Back

[Python] Clean Solution from DFS O(mn) to Union Find O(m + n) with Explanation

kkk1415 ★ 7 Last Edit: May 12, 2020 10:52 PM 29 VIEWS

2

What is given in the Question:
From A / B = k, we can have A / B = k and B / A = 1/k

let X and Y are two given value
X / Y = (X / a1) * (a1 / a2) * ..... * (an - 1 / an) * (an / Y)
where a1, a2, ... an are intermdiate values

Why this is a implicit Graph problem?

- Because X / Y is calculated using a chain of divisions, which implies this is a path
- Vertex: Each Value A
- Edge: If A / B = k is given, then (A, B) and (B, A) are edges and the edge length L(A, B) = k and L(B, A) = 1 / k
- Path: Between X and Y, the path is (X / a1) * (a1 / a2) * ..... * (an - 1 / an) * (an / Y), which is X / Y

Solution 1: DFS

After we build the graph, we can try to do the query using DFS:

- if query (u, v) is given, we start from u and perform DFS, and calculate the division value along the way
- If we reach v from u, we push the result. Otherwise, we push -1

Time Complexity: O(MN) where M = len(equations) and N = len(queries)
Space Complexity: O(M) where M = len(equations)

```
import collections

class Solution:
    def calcEquation(self, equations: List[List[str]], values: List[float], queries: List[List[str]]) -> List[float]:
        # Step 1: Build Graph
        graph = collections.defaultdict(list)

        for i in range(len(equations)):
            u, v = equations[i]
            l = values[i]

            graph[u].append((v, l))
            graph[v].append((u, 1 / l))

        # Step 2: DFS for each query
        result = []
        for u, v in queries:
            # skip if val is not found in the graph
            if u not in graph or v not in graph:
                result.append(-1.0)
                continue

            div_val = self.dfs(graph, u, v, set())
            if div_val is None:
                result.append(-1)
            else:
                result.append(div_val)

        return result

    def dfs(self, graph, start, end, visited):
        visited.add(start)

        if start == end:
            return 1.0

        for child, l in graph[start]:
            if child in visited:
                continue

            child_div_val = self.dfs(graph, child, end, visited)
            if child_div_val is not None:
                return l * child_div_val

        return None
```

Solution 2: Union Find

Based on our analysis in solution 1, we can easily tell that to calculate X / Y is a connectivity problem. Connectivity problems can be solved using Union Find or Disjoint Set data structure.

Since the problem is not just connectivity, but also the product of the edge length along the path, we need to slightly modify Union Find

- Apart from the original hash table `parent`, we need another hash table `val[x]` to store the value `x / parent[x]`, which the division value between x and it's parent
- Path compression needs to be modified. We need to update the `val[x]` after we set `parent[x] = root`, because the parent of x has been changed
- function `union` needs to be modified. We need to update the `val[root_a]` after we set `parent[root_a] = root_b`, because the parent of root_a has been changed

X / Y = (X / root) ÷ (Y / root) if X, Y connected and share a common root

Time Complexity: O(M + N) where M = len(equations) and N = len(queries)
Space Complexity: O(M) where M = len(equations)

```
class DisjointSet(object):
    def __init__(self):
        self.parent = {}
        self.val = {}

    def add(self, node):
        if node not in self.parent:
            self.parent[node] = node
            self.val[node] = 1

    def union(self, node_a, node_b, l):
        root_a, val_a = self.find(node_a)
        root_b, val_b = self.find(node_b)

        if root_a != root_b:
            self.parent[root_a] = root_b
            self.val[root_a] = l * val_b / val_a

    def find(self, node):
        if node not in self.parent: return None, None
        if self.parent[node] == node: return node, 1

        root, parent_root_val = self.find(self.parent[node])

        self.parent[node] = root
        self.val[node] *= parent_root_val

        return self.parent[node], self.val[node]

class Solution:
    def calcEquation(self, equations: List[List[str]], values: List[float], queries: List[List[str]]) -> List[float]:
        disjoint_set = DisjointSet()

        for i in range(len(equations)):
            u, v = equations[i]
            l = values[i]

            disjoint_set.add(u)
            disjoint_set.add(v)
            disjoint_set.union(u, v, l)

        result = []
        for u, v in queries:
            root_u, val_u = disjoint_set.find(u)
            root_v, val_v = disjoint_set.find(v)

            if not root_u or not root_v or root_u != root_v:
                result.append(-1)
                continue

            result.append(val_u / val_v)

        return result
```