2846.边权重均等查询 题解

> 计算学部十大打卡活动——“龙舞编程新春会”编程打卡（2024-1-26）
>
> [力扣——2846.边权重均等查询]()

一、题目

现有一棵由 `n` 个节点组成的无向树，节点按从 `0` 到 `n - 1` 编号。给你一个整数 `n` 和一个长度为 `n - 1` 的二维整数数组 `edges` ，其中 `edges[i] = [ui, vi, wi]` 表示树中存在一条位于节点 `ui` 和节点 `vi` 之间、权重为 `wi` 的边。

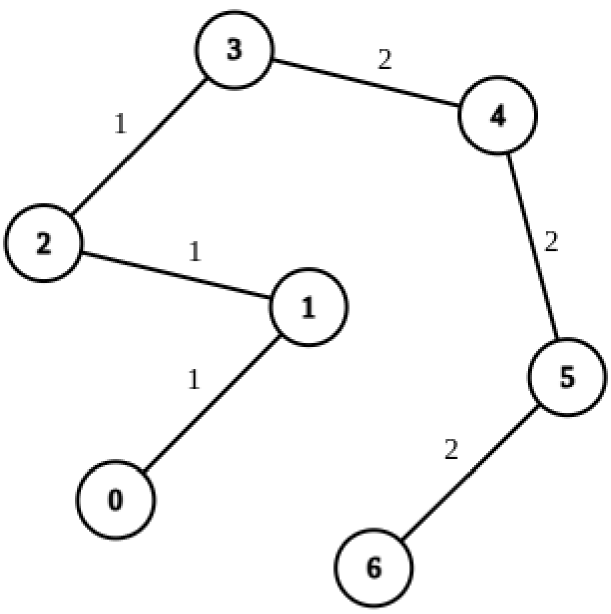
另给你一个长度为 `m` 的二维整数数组 `queries` ，其中 `queries[i] = [ai, bi]` 。对于每条查询，请你找出使从 `ai` 到 `bi` 路径上每条边的权重相等所需的 **最小操作次数** 。在一次操作中，你可以选择树上的任意一条边，并将其权重更改为任意值。

注意：

- 查询之间 **相互独立** 的，这意味着每条新的查询时，树都会回到 **初始状态** 。
- 从 `ai` 到 `bi` 的路径是一个由 **不同** 节点组成的序列，从节点 `ai` 开始，到节点 `bi` 结束，且序列中相邻的两个节点在树中共享一条边。

返回一个长度为 `m` 的数组 `answer` ，其中 `answer[i]` 是第 `i` 条查询的答案。

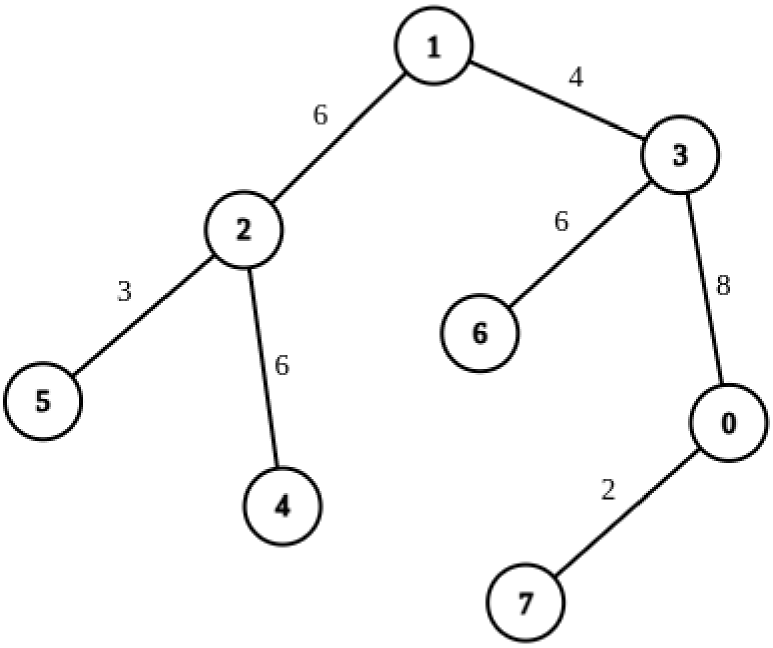
示例 1：



> 输入：n = 7, edges = [[0,1,1],[1,2,1],[2,3,1],[3,4,2],[4,5,2],[5,6,2]], queries = [[0,3],[3,6],[2,6],[0,6]]
> 输出：[0,0,1,3]
> 解释：第 1 条查询，从节点 0 到节点 3 的路径中的所有边的权重都是 1 。因此，答案为 0 。
> 第 2 条查询，从节点 3 到节点 6 的路径中的所有边的权重都是 2 。因此，答案为 0 。
> 第 3 条查询，将边 [2,3] 的权重变更为 2 。在这次操作之后，从节点 2 到节点 6 的路径中的所有边的权重都是 2 。因此，答案为 1 。
> 第 4 条查询，将边 [0,1]、[1,2]、[2,3] 的权重变更为 2 。在这次操作之后，从节点 0 到节点 6 的路径中的所有边的权重都是 2 。因此，答案为 3 。

> 对于每条查询 queries[i] ，可以证明 answer[i] 是使从 ai 到 bi 的路径中的所有边的权重相等的最小操作次数。

示例 2：



> 输入：n = 8, edges = [[1,2,6],[1,3,4],[2,4,6],[2,5,3],[3,6,6],[3,0,8],[7,0,2]], queries = [[4,6],[0,4],[6,5],[7,4]]
> 输出：[1,2,2,3]
> 解释：第 1 条查询，将边 [1,3] 的权重变更为 6 。在这次操作之后，从节点 4 到节点 6 的路径中的所有边的权重都是 6 。因此，答案为 1 。
> 第 2 条查询，将边 [0,3]、[3,1] 的权重变更为 6 。在这次操作之后，从节点 0 到节点 4 的路径中的所有边的权重都是 6 。因此，答案为 2 。
> 第 3 条查询，将边 [1,3]、[5,2] 的权重变更为 6 。在这次操作之后，从节点 6 到节点 5 的路径中的所有边的权重都是 6 。因此，答案为 2 。
> 第 4 条查询，将边 [0,7]、[0,3]、[1,3] 的权重变更为 6 。在这次操作之后，从节点 7 到节点 4 的路径中的所有边的权重都是 6 。因此，答案为 3 。
> 对于每条查询 queries[i] ，可以证明 answer[i] 是使从 ai 到 bi 的路径中的所有边的权重相等的最小操作次数。

提示：

- `1 <= n <= 104`
- `edges.length == n - 1`
- `edges[i].length == 3`
- `0 <= ui, vi < n`
- `1 <= wi <= 26`
- 生成的输入满足 `edges` 表示一棵有效的树
- `1 <= queries.length == m <= 2 * 104`
- `queries[i].length == 2`
- `0 <= ai, bi < n`

二、思路

以节点0为根节点，使用数组$count[i]$记录节点i到根节点0的路径上边权重的数量，$count[i][j]$表示节点i到根节点0的路径上权重为j的边数量。对于查询$queries[i]=[a_i,b_i]$，记节点lca_i为节点a_i与b_i的最近公共祖先，从节点a_i到b_i的路径上，权重为j的边数量
$t_j=count[a_i][j]+count[b_i][j]-2*count[lca_i][j]$，为了让节点$a_i$到$b_i$上每条边的权重都相等，则将路径上所有边都改为边数量最多的权重即可，则

$$res_i=\sum_{j=1}^{W}t_j-max_{1\leq j\leq W}t_j$$

由题意，W=26为表示权重的最大值。

三、代码

```
class Solution {
public:
    // 基于路径压缩查找的并查集
    class UnionFindSet {
    public:
        UnionFindSet(int n) : parents_(n) {
            for (int i = 0; i < n; ++i) {
                parents_[i] = i;
            }
        }

        // 基于路径压缩的查找
        int find(int x) {
            if (parents_[x] != x) {
                // 压缩子节点到根结点
                parents_[x] = find(parents_[x]);
            }
            return parents_[x];
        }

        // 合并sub结点的集合到master结点集合
        // 不能使用"带权并查集"优化,
        // 因为这里要保证并查集的根结点与数的祖先结点的一致性,要求sub合并到master而非相反方向
        void merge(int master, int sub) {
            int m_root = parents_[master], s_root = parents_[sub];
            if (m_root != s_root) {
                parents_[s_root] = m_root;
            }
        }
    private:
        vector<int> parents_;
    };

    void tarjan_lca(int node, int parent,
                    UnionFindSet& uf, vector<vector<int>>& count,
                    vector<vector<pair<int,int>>>& queries,
                    vector<bool>& visited,
                    vector<int>& lca,
                    vector<unordered_map<int,int>>& graph) {
        if (parent != -1) {
            count[node] = count[parent];     // 继承父结点的权重列表
            ++count[node][graph[node][parent]]; // 记录当前边的权重到列表
        }
```

```cpp
    visited[node] = true;   // 标记当前结点已访问
    // 遍历当前结点的每个子结点
    for (auto [dst, _]: graph[node]) {
        // 避免回路
        if (dst == parent) continue;
        // 递归tarjan
        tarjan_lca(dst, node, uf, count, queries, visited, lca, graph);
        // 并查集merge:将dst结点merge到node结点
        uf.merge(node, dst);
    }
    // 遍历与node结点相关的所有查询
    for (auto [dst, idx] : queries[node]) {
        if (visited[dst]) { // 如果query的另一个已经访问了,则可以求LCA
            // LCA即为dst结点的并查集根节点
            lca[idx] = uf.find(dst);
        }
    }
}

vector<int> minOperationsQueries(int n, vector<vector<int>>& edges,
                                 vector<vector<int>>& queries) {
    auto q_sz = queries.size();
    // 构建图(无向树) src-dst-weight
    vector<unordered_map<int,int>> graph(n);
    for (auto& edge: edges) {
        // 无向树需要两条边
        graph[edge[0]][edge[1]] = edge[2];
        graph[edge[1]][edge[0]] = edge[2];
    }

    // 按结点构造查询表 node1-node2-query_idx
    vector<vector<pair<int,int>>> query_pairs(n);
    for (int i = 0; i < q_sz; ++i) {
        // 需要构建对称的两个query
        auto& query = queries[i];
        query_pairs[query[0]].emplace_back(query[1], i);
        query_pairs[query[1]].emplace_back(query[0], i);
    }

    // 权重最大值
    constexpr int W = 26;
    // 记录0结点到每个结点的不同权重的边的列表
    vector<vector<int>> count(n, vector<int>(W+1));
    UnionFindSet uf(n);
    // 用于标记结点是否访问
    vector<bool> visited(n);
    // 记录每个query的两个结点的LCA
    vector<int> lca(q_sz);

    // tarjan算法求LCA
    tarjan_lca(0, -1, uf, count, query_pairs, visited, lca, graph);

    vector<int> res(q_sz, 0);
    for (int i = 0; i < q_sz; ++i) {
        int total = 0, maxx = 0;
        auto& query = queries[i];
        // 遍历每种数值的权重
        for (int w = 1; w <= W; ++w) {
```

```
                // query中两个结点权重为w的边的数目
                int t = count[query[0]][w] + count[query[1]][w] - 2 *
count[lca[i]][w];
                // 记录权重占比最多的边
                maxx = max(maxx, t);
                // 累加得到query中两个结点的路径的总边数
                total += t;
            }
            // 得到最小操作
            res[i] = total - maxx;
        }
        return res;
    }
};
```

复杂度分析

时间复杂度：$O((m+n) \times W + m \times logn)$，其中 n是节点数目，m 是查询数目，W是权重的可能取值数目。

空间复杂度：$O(n * W + m)$

四、LCA

LCA（最近公共祖先）是两个点在这棵树上距离最近的公共祖先节点，主要是用来处理当两个点仅有唯一一条确定的最短路径时的路径。下面介绍两种解法模板：

4.1 Tarjan

```
Tarjan(u)//marge和find为并查集合并函数和查找函数
{
    for each(u,v)    //访问所有u子节点v
    {
        Tarjan(v);        //继续往下遍历
        marge(u,v);    //合并v到u上
        标记v被访问过;
    }
    for each(u,e)    //访问所有和u有询问关系的e
    {
        如果e被访问过;
        u,e的最近公共祖先为find(e);
    }
}
```

4.2 倍增LCA

```
inline int lca(int u,int v)
{
    int deepu=deep[u],deepv=deep[v];
    if(deepu!=deepv)//先跳到同一深度
    {
        if(deep[u]<deep[v])
        {
            swap(u,v);
            swap(deepu,deepv);
```

```
        }
        int d=deepu-deepv;
        for(int i=0;i<=log2n;i++)
            if((1<<i)&d)u=fa[u][i];
    }
    if(u==v)return u;
    for(int i=log2n;i>=0;i--)
    {
        if(deep[fa[u][i]]<=0)continue;
        if(fa[u][i]==fa[v][i])continue;
        else u=fa[u][i],v=fa[v][i];//因为我们要跳到它们LCA的下面一层，所以它们肯定不相等，如果不相等就跳过去。
    }
    return fa[u][0];
}
```