Small-To-Large Merging

ARPAN BANERJEE

January 2022

Contents

§1 k-length paths

> **Problem**
> Find the number of simple paths of length k in a tree of n nodes.

§1.1 $\mathcal{O}(nk)$

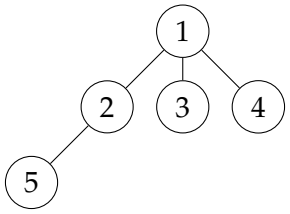
This can be done in $\mathcal{O}(nk)$ with dynamic programming. First root the tree arbitrarily. Let `paths[i][j]` be the number of paths of length j that start at node i and end somewhere within its subtree. Then for some node i, there are two types of k-length paths that could appear in its subtree:

1. i is the start/end of the path
2. i is in the middle of the path

Then

$$\texttt{paths[i][j]} = \sum_{c \in \texttt{children[i]}} \texttt{paths[c][j-1]}$$

and the answer can be incremented per the current `paths[i][j]` as follows:



Let $k = 3$ and let's say the DFS goes from left to right, starting at 1. Then if the left branch is done and we're back at 1, we start the middle branch. After the middle branch is done, we can search for paths of length k that are formed by the middle branch and the left branch. Similarly, when the right branch is done we search for paths of length k formed by the left two branches and the right branch. This procedure accounts for both of the above cases, as should be clear with the implementation below. This is in fact the intended solution for CF 161D.

```
#include <bits/stdc++.h>
#define int long long
using namespace std;

const int sz = 5e4 + 2;
vector<int> adj[sz];
int n, k, paths[sz][501], ans;  // paths has only endpoints

void dfs(int node, int par = -1){
        paths[node][0] = 1;
        for (int child : adj[node]){
                if (child == par) continue;
                dfs(child, node);

                for (int j = 0; j < k; j++){
                        ans += (paths[node][j] * paths[child][k - j - 1]);
                }

```

```
                for (int j = 1; j <= k; j++){
                        paths[node][j] += paths[child][j - 1];
                }
        }
}

signed main(){
        cin.tie(0)->sync_with_stdio(0); cin.exceptions(ios_base::failbit);
        cin >> n >> k;
        for (int i = 0; i < n - 1; i++){
                int u, v; cin >> u >> v;
                adj[u].emplace_back(v);
                adj[v].emplace_back(u);
        }

        dfs(1);
        cout << ans << endl;
}
```

§1.2 $\mathcal{O}(n \log n)$

Now onto the better solution. Let's store the depths of each subtree, where the depth of the respective root node is 0. For example, in subtree 2 in the above tree, the depth of node 2 is 0 and the depth of node 5 is 1. We can store a count `d` where `d[i][j]` is the count of depth j in subtree i.

Notice that the length of `d[i]` is bounded by the number of nodes in subtree i. We can build `d` from the bottom of the tree to the top, and we can attain `d[parent]` by combining `d` for its children. Though it may seem like it at first glance, this will not take $\mathcal{O}(n^2)$ due to efficient merging and since we are not storing independent `d` values; we are instead using them as the building blocks for later `d`'s.

When merging a `d[child]` into `d[parent]`, merge the smaller length into the larger. This takes $\mathcal{O}(n \log n)$.

§1.2.1 Proof

Assume `d[i]` has length p iff there are p nodes in subtree i. Let the smaller `d` have x elements, taking $\mathcal{O}(x)$ time to be merged into a larger `d`. The size of the resulting `d` is at least $2x$. So each x is involved in at most $\log n$ merges, resulting in $\mathcal{O}(n \log n)$.

```
#include <bits/stdc++.h>
#define sz(X) ((int)(X.size()))
#define eb emplace_back
#define int long long
using namespace std;

const int sz = 2e5 + 5;
int n, k, ans;
vector<int> adj[sz];
deque<int> d[sz]; // d[i][j] is how many depth j is there at subtree i

void merge(deque<int>& a, deque<int>& b){ // first parameter has the merged contents
```

```
        if (sz(a) < sz(b)) swap(a, b); // O(1)
        for (int i = 0; i < sz(b); i++)
                if (not ((k - i) < 0 or (k - i) >= sz(a)))
                        ans += (b[i] * a[k - i]);
        for (int i = 0; i < sz(b); i++) // merge
                a[i] += b[i];
}

void dfs(int node, int par = -1){
        d[node].push_front(1);
        for (int i : adj[node]){
                if (i == par) continue;
                dfs(i, node);
                d[i].push_front(0);
                merge(d[node], d[i]);
        }
}

signed main(){
        cin.tie(0)->sync_with_stdio(0); cin.exceptions(ios_base::failbit);

        cin >> n >> k;
        for (int i = 0; i < n - 1; i++){
                int u, v; cin >> u >> v;
                adj[u].eb(v); adj[v].eb(u);
        }

        dfs(1);
        cout << ans << endl;
}
```

§1.3 Other Solutions

You can use centroid decomposition for $O(n \log n)$. If the problem asks for the answer for k from 1 to n, you can do this in subquadratic time with fast fourier transform. That problem can be found here.