1743. Restore the Array From Adjacent Pairs

Medium Array Hash Table

Problem Description

In this problem, you have forgotten an integer array `nums` that consists of `n` unique elements. Despite not remembering the array itself, you do recall every pair of adjacent elements in `nums`. You are given a 2D integer array `adjacentPairs` with a size of `n - 1`. Each element `adjacentPairs[i] = [u_i, v_i]` indicates that the elements `u_i` and `v_i` are adjacent in the array `nums`.

It is guaranteed that every adjacent pair of elements `nums[i]` and `nums[i+1]` will be represented in `adjacentPairs`, either as `[nums[i], nums[i+1]]` or `[nums[i+1], nums[i]]`. These pairs can be listed in any order. Your task is to reconstruct and return the original array `nums`. If there are several possible arrays that fit the adjacent pairs, returning any one of them is acceptable.

Intuition

To restore the array, we need to recognize that the description constructs a path (like a one-dimensional graph) where each number except for the two endpoints has exactly two neighbors (think of nodes in a graph). The graph formed by `adjacentPairs` will have nodes with the following properties: the starting and ending elements (the ones found at the beginning and end of the array `nums`) will have only one adjacent element in `adjacentPairs` (they have a degree of one), while all other elements will have two adjacent elements (they have a degree of two). Identifying the unique starting or ending point allows us to sequentially rebuild the array from one end to the other.

Solution Approach

The `Solution` class in the provided Python code implements a function called `restoreArray` to solve the problem by reconstructing the original array from the given pairs of adjacent integers.

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

1. **Create a Graph Representation**: To keep track of the adjacent nodes (the integers that can come before or after a given integer in the array), we use a defaultdict of lists, named `g`. A defaultdict is used here because it conveniently allows appending to lists for keys that have not been explicitly set. For each `a, b` pair in `adjacentPairs`, we add `b` as an adjacent node to `a` and vice versa.

```
g = defaultdict(list)
for a, b in adjacentPairs:
    g[a].append(b)
    g[b].append(a)
```

2. **Identify an Endpoint**: Since we know the original array has unique elements with exactly two endpoints, we start by finding an integer that appears only once in the adjacency list (this has a graph degree of one) – meaning it's an endpoint.

```
for i, v in g.items():
    if len(v) == 1:
        ans[0] = i
        ans[1] = v[0]
        break
```

3. **Reconstruct the Array**: Once we have one endpoint, we initialize our answer array `ans` with the starting endpoint as the first element. We also know what follows it immediately given our graph.

   From there, we traverse the graph to rebuild the original array. We iterate from the second to the last element of the `ans` array and at each step, pull the adjacent vertices from our graph. Since each element has two neighboring elements (except the endpoints), we can easily determine the next element in the array by checking which of the two vertices is not the one we've just visited (i.e., not the second-to-last element in `ans`).

```
n = len(adjacentPairs) + 1
ans = [0] * n
# ... previous endpoint identification code ...
for i in range(2, n):
    v = g[ans[i - 1]]
    ans[i] = v[0] if v[1] == ans[i - 2] else v[1]
```

4. **Return the Result**: After the loop completes, `ans` contains the reconstructed array following the original order.

The nature of the problem aligns well with graph traversal algorithms – specifically, using a simple walk technique, as we are guaranteed a valid path that visits each node exactly once. The time complexity is O(N) because we visit each vertex exactly once, where N is the number of elements in the original array.

Example Walkthrough

To illustrate the solution approach, let's take a small example. Suppose we have the following pairs of adjacent elements from the problem statement:

`adjacentPairs = [[4,3], [1,2], [3,1]]`

From these pairs, we need to reconstruct the original integer array `nums`. Here's how we would apply the solution approach:

1. **Create a Graph Representation**: First, we create a graph by adding each pair to our adjacency list, representing each number and its corresponding adjacent numbers.

   We get the following graph:

```
{
  4: [3],
  3: [4, 1],
  1: [2, 3],
  2: [1]
}
```

2. **Identify an Endpoint**: We look for an integer that appears only once in the adjacency list, as this will be an endpoint. In our graph, both `4` and `2` appear only once, so either can be chosen as the starting endpoint.

   Assume we pick `4` as the starting point. Our answer array `ans` tentatively starts with:

```
ans = [4, _]
```

   Since `4` is connected to `3`, the next element is `3`:

```
ans = [4, 3, _]
```

3. **Reconstruct the Array**: From `3`, we look at its neighbors, which are `4` (which we've just visited) and `1`. So the next element is `1`, and our array now looks like:

```
ans = [4, 3, 1, _]
```

   Continuing this, the next number we add is the neighbor of `1` which isn't `3` (since we've already been there). That's `2`, resulting in:

```
ans = [4, 3, 1, 2]
```

   The array is now fully reconstructed.

4. **Return the Result**: Finally, the reconstructed array `ans = [4, 3, 1, 2]` is the original array, considering the given `adjacentPairs`. Therefore, the result of the `restoreArray` function with the given input would be `[4, 3, 1, 2]`.

In this example, the algorithm successfully reconstructs the original array from the pairs of adjacent elements. The use of graph representation simplifies the problem, making it more approachable and easier to solve efficiently.

Python Solution

```python
from collections import defaultdict

class Solution:
    def restoreArray(self, adjacentPairs):
        # Create a graph using a dictionary where each key has an associated list of its adjacent nodes
        graph = defaultdict(list)
        for pair in adjacentPairs:
            # Since it's an undirected graph, add each node to the list of its neighboring node
            graph[pair[0]].append(pair[1])
            graph[pair[1]].append(pair[0])

        # The original array would have a length of one more than the number of adjacent pairs
        array_length = len(adjacentPairs) + 1
        restored_array = [0] * array_length

        # Find the first and the second elements of the array, which are the endpoints of the path
        # (having only one adjacent node)
        for node, neighbors in graph.items():
            if len(neighbors) == 1:  # Endpoints will only have one neighbor
                restored_array[0] = node
                restored_array[1] = neighbors[0]
                break

        # Construct the rest of the array
        for i in range(2, array_length):
            # Find the next adjacent node that is not the previous element in the array
            current_neighbors = graph[restored_array[i - 1]]
            # If the current node has only one neighbor, it must be the next element; otherwise
            # it is the one that is not the previously used element
            restored_array[i] = current_neighbors[0] if current_neighbors[1] == restored_array[i - 2] else current_neighbors[1]

        # Return the reconstructed array
        return restored_array
```

Java Solution

```java
// Solution class to restore the array from its adjacent pairs
class Solution {
    // Method to restore array using adjacent pairs
    public int[] restoreArray(int[][] adjacentPairs) {
        // Calculate the total number of unique elements
        int n = adjacentPairs.length + 1;
        // Create a graph using a Map to store element and its adjacent elements
        Map<Integer, List<Integer>> graph = new HashMap<>();

        // Iterate over each adjacent pair
        for (int[] edge : adjacentPairs) {
            // Unpack the pair
            int a = edge[0], b = edge[1];
            // Build the adjacency list for each element in the pair
            graph.computeIfAbsent(a, k -> new ArrayList<>()).add(b);
            graph.computeIfAbsent(b, k -> new ArrayList<>()).add(a);
        }

        // Initialize the array to hold the restored sequence
        int[] restoredArray = new int[n];

        // Find the first element (head of array) which will be an element with only one neighbor
        for (Map.Entry<Integer, List<Integer>> entry : graph.entrySet()) {
            if (entry.getValue().size() == 1) {
                // Found the first element, which is the head of our restored array
                restoredArray[0] = entry.getKey();
                // The only neighbor is the second element
                restoredArray[1] = entry.getValue().get(0);
                break;
            }
        }

        // Reconstruct the array starting from index 2
        for (int i = 2; i < n; ++i) {
            // Get the neighbors of the last added element in restoredArray
            List<Integer> neighbors = graph.get(restoredArray[i - 1]);
            // Check the two neighbors to determine which is the next element in sequence
            // If one neighbor is the same as the previous element in restoredArray, choose the other one
            restoredArray[i] = neighbors.get(0) == restoredArray[i - 2] ? neighbors.get(1) : neighbors.get(0);
        }

        // Return the restored array
        return restoredArray;
    }
}
```

C++ Solution

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

class Solution {
public:
    vector<int> restoreArray(vector<vector<int>>& adjacent_pairs) {
        // Calculate the number of elements in the original array.
        int num_elements = adjacent_pairs.size() + 1;

        // Create a graph represented as an adjacency list.
        unordered_map<int, vector<int>> graph;
        for (auto& edge : adjacent_pairs) {
            int a = edge[0], b = edge[1];
            graph[a].push_back(b);
            graph[b].push_back(a);
        }

        // Initialize the answer array with the given size.
        vector<int> answer(num_elements);

        // Find the starting element which is the one with only one neighbor.
        for (auto& pair : graph) {
            if (pair.second.size() == 1) {
                answer[0] = pair.first;
                answer[1] = pair.second[0];
                break;
            }
        }

        // Iterate over each element in the answer to find the next element.
        for (int i = 2; i < num_elements; ++i) {
            // Access the neighbors of the last added element.
            auto neighbors = graph[answer[i - 1]];

            // The next element is the one which is not the previously added element.
            answer[i] = neighbors[0] == answer[i - 2] ? neighbors[1] : neighbors[0];
        }

        // Return the restored original array.
        return answer;
    }
};
```

Typescript Solution

```typescript
type AdjacencyList = Map<number, number[]>;

// Restore the array from the given adjacency pairs
function restoreArray(adjacentPairs: number[][]): number[] {
  // Calculate the number of elements in the original array
  const numElements: number = adjacentPairs.length + 1;

  // Create a graph represented as an adjacency list
  const graph: AdjacencyList = new Map<number, number[]>();
  for (const [a, b] of adjacentPairs) {
    if (!graph.has(a)) graph.set(a, []);
    if (!graph.has(b)) graph.set(b, []);
    graph.get(a)!.push(b);
    graph.get(b)!.push(a);
  }

  // Initialize the answer array with the given size
  const answer: number[] = new Array(numElements);

  // Find the starting element which is the one with only one neighbor
  for (const [key, neighbors] of graph) {
    if (neighbors.length === 1) {
      answer[0] = key;
      answer[1] = neighbors[0];
      break;
    }
  }

  // Iterate over each element in the answer to find the next element
  for (let i = 2; i < numElements; ++i) {
    // Access the neighbors of the last added element
    const neighbors = graph.get(answer[i - 1])!;

    // The next element is the one which is not the previously added element
    answer[i] = neighbors[0] === answer[i - 2] ? neighbors[1] : neighbors[0];
  }

  // Return the restored original array
  return answer;
}

// Example usage:
const adjacentPairs = [[2, 1], [3, 4], [3, 2]];
const restoredArray = restoreArray(adjacentPairs);
console.log(restoredArray);  // Output will be the restored array based on the given adjacent pairs
```

Time and Space Complexity

Time Complexity

The time complexity of the given code can be analyzed as follows:

- Building the graph `g` has a complexity of `O(m)`, where `m` is the number of adjacent pairs. Every adjacent pair requires two insert operations in the dictionary.
- Finding the starting node (where the degree is 1) has a maximum complexity of `O(n)` where `n` is the number of unique nodes. This is because in the worst case, we have to check every entry in the dictionary.
- Reconstructing the array `ans` requires us to iterate `n - 2` times (since the first two elements are already filled), and in each iteration, we find the next node in `O(1)` time (since we're dealing with at most two elements in a list, and we know one of them is the previous node). So, the complexity for this part is `O(n)`.

Overall, the time complexity is `O(m + n + n) => O(m + 2n)`. Since `m = n - 1`, we simplify the time complexity to `O(n)`.

Space Complexity

The space complexity of the given code can be analyzed as follows:

- The graph `g` will contain each unique node and its adjacent nodes. Since each edge contributes to two nodes' adjacency lists, the space needed for `g` is `O(2m)`.
- The `ans` array will contain `n` elements, so it requires `O(n)` space.

Together, the space complexity for storing the graph and the array amounts to `O(2m + n)`. Since `m = n - 1`, we can simplify the space complexity to `O(n)`.

Hence, both time and space complexities of the given code are `O(n)`.