_back(b);
            graph[b].push_back(a);
        }

        // Declaration of Depth-First Search (DFS) lambda function
        function<pair<int, int>(int)> dfs = [&](int node) -> pair<int, int> {
            visited[node] = true;
            int verticesCount = 1;  // 'x' has been replaced with 'verticesCount'
            int edgesCount = graph[node].size();  // 'y' has been replaced with 'edgesCount'

            // Recursive DFS calls
            for (int neighbor : graph[node]) {
                if (!visited[neighbor]) {
                    auto [subtreeVertices, subtreeEdges] = dfs(neighbor);
                    verticesCount += subtreeVertices;
                    edgesCount += subtreeEdges;
                }
            }
            return make_pair(verticesCount, edgesCount);
        };

        int completeComponents = 0;  // Using 'completeComponents' instead of 'ans' for clarity

        // Checking each connected component of the graph
        for (int i = 0; i < n; ++i) {
            if (!visited[i]) {
                auto [componentVertices, componentEdges] = dfs(i);
                // Check if the connected component is a complete graph
                // A complete graph with 'n' vertices has 'n*(n-1)/2' edges
                if (componentVertices * (componentVertices - 1) == componentEdges) {
                    completeComponents++;
                }
            }
        }
        return completeComponents;  // Return the count of complete components
    }
};

```

Typescript Solution

```typescript
type Graph = number[][];
let graph: Graph;
let visited: boolean[];

// Builds the undirected graph from the edges
const buildGraph = (n: number, edges: number[][]): void => {
  graph = Array.from({ length: n }, () => []);
  visited = new Array(n).fill(false);

  edges.forEach(edge => {
    const [a, b] = edge;
    graph[a].push(b);
    graph[b].push(a);
  });
};

// Defines a Depth-First Search (DFS) function to explore the graph
const depthFirstSearch = (node: number): [number, number] => {
  visited[node] = true;
  let verticesCount = 1;
  let edgesCount = graph[node].length;

  graph[node].forEach(neighbor => {
    if (!visited[neighbor]) {
      const [subtreeVertices, subtreeEdges] = depthFirstSearch(neighbor);
      verticesCount += subtreeVertices;
      edgesCount += subtreeEdges;
    }
  });

  return [verticesCount, edgesCount];
};

// Counts the number of complete components in the graph
const countCompleteComponents = (n: number, edges: number[][]): number => {
  buildGraph(n, edges);
  let completeComponents = 0;

  for (let i = 0; i < n; i++) {
    if (!visited[i]) {
      const [componentVertices, componentEdges] = depthFirstSearch(i);

      // Checks if the connected component is a complete graph
      // A complete graph with 'n' vertices has 'n*(n-1)/2' edges
      if (componentVertices * (componentVertices - 1) / 2 === componentEdges) {
        completeComponents++;
      }
    }
  }

  return completeComponents;
};

```

Time and Space Complexity

Time Complexity

The time complexity of the provided code is O(N + E), where `N` represents the number of nodes and `E` represents the number of edges in the graph. The reasoning behind this assessment is as follows:

- Iterating over all edges to create the graph `g` yields O(E), as it is directly proportional to the number of edges.

- The `dfs` function visits each node exactly once due to the guard condition `if not vis[i]:`, which prevents revisiting. Each call to `dfs` also iterates over all neighbors of the node, which results in a total of O(E) for all `dfs` calls across all nodes because each edge will be considered exactly twice (once from each incident node).

- The `for` loop over `n` nodes is O(N), as it will attempt a `dfs` from each unvisited node.

The combination of creating the adjacency list and the DFS traversal then constitutes O(N + E).

Space Complexity

The space complexity of the code is O(N + E) as well. This assessment considers the following components that utilize memory:

- The adjacency list `g` can potentially hold all the edges, which has a space requirement of O(E).

- The visited list `vis` has a space requirement of O(N), since it stores a boolean for each node.

- The recursion stack for DFS can also grow up to O(N) in the case of a deep or unbalanced tree.

- The `edges` list itself occupies O(E).

Taking all these factors into account, the upper bound on space utilized by the algorithm is O(N + E), which accounts for all nodes and edges stored and processed.