1319. Number of Operations to Make Network Connected

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

Problem Description

In this LeetCode problem, we are given a network of `n` computers, indexed from `0` to `n-1`. The connections between these computers are represented as a list of pairs, where each pair `[ai, bi]` indicates a direct ethernet cable connection between the computers `ai` and `bi`. Even though direct connections exist only between certain pairs, every computer can be reached from any other computer through a series of connections, forming a network.

Our task is to make all computers directly or indirectly connected with the least number of operations. An operation consists of disconnecting an ethernet cable between two directly connected computers and reconnecting it to a pair of disconnected computers.

We seek to find the minimum number of such operations needed to connect the entire network. If it is impossible to connect all computers due to an insufficient number of cables, the function should return `-1`.

Intuition

The solution revolves around the concept of the Union-Find algorithm, a common data structure used for disjoint-set operations. The core idea is to establish a way to check quickly if two elements are in the same subset and to unify two subsets into one.

Here's an intuitive breakdown of the approach:

1. **Initialization**: We start by assuming each computer forms its own single-computer network. This is represented by an array where each index represents a computer and contains its "parent." Initially, all computers are their own parents, forming `n` independent sets.
2. **Counting Redundant Connections**: As we iterate through the list of connections, we use the Union-Find with path compression to determine if two computers are already connected. If they are in the same subset, this connection is redundant. We count redundant connections because they represent extra cables that we can use to connect other more significant parts of the network.
3. **Unifying Sets**: For each non-redundant connection, we unify the sets that the two computers belong to. This action reduces the number of independent sets by one since we are connecting two previously separate networks.
4. **Determine Minimum Operations**: After all direct connections are processed, the number of independent sets remaining (`n`) minus one (since `n-1` connections are required to connect `n` computers in a single network) gives us the number of operations needed to connect the entire network.
5. **Check for Possibility**: If we have more independent networks than redundant connections (extra cables) remaining, it's impossible to connect all computers, and we return `-1`.
6. **Result**: If it's possible to connect all computers, we return the number of operations needed (`n-1`), which corresponds to the number of extra connections required to connect all disjoint networks.

By using Union-Find, we efficiently manage the merging of networks and avoid unnecessary complexity, leading us to the optimal number of operations required to connect the entire network.

Solution Approach

The solution for connecting all computers with a minimum amount of operations applies the Union-Find algorithm, which includes methods to find the set to which an element belongs, and to unite two sets. Here's a step-by-step walkthrough of how the implementation works:

1. **Creating a `find` function**: This function is recursive and is used to find the root parent of a computer. If the computer is its own parent, we return its value; otherwise, we recursively find the parent, applying path compression along the way by setting `p[x]` to the root parent. Path compression flattens the structure, reducing the time complexity of subsequent `find` operations.

```python
def find(x):
    if p[x] != x:
        p[x] = find(p[x])
    return p[x]
```

2. **Initializing the Parent List**: A list `p` of range `n` is initialized, signifying that each computer is initially its own parent, forming `n` separate sets.

```python
p = list(range(n))
```

3. **Iterating Over Connections**: For each connection `[a, b]` in `connections`, the algorithm checks whether `a` and `b` belong to the same set by comparing the root parent found by `find(a)` and `find(b)`.

```python
for a, b in connections:
    if find(a) == find(b):
        cnt += 1
    else:
        p[find(a)] = find(b)
        n -= 1
```

   - If they are in the same set, this connection is redundant (extra), and a counter `cnt` is incremented.
   - If they are not, we unite the sets by making the root parent of `a` the parent of the root parent of `b`, effectively merging the two sets. We also decrement the count of separate networks (`n`) by 1.

4. **Calculating Minimum Operations**: Once all pairs are processed, the algorithm checks if there are enough spare (redundant) connections to connect the remaining separate networks. This is done by comparing `n - 1` (the minimum number of extra connections required) with `cnt` (the number of spare connections). If `n - 1` is greater than `cnt`, it's impossible to connect all networks, and the function returns `-1`. Otherwise, it returns `n - 1` as the number of operations required.

```python
return -1 if n - 1 > cnt else n - 1
```

The underlying logic of Union-Find with path compression ensures that the time complexity of the `find` operation is almost constant (amortized `O(alpha(n))`), where `alpha` is the inverse Ackermann function, which grows extremely slowly and is less than 5 for all practical values of `n`. This property makes our solution highly efficient for the given problem.

Example Walkthrough

Let's assume we have a network with `n = 4` computers and the following list of ethernet cable connections (`connections`): `[[0, 1], [1, 2], [1, 3], [2, 3]]`. We want to find the minimum number of operations to connect the network.

1. **Creating the Parent List**: We initialize a list `p = [0, 1, 2, 3]`, which signifies that each computer is initially its own parent.
2. **Iterating Over Connections**: We process each pair in `connections` and track redundant connections with a counter `cnt = 0`.
   - For `[0, 1]`, `find(0)` returns `0` and `find(1)` returns `1`. Since they have different parents, we connect them by setting `p[0]` as the parent of `p[1]` (or vice versa). Now `p = [0, 0, 2, 3]`, and `n = 4 - 1 = 3`.
   - For `[1, 2]`, `find(1)` returns `0` and `find(2)` returns `2`. They are not in the same set, so we connect them, updating `p[2]` to `0` and `n` becomes `2`. Now `p = [0, 0, 0, 3]`.
   - For `[1, 3]`, `find(1)` gives `0` and `find(3)` gives `3`. They belong to different sets, so we merge them, setting `p[3]` to `0` and `n` becomes `1`. Now `p = [0, 0, 0, 0]`.
   - Finally, for `[2, 3]`, `find(2)` and `find(3)` both return `0`, indicating they are in the same set. This means the connection is redundant, we increment `cnt` to `1`.
3. **Calculating Minimum Operations**: We have `n - 1 = 1 - 1 = 0` minimum extra connections required to connect all computers and `cnt = 1` redundant connections.

   Since we already have one redundant connection and we need zero connections to unify the network (all computers have been connected through previous steps), we can perform `0` operations to connect the network.

Therefore, our function would return `0` for this example.

The example illustrates that by using Union-Find, we were able to merge disjoint sets efficiently, track redundant connections, and detect that no further operations were needed to fully connect the network.

Python Solution

```python
from typing import List

class Solution:
    def makeConnected(self, num_computers: int, connections: List[List[int]]) -> int:
        # Helper function to find the root of a given computer using path compression
        def find_root(computer):
            if parent[computer] != computer:
                parent[computer] = find_root(parent[computer])
            return parent[computer]

        # Initialization of a counter for spare cables (redundant connections)
        spare_cables = 0

        # Initialize the parent array where each computer is its own parent initially
        parent = list(range(num_computers))

        # Loop through each connection to unite the sets of computers
        for comp_a, comp_b in connections:
            root_a = find_root(comp_a)
            root_b = find_root(comp_b)

            # If the computers have the same root, we have found a redundant connection
            if root_a == root_b:
                spare_cables += 1
            else:
                # Union operation, setting the parent of one root to be the other
                parent[root_a] = root_b
                # Decrease the number of separate networks
                num_computers -= 1

        # If the number of available spare cables (redundant connections)
        # is at least the number of operations needed to connect all computers
        # we return the number of operations required (num_computers - 1)
        # Otherwise, we do not have enough cables and return -1
        return -1 if num_computers - 1 > spare_cables else num_computers - 1

# Example usage:
# sol = Solution()
# result = sol.makeConnected(4, [[0,1],[0,2],[1,2]])
# print(result)  # Output should be 1

```

Java Solution

```java
class Solution {
    // `parent` array where `parent[i]` represents the leader or parent of the ith node.
    private int[] parent;

    // Method to connect all computers using the minimum number of extra wires, if possible.
    public int makeConnected(int n, int[][] connections) {
        // Initialize the parent array to indicate each node is a leader of itself at the start.
        parent = new int[n];
        for (int i = 0; i < n; ++i) {
            parent[i] = i;
        }

        // `redundantConnections` counts the number of redundant connections.
        int redundantConnections = 0;

        // Loop through the given connections to unite the computers.
        for (int[] connection : connections) {
            int computer1 = connection[0];
            int computer2 = connection[1];

            // If two computers have the same leader, the connection is redundant.
            if (find(computer1) == find(computer2)) {
                ++redundantConnections;
            } else {
                // Union operation: join two sets; set leader of `computer1`'s set to `computer2`'s leader.
                parent[find(computer1)] = find(computer2);
                // Decrease the number of sets (or components) as two sets have been merged into one.
                --n;
            }
        }

        // To connect all computers, we need at least `n - 1` connections.
        // If `n - 1` is greater than `redundantConnections`, we can't make all computers connected.
        // Otherwise, return `n - 1` connections needed to make all computers connected.
        return n - 1 > redundantConnections ? -1 : n - 1;
    }

    // Method to recursively find the leader of the given node `x`.
    private int find(int x) {
        // Path compression: direct attachment of `x` to its leader to flatten the tree.
        if (parent[x] != x) {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    vector<int> parent; // Array to represent the parent of each node, initializing DSU structure.

    // The function to calculate the minimum number of connections to make all computers connected.
    int makeConnected(int n, vector<vector<int>>& connections) {
        parent.resize(n); // Assign the size of the parent vector based on the number of computers.
        for (int i = 0; i < n; ++i) parent[i] = i; // Initially, set each computer's parent to itself.
        int extraConnections = 0; // A counter to keep track of extra connections (redundant edges).

        // Loop through the list of connections
        for (auto& edge : connections) {
            int compA = edge[0], compB = edge[1]; // Computers A and B are connected.
            if (find(compA) == find(compB)) {
                // If they already have the same root, the connection is extra.
                ++extraConnections;
            } else {
                // Union operation: connect these two components
                parent[find(compA)] = find(compB);
                --n; // Decrement the count of disconnected components.
            }
        }
        // The minimum number of connections required is (number of components - 1).
        // If we don't have enough extra connections to connect all components, return -1.
        return n - 1 > extraConnections ? -1 : n - 1;
    }

    // The function to find the root of the component to which 'x' belongs.
    int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]); // Path compression: make all nodes in the path point to the root.
        }
        return parent[x];
    }
};

```

Typescript Solution

```typescript
const parent: number[] = []; // Array to represent the parent of each node in the DSU structure.

// The function to find the representative of the set that 'x' belongs to.
function find(x: number): number {
  if (parent[x] !== x) {
    parent[x] = find(parent[x]); // Path compression: make all nodes in the path point to the root.
  }
  return parent[x];
}

// The function to calculate the minimum number of connections to make all computers connected.
function makeConnected(n: number, connections: number[][]): number {
  for (let i = 0; i < n; ++i) parent[i] = i; // Initially, set each computer's parent to itself.
  let extraConnections = 0; // A counter to keep track of extra (redundant) connections.

  // Loop through the list of connections.
  for (const edge of connections) {
    const compA = edge[0], compB = edge[1]; // Computers A and B are connected.
    if (find(compA) === find(compB)) {
      // If they already have the same root, the connection is extra.
      ++extraConnections;
    } else {
      // Union operation: connect these two components.
      parent[find(compA)] = find(compB);
      --n; // Decrement the count of disconnected components.
    }
  }
  // The minimum number of connections required is (number of components - 1).
  // If there are not enough extra connections to connect all components, return -1.
  return (n - 1) > extraConnections ? -1 : n - 1;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code mainly depends on the number of connections and the union-find operations performed on them. In the best-case scenario, where all connections are already in their own set, the `find` operation takes constant time, thus the time complexity would be `O(C)`, where `C` is the number of connections. However, in the average and worst case, where the `find` function may traverse up to `n` nodes (with path compression, it would be less), the time complexity of each `find` operation can be approximated to `O(log*n)`, which is the inverse Ackermann function, very slowly growing and often considered almost constant. Therefore, for `C` connections, the average time complexity is `O(C * log*n)`.

The final check for whether the number of extra connections is sufficient to connect all components has a constant time complexity of `O(1)`.

So, the overall time complexity of the algorithm is `O(C * log*n)`.

Space Complexity

The space complexity is determined by the space needed to store the parent array `p`, which contains `n` elements. Hence, the space complexity is `O(n)` for the disjoint set data structure.

Additionally, no other auxiliary space that scales with the problem size is used, so the total space complexity remains `O(n)`.