Contents

树上启发式合并（dsu on tree、静态链分治）

> 快速了解：
>
> 针对问题：询问支持离线（没有修改），问题转化后发现询问只与子树有关（eg 统计树上一个节点的子树中具有某种特征的节点数。）
>
> 可以很方便地在 $O(n\log n)$ 的时间内完成。
>
> 提示：这里的dsu和并查集（Disjoint Set Union）没有没有没有半毛钱关系。

再看启发式

并查集按秩合并：小的并到大的，也叫启发式合并。

splay合并：小的并到大的。

通过小向大合并操作，让大的集体的信息尽可能的少维护，起到降低算法复杂度，优化算法的作用。

> 其实启发式就是一种朴素的优化。——czc

这个算法是怎么想出来的?

先看一个案例帮助理解：

【题目】

CF600E Lomsat gelral

大意：给出一棵树，每个节点有颜色，询问一些子树的颜色数量（颜色可重复）。

【思路】

遇事不决先想暴力：~~树套树、树上莫队等数据结构~~

再暴力一点：对每个节点都统计一遍颜色数量，那么n个节点的复杂度为n^2

暴力可以一口气先把树上所有节点数据都算出来，再回答问题。（虽然时间肯定不够）

发现父亲节点信息来源于子树信息。如何加速?

【尝试多维护一点信息】

直接对每个节点维护一个$check_i$ 数组表示节点i所在子树的颜色（具体是哪几种），cnt_i维护颜色数量

先递归到叶子节点，再在回溯的时候维护信息（合并当前fa的所有子树的check，根据fa的check算·出cnt）。

【问题】空间开支和时间开支不是很美妙

原因在于每个节点都要开一个check数组，并且每一对父子之间的check信息在合并的时候开销是O(n)的，怎么办?

【资源复用】

如何复用check数组?

有一个暴力的想法：

```
//来源说明，cf上关于dsu on tree的教程文件原文代码。（有删改）
int check[maxn];
void add(int v, int p, int x){
    check[ col[v] ] += x;
    for(auto u: g[v])//提示一下：这是c++11后的c++语法特性，意会为for(int u=g[v].front;u<=g[v].bound;u=g[v].next)
        if(u != p)
            add(u, v, x)
}
void dfs(int v, int p){
    add(v, p, 1);
    //now cnt[c] is the number of vertices in subtree of vertice v that has color c. You can answer the queries easily.
    add(v, p, -1);
    for(auto u : g[v])
        if(u != p)
            dfs(u, v);
}
```

但是这样仍然有很低效无用的资源统计，需要更高效率的add过程。

通过思考，，，，，

对于节点i，其`儿子数量最多的那个节点`的check数组保留，`其他儿子的check信息`暴力统计和合并到保留的check数组后撤销影响。

【具体做法】

1. 优先递归所有轻儿子，计算轻儿子的ans，然后消除递归的影响。
2. 递归重儿子，计算重儿子ans，将子树信息合并到根上，不消除递归影响。
3. 再暴力统计一遍轻儿子对于节点i的答案贡献，将子树信息合并到根上。
4. 判断是否上传信息，更新节点i的ans

【核心代码】

```
//来源说明，cf上关于dsu on tree的教程文件原文代码。（有删改）
int check[maxn];
bool big[maxn];
void add(int v, int p, int x){
    check[ col[v] ] += x;
    for(auto u: g[v])
        if(u != p && !big[u])
            add(u, v, x)
}
void dfs(int v, int p, bool keep){
    int mx = -1, bigChild = -1;
    for(auto u : g[v])
       if(u != p && sz[u] > mx)
          mx = sz[u], bigChild = u;
    for(auto u : g[v])
        if(u != p && u != bigChild)
            dfs(u, v, 0);  // run a dfs on small childs and clear them from check
    if(bigChild != -1)
        dfs(bigChild, v, 1), big[bigChild] = 1;  // bigChild marked as big and not cleared from cnt
    add(v, p, 1);
    //now check[c] is the number of vertices in subtree of vertice v that has color c. You can answer the queries easily.
    if(bigChild != -1)
        big[bigChild] = 0;
    if(keep == 0)
        add(v, p, -1);
}
```

其中最核心的代码在于这一段（用伪代码书写，方便理解）

```
dfs( x,  fa,  opt){
    for(all Edge):
        if (to == fa) : continue
        if (to !=BigSon[x]) : dfs(to,x,0) //暴力计算轻边贡献。
    if (son[x]) : dfs(BigSon[x],x,1) //统计重儿子信息，不撤销影响。
    add(x) //暴力统计
    ans[x]=NowAns
    if !opt : delet(x) //撤销影响
}
```

这个就是树上启发式合并。

简单来说：只把轻儿子的信息向重儿子的信息维护。

由于不停的向重儿子的check数组上合并信息，对于不同子树但是有父子关系的重儿子们check数组可以共用。（czc：也许这里需要一个图解释）

【复杂度分析】

遍历2次轻儿子，1次重儿子就可以获得整个子树的答案。

根据重剖性质：每个点到根的路径上最多经过 logn条边

> 每个节点合并到父亲节点后，子树大小至少增加两倍，总大小只有n，所以不能添加一个节点超过$O(logn)$次

所以时间复杂度为：

1. 递归轻子树，消除影响——$O(logn)$
2. 递归重子树，不消除影响，将信息合并到根上——$O(1)$
3. 再递归轻子树，将信息合并到根上——$O(logn)$
4. 判断是否上传信息

一共有$O(n)$个节点，所以复杂度为$O(nlogn)$

代码实现

```
#include <iostream>
#include <cstring>
#include <algorithm>
#include<vector>

#define LL long long
```

```cpp
using namespace std;
const int MAXN = 1e5 + 10;
inline int read() {
    char c = getchar(); int x = 0, f = 1;
    while(c < '0' || c > '9') {if(c == '-') f = -1; c = getchar();}
    while(c >= '0' && c <= '9') x = x * 10 + c - '0', c = getchar();
    return x * f;
}
int N, col[MAXN], son[MAXN], siz[MAXN], check[MAXN], Mx, Son;
//col数组存储每个节点的颜色。
//check 记录每种颜色的出现次数
//son 用来存储当前节点的重儿子。

LL sum = 0, ans[MAXN];
// sum用来存储出现次数最多的颜色的编号和
// ans用来存储每个节点的答案。

vector<int> v[MAXN];

//计算每个节点的子树大小和重儿子
void dfs(int x, int fa) {
    siz[x] = 1;
    for(int i = 0; i < v[x].size(); i++) {
        int to = v[x][i];
        if(to == fa) continue;
        dfs(to, x);
        siz[x] += siz[to];
        if(siz[to] > siz[son[x]]) son[x] = to;//轻重链剖分
    }
}

//递归处理这颗子树，val为1表示统计，-1表示撤回影响（消去影响）
//更新check, Mx和sum
//Mx 存储当前出现次数最多的颜色的出现次数
//sum用来存储出现次数最多的颜色的编号和
//col数组存储每个节点的颜色。
void add(int x, int fa, int val) {
    check[col[x]] += val;//根据题目要求统计颜色贡献。
    if(check[col[x]] > Mx) // 记录当前子树颜色最多的的那个颜色。
        Mx = check[col[x]], sum = col[x]; //注意是编号和
    else if(check[col[x]] == Mx) sum += (LL)col[x];

    for(int i = 0; i < v[x].size(); i++) {
        int to = v[x][i];
        if(to == fa || to == Son) continue;
        add(to, x, val);
    }
}

```

```
//树上启发式合并核心函数：通过add函数统计每个节点的答案，并根据opt决定是否消除对当前节点的影响
//处理节点x及其子树信息
void dfs2(int x, int fa, int opt) {
    for(int i = 0; i < v[x].size(); i++) {
        int to = v[x][i];
        if(to == fa) continue;
        if(to != son[x])
            dfs2(to, x, 0);//暴力统计轻边的贡献，opt = 0表示递归完成后消除对该点的影响
    }
    if(son[x]) //统计重儿子的贡献，不消除影响
        dfs2(son[x], x, 1), Son = son[x];
    add(x, fa, 1);
    Son = 0;//暴力统计所有轻儿子的贡献

    ans[x] = sum;//更新答案
    if(!opt) add(x, fa, -1), sum = 0, Mx = 0;//轻边的贡献要删除。
}
int main() {
    N = read();
    for(int i = 1; i <= N; i++) col[i] = read(); //读入节点颜色
    for(int i = 1; i <= N - 1; i++) {
        int x = read(), y = read();
        v[x].push_back(y); v[y].push_back(x);
    }
    dfs(1, 0); //处理子树大小、重儿子
    dfs2(1, 0, 0);//树上启发式合并
    for(int i = 1; i <= N; i++) printf("%I64d ", ans[i]);
    return 0;
}

```

【实现提示】

除了上面题目的公用check外，还可以给节点挂一个vector，那么在父子check信息传递的时候，可以通过指针交换在O（1）实现信息的流转。（如果你看不懂这句话，let it go.）

应用场景

询问支持离线（没有修改）, 问题转化后发现询问只与子树有关。

在一些静态树上问题可以骗分，供考试时间紧张/读题后发现自己一窍不通时使用

> 可以乱搞过一些静态树套树的题 并且 $O(n\log n)$ 吊打树上莫队 $O(n\sqrt{n})$, 所以有必要掌握。——xzh

变式： 阔力梯的树

（请配合xzh讲解的视频自主阅读）

给你一棵 n 个节点的树, 求每个节点的"结实程度"

一个节点的结实程度定义为：以该节点为根的子树里所有节点的编号，从小到大排列后, 相邻编号的平方和。

假设一个节点的子树中所有节点编号排序后构成的序列为 $a_1, a_2, \ldots a_k$,那么答案为

$$\sum_{i=1}^{k-1} \left(a_{i+1} - a_i\right)^2 \tag{1}$$

先考虑一个暴力的想法

用一个 set 维护子树内编号, 然后做 dsu on tree, 删除时清空, 统计时遍历 时间复杂度为 $O\left(n^2 \log^2 n\right)$

然后考虑优化

发现瓶颈在于每次统计时的遍历, 我们其实可以在插入时统计答案

考虑每插入一个编号 i 带来的贡献

1. 如果 i 是 set 中的最值, 那么可以直接统计贡献
2. 否则减去 i 在 set 中前驱后继产生的贡献, 再加上 i 与前驱, i 与后继带来的贡献 这样时间复杂度为 $O\left(n \log^2 n\right)$

练习题

FZUOJ3666 CF600E Lomsat gelral

树的节点有颜色，一种颜色占领了一个子树，当且仅当没有其他颜色在这个子树中出现得比它多。求占领每个子树的所有颜色之和。

FZUOJ5378 阔力梯的树

请见视频讲解

FZUOJ5492 CF570D Tree Requests

请见题面中文翻译

FZUOJ4927 CF741D Arpa’s letter-marked tree and Mehrdad’s Dokhtar-kosh paths

给一棵树，每个节点的权值是'a'到'v'的字母，每次询问要求在一个子树找一条路径，使该路径包含的字符排序后成为回文串。

因为是排列后成为回文串，所以一个字符出现了两次相当于没出现，也就是说，这条路径满足最多有一个字符出现奇数次

正常做法是对每一个节点dfs，每到一个节点就强行枚举所有字母找到和他异或后结果为1的个数<1的路径，再去最长值，这样 $O(n^2logn)$ 的，可以用dsu on tree优化到 $O(nlog^2n)$

拓展阅读——启发式合并

在算法竞赛中，启发式合并主要应用于解决一些与数据结构相关的问题，特别是那些涉及到维护、查询一些集合或子集属性的问题。这类问题往往要求高效地处理一些复杂操作，如合并两个集合、查询集合中的某种属性等。

下面是一些具体的应用例子：

1. **离线处理询问**：有时，我们需要在给定的树或图结构中，针对一些特定的查询进行快速的离线处理。比如，给定一个树，然后进行一些类型为"将一个节点添加到集合中"和"询问集合中的某种属性"的操作。启发式合并可以帮助我们将复杂度从$O(n^2)$降低到O(n log n)或者O(n)。
2. **动态树问题**：动态树问题通常涉及到在一棵树中进行一些插入、删除、修改等操作，并查询一些性质，如路径上的最大值，子树的信息等。启发式合并能高效地解决一些动态树问题。
3. **处理DSU（Disjoint Set Union）问题**：启发式合并常常用于处理并查集问题，尤其是那些涉及到需要合并大量小集合到大集合，或者从大集合中删除小集合的问题。启发式合并的思想可以保证这种合并的复杂度较低。
4. **处理一些图论问题**：启发式合并也常常用于处理一些图论问题，如求解图中的最大独立集，最小点覆盖等问题。

> hint splay 也可以启发式，但是用脑袋想一想，就会发现这个常数比较大。

refer

https://codeforces.com/blog/entry/44351

https://www.cnblogs.com/Aftglw/p/15880060.html

https://baijiahao.baidu.com/s?id=1613444794783555531&wfr=spider&for=pc

https://www.cnblogs.com/zwfymqz/p/9683124.html