Documentation

1. Problem Statement

The problem involves evaluating division equations and answering related queries. Each equation represents a division relationship between two variables, such as A/B=valueA / B. Given a series of such equations and their corresponding values, the task is to compute the result of division for various queries in the form C/D=?. If a query cannot be answered because the variables are not connected, the result should be -1.0. If a variable does not appear in any equation, it is considered undefined, and the result for such queries should also be -1.0.

2. Intuition

The problem can be intuitively framed as a graph traversal problem. In this case, each variable in the equations can be treated as a node, and the division relationships form directed edges with weights representing the division results. The key idea is to find a path between the two queried variables and calculate the product of the edge weights along that path. If no path exists between the variables, the result should be -1.0. Otherwise, the result is the product of the edge weights.

3. Key Observations

- If a variable does not appear in the equations, it is considered undefined, and any query involving that variable should return -1.0.
- The result of a variable divided by itself is always 1.0, as any number divided by itself is equal to 1.
- The problem can be solved using graph traversal, where the goal is to find the path between two nodes (variables) and compute the division result by multiplying the edge weights.
- The use of depth-first search (DFS) allows efficient traversal through the graph to compute the division results for each query.

4. Approach

The solution is based on representing the problem as a graph. Each variable is a node, and each equation creates a directed edge between two nodes with a weight corresponding to the division result. To compute the result for each query, we use a depth-first search (DFS) from the starting variable to the target variable, accumulating the product of the edge weights along the path. If no path exists, the result for that query is -1.0.

5. Edge Cases

There are a few edge cases to consider:

- Variables not present in the equations should return -1.0 for any query involving them.
- Queries that involve dividing a variable by itself should always return 1.0.
- If no path exists between the queried variables, the result should be -1.0.
- The solution should handle cases where some variables are connected while others are not, requiring different results for different queries.

6. Complexity Analysis

Time Complexity: The time complexity is driven by the graph construction and the DFS traversal. Constructing the graph takes O(E), where E is the number of equations. For each query, a DFS search may take O(V + E) time, where V is the number of variables and E is the number of equations. Therefore, the overall time complexity is O(Q·(V+E)), where Q is the number of queries.

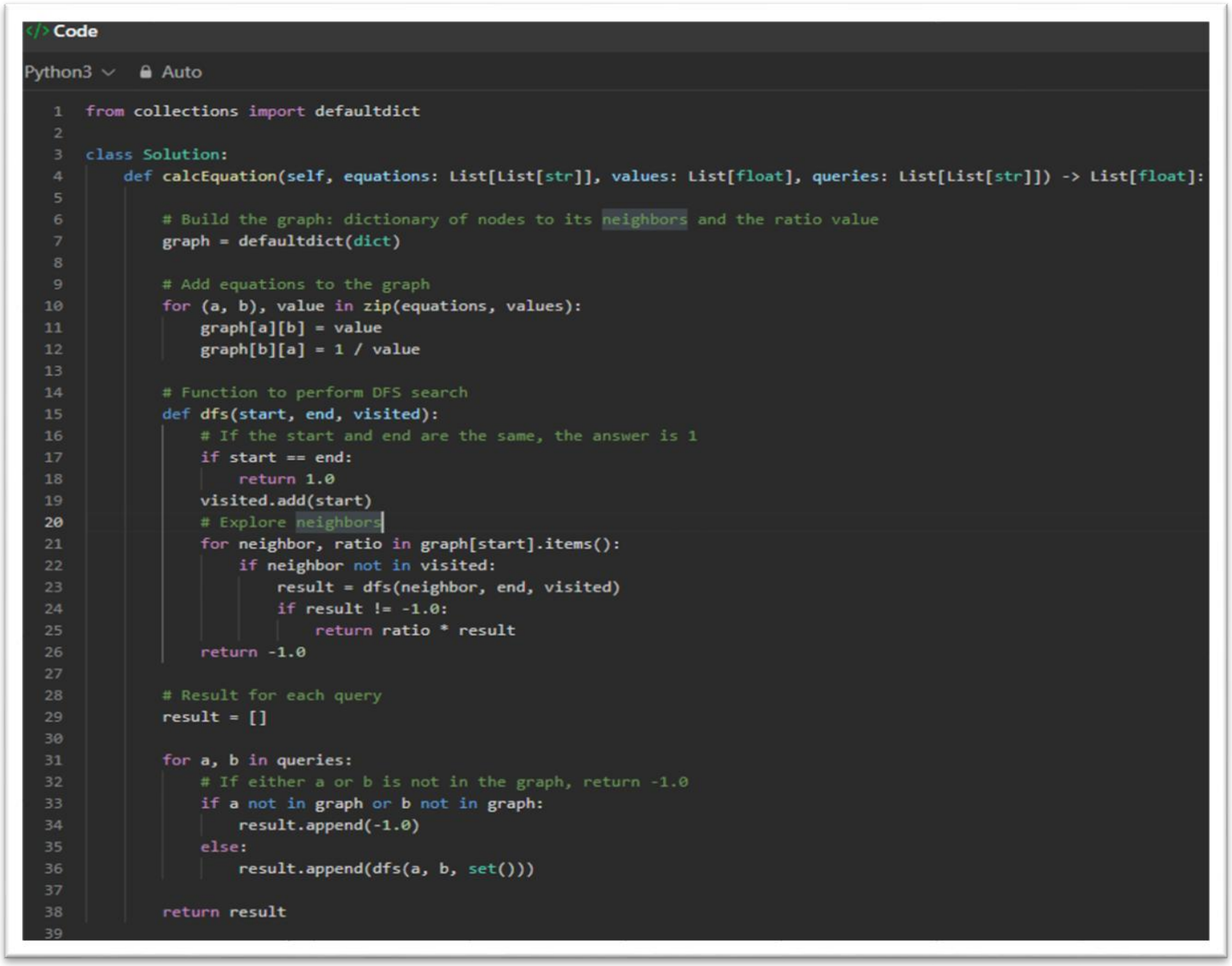
Space Complexity: The space complexity is O(V+E), as we need to store the graph, where V is the number of variables and E is the number of equations. Additionally, the recursion stack for DFS also contributes to the space complexity.

7. Alternative Approaches

An alternative approach could involve using Union-Find (Disjoint Set Union), where we keep track of connected components and their associated ratios. While this approach works well for dynamic connectivity problems, it requires more complex handling of ratios between connected components and may not be as straightforward as DFS in this case. DFS is simple to implement and effective for this problem, making it a preferred choice.

8. Code Implementation

The implementation uses a graph to represent the equations, with each node (variable) connected by edges that represent the division relationship between variables. For each query, a DFS is used to traverse the graph and compute the division result. If no path exists, the query result is -1.0. The solution efficiently handles multiple queries by reusing the graph structure.



```python
from collections import defaultdict

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

        # Build the graph: dictionary of nodes to its neighbors and the ratio value
        graph = defaultdict(dict)

        # Add equations to the graph
        for (a, b), value in zip(equations, values):
            graph[a][b] = value
            graph[b][a] = 1 / value

        # Function to perform DFS search
        def dfs(start, end, visited):
            # If the start and end are the same, the answer is 1
            if start == end:
                return 1.0
            visited.add(start)
            # Explore neighbors
            for neighbor, ratio in graph[start].items():
                if neighbor not in visited:
                    result = dfs(neighbor, end, visited)
                    if result != -1.0:
                        return ratio * result
            return -1.0

        # Result for each query
        result = []

        for a, b in queries:
            # If either a or b is not in the graph, return -1.0
            if a not in graph or b not in graph:
                result.append(-1.0)
            else:
                result.append(dfs(a, b, set()))

        return result
```

9. Test Cases

Test cases are essential to verify the correctness of the solution. For instance, a set of equations involving variables like a, b, and c is used to test the ability of the solution to compute answers for queries such as "What is a/ca / c?" or "What is b/ab / a?" Additional cases handle situations where variables are undefined or where no valid path exists between the queried variables, ensuring that the solution correctly returns -1.0 in such cases.

10. Final Thoughts

This problem is an excellent application of graph traversal techniques to solve a real-world problem involving ratios and connectivity. The depth-first search (DFS) approach is efficient and simple, making it an ideal solution for handling the problem's constraints. By modeling the problem as a graph and leveraging DFS, we can efficiently answer each query while managing edge cases involving undefined variables and self-division. This approach offers a clear and practical solution to the problem.