2204. Distance to a Cycle in Undirected Graph

[Leetcode Link]

Problem Description

You are given a connected undirected graph with exactly one cycle. The graph has `n` nodes, numbered from 0 to n - 1 (inclusive). You are also provided with a 2D integer array `edges`, where `edges[i] = [node1i, node2i]` denotes a bidirectional edge connecting the nodes `node1i` and `node2i`.

The distance between two nodes a and b is defined as the minimum number of edges needed to go from node a to node b.

The goal is to return an integer array `answer` of size n, where `answer[i]` is the minimum distance between the ith node and any node in the cycle.

Example

Let's consider the following graph:

```

0 -- 1 -- 2
 \       /
  \     /
     3
```

Here, the `edges` are [[0, 1], [1, 2], [2, 3], [0, 3]]. The cycle consists of nodes 0, 1, 2, and 3.

In this example, the distances to any node of the cycle are:

- Node 0: 0 (already in the cycle)
- Node 1: 0 (already in the cycle)
- Node 2: 0 (already in the cycle)
- Node 3: 0 (already in the cycle)

So, the `answer` array is [0, 0, 0, 0].

Approach

The solution uses Breadth First Search (BFS) and Depth First Search (DFS) to find the distance to the cycle for each node.

1. First, create an adjacency list (graph) from the edges.
2. Next, find the cycle in the graph using the DFS algorithm. Initialize a rank vector, and while visiting each node, update its rank.
3. Once the cycle is found, perform BFS on the cycle nodes to find the minimum distance to the cycle for each node.

Solution in Python

```

python
from collections import deque

class Solution:

    NO_RANK = -2

    def distanceToCycle(self, n, edges):
        ans = [0] * n
        graph = [[] for _ in range(n)]

        for u, v in edges:
            graph[u].append(v)
            graph[v].append(u)

        cycle = []
        self.getRank(graph, 0, 0, [self.NO_RANK] * n, cycle)

        q = deque(cycle)
        seen = [False] * n
        for u in cycle:
            seen[u] = True

        dist = 0
        while q:
            dist += 1
            for _ in range(len(q)):
                u = q.popleft()
                for v in graph[u]:
                    if not seen[v]:
                        q.append(v)
                        seen[v] = True
                        ans[v] = dist

        return ans

    def getRank(self, graph, u, currRank, rank, cycle):
        if rank[u] != self.NO_RANK:
            return rank[u]

        rank[u] = currRank
        minRank = currRank

        for v in graph[u]:
            if rank[v] == len(rank) or rank[v] == currRank - 1:
                continue
            nextRank = self.getRank(graph, v, currRank + 1, rank, cycle)

            if nextRank <= currRank:
                cycle.append(v)
            minRank = min(minRank, nextRank)

        rank[u] = len(rank)
        return minRank
```

This Python solution implements the approach explained above, using both DFS and BFS to find the minimum distance to the cycle for each node. The `getRank()` method implements DFS, while the Breadth First Search is performed after finding the cycle.## Solution in Javascript

```

javascript
class Solution {

    constructor() {
        this.NO_RANK = -2;
    }

    distanceToCycle(n, edges) {
        let ans = new Array(n).fill(0);
        let graph = new Array(n).fill(null).map(() => []);

        for (let [u,v] of edges) {
            graph[u].push(v);
            graph[v].push(u);
        }

        let cycle = [];
        this.getRank(graph, 0, 0, new Array(n).fill(this.NO_RANK), cycle);

        let queue = cycle.slice();
        let seen = new Array(n).fill(false);
        for (let u of cycle) {
            seen[u] = true;
        }

        let dist = 0;
        while (queue.length) {
            dist += 1;
            for (let i = queue.length; i > 0; i--) {
                let u = queue.shift();
                for (let v of graph[u]) {
                    if (!seen[v]) {
                        queue.push(v);
                        seen[v] = true;
                        ans[v] = dist;
                    }
                }
            }
        }

        return ans;
    }

    getRank(graph, node, currRank, rank, cycle) {
        if (rank[node] !== this.NO_RANK) {
            return rank[node];
        }

        rank[node] = currRank;
        let minRank = currRank;

        for (let v of graph[node]) {
            if (rank[v] === rank.length || rank[v] === currRank - 1) {
                continue;
            }
            let nextRank = this.getRank(graph, v, currRank + 1, rank, cycle);

            if (nextRank <= currRank) {
                cycle.push(v);
            }
            minRank = Math.min(minRank, nextRank);
        }

        rank[node] = rank.length;
        return minRank;
    }
}
```

This Javascript solution uses the same approach as the Python solution, with an implementation of DFS in the `getRank()` method and BFS for finding the minimum distance to the cycle for each node.

Solution in Java

```

java
import java.util.*;

class Solution {
    private static final int NO_RANK = -2;

    public int[] distanceToCycle(int n, int[][] edges) {
        int[] ans = new int[n];
        List<List<Integer>> graph = new ArrayList<>();

        for (int i = 0; i < n; i++) {
            graph.add(new ArrayList<>());
        }

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

        List<Integer> cycle = new ArrayList<>();
        getRank(graph, 0, 0, new int[n], cycle, NO_RANK);

        Queue<Integer> queue = new LinkedList<>(cycle);
        boolean[] seen = new boolean[n];

        for (int u : cycle) {
            seen[u] = true;
        }

        int dist = 0;
        while (!queue.isEmpty()) {
            dist++;
            int size = queue.size();
            for (int i = 0; i < size; i++) {
                int u = queue.poll();
                for (int v : graph.get(u)) {
                    if (!seen[v]) {
                        queue.add(v);
                        seen[v] = true;
                        ans[v] = dist;
                    }
                }
            }
        }

        return ans;
    }

    private int getRank(List<List<Integer>> graph, int node, int currRank, int[] rank, List<Integer> cycle, int defaultValue) {
        if (rank[node] != defaultValue) {
            return rank[node];
        }

        rank[node] = currRank;
        int minRank = currRank;

        for (int v : graph.get(node)) {
            if (rank[v] == rank.length || rank[v] == currRank - 1) {
                continue;
            }
            int nextRank = getRank(graph, v, currRank + 1, rank, cycle, defaultValue);

            if (nextRank <= currRank) {
                cycle.add(v);
            }
            minRank = Math.min(minRank, nextRank);
        }

        rank[node] = rank.length;
        return minRank;
    }
}
```

The Java solution is similar to the Python and Javascript solutions. It implements both DFS and BFS algorithms in separate methods. The `getRank()` method implements DFS to find the cycle, and the BFS is implemented in the `distanceToCycle()` method to find the minimum distance to the cycle for each node.

Got a question? [Ask the Teaching Assistant] anything you don't understand.