2277. Closest Node to Path in Tree

Problem Explanation

In this problem, we have a tree with `n` nodes numbered from 0 to n-1. We are given a 2D integer array `edges` that contains information about the bidirectional edges connecting the nodes of the tree. Our task is to answer a series of queries. Each query is represented as a 0-indexed integer array `[start, end, node]`, and for each query, we must find the node on the path from `start` to `end` that is closest to `node`.

To do this, we first need to calculate the shortest path distances between nodes. Then we need to walk through the path from `start` to `end` while tracking the closest node, making sure to update it whenever we find a node that is closer to `node`.

Algorithm Explanation

1. First, we define a function `fillDist` that takes the current node `u`, distance `d`, and fills the `dist` array with the shortest distance between `u` and all other nodes in the tree. To do this, we use a depth-first search approach, visiting all the neighbor nodes `v` of node `u` and calling `fillDist` recursively with `v` and distance `d+1`.
2. Next, we define a function `findClosest` that takes the current node `u`, destination node `end`, and target node `node`. It should return the node on the path from `u` to `end` that is closest to `node`. We use a similar depth-first search approach here as well, iterating over the neighbors `v` of the node `u`. If the distance from `v` to `end` is smaller than the distance from `u` to `end`, we update the closest node accordingly and call `findClosest` recursively with `v`.
3. With these helper functions, we can now implement the function `closestNode` that takes input parameters `n`, `edges`, and `query`. We first initialize an empty answer vector `ans` to store the results of each query.
4. We create a `tree` representation as an adjacency list using the information from `edges`.
5. We also initialize a 2D `dist` array with dimensions `n x n` and fill it with `-1`. Then, for each node `i`, we call `fillDist` function to fill the `dist` array with the shortest distance between node `i` and all other nodes in the tree.
6. Then, we iterate over each query `q`, extracting the `start`, `end`, and `node`. Afterwards, we call the `findClosest` function with `start`, `end`, and `node` to obtain the result for the current query. We store this result in our answer vector `ans`.
7. Finally, we return the answer vector `ans` containing the results of all queries.

```
plaintext
Example:

n = 6
edges = [[0, 1], [0, 2], [1, 3], [1, 4], [2, 5]]
query = [[1, 3, 2], [2, 0, 4]]

Tree edges:
 0
/ \
1   2
|   |
3   5
|
4

Dist array:
[[0 1 1 2 2]
 [1 0 2 1 1]
 [1 2 0 3 3]
 [2 1 3 0 2]
 [2 1 3 2 0]
 [2 3 1 4 4 0]]

For query [1, 3, 2], distance from 1 to 3 is 1 so we return node 3 as it is closer to node 2. Ans = [3]
For query [2, 0, 4], distance from 2 to 0 is 1 so we return node 0 due to smaller distance to node 4. Ans = [3, 0]

Output: [3, 0]
```

Java Solution

```
java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Solution {
    public int[] closestNode(int n, int[][] edges, int[][] query) {
        int[] ans = new int[query.length];
        ArrayList<ArrayList<Integer>> tree = new ArrayList<>();
        int[][] dist = new int[n][n];

        for (int i = 0; i < n; i++) {
            tree.add(new ArrayList<>());
            Arrays.fill(dist[i], -1);
        }

        for (int[] edge : edges) {
            int u = edge[0];
            int v = edge[1];
            tree.get(u).add(v);
            tree.get(v).add(u);
        }

        for (int i = 0; i < n; i++) {
            fillDist(tree, i, 0, dist);
        }

        for (int i = 0; i < query.length; i++) {
            int start = query[i][0];
            int end = query[i][1];
            int node = query[i][2];
            ans[i] = findClosest(tree, dist, start, end, node, start);
        }

        return ans;
    }

    private void fillDist(List<ArrayList<Integer>> tree, int start, int u, int d,
                          int[][] dist) {
        dist[start][u] = d;
        for (int v : tree.get(u))
            if (dist[start][v] == -1)
                fillDist(tree, start, v, d + 1, dist);
    }

    private int findClosest(List<ArrayList<Integer>> tree,
                            int[][] dist, int u, int end, int node,
                            int ans) {
        for (int v : tree.get(u))
            if (dist[v][end] < dist[u][end])
                return findClosest(tree, dist, v, end, node,
                                   dist[ans][node] < dist[v][node] ? ans : v);

        return ans;
    }
}
```

Python Solution

```
python
from collections import defaultdict

class Solution:
    def closestNode(self, n, edges, query):
        ans = []
        tree = defaultdict(list)
        dist = [[-1] * n for _ in range(n)]

        for edge in edges:
            u, v = edge
            tree[u].append(v)
            tree[v].append(u)

        def fillDist(start, u, d):
            dist[start][u] = d
            for v in tree[u]:
                if dist[start][v] == -1:
                    fillDist(start, v, d + 1)

        for i in range(n):
            fillDist(i, i, 0)

        def findClosest(u, end, node, ans):
            for v in tree[u]:
                if dist[v][end] < dist[u][end]:
                    ans = findClosest(v, end, node, ans if dist[ans][node] < dist[v][node] else v)
            return ans

        for q in query:
            start, end, node = q
            ans.append(findClosest(start, end, node, start))

        return ans
```

JavaScript Solution

```
javascript
class Solution {
    closestNode(n, edges, query) {
        const ans = [];
        const tree = Array.from({ length: n }, () => []);
        const dist = Array.from({ length: n }, () => Array(n).fill(-1));

        for (const edge of edges) {
            const [u, v] = edge;
            tree[u].push(v);
            tree[v].push(u);
        }

        const fillDist = (start, u, d) => {
            dist[start][u] = d;
            for (const v of tree[u]) {
                if (dist[start][v] === -1) {
                    fillDist(start, v, d + 1);
                }
            }
        };

        for (let i = 0; i < n; i++) {
            fillDist(i, i, 0);
        }

        const findClosest = (u, end, node, ans) => {
            for (const v of tree[u]) {
                if (dist[v][end] < dist[u][end]) {
                    ans = findClosest(v, end, node, dist[ans][node] < dist[v][node] ? ans : v);
                }
            }
            return ans;
        };

        for (let i = 0; i < query.length; i++) {
            const [start, end, node] = query[i];
            ans.push(findClosest(start, end, node, start));
        }

        return ans;
    }
}
```

C++ Solution

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

class Solution {
 public:
  vector<int> closestNode(int n, vector<vector<int>>& edges,
                          vector<vector<int>>& query) {
    vector<int> ans;
    vector<vector<int>> tree(n);
    vector<vector<int>> dist(n, vector<int>(n, -1));

    for (const vector<int>& edge : edges) {
      const int u = edge[0];
      const int v = edge[1];
      tree[u].push_back(v);
      tree[v].push_back(u);
    }

    for (int i = 0; i < n; ++i)
      fillDist(tree, i, i, 0, dist);

    for (const vector<int>& q : query) {
      const int start = q[0];
      const int end = q[1];
      const int node = q[2];
      ans.push_back(findClosest(tree, dist, start, end, node, start));
    }

    return ans;
  }

 private:
  void fillDist(const vector<vector<int>>& tree, int start, int u, int d,
                vector<vector<int>>& dist) {
    dist[start][u] = d;
    for (const int v : tree[u])
      if (dist[start][v] == -1)
        fillDist(tree, start, v, d + 1, dist);
  }

  int findClosest(const vector<vector<int>>& tree,
                  const vector<vector<int>>& dist, int u, int end, int node,
                  int ans) {
    for (const int v : tree[u])
      if (dist[v][end] < dist[u][end])
        return findClosest(tree, dist, v, end, node,
                           dist[ans][node] < dist[v][node] ? ans : v);

    return ans;
  }
};
```

C# Solution

```
csharp
using System;
using System.Collections.Generic;

public class Solution {
    public int[] ClosestNode(int n, int[][] edges, int[][] query) {
        int[] ans = new int[query.Length];
        List<int>[] tree = new List<int>[n];
        int[][] dist = new int[n][];

        for (int i = 0; i < n; i++) {
            tree[i] = new List<int>();
            dist[i] = new int[n];
            Array.Fill(dist[i], -1);
        }

        foreach (int[] edge in edges) {
            int u = edge[0];
            int v = edge[1];
            tree[u].Add(v);
            tree[v].Add(u);
        }

        for (int i = 0; i < n; i++) {
            FillDist(tree, i, i, 0, dist);
        }

        for (int i = 0; i < query.Length; i++) {
            int start = query[i][0];
            int end = query[i][1];
            int node = query[i][2];
            ans[i] = FindClosest(tree, dist, start, end, node, start);
        }

        return ans;
    }

    private void FillDist(List<int>[] tree, int start, int u, int d,
                          int[][] dist) {
        dist[start][u] = d;
        foreach (int v in tree[u]) {
            if (dist[start][v] == -1) {
                FillDist(tree, start, v, d + 1, dist);
            }
        }
    }

    private int FindClosest(List<int>[] tree,
                            int[][] dist, int u, int end, int node,
                            int ans) {
        foreach (int v in tree[u]) {
            if (dist[v][end] < dist[u][end]) {
                ans = FindClosest(tree, dist, v, end, node,
                                  dist[ans][node] < dist[v][node] ? ans : v);
            }
        }
        return ans;
    }
}
```

Time Complexity

The time complexity of our solution is O(n^2), where n is the number of nodes in the tree. This is because we perform depth-first searches on the tree for every node to compute the `dist` array.