1257. Smallest Common Region

Medium Tree Depth-First Search Breadth-First Search Array Hash Table String [Leetcode Link]

Problem Description

You are provided with a list that represents a hierarchy of regions where each list's first element is a region that includes all the following elements in the same list. Every region is considered to contain itself and any region it's listed with, where the first region is the largest and it is followed by regions it includes in descending order of size.

Now, you are given two specific regions, `region1` and `region2`, and your task is to find the smallest common region that contains both of these regions.

Imagine a scenario where you have regions defined in a hierarchical manner, like in a geographical context—continents include countries, countries include states, states include cities, and so on. If given two cities, you must find the smallest state or country that includes both cities.

Note that for any region `r3` included in `r1`, there will be no other `r2` that includes `r3`. This simplifies the problem, ensuring that the tree of regions is well-defined and does not have any overlapping or conflicting ownership. The problem guarantees that there is always a smallest common region that exists for any two given regions.

Intuition

To arrive at the solution, we need to identify the hierarchy or the path of parents from each region to the top-most region in the hierarchy. Once we have the path for both regions, we can traverse these paths to find the smallest common region, which will be the first common ancestor.

We start by creating a map `m` that holds the parent-child relationship, where every region except the very first has a corresponding parent region that includes it—this constructs a map that resembles a tree.

Consider this tree as a family tree, where every person apart from the family's progenitor has a parent. `region1` and `region2` then become two family members, and we are looking for their closest common ancestor.

We populate a set `s` with ancestors of `region1` by following the parent links up the hierarchy. Then, we start at `region2` and move up its ancestral line until we encounter the first region already present in the set `s`, which will be the least common ancestor. If `region2` or any of its ancestors do not appear in the set `s`, then `region1` itself is the smallest region (this can only happen if `region1` is the top-most region).

The use of the set ensures that we can efficiently check whether the region is a common ancestor, and iterating up the ancestral lines ensures we find the smallest such region containing both given regions.

Solution Approach

The solution follows a direct approach using a map and a set, which are simple yet powerful data structures in Python. Here's a step-by-step breakdown of the implementation based on the provided code:

1. Create a map `m` to store the parent-child relationships. After processing all the regions' lists, `m` will contain a mapping where the key is a 'child' region, and the value is its immediate 'parent' region. This is achieved with a nested loop: the outer loop iterates over each list of regions, and the inner loop (starting from the second element) maps each region to its 'parent', which is the first element in the current list.

```python
m = {}
for region in regions:
    for r in region[1:]:
        m[r] = region[0]
```

2. Instantiate a set `s`, which will hold all the ancestors of `region1`. We then traverse up from `region1` to the top-most ancestor (which will be a region with no parent in map `m`). As we traverse, we populate the set with each ancestor, creating a pathway from `region1` to its top-most ancestor.

```python
s = set()
while m.get(region1):
    s.add(region1)
    region1 = m[region1]
```

3. Starting with `region2`, the code checks if `region2` or any of its ancestors are in the set `s`, which would indicate a common ancestor has been found.

```python
while m.get(region2):
    if region2 in s:
        return region2
    region2 = m[region2]
```

4. If `region2` itself or one of its ancestors was not in set `s`, then the loop stops once `region2` becomes a region without a parent, which, as mentioned before, can only be the top-most ancestor (the very first region). In such a case, `region1` had to be the top-most ancestor, and it's returned as the smallest region that contains both `region1` and `region2`.

The implementation uses hashmap and hashset data structures, as map `m` and set `s` respectively, which facilitate the efficient lookup in constant time, `O(1)`, for checking the presence of ancestors and mapping children to parents. This is how the Python dictionary and set are generally implemented.

This approach is efficient in terms of time complexity because it has to process each region only once when creating the map, which is `O(N)` where `N` is the total number of regions, and then the ancestral path for both regions, each being at most `O(H)` where `H` is the height of the hierarchy. Therefore, the overall complexity is `O(N + H)`. The space complexity is `O(N)` for storing the map and the set of ancestors for `region1`.

Example Walkthrough

Let's illustrate the solution approach with an example. Suppose we have five regions arranged in the following hierarchy:

```
[
 ["Earth", "North America", "United States"],
 ["United States", "California"],
 ["United States", "New York"],
 ["California", "Los Angeles"],
 ["New York", "New York City"]
]
```

And we are given two regions to find the smallest common region for: `region1 = "Los Angeles"` and `region2 = "New York City"`.

1. First, we create the map `m` listing each child's parent. It looks like this after processing the lists:

```
m = {
    "North America": "Earth",
    "United States": "North America",
    "California": "United States",
    "New York": "United States",
    "Los Angeles": "California",
    "New York City": "New York"
}
```

2. Next, we add all ancestors of `region1` ("Los Angeles") into the set `s`. We follow up the hierarchy from "Los Angeles" to "California", and then to "United States", which is an ancestor to both "California" and "New York":

```python
s = set(["Los Angeles", "California", "United States"])
```

3. After this, we check if `region2` ("New York City") or any of its ancestors are in set `s`. We start with "New York City", whose parent is "New York", and then the parent of "New York" is "United States" which is in `s`. Thus, we find that "United States" is the smallest common region for "Los Angeles" and "New York City":

```python
# "New York City" is not in s
# "New York"'s parent is "United States" which is in s
smallest_common_region = "United States"
```

4. Now we have the result: the smallest common region containing both "Los Angeles" and "New York City" is "United States".

The example confirms our approach, showing how it finds the least common ancestor efficiently by constructing a map to represent the hierarchy and then using a set to track the ancestors of one region to compare with the ancestral line of the other region.

Python Solution

```python
class Solution:
    def findSmallestRegion(self, regions, region1, region2):
        # Create a dictionary to map each region to its parent region.
        parent_map = {}
        for region in regions:
            # region[0] is the parent region, and the rest are the child regions.
            for child_region in region[1:]:
                parent_map[child_region] = region[0]

        # A set to store all the ancestors of region1.
        ancestors = set()
        # Loop to find all ancestors of region1 and add them to the set.
        while region1 in parent_map:
            ancestors.add(region1)
            # Move to the parent region.
            region1 = parent_map[region1]

        # Loop to traverse the ancestors of region2.
        while region2 in parent_map:
            # Check if the current region2 is also an ancestor of region1.
            if region2 in ancestors:
                # If it is, we have found the smallest common region.
                return region2
            # Move to the parent region of region2.
            region2 = parent_map[region2]

        # If there is no common ancestor in the set, the smallest common
        # region is the topmost ancestor of region1.
        return region1

```

Java Solution

```java
class Solution {
    public String findSmallestRegion(List<List<String>> regions, String region1, String region2) {
        // Create a map to store each region's immediate parent region
        Map<String, String> parentMap = new HashMap<>();
        // Iterate through the list of regions to fill the parentMap
        for (List<String> region : regions) {
            for (int i = 1; i < region.size(); i++) {
                // Map each child (i > 0) to its parent (0th element)
                parentMap.put(region.get(i), region.get(0));
            }
        }

        // Initialize a set to keep track of all ancestors of region1
        Set<String> ancestors = new HashSet<>();
        // Go up the hierarchy of regions from region1, adding each to the ancestor set
        while (parentMap.containsKey(region1)) {
            ancestors.add(region1);
            region1 = parentMap.get(region1); // Move to the parent region
        }

        // Start from region2 and move up the hierarchy until you find a common ancestor
        while (parentMap.containsKey(region2)) {
            if (ancestors.contains(region2)) {
                // If region2 or any of its ancestors is in the ancestors set, we've found the smallest common region
                return region2;
            }
            region2 = parentMap.get(region2); // Move to the parent region
        }

        // If no common ancestor is found while traversing from region2, return the highest ancestor of region1
        return region1;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <unordered_map>
#include <unordered_set>

class Solution {
public:
    // Function to find the smallest common region between region1 and region2
    std::string findSmallestRegion(std::vector<std::vector<std::string>>& regions,
                                   std::string region1, std::string region2) {
        // Create a map to store the immediate parent for each region
        std::unordered_map<std::string, std::string> parentMap;

        // Fill in the parent map with each region's parent
        for (const auto& region : regions) {
            for (size_t i = 1; i < region.size(); ++i) {
                parentMap[region[i]] = region[0];
            }
        }

        // Create a set to store the ancestors of region1
        std::unordered_set<std::string> ancestors;

        // Add all ancestors of region1 to the set
        while (parentMap.count(region1)) {
            ancestors.insert(region1);
            region1 = parentMap[region1]; // Move up to the parent region
        }

        // Find the first common ancestor of region2 which is in the ancestors set
        while (parentMap.count(region2)) {
            if (ancestors.count(region2)) {
                return region2; // This is the closest common ancestor
            }
            region2 = parentMap[region2]; // Move up to the parent region
        }

        // If there was no common ancestor found in the loop above,
        // return the last ancestor of region1 which should be the root region
        return region1;
    }
};

```

Typescript Solution

```typescript
type Region = string;
type Regions = Region[][];

// We use Map instead of unordered_map for parent mapping
const parentMap = new Map<Region, Region>();

/**
 * Function to fill in the parent map with each region's parent.
 */
function fillParentMap(regions: Regions): void {
    // Iterate through each list of regions
    for (const regionList of regions) {
        // The first region is the parent, the rest are its children
        const parentRegion = regionList[0];
        for (let i = 1; i < regionList.length; i++) {
            const childRegion = regionList[i];
            // Assign parent to the child region
            parentMap.set(childRegion, parentRegion);
        }
    }
}

/**
 * Function to find the smallest common region between region1 and region2.
 */
function findSmallestRegion(regions: Regions, region1: Region, region2: Region): Region {
    // Fill the parent map before processing
    fillParentMap(regions);

    // Use Set to find the ancestors of region1
    const ancestors = new Set<Region>();

    // Add all ancestors of region1 to the set
    while (parentMap.has(region1)) {
        ancestors.add(region1);
        region1 = parentMap.get(region1)!; // Move up to the parent region, using ! to assert that value is not undefined
    }

    // Find the first common ancestor of region2 which is in the ancestors set
    while (parentMap.has(region2)) {
        if (ancestors.has(region2)) {
            return region2; // This is the closest common ancestor
        }
        region2 = parentMap.get(region2)!; // Move up to the parent region, using ! to assert that value is not undefined
    }

    // No common ancestor found in the loop above, return the root region instead
    return region1;
}

```

Time and Space Complexity

The given Python code defines a function to find the smallest common region between `region1` and `region2`.

- Time complexity:

The time complexity of the first loop is `O(N)` where `N` is the total number of regions inside all the lists combined. This is because each region list is traversed exactly once, and each traversal takes `O(K)`, where `K` is the number of sub-regions within the current region. Hence, the overall time for this loop is proportional to the total number of regions in all lists.

For the second and third while loops, the time complexity is `O(H1 + H2)`, where `H1` is the height of the region hierarchy (region tree) from `region1` to the top, and `H2` is the height from `region2` to the top. In the worst-case scenario, this would be `O(H)` where `H` is the height of the tree.

Combining the two parts, the overall time complexity of the function can be expressed as `O(N + H)`, assuming `H` can be considered smaller than `N` in a large dataset.

- Space complexity:

The space complexity consists of the additional space used by the map `m` and set `s`. The map `m` has `O(N)` complexity as it contains all regions except the global root. The set `s` has `O(H1)` complexity in the worst case as it stores the path from `region1` to the root.

Hence, the total space complexity of the function is `O(N + H1)`. Since `H1` is generally smaller compared to `N`, we can approximate the space complexity as `O(N)`.