# **树的直径**

## **定义：**

一棵树的直径就是这棵树上存在的最长路径。

给定一棵树，树中每条边都有一个权值，树中两点之间的距离定义为连接两点的路径边权之和。树中最远的两个节点之间的距离被称为树的直径，连接这两点的路径被称为树的最长链。后者通常也可称为直径，即直径是一个数值概念，也可代指一条路径。

## **求法：**

### **一、树形dp**

时间复杂度：O( n )；

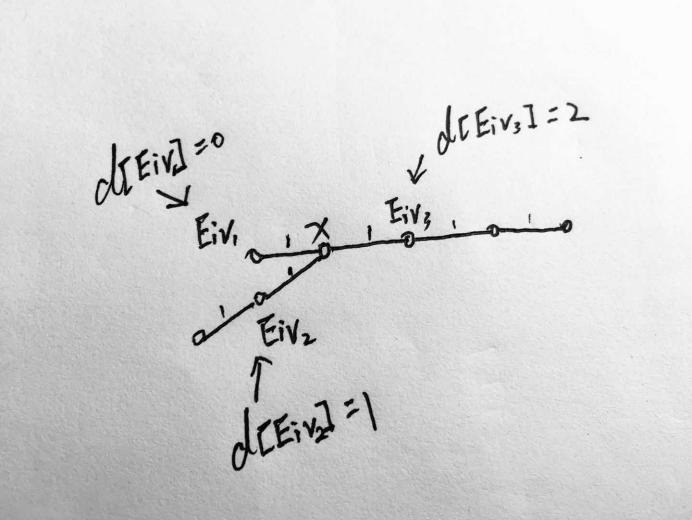
优点：代码量少实现方便。

不足：不容易记录路径。

#### **实现过程：**

状态：d[ x ] 以当前结点 x 为根的 子树的直径。

我们枚举每一个结点 x 以及 它要到达的下一个结点 Eiv。



这两个结点所能达到的最大距离之和 加上 这两个结点的边权就有可能去更新树的直径。

即：树的直径 ans\_max = max{ d[ x ] + d[ Eiv ] + edge[x, Eiv] } （1 <= x <= N）

那么 d[ x ] 通过什么更新呢？当然是由 它所连接下一个结点所能达到最大距离 来更新了；

即 d[ x ] = max{ d[ x ], d[ Eiv ] + edge[ x, Eiv ] }；

核心代码：

### void dp(int st)

### {

### vis[st] = true; //当前结点已访问

### for(int i = head[st]; i != -1; i = edge[i].nxt){

### int Eiv = edge[i].v;

### if(vis[Eiv]) continue; //不走回头路

### dp(Eiv);

### ans\_max = max(ans\_max, d[st] + d[Eiv] + edge[i].w); //更新树的直径（由当前结点两段之和更新）

### d[st] = max(d[st], d[Eiv]+edge[i].w); //更新当前结点所能走的最长路径(保留较长的那边）

### }

### }

### **二、DFS || BFS**

时间复杂度： O( n )；

优点：可以在第一次 dfs/bfs记录前驱

不足：代码量稍大

#### **实现过程：**

**前提：**

两次dfs或bfs。第一次任意选一个点进行dfs(bfs)找到离它最远的点，此点就是最长路的一个端点，再以此点进行dfs（bfs），找到离它最远的点，此点就是最长路的另一个端点，于是就找到了树的直径。

**证明：**

主要是利用了反证法：

假设 s-t这条路径为树的直径，或者称为树上的最长路

现有结论，从任意一点u出发搜到的最远的点一定是s、t中的一点，然后在从这个最远点开始搜，就可以搜到另一个最长路的端点，即用两遍广搜就可以找出树的最长路

证明：

设u为s-t路径上的一点，结论显然成立，否则设搜到的最远点为T则

dis(u,T) >dis(u,s)     且  dis(u,T)>dis(u,t)   则最长路不是s-t了，与假设矛盾

设u不为s-t路径上的点

首先明确，假如u走到了s-t路径上的一点，那么接下来的路径肯定都在s-t上了，而且终点为s或t，在1中已经证明过了

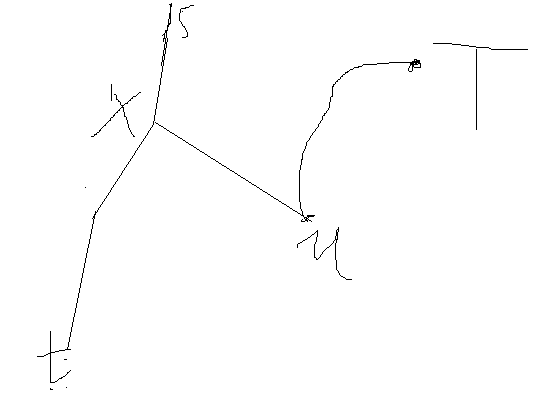
所以现在又有两种情况了：

1：u走到了s-t路径上的某点，假设为X，最后肯定走到某个端点，假设是t ，则路径总长度为dis(u,X)+dis(X,t)

2：u走到最远点的路径u-T与s-t无交点，则dis(u-T) >dis(u,X)+dis(X,t);显然，如果这个式子成立，

则dis(u,T)+dis(s,X)+dis(u,X)>dis(s,X)+dis(X,t)=dis(s,t)最长路不是s-t矛盾

附上一张第二种情况的图



核心代码：

void dfs(int s)

{

for(int i = head[s]; i != -1; i = edge[i].nxt){

int Eiv = edge[i].v;

if(fa[s] == Eiv) continue; //不走回头路，也可以递归父亲结点省去fa数组空间

fa[Eiv] = s;

dis[Eiv] = dis[s] + edge[i].w; //当前结点最长路径

dfs(Eiv);

}

}