886. Possible Bipartition

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

Problem Description

The problem presents a scenario where we need to split n people, each labeled uniquely from 1 to n, into two separate groups. These groups could be of any size, which implies that even a single individual in a group is allowable. The key constraint is that some pairs of individuals do not get along, denoted by the `dislikes` array. Each element of the `dislikes` array is a subarray of two integers where `a_i, b_i` signifies that persons `a_i` and `b_i` cannot be in the same group, and they should not end up in the same group.

The task is to determine if it's possible to successfully divide all the people into two such groups where none of the individuals in the same group dislike each other.

Intuition

To intuitively approach the problem, we can consider the 'dislike' relationships as edges in a graph with the people as vertices. If we can color all vertices of this graph using only two colors in such a way that no two adjacent vertices (disliking each other) have the same color, then we can successfully divide the people into two groups—where each color represents a separate group.

The problem thus can be related to a graph theory concept known as bipartite checking. A graph is bipartite if we can split its vertices into two groups such that no edges exist between vertices of the same group.

The solution adopts the union-find algorithm to solve this problem efficiently. Union-find is a data structure that keeps track of elements divided into sets and is capable of efficiently combining these sets and finding representatives of these sets. It can help in the quick determination of whether two elements belong to the same set or not — crucial for us to understand if connecting two individuals (via their dislikes) would create a conflict within a group.

The provided Python solution follows these steps:

1. Initialize each person's parent as themselves, creating 'n' separate sets.
2. Iteratively traverse through the `dislikes` list:
   - For each dislike relationship, find the parents (representatives) of the two individuals.
   - If both individuals already have the same parent, this means they're supposed to be in the same group, indicating a conflict, and thus, bipartition isn't possible.
   - Otherwise, union the sets by assigning one individual's parent as the other individual's dislikes' parent, effectively grouping them in the opposite groups.
3. If we pass through the dislikes list without encountering any conflicts, a bipartition is possible, and we return `True`.

The union-find algorithm provides an efficient way to dynamically group the individuals while keeping track of their relationships and allows us to assess the possibility of a bipartition.

Solution Approach

The solution approach relies on the union-find data structure and algorithm to solve the problem. Here is a step-by-step walkthrough of the implementation provided in the reference solution:

1. **Initialize Union-Find Structure**: We start by initializing an array `p = list(range(n))` where each person is initially set as their own parent. This setup helps us track which group each person belongs to as we progress through the dislikes.
2. **Graph Construction**: Using a dictionary `g`, we map each person to a list of people they dislike (building an adjacency list representation of our graph). The code `for a, b in dislikes: a, b = b - 1, a - 1; g[a].append(b); g[b].append(a)` populates this dictionary. Note that we use `a - 1` and `b - 1` because the people are labeled from 1 to n, but Python lists are 0-indexed.
3. **Union-Find Algorithm**: The core logic lies in the following part of the code:
   - We iterate over each person using `for i in range(n)` and then over the list of people each person dislikes using `for j in g[i]:`.
   - We perform a **find** operation for both the current person `i` and the people they dislike `j` by calling the `find()` function, which recursively locates the root parent of the given person's set, with path compression. If `find(i)` is equal to `find(j)`, this would mean attempting to add an edge between two vertices already part of the same group, which isn't allowed; thus, we return `False`.
   - If there's no immediate conflict, we perform a **union** operation. We union the set containing the disliked individual `j` with the next person disliked by `i` (`g[i][0]`), ensuring that 'dislikers' and 'dislikees' end up in different sets by having different roots. This is achieved by setting the parent of `find(j)` to the person that `i` dislikes first: `p[find(j)] = find(g[i][0])`.
4. **Conclusion**: After going through all dislikes pair without encountering a conflict, we can conclude that a bipartition is possible and return `True`.

This approach utilizes union-find's quick union and find operations to keep time complexity low, typically the near-constant time for each operation due to path compression, allowing efficient checks and unions while iterating through the dislikes array. Thus, the solution effectively groups individuals into two sets where disliked individuals are kept apart, which aligns with attempting to color a bipartite graph with two colors.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we have $n = 4$ people, and the `dislikes` array is given as `dislikes = [[1,2], [1,3], [2,4]]`. Here's how the union-find approach would work on this input:

1. **Initialize Union-Find Structure**: We start with a list `p = [0, 1, 2, 3]`. Each person is initially in their own separate group.
2. **Graph Construction**: Construct the dislike graph `g`:

```
g = {
  0: [1, 2], // Person 1 dislikes 2 and 3.
  1: [0, 3], // Person 2 dislikes 1 and 4.
  2: [0],    // Person 3 dislikes 1.
  3: [1]     // Person 4 dislikes 2.
}
```

   Note that we have adjusted for 0-indexing by subtracting 1 from each person's label.

3. **Union-Find Algorithm**: Now, we will proceed through each person and their dislikes:
   - Start with person `0` dislikes `[1, 2]`.
     - For disliking person `1`, since `find(0)` is `0` and `find(1)` is `1`, there's no conflict. Union them by setting the representative or parent of dislikee to the dislikee of person `0`, which is `p[find(1)] = find(0)` resulting in `p = [0, 0, 2, 3]`. This effectively assigns person `2` and `1` to different groups.
     - For disliking person `2`, `find(0)` is `0` and `find(2)` is `2`. No conflict. Union again: `p[find(2)] = find(0)` leading to `p = [0, 0, 0, 3]`.
   - Person `1` dislikes `[0, 3]`.
     - Disliking person `0` doesn't need a union as it's already handled.
     - For disliking person `3`, `find(1)` is `0` and `find(3)` is `3`. No conflict. Union performed by `p[find(3)] = find(1)`, leading to `p = [0, 0, 0, 0]`. However, given `find(1) = 0`, to union them correctly, we must set `p[find(3)] = find(0)` (person `1` dislikes first), but since `p[3]` is already `0` due to prior operations, no change occurs.
   - Person `2` and `3` do not need separate processing as their dislike relationships have already been handled.
4. **Conclusion**: No conflict was found during the pairwise checks, indicating that we can successfully divide the four people into two groups despite their dislikes. For this particular small example, everyone ended up with the same root parent, but in a larger, more complex graph, the union-find algorithm would maintain separate roots where appropriate to reflect the correct bipartition into two groups.

Python Solution

```python
from collections import defaultdict

class Solution:
    def possibleBipartition(self, n: int, dislikes: List[List[int]]) -> bool:
        # Helper function to find the root of the set that element `x` belongs to.
        def find_root(x):
            if parent[x] != x:
                # Path compression: make each looked-at node point to the root
                parent[x] = find_root(parent[x])
            return parent[x]

        # Graph representation where key is a node and values are the nodes that are disliked by the key
        graph = defaultdict(list)
        for a, b in dislikes:
            # Since people are numbered from 1 to N, normalize to 0 to N-1 for zero-indexed arrays
            a, b = a - 1, b - 1
            graph[a].append(b)  # Add b to the dislike list of the other
            graph[b].append(a)

        # Initialize the parent array for disjoint-set union-find, each node is its own parent initially
        parent = list(range(n))

        # Process every person
        for i in range(n):
            # Check dislikes for person 'i'
            for j in graph[i]:
                # If person 'i' and one of the people who dislike 'i' has the same root, partition is not possible
                if find_root(i) == find_root(j):
                    return False
                # Union operations: Connect the groups of the first disliked person to other disliked nodes' group
                parent[find_root(j)] = find_root(graph[i][0])

        # If no conflicts found, partitioning is possible
        return True
```

Java Solution

```java
class Solution {
    private int[] parent; // array to store the parent of each node

    // Function to check if it is possible to bipartition the graph based on dislikes
    public boolean possibleBipartition(int n, int[][] dislikes) {
        parent = new int[n+1];
        List<Integer>[] graph = new List[n+1]; // adjacency list to represent the graph
        Arrays.setAll(graph, k -> new ArrayList<>());
        for (int i = 0; i <= n; ++i) {
            parent[i] = i; // initially, each node is its own parent
        }
        for (var edge : dislikes) {
            int node1 = edge[0] - 1, node2 = edge[1] - 1; // adjusting index to be 0-based
            graph[node1].add(node2); // since the graph is undirected, add edge in both directions
            graph[node2].add(node1);
        }
        for (int i = 0; i <= n; ++i) {
            for (int adjacentNode : graph[i]) {
                if (findParent(i) == findParent(adjacentNode)) {
                    // If two adjacent nodes have the same set representative, bipartition isn't possible
                    return false;
                }
                // Union the sets of the first neighbor of i and the current neighbor (j)
                parent[findParent(adjacentNode)] = findParent(graph[i].get(0));
            }
        }
        return true; // If no conflicts are found, bipartition is possible
    }

    // Function to find the representative (root parent) of a node using path compression
    private int findParent(int node) {
        if (parent[node] != node) {
            parent[node] = findParent(parent[node]); // path compression for optimization
        }
        return parent[node];
    }
}
```

C++ Solution

```cpp
#include <vector>   // Required for using the vector type
#include <numeric>  // Required for using the iota function
#include <unordered_map> // Required for using unordered_map type
#include <functional>  // Required for using std::function

class Solution {
public:
    // Function to check if it is possible to bipartition the graph
    bool possibleBipartition(int n, std::vector<std::vector<int>>& dislikes) {
        // Initialize the parent vector and assign each node to be its own parent
        std::vector<int> parent(n);
        iota(parent.begin(), parent.end(), 0);

        // Create an adjacency list for the graph
        std::unordered_map<int, std::vector<int>> graph;
        for (const auto& edge : dislikes) {
            int a = edge[0] - 1; // Adjusting index to be zero based
            int b = edge[1] - 1; // Adjusting index to be zero based
            graph[a].push_back(b); // Add b to a's dislike list
            graph[b].push_back(a); // Add a to b's dislike list
        }

        // Define find function to find the set representative of a node
        std::function<int(int)> find = [&](int node) -> int {
            if (parent[node] != node) {
                // Path compression: collapse the find-path by setting the parent
                // of the intermediate nodes to the root node
                parent[node] = find(parent[node]);
            }
            return parent[node];
        };

        // Iterate through each node to check for conflict in the bipartition
        for (int i = 0; i < n; ++i) {
            // If the node has dislikes
            if (!graph[i].empty()) {
                int parentOfNode = find(i); // Find set representative of i
                int firstDislike = graph[i][0]; // Get first element in dislikes list
                for (int dislikeNode : graph[i]) {
                    // If the set representative of i equals that of a disliked node,
                    // the graph cannot be bipartitioned
                    if (find(dislikeNode) == parentOf) return false;

                    // Union operation: set the parent of the set representative of the
                    // current dislike node to be the set representative of the first dislike node
                    parent[find(dislikeNode)] = find(firstDislike);
                }
            }
        }

        // If no conflicts are found, the graph can be bipartitioned
        return true;
    }
};
```

Typescript Solution

```typescript
function possibleBipartition(n: number, dislikes: number[][]): boolean {
    // Array to store the group color assignment for each person, initialized with 0 (no color).
    const colors = new Array(n + 1).fill(0);

    // Adjacency list to represent the graph of dislikes.
    const graph = Array.from({ length: n + 1 }, () => []);

    // Helper function to perform depth-first search for coloring and checking the graph.
    // If a conflict is found, it returns true, indicating that the graph is not bipartite.
    const depthFirstSearch = (person: number, colorValue: number): boolean => {
        colors[person] = colorValue; // Assign color to the person.
        for (const neighbor of graph[person]) {
            // If neighbor has the same color or if neighbor is uncolored but a conflict is detected upon coloring
            if (colors[neighbor] === colors[person] || (colors[neighbor] === 0 && depthFirstSearch(neighbor, 3 - colorValue))) {
                return true;
            }
        }
        return false;
    };

    // Build the graph based on input dislike pairs.
    for (const [a, b] of dislikes) {
        graph[a].push(b);
        graph[b].push(a);
    }

    // Attempt to color the graph using two colors (1 and 2), starting from each uncolored person.
    // If a conflict arises during coloring, the graph cannot be bipartitioned, so return false.
    for (let i = 1; i <= n; ++i) {
        if (colors[i] === 0 && depthFirstSearch(i, 1)) {
            return false;
        }
    }

    // If no conflicts were found, the graph can be bipartitioned, so return true.
    return true;
}
```

Time and Space Complexity

The given Python code is designed to determine if it is possible to bipartition a graph such that no two nodes that dislike each other are in the same group. Here is the analysis of its time and space complexity:

Time Complexity:

- The program creates a graph `g` from the `dislikes` array. If there are `E` dislike pairs, this step has a time complexity of $O(E)$ because it iterates through the `dislikes` array once.
- The disjoint set function `find` has an amortized time complexity of $O(\alpha(N))$ per call, where α is the inverse Ackermann function and is nearly constant (very slowly growing) for all practical values of N. This function is called up to $2E$ times—twice for each edge.
- The outer `for` loop runs N times (once for each node), and the inner loop runs as many times as there are edges for a particular node. In the worst case, every node is connected to every other node, which would make this step take $O(N^2)$. However, due to path compression in the `find` function, the complexity of the union find operation across all iterations is $O(E * \alpha(N))$.
- Combining these factors, the overall time complexity of the algorithm is $O(E + E * \alpha(N))$ which simplifies to $O(E * \alpha(N))$ since $\alpha(N)$ is nearly constant and not large.

Space Complexity:

- There is a graph `g` which stores up to $2E$ edges in an adjacency list, so the space complexity for `g` is $O(E)$.
- The parent list `p` has a space complexity of $O(N)$ because it stores a representative for each of the N nodes.
- Ignoring the input `dislikes`, the auxiliary space used by the algorithm (the space exclusive of inputs) is $O(N + E)$.
- Therefore, the total space complexity of the function is $O(N + E)$ taking into account the space for the graph and the disjoint set.

In summary, the time complexity is $O(E * \alpha(N))$ and the space complexity is $O(N + E)$.