1042. Flower Planting With No Adjacent

Medium Depth-First Search Breadth-First Search Graph

Leetcode Link

Problem Description

This problem involves `n` gardens, each needing to be planted with one of 4 types of flowers. The gardens are connected by `paths`, bidirectional paths, and each garden can have at most 3 paths connected to it. The crux of the problem is to choose the type of flower for each garden in such a way that no two gardens connected by a path have the same type of flower. The goal is to find any one configuration that satisfies this condition. The final output should be an array where the element at the `i`-th position refers to the type of flower in the `(i+1)`th garden. You're assured that there is at least one solution to this problem.

Intuition

Imagine each garden as a node in a graph and each path as an edge connecting two nodes. Since a garden can have at most 3 paths, a node can have a maximum of 3 edges. To solve this, we can iterate through each garden and choose for it the lowest numbered flower type which hasn't been used by its connected neighbors.

The intuition behind this approach is that since each garden can have at most 3 paths, there is always at least one type of flower (out of the 4 available types) that is not used by any neighbor garden. By iterating through each garden, we keep track of the types used by its neighbors and then assign the first available type.

The solution uses a graph represented as an adjacency list, where the key is the garden and the value is a list of connected gardens. As it iterates through each garden, it dynamically updates the assigned flower types and maintains a set of used flower types for the current garden's neighbors. By ensuring that no adjacent gardens share a flower type, we respect the constraint and construct a valid configuration for planting flowers.

Solution Approach

The implementation employs a graph-based approach and makes use of Python's built-in data structures:

1. **Default Dictionary for Adjacency List**: The graph is represented using a `defaultdict` from Python's `collections` module, which provides a convenient way to store the adjacency list. Every garden is keyed to a list of neighbor gardens that are connected by paths.

2. **Iterating Over Gardens**: We iterate sequentially over each garden. Here, the garden's index is treated as its label (decremented by 1 as indices in Python are 0-based, but gardens are labeled from 1 to `n`).

3. **Tracking Used Flower Types**: A set is created for each garden to keep track of the flower types used by its neighboring gardens.

4. **Assigning Flower Types**: For each garden, we loop over the possible flower types (1 through 4). We pick the first flower type that is not present in the set of used types and assign this flower type to the current garden.

5. **Updating the Answer**: After a valid flower type is chosen for a garden, it is saved to the `ans` list, which initially is filled with zeroes.

6. **Returning the Solution**: After all gardens have been assigned a flower type, we return the `ans` list, which now contains the type of flower for each garden satisfying the problem's constraints.

In more detail, for each garden `x`, we perform the following steps:

- Get the set `used` of flower types that have already been used by the neighbors of garden `x`.
- Iterate through the flower types `c` from 1 to 4, checking if `c` is not in the `used` set.
- When we find such a `c`, we assign it to `ans[x]` and break the inner loop since we've found a valid flower for this garden.

This greedy algorithm ensures that since no more than 3 gardens can be adjacent to a garden, there will always be at least one type of flower from the 4 available that we can plant in every garden, thus always providing a valid solution.

Example Walkthrough

Consider a scenario with five gardens and the following paths between them: `[1,2]`, `[2,3]`, `[3,4]`, `[4,5]`, `[1,5]`. This means garden 1 is connected to gardens 2 and 5, garden 2 is connected to gardens 1 and 3, and so on.

Let's apply the solution approach to this example:

- Start with an empty adjacency list and an `ans` list filled with zeroes, denoting that no flowers have been planted yet.
- Interpret the paths to fill the adjacency list, which in this case results in:

```
1 1 -> [2, 5]
2 2 -> [1, 3]
3 3 -> [2, 4]
4 4 -> [3, 5]
5 5 -> [4, 1]
```

- Begin iterating from garden 1 to 5.
- For garden 1, check the flower types of its neighbors (gardens 2 and 5 which have not been planted yet) and note that all flower types (1, 2, 3, 4) are available. Choose the lowest number which is 1, and assign it to the garden, so `ans[1] = 1`.
- Move to garden 2, which is connected to garden 1 (planted with flower type 1) and garden 3 (not yet planted). The available flower types are 2, 3, and 4. Pick the lowest, which is 2, so `ans[2] = 2`.
- For garden 3, connected to garden 2 (flower type 2) and garden 4 (not yet planted). Available types are 1, 3, and 4. Choose the lowest, so `ans[3] = 1`.
- Move to garden 4, which has neighbors 3 and 5. Garden 3 has flower type 1, and garden 5 is unplanted. The choices are 2, 3, or 4. Pick the lowest, so `ans[4] = 2`.
- Finally, garden 5 is connected to gardens 1 and 4, which have flower types 1 and 2, respectively. The remaining flowers are 3 and 4. So we pick 3, and `ans[5] = 3`.
- The flower assignments for each of the five gardens represented by the `ans` list are now `[1, 2, 1, 2, 3]`.

This walkthrough demonstrates the straightforward greedy strategy described in the solution approach, reflecting how we always ensure that each garden has a unique type of flower compared to its immediate neighbors.

Python Solution

```python
from collections import defaultdict

class Solution:
    def gardenNoAdj(self, n: int, paths: List[List[int]]) -> List[int]:
        # Build a graph where each node represents a garden and edges represent paths
        graph = defaultdict(list)
        for path in paths:
            # Decrementing by 1 to convert the 1-indexed gardens to 0-indexed for easier array manipulation
            x, y = path[0] - 1, path[1] - 1
            graph[x].append(y)
            graph[y].append(x)

        # Initialize the answer array where each garden's flower type will be stored
        garden_flower_types = [0] * n

        # Loop through each garden and choose a flower type
        for garden in range(n):
            # Build a set of used flower types for the current garden's adjacent gardens
            used_flower_types = {garden_flower_types[x] for x in graph[garden]}

            # Assign the lowest flower type (1-4) that is not used by adjacent gardens
            for flower_type in range(1, 5):
                if flower_type not in used_flower_types:
                    garden_flower_types[garden] = flower_type
                    break

        # Return the list of assigned flower types for each garden
        return garden_flower_types
```

Java Solution

```java
class Solution {
    public int[] gardenNoAdj(int n, int[][] paths) {
        // Create an adjacency list to represent gardens and their paths
        List<Integer>[] graph = new List[n];
        // Initialize each list within the graph
        Arrays.setAll(graph, x -> new ArrayList<>());
        // Fill the adjacency list with the paths provided
        for (int[] path : paths) {
            int garden1 = path[0] - 1; // Subtract 1 to convert to 0-based index
            int garden2 = path[1] - 1; // Subtract 1 to convert to 0-based index
            graph[garden1].add(garden2); // Add a path from garden1 to garden2
            graph[garden2].add(garden1); // Add a path from garden2 to garden1 (undirected graph)
        }

        // Answer array to store the type of flowers in each garden
        int[] flowerTypes = new int[n];
        // Array to keep track of used flower types
        boolean[] usedFlowers = new boolean[5]; // Index 0 is unused, as flower types are 1-4

        // Assign flower types to each garden
        for (int garden = 0; garden < n; ++garden) {
            // Reset the usedFlowers array for the current garden
            Arrays.fill(usedFlowers, false);
            // Mark the flower types used by adjacent gardens
            for (int adjacentGarden : graph[garden]) {
                usedFlowers[flowerTypes[adjacentGarden]] = true;
            }
            // Find the lowest number flower type that hasn't been used by adjacent gardens
            for (int type = 1; type < 5; ++type) {
                if (!usedFlowers[type]) {
                    flowerTypes[garden] = type; // Assign this flower type to the current garden
                    break; // Exit loop after assigning a type
                }
            }
        }

        // Return the array containing the flower types for each garden
        return flowerTypes;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <cstring> // Include C string library for memset

using namespace std; // Use the standard namespace

class Solution {
public:
    // Method to assign garden types without adjacent gardens having the same type.
    vector<int> gardenNoAdj(int n, vector<vector<int>>& paths) {
        // Create an adjacency list for the gardens.
        vector<vector<int>> adjList(n);
        for (const auto& garden : paths) {
            int garden1 = garden[0] - 1; // Convert to 0-based index.
            int garden2 = garden[1] - 1; // Convert to 0-based index.
            // Add each path to the adjacency list in both directions.
            adjList[garden1].push_back(garden2);
            adjList[garden2].push_back(garden1);
        }

        // Vector to store the type of flowers planted in each garden.
        vector<int> flowerTypes(n, 0);
        // Boolean array to keep track of used flower types for adjacency.
        bool usedFlowerTypes[5];

        // Iterate through each garden to assign flower types.
        for (int garden = 0; garden < n; ++garden) {
            // Reset the used flower types.
            memset(usedFlowerTypes, false, sizeof(usedFlowerTypes));

            // Mark the flower types already used by adjacent gardens.
            for (int adjacentGarden : adjList[garden]) {
                usedFlowerTypes[flowerTypes[adjacentGarden]] = true;
            }

            // Assign the lowest flower type that isn't used.
            for (int flowerType = 1; flowerType <= 4; ++flowerType) {
                if (!usedFlowerTypes[flowerType]) {
                    flowerTypes[garden] = flowerType;
                    break; // Stop looping once a flower type is assigned.
                }
            }
        }

        return flowerTypes; // Return the final assignment of flower types.
    }
};
```

Typescript Solution

```typescript
function gardenNoAdj(numberOfGardens: number, paths: number[][]): number[] {
    // Create an adjacency list to store the connections between gardens
    const graph: number[][] = new Array(numberOfGardens).fill(0).map(() => []);
    // Build the graph from the given paths
    for (const [garden1, garden2] of paths) {
        graph[garden1 - 1].push(garden2 - 1);
        graph[garden2 - 1].push(garden1 - 1);
    }
    // Initialize an array to hold the flower type for each garden
    const flowerTypes: number[] = new Array(numberOfGardens).fill(0);
    // Iterate through each garden to assign a flower type
    for (let currentGarden = 0; currentGarden < numberOfGardens; ++currentGarden) {
        // Track usage of flower types from 1 to 4, as each garden can have 1 of 4 different types of flowers
        const usedFlowerTypes: boolean[] = new Array(5).fill(false);
        // Mark the flower types used by the neighboring gardens
        for (const adjacentGarden of graph[currentGarden]) {
            usedFlowerTypes[flowerTypes[adjacentGarden]] = true;
        }
        // Assign the lowest numbered flower type that isn't used by any neighboring gardens
        for (let flowerType = 1; flowerType < 5; ++flowerType) {
            if (!usedFlowerTypes[flowerType]) {
                flowerTypes[currentGarden] = flowerType;
                break;
            }
        }
    }
    // Return the array containing the flower type for each garden
    return flowerTypes;
}
```

Time and Space Complexity

The code defines a Solution class with the `gardenNoAdj` method, which assigns a unique type of flower to each garden given the constraints that no adjacent gardens can have the same type of flower. There are `n` gardens numbered from `1` to `n` and at most `4` types of flowers.

Time Complexity:

The time complexity is determined by the following factors:

1. Building the graph: The loop iterates through the `paths` array, which might contain up to `n*(n-1)/2` paths in the worst case (a complete graph). This process has a time complexity of `O(E)` where `E` is the number of paths (edges).

2. Assigning flowers to gardens: There's an outer loop iterating `n` times for each garden and, inside it, a nested loop that iterates through the adjacent gardens (up to `n-1` times in the worst-case scenario for a complete graph). However, since each garden is limited to 3 edges (paths) to prevent excessive adjacency according to the problem statement, the inner loop has a constant factor, and thus, this part has a time complexity of `O(n)`.

3. Choosing a flower that hasn't been used: We iterate through the 4 types of flowers. This is a constant operation since the number of flower types does not change with `n`.

Thus, the total time complexity is `O(E + n)`, where `E` is the number of edges or paths.

Space Complexity:

The space complexity is determined by the following factors:

1. The graph `g`: In the worst-case scenario (a complete graph), each node connects to `n-1` other nodes. Therefore, the space required by this graph is `O(E)` where `E` is the number of edges.

2. `ans` array: This is an array of size `n`, so it consumes `O(n)` space.

3. `used` set: At most, the set contains 4 elements because there are only 4 different types of flowers. This is a constant space `O(1)`.

Considering all factors, the total space complexity of the algorithm is `O(E + n)`. The space taken by the `used` set is negligible in comparison to the graph and `ans` array.

Combining both the time and space complexities, we sum them up as `O(E + n)` for time complexity and `O(E + n)` for space complexity.