785. Is Graph Bipartite

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

Problem Description

In this problem, we are dealing with an undirected graph consisting of `n` nodes, labeled from `0` to `n - 1`. The graph is represented by a 2D array, where each entry `graph[u]` contains a list of nodes that are adjacent to node `u`. The graph has certain characteristics: no node is connected to itself, there are no multiple edges between the same set of nodes, the edges are bidirectional (if node `v` is in `graph[u]`, then node `u` will be in `graph[v]`), and the graph may not be fully connected.

The task is to determine whether the graph is bipartite. A graph is considered bipartite if we can split all the nodes into two distinct sets such that no two nodes within the same set are connected by an edge. In other words, each edge should connect a node from one set to a node in the other set.

Intuition

To determine if a graph is bipartite, one well-known approach is to try to color the graph using two colors in such a way that no two adjacent nodes have the same color. If you can successfully color the graph this way, it is bipartite. Otherwise, it is not.

This solution follows a depth-first search (DFS) approach. Starting from any uncolored node, we assign it a color (say color 1), then all of its neighbors get the opposite color (color 2), their neighbors get color 1, and so on. If at any point we find a conflict (i.e., we try to assign a node a color different from what it has been assigned already), we know that the graph cannot be bipartite.

The `dfs` function in the solution plays a crucial role in this process. It tries to color a node `u` with a given color `c` and then recursively tries to color all of the adjacent nodes with the opposite color, `3-c`, because if `c` is `1`, then `3-c` is `2`, and if `c` is `2`, then `3-c` is `1`. If it ever finds that it cannot color a node because it has already been colored with the same color as `u`, the function returns `False`.

The array `color` of size `n` keeps track of the colors of each node, with a `0` value meaning uncolored. The outer loop of the algorithm ensures that we start a DFS on each component of the graph since the graph may not be connected, and every node needs to be checked.

The algorithm returns `False` as soon as it finds a coloring conflict. If no conflict is found, it returns `True` after all nodes have been visited, indicating the graph is bipartite.

Solution Approach

The solution to determine if an undirected graph is bipartite involves a graph traversal algorithm, specifically Depth-First Search (DFS). The DFS is chosen here because it allows us to go as deep as possible through each branch before backtracking, which is suitable for the coloring problem where adjacent nodes need to be considered closely and immediately.

Here are the key steps of the implementation using DFS:

1. Initialize an array `color` with length equal to the number of nodes `n`. This array will track the color of each node. A value of `0` in this array signifies that the node has not been colored yet.
2. Start a loop from `i = 0` to `n - 1` to initiate a DFS on each node if it is not colored already. We need to ensure that disconnected parts of the graph are also considered, which is why a loop is required instead of a single DFS call.
3. For the DFS, define a helper function named `dfs` that will attempt to color a node `u` with a given color `c`. The function follows these steps:
    - Assign the color `c` to `color[u]`.
    - Iterate over all adjacent nodes `v` of `u`.
    - If the neighbor `v` is not yet colored (`color[v]` is `0`), recursively call `dfs(v, 3 - c)` to color `v` with the opposite color.
    - If the neighbor `v` is already colored with the same color as `u` (`color[v]` is `c`), then we have found a conflict indicating that the graph is not bipartite. Return `False` in this case.
4. If the DFS is able to color the component starting from node `i` without conflicts (the `dfs` calls all return `True`), the function continues to the next component by advancing in the outer loop.
5. If any call to `dfs` returns `False`, the main function `isBipartite` immediately returns `False` as well, as a conflict has been detected.
6. If the loop completes without finding a conflict, return `True`, signaling that the graph is bipartite, since it was possible to color the graph using two colors according to the bipartite rules.

To summarize, the solution structure consists of a DFS helper function encapsulated within the main function that manages the loop through all nodes and the color array. The algorithm relies on the properties of the DFS and a simple coloring heuristic to solve the bipartite check efficiently.

Example Walkthrough

Let's use a small example to illustrate the solution approach. Suppose we have the following undirected graph represented with `n = 4` nodes:

```
0 -- 1
|    |
3 -- 2
```

In graph representation, it would be:

- `graph[0]` contains `[1, 3]`
- `graph[1]` contains `[0, 2]`
- `graph[2]` contains `[1, 3]`
- `graph[3]` contains `[0, 2]`

Now let's walk through the steps:

1. Initialize the `color` array with length `4` (since we have `4` nodes). The array starts as `[0, 0, 0, 0]`.
2. Start a loop from `0` to `3`. We look at each node to determine if it needs to be colored.
3. When `i = 0`, the node is not colored. We call `dfs(0, 1)`. For clarity, `dfs(u, c)` means we're trying to color node `u` with color `c`.
    - `color[0]` becomes `1`.
    - We look at `graph[0]` which is `[1, 3]`.
    - For `1`, since `color[1]` is `0`, we call `dfs(1, 3 - 1)` which simplifies to `dfs(1, 2)`.
        - `color[1]` becomes `2`.
        - We look at `graph[1]` which is `[0, 2]`.
        - Node `0` is already colored with a different color, so there's no conflict.
        - For node `2`, since `color[2]` is `0`, we call `dfs(2, 3 - 2)` simplifying to `dfs(2, 1)`.
            - `color[2]` becomes `1`.
            - We look at `graph[2]` which is `[1, 3]`.
            - Node `1` is already colored with a different color, so there's no conflict.
            - For node `3`, since `color[3]` is `0`, we call `dfs(3, 3 - 1)` which is `dfs(3, 2)`.
                - `color[3]` becomes `2`.
                - We look at `graph[3]` which is `[0, 2]`.
                - Both nodes `0` and `2` are already colored with different colors, so there's no conflict.
    - The `dfs` call stack for node `0` completes successfully without conflicts.
4. As the outer loop continues, `i` advances but all nodes are already colored, so no further `dfs` calls are necessary.
5. Since none of the `dfs` calls returned `False`, the graph has been successfully colored with two colors without conflict—`color` array is `[1, 2, 1, 2]`.
6. The loop completes without finding a conflict, so the function would return `True`. This indicates that the graph is bipartite since it's possible to color the graph using two colors, following the rules.

Python Solution

```python
from typing import List

class Solution:
    def isBipartite(self, graph: List[List[int]]) -> bool:
        # Depth-First Search function to determine
        # if we can color the graph using two colors
        def dfs(node_index, color_value):
            # Assign the color to the current node
            node_colors[node_index] = color_value
            # Traverse all adjacent nodes (neighbors)
            for neighbor in graph[node_index]:
                # If the neighbor hasn't been colored, color it with the alternate color
                if node_colors[neighbor] == 0:
                    if not dfs(neighbor, 3 - color_value):  # 1 -> 2 or 2 -> 1
                        return False
                # If the neighbor is already colored with the same color, graph is not bipartite
                elif node_colors[neighbor] == color_value:
                    return False
            # If all neighbors are colored correctly, return True
            return True

        # Get the number of nodes in the graph
        num_nodes = len(graph)
        # Initialize the node_colors list to zero for all nodes; 0 means not yet colored
        node_colors = [0] * num_nodes
        # Iterate over all nodes
        for i in range(num_nodes):
            # If the node hasn't been colored, start DFS and try to color it with color 1
            if node_colors[i] == 0 and not dfs(i, 1):
                # If DFS returns False, the graph is not bipartite
                return False
        # If all nodes are colored without conflict, return True
        return True

```

Java Solution

```java
class Solution {
    // Array 'colors' will store the colors of nodes. If uncolored, it stores 0; otherwise 1 or 2.
    private int[] colors;
    // The adjacency list representation of the graph.
    private int[][] graph;

    /**
     * Function to check if a graph is bipartite or not.
     * A graph is bipartite if we can split its set of nodes into two independent subsets A and B
     * such that every edge in the graph connects a node in set A and a node in set B.
     *
     * @param graph The adjacency list representation of the graph.
     * @return true if the graph is bipartite, false otherwise.
     */
    public boolean isBipartite(int[][] graph) {
        int numNodes = graph.length; // Number of nodes in the graph.
        colors = new int[numNodes]; // Initialize the colors array.
        this.graph = graph; // Assign the graph to the instance variable.

        // Process every node.
        for (int node = 0; node < numNodes; ++node) {
            // If the node is not colored and the depth-first search (DFS) returns false,
            // then the graph is not bipartite.
            if (colors[node] == 0 && !depthFirstSearch(node, 1)) {
                return false;
            }
        }
        // If all nodes are successfully colored with DFS, the graph is bipartite.
        return true;
    }

    /**
     * Depth First Search (DFS) method to assign colors to the nodes of the graph.
     *
     * @param node The current node to color.
     * @param color The color to assign to the node. It can either be 1 or 2.
     * @return true if successful, false if there's a conflict in coloring.
     */
    private boolean depthFirstSearch(int node, int color) {
        // Color the current node.
        colors[node] = color;
        // Visit all adjacent nodes.
        for (int adjacent : graph[node]) {
            // If the adjacent node is not colored, color it with the opposite color.
            if (colors[adjacent] == 0) {
                if (!depthFirstSearch(adjacent, 3 - color)) { // 3 - color gives the opposite color.
                    return false;
                }
            } else if (colors[adjacent] == color) { // If the adjacent node has the same color, return false.
                return false;
            }
        }
        // All adjacent nodes can be colored with opposite color, return true.
        return true;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to check if the graph is bipartite.
    bool isBipartite(vector<vector<int>>& graph) {
        int numNodes = graph.size(); // Get the number of nodes in the graph.
        vector<int> colors(numNodes, 0); // Vector to store colors for each node, initialized to 0.

        // Iterate through each node in the graph.
        for (int node = 0; node < numNodes; ++node) {
            // If the node is uncolored and the DFS coloring fails, the graph is not bipartite.
            if (colors[node] == 0 && !dfsColorGraph(node, 1, colors, graph)) {
                return false;
            }
        }
        // All nodes have been successfully colored without conflicts, hence the graph is bipartite.
        return true;
    }

    // Helper function to perform DFS and color the graph.
    bool dfsColorGraph(int currentNode, int currentColor, vector<int>& colors, vector<vector<int>>& graph) {
        colors[currentNode] = currentColor; // Color the current node.

        // Iterate through all adjacent nodes of the current node.
        for (int adjacentNode : graph[currentNode]) {
            // If the adjacent node is uncolored, attempt to color it with the opposite color.
            if (colors[adjacentNode] == 0) {
                if (!dfsColorGraph(adjacentNode, 3 - currentColor, colors, graph)) {
                    return false; // If coloring fails, the graph is not bipartite.
                }
            }
            // If the adjacent node has the same color, the graph cannot be bipartite.
            else if (colors[adjacentNode] == currentColor) {
                return false;
            }
        }
        // All adjacent nodes can be colored with the opposite color.
        return true;
    }
};

```

Typescript Solution

```typescript
// Function to determine if a graph is bipartite
// A graph is bipartite if the nodes can be divided into two sets such that
// no two nodes in the same set are adjacent.
function isBipartite(graph: number[][]): boolean {
    // `n` stores the total number of nodes in the graph
    const n: number = graph.length;
    // `isValid` keeps track of whether the graph is bipartite
    let isValid: boolean = true;
    // `colors` array will store the colors assigned to each node,
    // where 0 means uncolored, 1 is the first color, and 2 is the second color
    let colors: number[] = new Array(n).fill(0);

    // Helper function to perform depth-first search and color the nodes
    function dfs(nodeIndex: number, color: number, graph: number[][]): void {
        // Color the current node
        colors[nodeIndex] = color;
        // Determine the color to be given to adjacent nodes (1 or 2)
        const nextColor: number = 3 - color;

        // Iterate over all adjacent nodes
        for (let adjacentNode of graph[nodeIndex]) {
            // If the adjacent node is not yet colored, color it with nextColor
            if (!colors[adjacentNode]) {
                dfs(adjacentNode, nextColor, graph);
                // If at any point, the graph is found not to be valid, exit early
                if (!isValid) return;
            } else if (colors[adjacentNode] !== nextColor) {
                // If the adjacent node has been colored with the wrong color,
                // the graph is not bipartite
                isValid = false;
                return;
            }
        }
    }

    // Iterate over each node, coloring them if not already colored,
    // while the graph remains valid
    for (let i = 0; i < n && isValid; i++) {
        if (!colors[i]) {
            // Start coloring nodes from the first color
            dfs(i, 1, graph);
        }
    }
    // Return whether the graph is bipartite
    return isValid;
}

```

Time and Space Complexity

The provided code defines a function `isBipartite` which checks if a given graph can be colored with two colors such that no two adjacent nodes have the same color, which is a characteristic of a bipartite graph.

Time Complexity

The time complexity of the code is `O(V + E)`, where `V` is the number of vertices in the graph, and `E` is the number of edges. This is because the function uses a Depth-First Search (DFS) to traverse the graph. Each node is visited exactly once, and for each node, all its adjacent nodes are explored. Visiting each node takes `O(V)` time, and exploring adjacent nodes (edges) takes `O(E)` time in total.

Space Complexity

The space complexity is also `O(V)`. The `color` array is used to store the color of each node and its size is proportional to the number of nodes (`n`), which is `V`. The space is also used by the call stack due to the recursive DFS calls, which in the worst case can go as deep as `V` levels, if the graph is a long path or a chain of nodes. Therefore the total space used by the algorithm is proportional to the number of nodes.