

树链剖分

树链剖分的思想及能解决的问题

树链剖分用于将树分割成若干条链的形式，以维护树上路径的信息。

具体来说，将整棵树剖分为若干条链，使它组合成线性结构，然后用其他的数据结构维护信息。

树链剖分（树剖/链剖）有多种形式，如重链剖分、长链剖分和用于 Link/cut Tree 的剖分（有时被称作“实链剖分”），大多数情况下（没有特别说明时），“树链剖分”都指“重链剖分”。

重链剖分可以将树上的任意一条路径划分成不超过 $O(\log n)$ 条连续的链，每条链上的点深度互不相同（即是自底向上的一条链，链上所有点的 LCA 为链的一个端点）。

重链剖分还能保证划分出的每条链上的结点 DFS 序连续，因此可以方便地用一些维护序列的数据结构（如线段树）来维护树上路径的信息。

如：

1. 修改树上两点之间的路径上所有点的值。
2. 查询树上两点之间的路径上结点权值的和/极值/其它（在序列上可以用数据结构维护，便于合并的信息）

除了配合数据结构来维护树上路径信息，树剖还可以用来 $O(\log n)$ （且常数较小）地求 LCA。在某些题目中，还可以利用其性质来灵活地运用树剖。

重链剖分

我们给出一些定义：

定义重子结点表示其子结点中子树最大的子结点。如果有多个子树最大的子结点，取其一。如果没有子结点，就无重子结点。

定义轻子结点表示剩余的所有子结点。

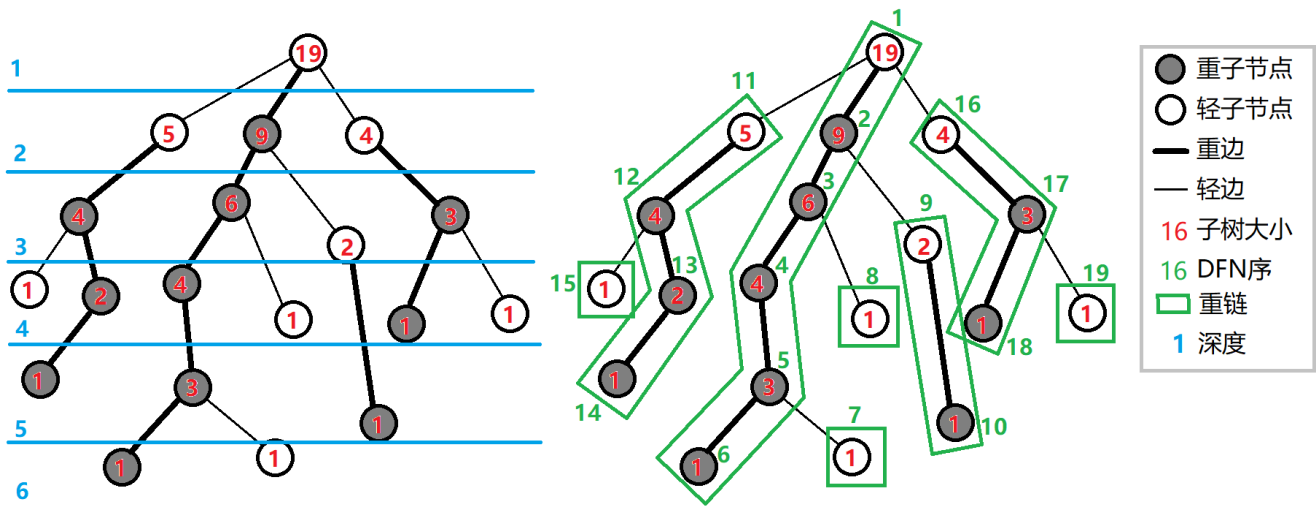
从这个结点到重子结点的边为重边

到其他轻子结点的边为轻边

若干条首尾衔接的重边构成重链

把落单的结点也当作重链，那么整棵树就被剖分成若干条重链。

如图：



实现

树剖的实现分两个 DFS 的过程。伪代码如下：

第一个 DFS 记录每个结点的父节点 `father`、深度 `deep`、子树大小 `size`、重子结点 `hson`。\$
`\begin{array}{l} \text{TREE-BUILD}(u,dep) \\ \begin{array}{l} 1 \ \& \ u.hson \ \text{gets } 0 \\ 2 \ \& \ u.hson.size \ \text{gets } 0 \\ 3 \ \& \ u.deep \ \text{gets } dep \\ 4 \ \& \ u.size \ \text{gets } 1 \\ 5 \ \& \ \text{for } \{ \text{each } \ u \ \text{'s son } \ v \} \\ \quad u.size \ \text{gets } u.size + 1 \\ \text{TREE-BUILD}(v,dep+1) \\ 7 \ \& \ \text{quad } v.father \ \text{gets } u \\ 8 \ \& \ \text{quad } \\ \text{if } \ v.size > u.hson.size \\ 9 \ \& \ \text{quad} \ \text{quad } u.hson \ \text{gets } v \\ 10 \ \& \ \text{return } \ u.size \end{array} \end{array}` \$
 第二个 DFS 记录所在链的链顶 `top` (应初始化为结点本身)、重边优先遍历时的 DFS 序 (`dfn`、DFS 序对应的结点编号 `rank`)。\$
`\begin{array}{l} \text{TREE-DECOMPOSITION}(u,top) \\ \begin{array}{l} 1 \ \& \ u.top \ \text{gets } top \\ 2 \ \& \ tot \ \text{gets } tot+1 \\ 3 \ \& \ u.dfn \ \text{gets } tot \\ 4 \ \& \ rank(tot) \ \text{gets } u \\ 5 \ \& \ \text{if } \ u.hson \ \text{is not } \ 0 \\ 6 \ \& \ \text{quad } \text{TREE-DECOMPOSITION}(u.hson,top) \\ 7 \ \& \ \text{quad } \text{for } \{ \text{each } \ u \ \text{'s son } \ v \} \\ 8 \ \& \ \text{quad} \ \text{quad} \ \text{if } \ v \ \text{is not } \ u.hson \\ 9 \ \& \ \text{quad} \ \text{quad} \ \text{TREE-DECOMPOSITION}(v,v) \end{array} \end{array}` \$ 以下为代码实现。

我们先给出一些定义：

- $fa(x)$ 表示结点 x 在树上的父亲。
- $dep(x)$ 表示结点 x 在树上的深度。
- $siz(x)$ 表示结点 x 的子树的结点个数。
- $son(x)$ 表示结点 x 的重儿子。
- $top(x)$ 表示结点 x 所在重链的顶部结点 (深度最小)。
- $dfn(x)$ 表示结点 x 的 DFS 序，也是其在线段树中的编号。
- $rnk(x)$ 表示 DFS 序所对应的结点编号，有 $rnk(dfn(x))=x$ 。

我们进行两遍 DFS 预处理出这些值，其中第一次 DFS 求出 $fa(x), dep(x), siz(x), son(x)$ ，第二次 DFS 求出 $top(x), dfn(x), rnk(x)$ 。

```
void dfs1(int o){
    son[o]=-1;
    siz[o]=1;
    for(int j=h[o];j;j=nxt[j]){
```


values between u and v 链上的 DFS 序是连续的，可以使用线段树、树状数组维护。

每次选择深度较大的链往上跳，直到两点在同一条链上。

同样的跳链结构适用于维护、统计路径上的其他信息。

子树维护

有时会要求维护子树上的信息，譬如将以 x 为根的子树的所有结点的权值增加 v

在 DFS 搜索的时候，子树中的结点的 DFS 序是连续的。

每一个结点记录 bottom 表示所在子树连续区间末端的结点。

这样就把子树信息转化为连续的一段区间信息。

求最近公共祖先

不断向上跳重链，当跳到同一条重链上时，深度较小的结点即为 LCA

向上跳重链时需要先跳所在重链顶端深度较大的那个。

参考代码：

```
int lca(int u,int v){
    while(top[u]!=top[v]){
        if(dep[top[u]]>dep[top[v]])
            u=fa[top[u]];
        else
            v=fa[top[v]];
    }
    return dep[u]>dep[v]?v:u;
}
```

参考资料

[OI Wiki](#)

From: <https://wiki.cvbbacm.com/> - CVBB ACM Team

Permanent link: https://wiki.cvbbacm.com/doku.php?id=2020-2021:teams:legal_string:%E6%A0%91%E9%93%BE%E5%89%96%E5%88%86_lgwza&rev=1594106316

Last update: 2020/07/07 15:18