树形DP

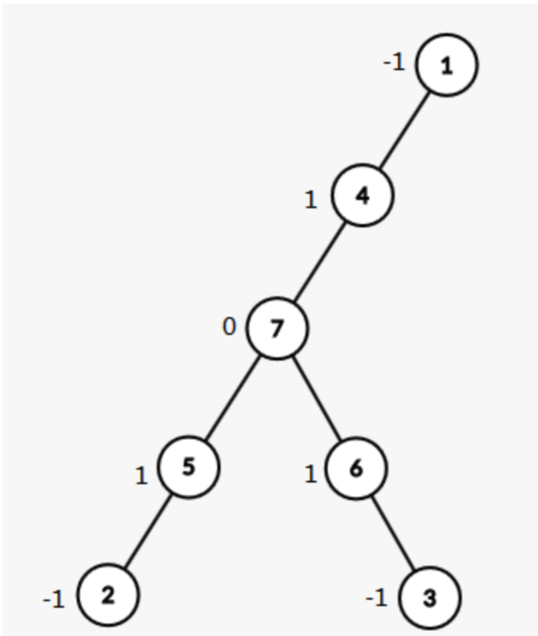
顾名思义，就是在树上做dp这个过程，根据树的特点，我们很容易想到树形dp的转移方程和状态应该是和树上结点和结点之间的关系有关，那么，在树形动态规划当中，我们一般先算子树再进行合并，都是先遍历子树，遍历完之后将子树的值传给父亲。简单来说我们动态规划的过程大概就是访问所有子树，再在根上合并，所以在子问题的解决上，我们需要用到递归。

对于每个节点 u，先递归它的子节点v，**回溯时，从子节点v向父节点u进行状态转移**

P1122 最大子树和

给定一棵N个节点的树，点有点权，点权有正有负，求这棵树的连通块的最大权值之和是多少。

这道题很水，但它能很好说明树形dp的执行过程



对于每个连通块，我们可以把他看成由一个节点和这个节点的子树所组成，那么我们可以计算每个结点和它的子树的连通块的最大权值之和，我们可以用dp来处理，将根节点的值算出来后，问题就解决了。

dp[i]表示结点i和它的子树的连通块的最大权值之和。
dp方程非常简单：

$$dp[i] = \sum_{k=0}^{cnt[i]-1} max(0, dp[son[k]]) + w[i]$$

即：
选择最大值>0的子树。

重点在于代码的实现：
观察dp方程，我们需要子树的信息，那就要先处理子树，所以在dp过程中，我们需要递归：

```
void DP(int p,int fa){
    for(int i=head[p];i;i=edge[i].nxt){
        if(edge[i].v != fa && edge[i].v){
            DP(edge[i].v,p);
            dp[p] += max(0,dp[edge[i].v]);
        }
    }
    dp[p]+=b[p];
    return ;
}
```

建树最好用前向星见图，其他就是常规操作了：

```
#include<bits/stdc++.h>
using namespace std;
int n;
int b[100000];
int head[100000];
int cnt;
int dp[100000];
int maxx=-1e9-7;
struct Edge{
    int v;
    int nxt;
}edge[100000];
void add(int u,int v){
    edge[++cnt].v = v;
    edge[cnt].nxt = head[u];
    head[u] = cnt;
}
void DP(int p,int fa){
    for(int i=head[p];i;i=edge[i].nxt){
        if(edge[i].v != fa && edge[i].v){
            DP(edge[i].v,p);
            dp[p] += max(0,dp[edge[i].v]);
        }
    }
    dp[p]+=b[p];
    return ;
}
int main(){
    cin >> n;
    for(int i=1;i<=n;i++){
        cin >> b[i];
    }
    for(int i=1;i<n;i++){
        int u,v;
        cin >> u >> v;
        add(u,v);
        add(v,u);
    }
    DP(1,0);
    for(int i=1;i<=n;i++){
        maxx = max(dp[i],maxx);
    }
    cout << maxx;
```

```
    return 0;
}
```

树的重心:

n 个结点的无根树，找到一个结点 u ， 删除结点 u 后，最大的连通块的结点数最小，则节点 u 就为重心。

给定n(2≤n≤10000)个点的树，输出重心的节点编号。

需要知道删除节点 u 后，剩下连通块的数量，即节点 u 的孩子节点 v 为根的子树节点数。

设 **f[u]** 表示节点 u 为根的子树的节点数量，初始时全为1。

设节点 u 的孩子节点为 v ， 则:

$$f[u] = f[u] + \sum f[v]$$

设 **mx[u]** 表示删除节点 u 后，最大的连通块数量。

$$mx[u] = max(f[v], n - f[u])$$

最小的 mx[u] 就是重心节点。

通过一次DFS，可以求解出 f、mx 数组，时间复杂度为O(n)。

树的最长路径:

以节点u为起点的最长路径有两种情况:
1.①走到节点 u 为子树的叶子节点的最远距离

2.①通过父节点走到其他点的最远距离

d(u,0):表示节点u走到子树节点的最远距离。

d(u,1):表示节点u通过父节点走到其他点的最远距离。

d(u,0) = max(dp(v,0)+w(u,v),dp(u,0))往子节点所走的路径，肯定也要从子节点自身往下走的转移

d(u,1)则有两种方案，到达u后，有可能到另外的子树去，也可能到父节点的父节点去，可是，我们不能确定dp(u,0)一定是到达其他子树的也有可能选择我当前这个点，为了避免这种状态，我们需要增加状态:
d(u,2)，表示节点u**走到子树节点的次远距离**，我们可以在求解d(u,0)时顺带求解。

这个转移是这样的:
1.d(v,0)+w(u,v) > d(u,0):d(u,2)=d(u,0),d(u,0)=d(v,0)+w(u,v);

2.d(v,0+w(u,v)<d(u,0)):d(u,2)=max(d(u,v),d(v,0+w(u,v));

那么d(u,1)的转移是这样的:

d(u,0) = d(v,0)+w(u,v): d(v,1)=max(dp(u,2),d(u,1))+w(u,v)

else: d(v,1)=max(dp(u,0),d(u,1))+w(u,v)

注意，这两次转移的顺序是不一样的，要根据所需要的的信息判断，所以进行两次DFS即可

[CTSC1997] 选课

在大学里每个学生，为了达到一定的学分，必须从很多课程里选择一些课程来学习，在课程里有些课程必须在某些课程之前学习，如高等数学总是在其它课程之前学习。现在有 N 门功课，每门课有个学分，每门课有一门或没有直接先修课（若课程 a 是课程 b 的先修课即只有学完了课程 a，才能学习课程 b）。一个学生要从这些课程里选择 M 门课程学习，问他能获得的最大学分是多少？

根据每门课只有0或1门选修课，可以看出课程与课程的关系像一棵树，但是，这里有很多棵树，我们可以再建立一个节点作为他们的根来方便处理，这个点本身没有意义。

我们要 求的就是以这个点为根的树所获得的最大学分

那么就可以用树形dp来解决

设dp(i,j,k)表示选择课程i和在它前j个子节点中选择k门课程

转移方程：

$$dp[now][j][k] = max(dp[now][j-1][k], dp[son][son\text{的节点数}][l] + dp[now][j-1][k-l]);$$

而j其实可以借助背包的滚动数组优化掉：

$$dp[now][j] = max(dp[now][j], dp[v][k] + dp[now][j-k])$$

转移的时候要倒着来

代码：

```
#include<bits/stdc++.h>
using namespace std;
int n,m,dp[1005][1005],head[1005],cnt;
struct Edge{
    int v;
    int nxt;
}edge[1005];
void add(int u,int v){
    edge[++cnt].nxt=head[u];
    edge[cnt].v=v;
    head[u]=cnt;
}
void f(int now){
    for(int i=head[now];i;i=edge[i].nxt){//枚举子树
        int v=edge[i].v;
        f(v);
        for(int j=m+1;j>=1;j--){//枚举选择课程数量
            for(int k=0;k<j;k++){//枚举在当前子树选择多少个节点
                dp[now][j]=max(dp[now][j],dp[v][k]+dp[now][j-k]);//滚动数组的优化
            }
        }
    }
}
int main(){
    cin >> n >> m;
    for(int i=1;i<=n;i++){
        int fa;
        cin >> fa >> dp[i][1];
        add(fa,i);
    }
    f(0);
```

```
    printf("%d\n",dp[0][m+1]);
    return 0;
}
```

P1352 没有上司的舞会

某大学有 n 个职员，编号为 1~n。

他们之间有从属关系，也就是说他们的关系就像一棵以校长为根的树，父结点就是子结点的直接上司。

现在有个周年庆宴会，宴会每邀请来一个职员都会增加一定的快乐指数 r_i，但是呢，

如果某个职员的直接上司来参加舞会了，那么这个职员就无论如何也不肯来参加舞会了。

所以，请你编程计算，邀请哪些职员可以使快乐指数最大，求最大的快乐指数。

第i个职员来的话,他的下属一定不能来

第i个职员不来的话，下属就可来可不来

对于一个以i为根的子树，答案分两种情况，就是根节点有没有使用。

所以状态就可以设为dp(i,0/1)表示以i为根的子树根是否使用时的最大值

方程就按照上面的描述推就可以了

```
#include<bits/stdc++.h>
using namespace std;
int n;
int r[10000];
int root;
vector<int> son[10000];
int f[10000][2];
int vis[10000];
void dp(int p){
    for(int i=0;i<son[p].size();i++){
        dp(son[p][i]);
        f[p][0] += max(f[son[p][i]][1],f[son[p][i]][0]);
        f[p][1] += f[son[p][i]][0];
    }
    f[p][1]+=r[p];
    return ;
}
int main(){
    cin.tie(0);
    cout.tie(0);
    cin >> n;
    for(int i=1;i<=n;i++){
        cin >> r[i];
    }
    for(int i=1;i<n;i++){
        int l,k;
        cin >> l >> k;
        son[k].push_back(l);
        vis[l] = 1;
    }
```

```
    for(int i=1;i<=n;i++){
        if(!vis[i]){
            root=i;
            break;
        }
    }
    dp(root);
    cout << max(f[root][0],f[root][1]);
    return 0;
}
```

换根DP

就是树形DP中的一种。

它的核心在于不同根的信息的转移和关系。

[[POI2008] STA-Station]

给出一个 N 个点的无根树,找出一个点来,以这个点为根的树时,所有点的深度之和最大（N<=1000000)

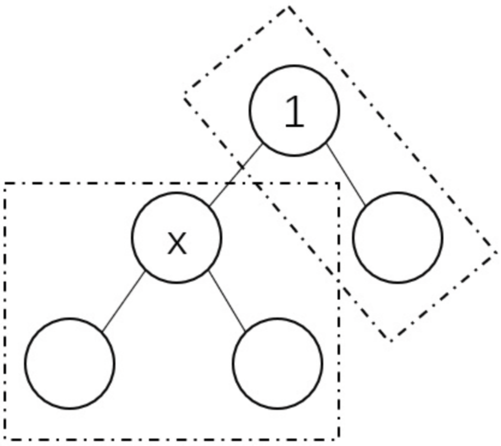
如果用普通的树形DP感觉并不好做，可以先想想暴力的思路看

暴力的思路就是对每一个结点都进行搜索，在搜索的时候统计深度，这样下来时间复杂度是O(n^2)

借助递推的思想，我们看看每个不同的深度之间有没有什么联系，也就是搜索一次便可以得到所有答案：

我们假设第一次搜索时的根节点为1号节点，则此时只有1号节点的答案是已知的。同时第一次搜索可以统计出所有子树的大小。

我们假设此时将根节点换成节点x，x节点为1号节点的子节点，则其子树由两部分构成，第一部分是其原子树，第二部分则是1号节点的其他子树（如下图）。



可以发现，这个形状的树由两个部分组成，一个是x的子树，一个是1号节点的其他子树

观察可以发现，第一个部分的子树中的每一个节点的深度-1，第二部分的子树中的每一个节点深度+1，有了这样的规律，我们就可以递推了！

设以i为根节点的答案是ans[i]，以i为根节点的子树大小是size[i]，那么整个树的大小就是size[1]，则递推公式为：

$$
\begin{aligned}
& ans[v] \\
& = ans[u] - size[v] + (size[1] - size[v]) \\
& = ans[u] - 2 * size[v] + size[1]
\end{aligned}
$$

那么问题就解决了。

```
#include<bits/stdc++.h>
using namespace std;
typedef long long int ll;
int n;
int head[1000006],cnt;
ll ans[1000006],siz[1000006],dep[1000006];
struct Edge{
    int v;
    int nxt;
}edge[2000006];
void add(int u,int v){
    edge[++cnt].nxt = head[u];
    edge[cnt].v = v;
    head[u] = cnt;
}
void dfs1(int node,int fa){
    dep[node]=dep[fa]+1;//节点的深度
    siz[node] = 1;//siz记录子树大小
    for(int i=head[node];i;i=edge[i].nxt){
        int v=edge[i].v;
        if(v==fa) continue;
        dfs1(v,node);
        siz[node] += siz[v];
    }
    return ;
}
void dfs2(int u,int fa){
    for(int i=head[u];i;i=edge[i].nxt){
        int v=edge[i].v;
        if(v == fa) continue;
        ans[v] = ans[u]+siz[1]-2*siz[v];
        dfs2(v,u);
    }
}
//注意搜索顺序
ll res=-1e9,tmp=-1e9;
int main(){
    scanf("%d",&n);
    for(int i=1;i<n;i++){
        int u,v;
        scanf("%d %d",&u,&v);
        add(u,v);
        add(v,u);
    }
    dfs1(1,0);
    for(int i=1;i<=n;i++){
        ans[1]+=dep[i];
    }//统计根的答案
    dfs2(1,0);//递推求解
    for(int i=1;i<=n;i++){
        if(ans[i]>tmp){
```

```
            res = i;
            tmp = ans[i];
        }
    }
    printf("%d",res);
    return 0;
}
```

由此我们可以大概看出换根DP的套路：

1，指定某个节点为根节点。

2，第一次搜索完成预处理（如子树大小等），同时得到该节点的解。

3，第二次搜索进行换根的动态规划，由已知解的节点推出相连节点的解。

有点类似于将dp的转移变成找规律（？），就不是根据子问题来推导了。

CF1324F Maximum White Subtree

- 给定一棵n个节点无根树，每个节点有一个颜色Au 0/1,表示白色/黑色，对于每个节点u，选择一个包含u的连通子图,设白点个数=cnt1,黑点个数=cnt2，使得cn1-cnt2最大化，输出这个值
- 1<=n<=200000

按照上述思路来分析，这个很显然是换根DP，我们先考虑对于单个节点要求什么，再思考节点之间的关系

考虑将问题细化，一个包含连通子图可以至多由两个部分组成，一个由u的子树组成，一个由他的父节点的子图，而他的父节点的子图有可能包含了u的子树，还需要分情况讨论。

对于u的子树，我们可以dp求解他的max(cnt1-cnt2)

那么他父节点的子图是否包含u的子树就可以通过u子树的max(cnt1-cnt2)判断了。

任意设一个点为根，第一次dfs求解dp数组（类似于最大子树和），dp[root]就是root的答案，第二次dfs递推求解每个节点的答案，输出即可。