261. Graph Valid Tree

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

Problem Description

In this problem, we're given a graph that is composed of 'n' nodes which are labeled from 0 to n - 1. The graph also has a set of edges given in a list where each edge is represented by a pair of nodes like `[a, b]` that signifies a bidirectional (undirected) connection between node `a` and node `b`. Our main objective is to determine if the given graph constitutes a valid tree.

To understand what makes a valid tree, we should recall two essential properties of trees:

1. A tree must be connected, which means there should be some path between every pair of nodes.
2. A tree cannot have any cycles, meaning no path should loop back on itself within the graph.

Therefore, our task is to check for these properties in the given graph. We need to verify if there's exactly one path between any two nodes (confirming a lack of cycles) and that all the nodes are reachable from one another (confirming connectivity). If both conditions are met, we return `true`; otherwise, we return `false`.

Intuition

To determine if the set of edges forms a valid tree, the "Union Find" algorithm is an excellent choice. This algorithm is a classic approach used in graph theory to detect cycles and check for connectivity within a graph.

Here's why Union Find works for checking tree validity:

1. **Union Operation**: This is used to connect two nodes. If they are already connected, it means we're trying to add an extra connection to a connected pair, indicating the presence of a cycle.
2. **Find Operation**: This operation helps in finding the root of each node, generally used to check if two nodes have the same root. If they do, a cycle is present since they are already connected through a common ancestor. If not, we can perform an union operation without creating a cycle.
3. **Path Compression**: This optimization helps in flattening the structure of the tree, which improves the time complexity of subsequent "Find" operations.

In our solution, we start with each node being its own parent (representing a unique set). Then, we iterate through each edge, applying the "Find" operation to see if any two connected nodes share the same root:

- If they do, we've detected a cycle and return `false` as a cycle indicates it's not a valid tree.
- If they don't, we connect (union) them by updating the root of one node to be the root of the other.

As we connect nodes, we also decrement 'n' as an additional check. If at the end of processing all edges, there's more than one disconnected component, 'n' will be greater than 1, indicating the graph is not fully connected, thus not a valid tree. If 'n' is exactly 1, it means the graph is fully connected without cycles, so it's a valid tree and we return `true`.

Solution Approach

The solution provided leverages the Union Find algorithm to check the validity of a tree. Here's a step-by-step walkthrough of the algorithm as implemented in the Python code:

1. **Initialization**:
   - We start by creating an array `p` to represent the parent of each node. Initially, each node is its own parent, thus `p[i] = i` for `i` from `0` to `n-1`.
   - The variable `n` tracks the number of distinct sets or trees; initially, each node is separate, so there are `n` trees.
2. **Function Definition - `find(x)`**:
   - This function is critical to Union Find. Its purpose is to find the root parent of a node `x`.
   - The function is implemented with path compression, meaning every time we find the root of a node, we update the parent along the search path directly to the root. This helps reduce the tree height and optimize future searches.
3. **Process Edges**:
   - We iterate over each edge in the list `edges`.
   - For each edge `[a, b]`, we find the roots of both nodes `a` and `b` using the `find` function.
   - We check if the roots are equal. If they are, `find(a) == find(b)`, this indicates that nodes `a` and `b` are already connected, thus forming a cycle. In this case, we immediately return `False`, as a valid tree cannot contain cycles.
   - If the roots are different, it means that connecting them doesn't form a cycle, so we perform a union operation by setting the parent of `find(a)` to `find(b)`.
   - After successfully adding an edge without forming a cycle, we decrement `n` since we have merged two separate components into one.
4. **Final Verification**:
   - After processing all edges, we check if `n` is exactly `1`. If it is, it means all nodes are connected, forming a single connected component without cycles, which satisfies the definition of a tree.
   - If `n` is not `1`, it means the graph is either not fully connected, or we have returned `False` earlier due to a cycle. In either case, the graph does not form a valid tree.

The simplicity of this algorithm comes from the elegant use of the Union Find pattern to quickly and efficiently find cycles and check connectivity. The solution runs in near-linear time, making it very efficient for large graphs.

Example Walkthrough

Let's consider a simple example to illustrate the solution approach using the Union Find algorithm.

Suppose we have a graph with `n = 4` nodes labeled from `0` to `3`, and we're given an edge list `edges = [[0, 1], [1, 2], [2, 3]]`.

Step 1: Initialization

- We begin by initializing the parent array `p` with `p = [0, 1, 2, 3]`.

Step 2: Function Definition - `find(x)`

- We define the `find` function to find the root parent of a given node `x`. Additionally, this function uses path compression.

Step 3: Process Edges

- We iterate through each edge `[a, b]` in the given `edges` list:
  - The first edge is `[0, 1]`. We find the roots of `0` and `1`, which are `0` and `1`, respectively. Since they have different roots, we perform the union operation by setting `p[1]` to the root of `0` (which is `0`). Now `p = [0, 0, 2, 3]`, and we decrement `n` to `3`.
  - The next edge is `[1, 2]`. The roots of `1` and `2` are `0` and `2`. They have different roots, so we union them by setting `p[2]` to the root of `1` (which is `0`). Now `p = [0, 0, 0, 3]`, and we decrement `n` to `2`.
  - The final edge is `[2, 3]`. The roots of `2` and `3` are `0` and `3`. Again, different roots allow us to union them, updating `p[3]` to `0`. Our parent array is now `p = [0, 0, 0, 0]`, and `n` is decremented to `1`.

Step 4: Final Verification

- After iterating over all the edges, we now check if `n` equals `1`. Since that's the case in our example, it means that all nodes are connected in a single component, and since we didn't encounter any cycles during the union operations, the graph forms a valid tree.

So, for this input graph represented by `n = 4` and `edges = [[0, 1], [1, 2], [2, 3]]`, our algorithm would return `True` indicating that the graph is a valid tree.

Python Solution

```python
from typing import List

class Solution:
    def validTree(self, num_nodes: int, edges: List[List[int]]) -> bool:
        # Helper function to find the root of a node 'x'.
        # Uses path compression to flatten the structure for faster future lookups.
        def find_root(node):
            if parent[node] != node:
                parent[node] = find_root(parent[node])  # Path compression
            return parent[node]

        # Initialize the parent list where each node is initially its own parent.
        parent = list(range(num_nodes))

        # Iterate over all the edges in the graph.
        for node_1, node_2 in edges:
            # Find the root of the two nodes.
            root_1 = find_root(node_1)
            root_2 = find_root(node_2)

            # If the roots are the same, it means we encountered a cycle.
            if root_1 == root_2:
                return False

            # Union the sets - attach the root of one component to the other.
            parent[root_1] = root_2

            # Each time we connect two components, reduce the total number of components by one.
            num_nodes -= 1

        # A tree should have exactly one more node than it has edges.
        # After union operations, we should have exactly one component left.
        return num_nodes == 1

```

Java Solution

```java
class Solution {

    private int[] parent; // The array to track the parent of each node

    // Method to determine if the input represents a valid tree
    public boolean validTree(int n, int[][] edges) {
        parent = new int[n]; // Initialize the parent array

        // Set each node's parent to itself
        for (int i = 0; i < n; ++i) {
            parent[i] = i;
        }

        // Loop through all edges
        for (int[] edge : edges) {
            int nodeA = edge[0];
            int nodeB = edge[1];

            // If both nodes have the same root, there's a cycle, and it's not a valid tree
            if (find(nodeA) == find(nodeB)) {
                return false;
            }

            // Union the sets to which both nodes belong by updating the parent
            parent[find(nodeA)] = find(nodeB);

            // Decrement the number of trees - we are combining two trees into one
            --n;
        }

        // If there's exactly one tree left, the structure is a valid tree
        return n == 1;
    }

    // Method to find the root (using path compression) of the set to which x belongs
    private int find(int x) {
        // If x is not the parent of itself, recursively find the root parent and apply path compression
        if (parent[x] != x) {
            parent[x] = find(parent[x]);
        }

        return parent[x]; // Return the root parent of x
    }
}

```

C++ Solution

```cpp
#include <vector>
using namespace std;

class Solution {
public:
    // Parent array representing the disjoint set forest
    vector<int> parent;

    // Function to check if given edges form a valid tree
    bool validTree(int n, vector<vector<int>>& edges) {
        // Initialize every node to be its own parent, forming n disjoint sets
        parent.resize(n);
        for (int i = 0; i < n; ++i) parent[i] = i;

        // Iterate through each edge
        for (auto& edge : edges) {
            int node1 = edge[0], node2 = edge[1];
            // Check if both nodes have the same root
            // If they do, a cycle is detected and it's not a valid tree
            if (find(node1) == find(node2)) return false;

            // Union the sets of the two nodes
            parent[find(node1)] = find(node2);

            // Decrement the count of trees because one edge connects two nodes in a single tree
            --n;
        }
        // For a valid tree, after connecting all nodes there should be exactly one set left (one tree)
        return n == 1;
    }

    // Helper function for finding the root of a node
    int find(int x) {
        // Path compression: Update the parent along the find path directly to the root
        if (parent[x] != x) parent[x] = find(parent[x]);
        return parent[x];
    }
};

```

Typescript Solution

```typescript
// Function to determine if a given undirected graph is a valid tree
/**
 * @param {number} numberOfNodes - the number of nodes in the graph
 * @param {number[][]} graphEdges - the edges of the graph
 * @return {boolean} - true if the graph is a valid tree, false otherwise
 */
const validTree = (numberOfNodes: number, graphEdges: number[][]): boolean => {
  // Parent array to track the root parent of each node
  let parent: number[] = new Array(numberOfNodes);

  // Initialize parent array so each node is its own parent initially
  for (let i = 0; i < numberOfNodes; ++i) {
    parent[i] = i;
  }

  // Helper function to find the root parent of a node
  const findRootParent = (node: number): number => {
    if (parent[node] !== node) {
      // Path compression: make the found root parent the direct parent of 'node'
      parent[node] = findRootParent(parent[node]);
    }
    return parent[node];
  };

  // Explore each edge to check for cycles and connect components
  for (const [nodeA, nodeB] of graphEdges) {
    // If two nodes have the same root parent, a cycle is detected
    if (findRootParent(nodeA) === findRootParent(nodeB)) {
      return false;
    }
    // Union operation: connect the components by making them share the same root parent
    parent[findRootParent(nodeA)] = findRootParent(nodeB);
    // Decrement the number of components by 1 for each successful union
    --numberOfNodes;
  }

  // A valid tree must have exactly one connected component with no cycles
  return numberOfNodes === 1;
};

// Example usage:
console.log(validTree(5, [[0, 1], [1, 2], [2, 3], [1, 4]])); // Outputs: true

```

Time and Space Complexity

The given Python code implements a union-find algorithm to determine if an undirected graph forms a valid tree. It uses path compression to optimize the find operation.

Time Complexity:

The time complexity of the `validTree` function mainly depends on the two operations: `find` and `union`.

- The `find` function has a time complexity of $O(\alpha(n))$ per call, where $\alpha(n)$ represents the Inverse Ackermann function which grows very slowly. In practical scenarios, $\alpha(n)$ is less than 5 for all reasonable values of n.

- Each edge leads to a single call to the `find` function during processing, and possibly a union operation if the vertices belong to different components. Thus, with m edges, there would be 2m calls to `find`, taking $O(\alpha(n))$ each.

- Also, the loop through the edges happens exactly m times, where m is the number of edges.

Bringing it all together, the overall time complexity is $O(m\alpha(n))$.

However, since the graph must have exactly n - 1 edges to form a tree (where n is the number of nodes), m can be replaced by n - 1. Therefore, the time complexity simplifies to $O((n - 1)\alpha(n))$ which is also written as $O(n\alpha(n))$, since the -1 is negligible compared to n for large n.

Space Complexity:

The space complexity is determined by the storage required for the parent array `p`, which has length n.

- The space taken by the parent array is O(n).

- Aside from the parent array `p` and the input `edges`, only a fixed amount of space is used for variables like `a`, `b`, and `x`.

Thus, the space complexity of the algorithm is O(n).