Pancakes - Editorial

Timothy Gao

Essentially, we are given a tree of $N - 1$ nodes ($1 \leq N \leq 4 * 10^5$) rooted at node 1. Each node takes on a bit value of 0 or 1 at the start. In one move, we can select a node i and flip the bit values of all the nodes in its subtree (including i) for a cost of $cost[i]$. Not all nodes are able to be selected and flipped. We are tasked with finding the maximum number of nodes that can have bit value of 1 after some sequence of flips, as well as the minimum cost that this takes.

1 Initial Observations

Let's focus on each node individually. Because each flip only affects a subtree, the bit value of each node is independent of the flips of nodes in its subtree. In fact, the node's bit value will be strictly dependent on the total number of flips of its ancestors (which include its parent, parent of parent, etc. and must include node 1) and its original value.

Now we may consider some kind of dynamic programming solution, $dp[i][j]$, where i indicates the current node and j is how many times its ancestors are flipped. j uniquely determine the bit value of node i. Specifically, $val[i] = (val_{initial}[i] + j) \mod 2$. This DP runs in $O(N^2)$, which is well over the time limit. One important observation to make here is that number of flips is cyclic as apparent from the mod, so we really only care about the parity of j. This comes from the fact that flipping a bit twice is equivalent to not flipping at all.

2 Arriving at the Solution

So now we have the motivation for our DP state, $dp[i][j]$, where i is the current node and j is the parity of the number of times its ancestors nodes are flipped. Our DP also stores a pair, $(ones, cost)$ which indicates the maximum ones we can attain at this statement, then the minimum cost of doing so. The transition for this state is also fairly intuitive. First we consider not flipping the current node. The result from this would be $\sum_{k=1}^{n} dp[k][j]$ over all the children of i (k is each child, n is the number of children). If we have the option to flip the current node, then we consider flipping it. This yields $\sum_{k=1}^{n} dp[k][(j+1) \mod 2]$, and we also add $cost[i]$ of flipping the current node to the second value in the resulting pair. Now, we simply select the best option between not flipping and flipping if we are able to (the *best* function in my code). This algorithm yields a final time complexity of $O(N)$, with $O(N)$ states and $O(1)$ transitions.

3 Code

```
//@timothyg

#include <bits/stdc++.h>

using namespace std;

typedef long long ll;
typedef pair<int, ll> pii;

#define pb push_back

#define f first
```

```cpp
#define s second

const int maxn = 400005;

int cost[maxn], init[maxn];
pii dp[maxn][2];
vector<int> adj[maxn];

//Select best cost: we wish to maximize ones, then minimize the cost
pii best(pii a, pii b){
    if(a.f == b.f){
        return (a.s < b.s ? a : b);
    }
    return (a.f < b.f ? b : a);
}

pii dfs(int u, int p = -1, int mode = 0){
    if(dp[u][mode] != pii(-1, -1)) return dp[u][mode];
    pii flip(init[u] ^ mode ^ 1, cost[u]);
    pii noflip(init[u] ^ mode, 0);
    for(int n : adj[u]){
        if(n == p) continue;
        pii flipped = dfs(n, u, mode ^ 1);
        pii noflipped = dfs(n, u, mode);
        flip.f += flipped.f; flip.s += flipped.s;
        noflip.f += noflipped.f; noflip.s += noflipped.s;
    }
    if(cost[u] == -1){ //are we able to flip this node?
        dp[u][mode] = noflip;
    }else{
        dp[u][mode] = best(flip, noflip);
    }
    return dp[u][mode];
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(0);
    int N; cin >> N;
    string str; cin >> str; //indicating which bulbs are initially turned on
    for(int i = 0; i<N; i++){
        init[i] = str[i] == '1';
        dp[i][0] = dp[i][1] = pii(-1, -1); //this is a marker that the dp state hasn't been
            visited
        cin >> cost[i]; //indicating the cost to flip the switch at node i+1, -1 means no
            switch
    }

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

    pii ret = dfs(0);
    cout << ret.f << " " << ret.s << '\n';
    return 0;
}
```