3203. Find Minimum Diameter After Merging Two Trees

David Chen

2024 年 12 月 26 日

问题 5：合并两棵树后求最小直径

题目描述

给定两棵无向树，分别有 n 和 m 个节点，编号为 0 到 $n-1$ 和 0 到 $m-1$。给定两个二维整数数组 `edges1` 和 `edges2`，长度分别为 $n-1$ 和 $m-1$，其中 `edges1[i]` $= [a_i, b_i]$ 表示第一棵树中节点 a_i 和节点 b_i 之间有一条边，`edges2[i]` $= [u_i, v_i]$ 表示第二棵树中节点 u_i 和节点 v_i 之间有一条边。

你需要将第一棵树中的某个节点与第二棵树中的某个节点连接起来，形成一棵新的树。返回合并后树的最小直径。

树的直径定义为树中任意两点之间最长路径的长度。

原始解答分析

以下是用户提供的 C++ 代码实现：

```cpp
int minimumDiameterAfterMerge(vector<vector<int>> &edges1,
                              vector<vector<int>> &edges2) {
    long long n = edges1.size() + 1;
    long long m = edges2.size() + 1;
    vector<vector<int>> next1(n);
    vector<vector<int>> next2(m);
    unordered_map<long long, int> dp1;
    unordered_map<long long, int> dp2;
    for (auto &i: edges1) {
        next1[i[0]].push_back(i[1]);
        next1[i[1]].push_back(i[0]);
    }
    for (auto &i: edges2) {
        next2[i[0]].push_back(i[1]);
        next2[i[1]].push_back(i[0]);
    }
    int min1 = n - 1;
    int min2 = m - 1;
    int d1 = 0;
```

```
int d2 = 0;
for (int k = 0; k < n; ++k) {
    vector<bool> visited1(n, false);
    auto dfs1 = [&](auto &&dfs1, int node) -> int {
        int max_depth = 0;
        for (auto i: next1[node]) {
            if (visited1[i]) {
                continue;
            }
            visited1[i] = true;
            int tmp = 1;
            if (dp1.count(node * n + i))
                tmp += dp1[node * n + i];
            else
                tmp += (dp1[node * n + i] = dfs1(dfs1, i));
            d1 = max(d1, tmp + max_depth);
            max_depth = max(max_depth, tmp);
        }
        return max_depth;
    };
    visited1[k] = true;
    min1 = min(dfs1(dfs1, k), min1);
}
for (int k = 0; k < m; ++k) {
    vector<bool> visited2(m, false);
    auto dfs2 = [&](auto &&dfs2, int node) -> int {
        int max_depth = 0;
        for (auto i: next2[node]) {
            if (visited2[i]) {
                continue;
            }
            visited2[i] = true;
            int tmp = 1;
            if (dp2.count(node * m + i))
                tmp += dp2[node * m + i];
            else
                tmp += (dp2[node * m + i] = dfs2(dfs2, i));
            d2 = max(d2, tmp + max_depth);
            max_depth = max(max_depth, tmp);
        }
        return max_depth;
    };
```

```
            visited2[k] = true;
            min2 = min(dfs2(dfs2, k), min2);
        }
        return max(max(d1, d2), min1 + min2 + 1);
}
```

当前算法的思路

1. ** 构建邻接表 **：将两棵树的边信息转换为邻接表表示，分别存储在 `next1` 和 `next2` 中。
2. ** 计算直径 **：

- 对于每棵树，遍历每个节点，使用深度优先搜索（DFS）计算以该节点为根的子树的最大深度。
- 使用记忆化技术（通过 `dp1` 和 `dp2`）避免重复计算。
- 更新全局最大深度（即直径）。

3. ** 合并两棵树后的直径计算 **：最终返回两棵树直径的最大值与两棵树直径之和加一中的最大值。

优化建议

当前算法在计算两棵树的直径时存在以下问题：

1. **时间复杂度较高**：对于每个节点进行 DFS，整体时间复杂度为 $\mathcal{O}(n^2)$，其中 n 是节点数。
2. **空间复杂度高**：使用了两个 `unordered_map` 进行记忆化，增加了空间消耗。
3. **不必要的重复计算**：尽管使用了记忆化，但对于树这种结构，仍有优化空间。

以下将逐一分析并提出优化方案。

1. 使用双 BFS 计算直径

对于树结构，计算直径的高效方法是使用双广度优先搜索（BFS）：

1. 从任意节点开始 BFS，找到距离最远的节点 u。
2. 从节点 u 开始 BFS，找到距离 u 最远的节点 v，此时 u 到 v 的距离即为树的直径。

这种方法的时间复杂度为 $\mathcal{O}(n)$，远优于当前的 $\mathcal{O}(n^2)$。

2. 优化代码结构

通过重新组织代码结构，可以减少不必要的变量和数据结构的使用。例如，使用全局变量或传递引用来避免在递归中传递大量参数。

3. 避免使用 `unordered_map`

对于树这种结构，节点编号连续且有限，可以使用简单的数组来存储深度信息，代替 `unordered_map`，提高访问效率。

优化后的代码实现

基于上述优化建议，以下是优化后的 C++ 代码实现：

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

class Solution {
public:
    // BFS to find the farthest node and its distance from the start node
    pair<int, int> bfs(int start, const vector<vector<int>> &adj) {
        int n = adj.size();
        vector<int> dist(n, -1);
        queue<int> q;
        q.push(start);
        dist[start] = 0;
        int farthest_node = start;
        int max_dist = 0;

        while (!q.empty()) {
            int node = q.front(); q.pop();
            for (auto &neighbor : adj[node]) {
                if (dist[neighbor] == -1) {
                    dist[neighbor] = dist[node] + 1;
                    q.push(neighbor);
                    if (dist[neighbor] > max_dist) {
                        max_dist = dist[neighbor];
                        farthest_node = neighbor;
                    }
                }
            }
        }
        return {farthest_node, max_dist};
    }

    // Function to compute the diameter of a tree using double BFS
    int treeDiameter(const vector<vector<int>> &adj) {
        // First BFS to find one end of the diameter
        pair<int, int> first = bfs(0, adj);
        // Second BFS from the farthest node found in the first BFS
        pair<int, int> second = bfs(first.first, adj);
        return second.second;
```

```
}

int minimumDiameterAfterMerge(vector<vector<int>> &edges1,
                              vector<vector<int>> &edges2) {
    // Number of nodes in each tree
    int n = edges1.size() + 1;
    int m = edges2.size() + 1;

    // Build adjacency lists
    vector<vector<int>> adj1(n, vector<int>());
    vector<vector<int>> adj2(m, vector<int>());
    for (auto &edge : edges1) {
        adj1[edge[0]].push_back(edge[1]);
        adj1[edge[1]].push_back(edge[0]);
    }
    for (auto &edge : edges2) {
        adj2[edge[0]].push_back(edge[1]);
        adj2[edge[1]].push_back(edge[0]);
    }

    // Compute diameters of both trees
    int diameter1 = treeDiameter(adj1);
    int diameter2 = treeDiameter(adj2);

    // To minimize the diameter after merging,
    // connect the centers of both trees
    // However, for simplicity, we can consider
    // that the new diameter is the maximum of:
    // - diameter1
    // - diameter2
    // - (ceil(diameter1/2) + ceil(diameter2/2) + 1)
    // This is based on the property that connecting
    // two trees via their centers minimizes the diameter.

    // Find the radius (ceil(diameter / 2)) of both trees
    int radius1 = (diameter1 + 1) / 2;
    int radius2 = (diameter2 + 1) / 2;

    // The new diameter after merging
    int new_diameter = max({diameter1, diameter2, radius1 + radius2 + 1});

    return new_diameter;
```

```
    }
};
```

优化说明

1. 使用双 BFS 计算直径

- **第一步**：从任意节点（如节点 0）开始 BFS，找到距离起始节点最远的节点 u。
- **第二步**：从节点 u 开始 BFS，找到距离 u 最远的节点 v，此时 u 到 v 的距离即为树的直径。

这种方法的时间复杂度为 $\mathcal{O}(n)$，显著优于原算法的 $\mathcal{O}(n^2)$。

2. 简化数据结构

- 使用简单的数组 `vector<int>` 代替 `unordered_map`，提高访问效率。
- 通过直接传递引用，避免在递归中传递大量参数。

3. 减少不必要的变量

- 移除 `min1`, `min2`, `d1`, `d2` 等不必要的变量，简化逻辑。
- 直接通过计算半径来推导合并后的直径，减少计算步骤。

时间复杂度分析

优化后的算法主要包括以下部分：

1. **构建邻接表**：$\mathcal{O}(n+m)$，其中 n 和 m 分别是两棵树的节点数。
2. **计算直径**：每棵树使用双 BFS，时间复杂度为 $\mathcal{O}(n)$ 和 $\mathcal{O}(m)$。
3. **计算合并后直径**：$\mathcal{O}(1)$。

因此，总时间复杂度为 $\mathcal{O}(n+m)$，远优于原始算法的 $\mathcal{O}(n^2+m^2)$。

算法优缺点

优点

- **高效性**：时间复杂度由 $\mathcal{O}(n^2+m^2)$ 降低到 $\mathcal{O}(n+m)$，大幅提高了算法效率。
- **简洁性**：简化了代码结构，去除了不必要的变量和数据结构，使代码更易于理解和维护。
- **空间优化**：减少了空间占用，避免了使用 `unordered_map` 带来的额外空间开销。

缺点

- **适用性限制**：该优化方法基于树的性质，仅适用于无环连通图（树）。对于其他图结构，需采用不同的算法。
- **假设中心连接**：为了最小化合并后的直径，假设连接两棵树的中心节点。但在实际情况中，可能需要更复杂的处理以进一步优化。

总结

通过采用双 BFS 方法计算树的直径，并优化数据结构和算法逻辑，可以显著提升原始解答的效率和简洁性。优化后的算法不仅减少了时间和空间复杂度，还提高了代码的可读性和可维护性，是处理此类树结构问题的有效方法。