SPOJ375 QTREE 解法的一些研究

Yang Zhe*

2007 年 1 月 3 日

摘 要

虽然本题已有前辈进行过研究[3], 并且得到了一个令人满意的解法(其时间复杂度为 $O(n\log n + q\sqrt{n}\log n)$), 但仍有一些优秀的算法未被提及. 本文从这类问题的一般模型——动态树问题入手, 简要介绍了如何将这个模型加以改造并应用到本题.

本文首先介绍了直接运用 Link-Cut Trees 的 $O((n+q)\log n)$ 时间复杂度的解法, 由于它没有对本题的特点加以充分利用, 所以其实际效果不佳. 随后本文又介绍了一个利用了本题的特点的实际表现不错的 $O(n+q\log^2 n)$ 时间复杂度的解法, 在提交最后一个解法之前, 这个解法在 SPOJ 上排名第二.

但已有的成绩不能阻止我们的探索. 将这个解法中维护路径的数据结构修改为 Splay Tree, 我们发现其时间复杂度降到了 $O((n+q)\log n)$, 但由于 Splay Tree 的常数因子巨大, 其实际运行效果也不理想. 经过对这些解法的比较与分析, 在最后, 本文提出了一种静态的"*全局平衡二叉树*"数据结构. 并得到了用这个数据结构维护整棵树的, 时间复杂度为 $O((n+q)\log n)$ 的解法. 这个解法的实际表现也很不错, 比前面提及的 $O(n+q\log^2 n)$ 时间复杂度的算法又快了一些.

希望本文能对大家进一步研究这类问题起到抛砖引玉的作用.

目 录

*yangzhe1990@gmail.com

1 题目描述及简要分析

给一棵共有 $n(n \leq 10000)$ 个结点的树, 每条边都有一个权值, 要求维护一个数据结构, 支持如下操作:

1. 修改某条边的权值;
2. 询问某两个结点之间的唯一通路上的最大边权.

其中操作的总的次数为 q.

对于操作 2, 我们只需要求出被询问的两结点的最近公共祖先, 并分别求出这两条路径上的最大边权.

显然, 直接计算是会超时的. 不难发现, 这个问题要求我们使用一种数据结构, 用来维护从某个点向根的路径的权值. 然而, 这离问题的解决还有一定距离. 我们不妨先看一个更一般的问题: 动态树问题.

2 动态树问题(Dynamic Tree Problems)

动态树问题, 即要求我们维护一个由若干棵子结点无序的有根树组成的森林. 要求这个数据结构支持对树的分割, 合并, 对某个点到它的根的路径的某些操作, 以及对某个点的子树进行的某些操作. 其具体内容读者可以参考[1].

在这里我们考虑一个简化的动态树问题, 它只包含对树的形态的操作和对某个点到根的路径的操作(如果包含对子树的操作, 就需要用到另一种称作 Euler-Tour Trees 的数据结构, 这超出了本文的讨论范围):

维护一个数据结构, 支持以下操作:

- $MAKE\_TREE()$ — 新建一棵只有一个结点的树.
- $CUT(v)$ — 删除 v 与它的父亲结点 $parent(v)$ 的边, 相当于将点 v 的子树分离了出来.
- $JOIN(v, w)$ — 让 v 成为 w 的新的儿子. 其中 v 是一棵树的根结点, 并且 v 和 w 是不同的两棵树中的结点.
- $FIND\_ROOT(v)$ — 返回 v 所在的树的根结点.

搞清了这个问题, 我们也容易扩充这个数据结构, 维护每个点到它所属的树的根结点的路径的一些信息, 例如权和, 边权的最大值, 路径长度等.

3 Link-Cut Trees

Link-Cut Trees 是由 Sleator 和 Tarjan 发明的解决这类动态树问题的一种数据结构 [1]. 这个数据结构可以在均摊 $O(\log n)$ 的时间内实现上述动态树问题的每个操作.

下面我们先介绍 Link-Cut Trees 的概念, 然后介绍它的支持各种操作, 最后对它的复杂度进行分析.

3.1 Link-Cut Trees 的定义

称一个点被访问过, 如果刚刚执行了对这个点的 $ACCESS$ 操作.

如果结点 v 的子树中, 最后被访问的结点在子树 w 中, 这里 w 是 v 的儿子, 那么就称 w 是 v 的 Preferred Child. 如果最后被访问过的结点就是 v 本身, 那么它没有 Preferred Child. 每个点到它的 Preferred Child 的边称作 Preferred Edge. 由 Preferred Edge 连接成的不可再延伸的路径称为 Preferred Path.

这样, 整棵树就被划分成了若干条 Preferred Path. 对每条 Preferred Path, 用这条路上的点的深度作为关键字, 用一棵平衡树来维护它(在这棵平衡树中, 每个点的左子树中的点, 都在 Preferred Path 中这个点的上方; 右子树中的点, 都在 Preferred Path 中这个点的下方). 需要注意的是, 这种平衡树必须支持分离与合并. 这里, 我们选择 Splay Tree 作为这个平衡树的数据结构. 我们把这棵平衡树称为一棵 Auxiliary Tree.

知道了树 T 分解成的这若干条 Preferred Path, 我们只需要再知道这些路径之间的连接关系, 就可以表示出这棵树 T. 用 Path Parent 来记录每棵 Auxiliary Tree 对应的 Preferred Path 中的最高点的父亲结点, 如果这个 Preferred Path 的最高点就是根结点, 那么令这棵 Auxiliary Tree 的 Path Parent 为 $null$.

Link-Cut Trees 就是将要维护的森林中的每棵树 T 表示为若干个 Auxiliary Tree, 并通过 Path Parent 将这些 Auxiliary Tree 连接起来的数据结构.

3.2 Link-Cut Trees 的操作

3.2.1 Access

ACCESS 操作是 Link-Cut Trees 的所有操作的基础. 假设调用了过程 *ACCESS*(v), 那么从点 v 到根结点的路径就成为一条新的 Preferred Path. 如果路径上经过的某个结点 u 并不是它的父亲 *parent*(u) 的 Preferred Child, 那么由于 *parent*(u) 的 Preferred Child 会变为 u , 原本包含 *parent*(u) 的 Preferred Path 将不再包含结点 *parent*(u) 及其之上的部分.

下图为 Link-Cut Trees 的一个结构示意图, 及一次 *ACCESS* 操作的前后对比图.

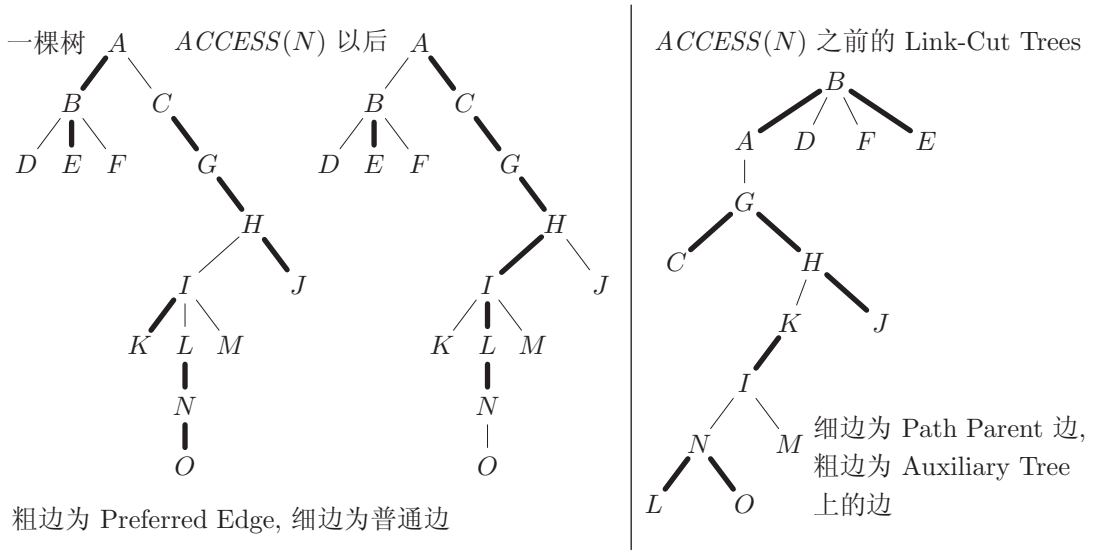

图 1: 一棵树及其 Link-Cut Trees, 以及一次 *ACCESS* 操作的前后对比

ACCESS 的操作有些出人意料, 就是直接更新需要被更新的 Auxiliary Tree. 这看起来并不怎么高效, 但事实上, 在后面我们将证明它的时间复杂度是均摊 $O(\log n)$ 的.

首先, 由于访问了点 v, 那么它的 Preferred Child 应当消失. 先将点 v 旋转到它所属的 Auxiliary Tree 的根, 如果 v 在 v 所属的 Auxiliary Tree 中有右儿子(也就是 v 原来的 Preferred Child), 那么应该将 v 在 v 所属的 Auxiliary Tree 中的右子树(对应着它的原来的 Preferred Child 之下的 Preferred Path)从 v 所属的 Auxiliary Tree 中分离, 并设置这个新的 Auxiliary Tree 的 Path Parent 为 v.

然后, 如果点 v 所属的 Preferred Path 并不包含根结点, 设它的 Path Parent 为 u, 那么需要将 u 旋转到 u 所属的 Auxiliary Tree 的根, 并用点 v 所属的 Auxiliary Tree 替换到点 u 所属的 Auxiliary Tree 中点 u 的右子树, 再将原来点 u 所属的 Auxiliary Tree 中点 u 的右子树的 Path Parent 设置为 u. 如此操作, 直到到达包含根结点的 Preferred Path.

3.2.2 Find Root

在 *ACCESS*(v) 之后, 根结点一定是 v 所属的 Auxiliary Tree 的最小结点. 我们先把 v 旋转到它所属的 Auxiliary Tree 的根. 再从 v 开始, 沿着 Auxiliary Tree 向左走, 直到不能再向左, 这个点就是我们要找的根结点. 由于使用的是 Splay Tree 数据结构保存 Auxiliary Tree, 我们还需要对根结点进行 Splay 操作.

3.2.3 Cut

先访问 v, 然后把 v 旋转到它所属的 Auxiliary Tree 的根, 然后再断开 v 在它的所属 Auxiliary Tree 中与它的左子树的连接, 并设置.

3.2.4 Join

先访问 v , 然后修改 v 所属的 Auxiliary Tree 的 Path Parent 为 w, 然后再次访问 v .

3.2.5 上述操作的伪代码

Link-Cut Trees 的伪代码:

```
1: procedure ACCESS(v)
2:     u ← v
3:     v ← null
4:     repeat
5:         SPLAY(u)
6:         path-parent[right-child[u]] ← u
7:         right-child[u] ← v
8:         path-parent[v] ← null
9:         v ← u
10:        u ← path-parent[u]
11:    until u = null
12: end procedure

13: function FIND_ROOT(v)
14:    ACCESS(v)
15:    SPLAY(v)
16:    while left-child(v) ≠ null do
17:        v ← left-child(v)
18:    end while
19:    SPLAY(v)
20:    return v
21: end function

22: procedure CUT(v)
23:    ACCESS(v)
24:    left-child[v] ← null
25: end procedure

26: procedure JOIN(v, w)
27:    ACCESS(v)
28:    path-parent[v] ← w
29:    ACCESS(v)
30: end procedure
```

3.3 Link-Cut Trees 的时间复杂度分析

从伪代码中可以看出, 除 *ACCESS* 操作之外的其它操作, 其均摊时间复杂度至多为 $O(\log n)$. 所以只用分析 *ACCESS* 的时间复杂度.

在 *ACCESS* 操作的分析中, 我们将 *ACCESS* 的时间分为两部分计算. 第一部分我们证明切换 Preferred Child 的次数是均摊 $O(\log n)$ 的; 第二部分我们证明一次 *ACCESS* 中的所有 Splay 操作的总时间是均摊 $O(\log n)$ 的.

3.3.1 轻重边路径剖分(Heavy-Light Decomposition)

轻重边路径剖分是一种对任意树均适用的分析技巧, 它将树的边分为两类, 一条边要么是重(Heavy)的, 要么是轻(Light)的.

记 $size(i)$ 为树中以 i 为根的子树的结点个数. 令 u 是 v 的儿子中 $size(u)$ 最大的儿子(如果有多个, 那么取任意一个), 称边(v,u)为重边, 称 v 到它的其它儿子 w 的边(v,w)为轻边(即使 $size(w) = size(u)$).[1]如果一个点有儿子, 那么存在唯一一条从它出发的重边. 根据这个定义直接可得:

性质 3.1 *如果 u 是 v 的儿子, 并且 (v,u) 是一条轻边, 那么*

$$size(u) < size(v)/2.$$

[1]为了使划分出的路径尽可能少, 这里介绍的轻重边的定义是修改过的. 在原来的定义中, 所有满足 $size(u) < \frac{1}{2}size(v)$ 的边 (v,u) 为轻边.

证明 假设$size(u) \geq size(v)/2$, 因为 (v,u) 是一条轻边, 所以从 v 出发存在一条重边 (v,w). 根据重边的定义,

$$size(v) \geq 1 + size(w) + size(u) \geq 2size(u) + 1 \geq 1 + size(v).$$

矛盾! ■

令 $light\text{-}depth(v)$ 为从 v 到根经过的轻边个数, 则有

引理 3.1

$$light\text{-}depth(v) \leq \lg n.$$

证明 根据性质 3.1, 如果经过了一条轻边, 那么当前子树的点个数至多变为原来的一半. 最初的子树的点的个数为 n, v 的子树的点数至少为 1, 所以至多经过了 $\lg n$ 条轻边. ■

3.3.2 Preferred Child 变化次数的均摊 $O(\log n)$ 的证明

为了计算 Preferred Child 的变化次数, 我们首先对要操作的树 T 作轻重边剖分.

除了 v 的 Preferred Child 消失之外, 每次 Preferred Child 变化, 必然产生一条新的 Preferred Edge. 我们只需要计数新产生的 Preferred Edge 的个数.

根据引理 3.1, *ACCESS* 过程中至多会新产生 $\lg n$ 条轻的 Preferred Edge. 但是一次可以产生很多重的 Preferred Edge. 那么新产生的重的 Preferred Edge 的个数如何计数呢? 由于每条重边变成 Preferred Edge 的次数至多为重边由 Preferred Edge 变回普通边的次数加上边的总数(有可能最后剩下所有的边都是重的 Preferred Edge). 一条重边由 Preferred Edge 变回普通边, 必然对应一条轻边变为 Preferred Edge, 于是重边变成 Preferred Edge 的总次数至多为轻边变为 Preferred Edge 的总次数加上边的总数. 于是平均每次 *ACCESS* 操作中的重边变为 Preferred Edge 的次数就是 $O(\log n)$ 的.

综上, Preferred Child 的变化次数是均摊 $O(\log n)$ 的.

3.3.3 Splay 操作总时间的均摊 $O(\log n)$ 的证明

对 Splay 操作的时间复杂度分析常用势能分析法.

给 Splay Tree 中的每个结点 u 赋予一个正权 $w(u)$, 令 $s(u)$ 为 u 的子树中的结点的权和. 定义势能函数 $\Phi = \sum_u \lg s(u)$. 可以证明:

引理 3.2 *$SPLAY(v)$ 操作中 zig-zig, zag-zag 的均摊时间花费*

$$\widehat{cost} \leq 3(\lg s'(v) - \lg s(v)).$$

(其中常数 3 是不可改进的.)

引理 3.3 *$SPLAY(v)$ 操作中 zig-zag, zag-zig 的均摊时间花费*

$$\widehat{cost} \leq 2(\lg s'(v) - \lg s(v)) \leq 3(\lg s'(v) - \lg s(v)).$$

引理 3.4 *$SPLAY(v)$ 操作中 zig, zag 的均摊时间花费*

$$\widehat{cost} \leq \lg s'(v) - \lg s(v) + 1 \leq 3(\lg s'(v) - \lg s(v)) + 1.$$

这三个引理的证明比较简单, 有兴趣的读者可以自己证明或查阅相关资料. 由这三个引理直接可得:

定理 3.5 (Access Theorem) *$SPLAY(v)$ 操作的均摊时间花费*

$$\widehat{cost} \leq 3(\lg s'(v) - \lg s(v)) + 1.$$

(其中常数 3 是不可改进的.)

在本问题中, 由于一次 *ACCESS* 操作会执行多次(均摊 $O(\log n)$ 次) *SPLAY* 操作, 所以我们应当寻找一个合适的权函数而不能简单的令 $w(u) = 1$. 如果我们将 Auxiliary Tree 中的边想象成实边, 把 Path Parent 想象为虚边, 一棵子树的结点总数等于 1 + 它的所有子树(包括虚边所连接的子树)的结点总数. 让 $w(u) = 1 +$ 以 u 为 Path Parent 的子树的结点总数, 那么 $s(u)$ 就是 u 的子树的结点总数.

假设在 $ACCESS(v)$ 的过程中, 分别对 $v_0 = v$, $v_1 = path\text{-}parent[v_0]$, ... , $v_k = path\text{-}parent[v_{k-1}]$ 进行了 $SPLAY$ 操作, 那么 $ACCESS$ 操作的均摊时间

$$\begin{aligned}
\widehat{cost} &\leq \sum_{i=0}^{k} 3\left(\lg s'(v_i) - \lg s(v_i)\right) + 1 \\
&\leq 3\left[\sum_{i=1}^{k}\left(\lg s'(v_i) - \lg s'(v_{i-1})\right) + \lg s'(v_0) - \lg s(v_0)\right] + k \\
&= 3\left(\lg s'(v_k) - \lg s(v_0)\right) + k \\
&\leq 3\lg n + \text{Preferred Child 的改变次数}
\end{aligned}$$

因为在前面我们已经证明 Preferred Child 的改变次数是均摊 $O(\log n)$ 的, 所以 $ACCESS$ 操作的均摊时间复杂度是 $O(\log n)$.

4 本题的解法

4.1 解法概览

有了上面的知识, 我们就可以很好地解决这个问题了. 下面将讨论本题的四种解法, 如表 1.

解法编号	解法概述	时间复杂度
解法一	动态树, 使用 Link-Cut Trees 数据结构	$O((n+q)\log n)$
解法二	轻重边路径剖分, 用线段树或虚二叉树维护每条路径	$O(n+q\log^2 n)$
解法三	轻重边路径剖分, 用 Splay Tree 维护每条路径	$O((n+q)\log n)$
解法四	轻重边路径剖分, 用一棵“全局平衡二叉树”维护所有路径	$O((n+q)\log n)$

表 1: 本文所述解法

本文介绍的解法均附有程序供大家参考, 它们的运行时间如表 2.

解法编号	相应参考程序	运行时间
解法一	QTREE_dynamic-tree_link-cut-trees.pas	4.58 秒
解法二(线段树)	QTREE_heavy-light-decomposition_segment-tree.pas	2.65 秒
解法二(虚二叉树)	QTREE_heavy-light-decomposition_imaginary-bst.pas	2.37 秒
解法三	QTREE_heavy-light-decomposition_splay-tree.pas	4.28 秒
解法四	QTREE_heavy-light-decomposition_global-balanced-bst.pas	2.06 秒

表 2: 相应程序的运行时间对比

需要注意的是, 解法二和解法三虽然只是维护路径使用的数据结构不同, 但由于二者时间复杂度不同, 并且解法二和解法三的比较对解法四的得出起很大作用, 故将它们分开列出. 解法二对应的两个程序只是实现方法有所不同, 运行时间相差很小, 并不能以此为据评价线段树与虚二叉树的优劣.

4.2 动态树解法(解法一)

使用 Link-Cut Trees 数据结构来维护这棵树, 我们只需要对上文提到的 Auxiliary Tree 做一些扩充, 在 Auxiliary Tree 中, 记录每个结点到它的父亲结点的边权 $cost$, 和以它为根的子树中所有结点的 $cost$ 的最大值 $maxcost$. 由于求最大值的运算是满足结合律的, 所以 $maxcost$ 可以在 Auxiliary Tree 旋转的过程中用 $O(1)$ 的时间维护.

对于修改边权的操作, 我们只需要更新相应结点在它所属的 Auxiliary Tree 中的 $cost$, 并对这个结点进行 Splay 操作来维护这棵 Auxiliary Tree 中相关结点的 $maxcost$. 显然, 这个操作的时间复杂度是 $O(\log n)$.

对于询问 u 到 v 的路径的最大边权的操作, 我们先 $ACCESS(u)$, 然后在 $ACCESS(v)$ 的过程中, 一旦走到了包含根结点(也就是包含 u)的 Auxiliary Tree, 这时我们走到的结点恰好就是 u 和 v 的最近公共祖先, 不妨称这个结点为 d. 这时 d 所在的 Auxiliary Tree 中 d 的右子树的 $maxcost$ 即为 u 到 d 的路径的最大边权, v 所在的 Auxiliary Tree 的 $maxcost$ 则是 v 到 d 的路径的最大边权. 于是我们所求的答案就是

这两个 $maxcost$ 中的最大值. 因为 Access 操作的均摊复杂度为 $O(\log n)$, 所以回答这个询问所花时间也是 $O(\log n)$ 的.

因为 Link-Cut Trees 的初势能是 $O(n \log n)$ 的, 所以此方法总的时间复杂度是 $O((n+q)\log n)$. 由于使用了 Splay Tree 作为 Auxiliary Tree 的数据结构, 而 Splay Tree 的常数因子巨大, 所以这个算法并不能轻易的在规定时间内(5 秒)出解, 包括文[3]的作者, 也没有用这个方法通过所有测试数据. 下面对我的程序中 Splay Tree 实现的一点技巧加以说明.

4.2.1 Splay Tree 实现的一点技巧

在定理 3.5 中我们看到, Splay Tree 的常数因子是一次单旋转所花时间的三倍. 由于一次单旋转的过程中, 我们不仅要对 Splay Tree 的局部形态进行修改, 还要维护顶点的 $maxcost$ 属性, 如何写好这部分代码, 对 Splay Tree 的实际运行效果起非常重要的作用. 同时, 由于通常的 Splay Tree 的写法中, 需要进行对结点是否根结点, 或是父亲的哪个儿子的判断, 这也消耗了很多时间, 于是我对这些操作的代码进行了一些优化, 并且得到了一种简洁高效的 $SPLAY$, $ROTATE$, $UPDATE$ 的写法.

一些变量的解释: 为了方便 $UPDATE$ 的实现, 我们设置一个虚拟的空结点 $null$, 这个空结点的 $maxcost = -\infty$. 为了方便 $ROTATE$ 的实现, 我们假设结点 x 的左儿子结点存储在数组 $child[x][0]$ 中, 右儿子结点存储在数组 $child[x][1]$ 中. 如果 x 不是根, 那么 $parent[x]$ 存储 x 的父亲结点, $nodetype[x]$ 记录 x 是它的父亲结点的哪个儿子, 即 $child[parent[x]] = x$; 否则 $parent[x]$ 存储这课 Auxiliary Tree 的 Path Parent, $nodetype[x] = 2$.

Splay Tree 的伪代码:

```
 1: procedure UPDATE(node)                          ▷ 更新结点 node 的 maxcost
 2:     k ← cost[node]
 3:     x ← maxcost[child[node][0]]
 4:     y ← maxcost[child[node][1]]
 5:     if x < y then
 6:         x ← y
 7:     end if
 8:     if k < x then
 9:         k ← x
10:     end if
11:     maxcost[node] ← k
12: end procedure


13: procedure ROTATE(node, x)        ▷ 将 nodetype 为 x 的结点 node 旋转到它父亲的位置
14:     p ← parent[node]
15:     y ← nodetype[p]

16:     tmp ← parent[p]
17:     parent[node] ← tmp
18:     nodetype[node] ← y
19:     if y ≠ 2 then
20:         child[tmp][y] ← node
21:     end if

22:     y ← 1 − x
23:     tmp ← child[node][y]
24:     child[p][x] ← tmp
25:     parent[tmp] ← p
26:     nodetype[tmp] ← x

27:     parent[p] ← node
28:     nodetype[p] ← y
29:     child[node][y] ← p

30:     UPDATE(p)
31: end procedure
```

```
32: procedure SPLAY(node)
33:     repeat
34:         a ← nodetype[node]
35:         if a = 2 then
36:             break
37:         end if
38:         p ← parent[node]
39:         b ← nodetype[p]

40:         if a = b then
41:             ROTATE(p, a)                          ▷ zig-zig 或 zag-zag 的第一步
42:         else
43:             ROTATE(node, a)                       ▷ zig-zag 或 zag-zig 或 zig 或 zag 的第一步
44:         end if
45:         if b = 2 then
46:             break                                 ▷ p 是原来的根结点, node 已经被旋转到了根
47:         end if
48:         ROTATE(node, b)                           ▷ zig/zag-zig/zag 的第二步
49:     until FALSE
50:     UPDATE(node)
51: end procedure
```

在 *SPLAY*(*node*) 的过程中, 通常的 *SPLAY* 的写法, 会对有些点执行很多次 *UPDATE*, 事实上这是不必要的. 我们只需要在 *ROTATE* 操作中, *UPDATE* 被旋转结点的父亲结点, 并在 *SPLAY* 的最后对点 *node* 进行一次 *UPDATE* 即可. 由于 *UPDATE* 是一个费时的操作, 这个优化除去了大约一半的多余 *UPDATE* 操作, 所以对程序的性能起到了较大幅度的提高作用.

4.3 轻重边路径剖分方法(解法二至四)

在这个题目中, 由于树的结构并不发生变化, 我们根本没有用到动态树问题中对树的形态进行改变的操作. 对于这样一个更特殊的问题, 我们理应找到一个不比 Link-Cut Trees 复杂的, 不比 $O((n+q)\log n)$ 慢的算法.

考虑到 Link-Cut Trees 是为了让树的形态可以变化才使用了 Preferred Child 等概念, 我们可不可以去掉它们呢? 经过分析, 我们发现这是可行的. 我们只需要应用 Link-Cut Trees 的复杂度分析中的"轻重边路径剖分", 就可以得到本题的快速解法.

4.4 使用线段树或虚二叉树维护路径的 $O(n+q\log^2 n)$ 时间复杂度的解法(解法二)

首先, 对这棵树进行轻重边路径剖分. 称一条不可再延伸的重边组成的路径为重路径. 那么这棵树就被划分为若干条重路径. 我们把每条重路径用线段树或虚二叉树来维护. 我们借用 Auxiliary Tree 这个名称来称呼这里的每棵线段树(或虚二叉树).

对于修改边权的操作, 我们只需要更改相应结点的 *cost*, 并维护它所在的 Auxiliary Tree.

对于询问 u, v 间路径上的边权的最大值的操作, 我们用朴素的方法求 u, v 对应的两棵 Auxiliary Tree 的最近公共祖先. 根据引理 3.1 可得, 从任何一个点到根, 至多会经过 $\lg n$ 条路径, 所以至多会经过 $O(\log n)$ 棵 Auxiliary Tree. 在找最近公共祖先的过程中, 我们可以在 $O(\log n)$ 的时间内求出 u, v 的路径在每棵经过的 Auxiliary Tree 上的部分的边权的最大值. 最后我们再求出 u, v 路径在那棵最近公共祖先 Auxiliary Tree 上的部分的最大值. 这些最大值中的最大值, 就是所求答案.

因为修改边权的时间复杂度是 $O(\log n)$, 回答询问的时间复杂度是 $O(\log^2 n)$, 所以总的时间复杂度就是 $O(n+q\log^2 n)$.

4.5 使用 Splay Tree 维护路径的 $O((n+q)\log n)$ 时间复杂度的解法(解法三)

这个解法中, 我们只需要将上个解法中维护路径的数据结构的改为 Splay Tree. 用到动态树复杂度分析同样的势能函数, 就可以证明, 回答询问的均摊时间复杂度是 $O(\log n)$ 的. 因为初势能是 $O(n\log n)$ 的, 所以这个解法的总时间复杂度为 $O((n+q)\log n)$.

但是在表 2 中我们看到, 这个程序的运行时间并不理想, 我们自然想到了如下问题——

4.6 探索：是否存在一种维护路径的数据结构，它不必具有和 Splay Tree 一样强大的功能，但它同时具有与解法三相同的理论时间复杂度，以及和解法二一样好的实际效果

在上面的讨论中，我们发现，使用线段树或虚二叉树维护路径，虽然每棵 Auxiliary Tree 都是绝对平衡的，但是其理论时间复杂度却比使用 Splay Tree 的高. 这是为什么呢?

从使用 Splay Tree 来维护 Auxiliary Tree 的时间复杂度分析中，可以看到，Splay Tree 并不是孤立地对待每棵 Auxiliary Tree，而是将这些 Auxiliary Tree 通过 Path Parent 看成一个整体，带来了整体的平衡，于是得到了每次询问的均摊 $O(\log n)$ 的时间复杂度.

而线段树或虚二叉树，就没有利用到这些 Auxiliary Tree 之间的联系，即使每棵 Auxiliary Tree 多么平衡，由于丧失了全局方向感，带来的却是整体的不平衡.

这样看来，我们只要朝着利用 Auxiliary Tree 之间的 Path Parent，通过给每个 Auxiliary Tree 定制合适形态，以达到整体的平衡的方向考虑，就有希望得到我们寻觅已久的算法.

4.7 使用“全局平衡二叉树”的 $O((n+q)\log n)$ 时间复杂度的解法(解法四)

顺着上一节的思路，我们把所有的 Auxiliary Tree 看成一个整体，我们只需要找到一个给 Auxiliary Tree 定制合适形态的方法. 正因为这个方法达到了整体的平衡，我们不妨称之为“*全局平衡二叉树*”[2].

4.7.1 “虚拟树”

首先，我们从全局入手，先不考虑每棵 Auxiliary Tree 的具体形态，而是考虑这些 Auxiliary Tree 之间的联系.

我们称 Auxiliary Tree 中的一个结点的儿子为它的实儿子. 如果一棵 Auxiliary Tree 的根结点为 r，$\textit{path-parent}[r]=u$，那么我们称 r 为 u 的一个虚儿子. 我们考虑这样一棵，以最顶端 Auxiliary Tree 的根结点为根的，由实边和虚边连接而成的树(回忆解法二和 Link-Cut Trees)，我们不妨称它为“虚拟树”. 我们定义一个结点的子树，就是它和它发出的实边和虚边所连接到的子树构成的树；左(右)儿子，就是它在它所属的 Auxiliary Tree 中的左(右)儿子；虚儿子，就是由虚边和它相连的儿子.

在上面所讨论的每种方法中都能找到“虚拟树”的影子. 在解法一和三中，我们维护的是一棵动态的“虚拟树”；在解法二中，我们维护的是一棵静态的“虚拟树”，但它的深度是 $O(\log^2 n)$ 的.

在解法四中，我们正是要寻找一种深度为 $O(\log n)$ 的“虚拟树”. 当然，这样的“虚拟树”可以有很多，我们将在 4.7.4 中讨论. 我们只把符合下文给出的“全局平衡二叉树定义”的虚拟树为一棵“全局平衡二叉树”.

4.7.2 全局平衡二叉树的定义及构建

如何构建一棵全局平衡二叉树呢? 首先假设我们用轻重边路径剖分得到了一棵虚拟树，并且我们成功地构造出了一棵全局平衡二叉树，我们来研究一下这棵全局平衡二叉树的性质.

要求 4.1 *这棵全局平衡二叉树中，任何子树的左(右)子树的结点总数小于等于其结点总数的一半.*

事实上，这个要求很容易满足. 假设这棵子树的根结点所属的 Auxiliary Tree 中的结点按照关键字从小到大的顺序排序之后为 $v_1, v_2, \ldots, v_k$. 其中 k 为这棵 Auxiliary Tree 的结点总数. 令 $w(u)=1+$ 以 u 为父亲的所有虚子树的结点总数和. 设 $s_i=\sum_{j=1}^{i} w(v_j)$，则 s_k 为这棵子树的结点总数. 因为 $s_0=0$，$s_{k-1}<s_k$，所以存在一个整数 t，满足 $s_{t-1}<s_k/2\le s_t$. 选择结点 v_t 作为这棵 Auxiliary Tree 的根，于是左子树的结点总数 $=s_{t-1}<s_k/2$，右子树的结点总数 $=s_k-s_t\le s_k/2$.

性质 4.2 *这棵全局平衡二叉树中，任何子树的虚子树的结点总数小于其结点总数的一半.*

证明 根据性质 3.1，可得任何子树的虚子树的结点总数小于这棵子树的结点总数与它的左子树的结点总数之差的一半，也就小于这棵子树的结点总数的一半. ■

要求 4.3 (子结构) *一棵全局平衡二叉树的任何子树都是全局平衡二叉树.*

定理 4.1 (“全局平衡”定理) *一棵有 n 个结点的全局平衡二叉树的最大可能深度(根的深度为 0)*

$$\textit{max-depth}(n)\le \lg n.$$

证明 由于要求 4.1，性质 4.2 与要求 4.3，这棵全局平衡二叉树根结点的左(右/虚)子树的结点个数至多为原来的一半，并且其仍为一棵全局平衡二叉树，所以$\textit{max-depth}(n)\le \textit{max-depth}(n/2)+1\le \lg n$. ■

有了上述知识，我们就可以写出全局平衡二叉树的构建过程.

[2]因我对这些概念的命名可能不够规范，故在其第一次出现的时候用引号括住，加以注明

构建全局平衡二叉树的代码:

```
1: function BUILD(l, r)
2:     二分查找满足 s[t − 1] < (s[r] + s[l − 1])/2 ≤ s[t] 的 t
3:     if t > l then
4:         left-child[t] ← BUILD(l, t − 1)
5:     end if
6:     if t < r then
7:         right-child[t] ← BUILD(t + 1, r)
8:     end if
9:     return t
10: end function

11: procedure CONSTRUCT-AUXILIARY-TREE(v[1..k])
12:     计算 s[0..k]
13:     BUILD(1, k)
14: end procedure

15: procedure COMPUTE-WEIGHT(node)
16:     size(node) ← 1
17:     for all 边(node, v) do
18:         COMPUTE-WEIGHT(v)
19:         size(node) ← size(node) + size(v)
20:     end for
21:     w(node) ← size(node)
22:     if 存在重边(node, v) then
23:         w(node) ← w(node) − size(v)
24:     end if
25: end procedure

26: procedure MAKE-GLOBAL-BALANCED-BST(tree)
27:     Heavy-Light-Decomposition(tree)
28:     Compute-Weight(root)
29:     for all 重路径 (v[1..k]), 这里 v[i] 是 v[i + 1] 的父亲 do
30:         CONSTRUCT-AUXILIARY-TREE(v[1..k])
31:     end for
32: end procedure
```

根据代码可以看出构建全局平衡二叉树的过程是 $O(n \log n)$ 的.
一个全局平衡二叉树的示例如下:

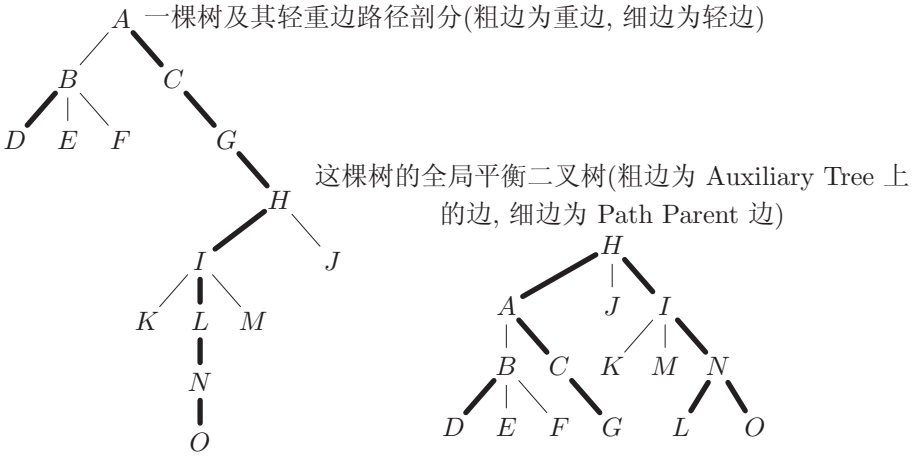

图 2: 一棵树及其全局平衡二叉树

4.7.3 使用全局平衡二叉树的解法

对本题要求的修改权值的操作, 我们只需要维护相应的 Auxiliary Tree. 这个过程我们不再赘述. 因为全局平衡二叉树的深度是 $O(\log n)$的, 所以这个操作的时间复杂度是 $O(\log n)$ 的.

对于询问路径上的最大边权的操作, 我们不妨先回忆一下和解法二中的做法. 我们将询问分割为在“途经”Auxiliary Tree 和“最近公共祖先”Auxiliary Tree 的询问. 在“最近公共祖先” Auxiliary Tree 的部分的询问我们可以在 $O(\log n)$ 的时间内回答, 因为这不会影响这个解法的时间复杂度, 我们不妨把注意集中在处理“途经”Auxiliary Tree 上.

在每棵“途经”Auxiliary Tree 上. 我们只需要知道当前结点, 它的父亲结点, ..., 这条重路径的最高点的 *cost* 的最大值. 在 Auxiliary Tree 上, 就是求出所有关键字小于等于当前结点的关键字的结点的 *cost* 的最大值. 显然, 只需要从 Auxiliary Tree 的根结点开始向下走到当前结点, 并根据已经维护过的 *maxcost*, 就可以计算出所求的最大值. 假设这个 Auxiliary Tree 的根结点的深度为 $depth(root)$, 当前结点的深度为 $depth(node)$, 则这次询问所花时间为 $O(depth(root) - depth(node) + 1)$.

假设询问点 v_0 过程中经过的“途经”Auxiliary Tree 分别为 $r_0, r_1, \ldots, r_n$. r_k 的 Path Parent 为 v_{k+1}, 显然, 我们要做的就是分别求出了 v_k 到 r_k 的结点的 *cost* 的最大值.

根据上面的结论, 我们在“途经”Auxiliary Tree 上花费的总时间就是

$$\begin{aligned} & O(\sum_{i=0}^{k} depth(r_k) - depth(v_k) + 1) \\ = \; & O(\sum_{i=0}^{k} depth(v_{k+1}) - depth(v_k)) \\ = \; & O(depth(v_{k+1}) - depth(v_0)) \\ = \; & O(\log n). \end{aligned}$$

所以我们回答一次询问的总时间是 $O(\log n)$.

我们也可以这样直观地理解回答询问的过程: 我们在这棵“全局平衡二叉树”上找这两个结点的最近公共祖先(这个最近公共祖先并不一定是原树中这两个结点的最近公共祖先), 并在途径的结点收集相关信息. 由于“全局平衡二叉树”的深度是 $O(\log n)$ 的, 所以我们回答询问的总时间是 $O(\log n)$ 的.

4.7.4 “全局平衡”的二叉树

在全局平衡二叉树中, 影响树的深度的关键其实是性质 4.2. 那么其实这个条件可以放宽为: 任何子树的虚子树的结点总数小于其结点总数的 c 倍, 其中 c 是一个小于 1 的常数. 我们称满足这个条件的树为一棵“全局平衡”的二叉树. 显然, 要求 4.1 和要求 4.3 也可以满足. 同时, 类似定理 4.1, 我们可以证明, 这棵“全局平衡”的二叉树的深度 $\leq \log_{1/c} n = O(\log n)$, 只不过当 c 很大的时候, 深度的常数因子较大.

显然, 一棵“全局平衡”的二叉树也可以用来解决本题. 但实际上, 我们可以保证构造出“全局平衡”的二叉树的 c, 是大于等于 1/2 的. 所以上文的方法再无改进之处. 即使我们用尽各种办法得到了一个小于 1/2 的 c, 也是徒劳的. 因为这时, 制约深度的关键已经变为了要求 4.1, 考虑到整棵树是一条链的情况, 这个要求是无法再改进的. 于是, 上文所述的全局平衡二叉树应该是足够好的了.

4.8 解法的对比与总结

分析表 2, 我们可以发现:

解法三和解法二, 虽然出自同一个思路, 但是由于 Splay Tree 时间复杂度分析的特殊性, 解法三具有更低的理论时间复杂度. 只不过由于 Splay Tree 的常数因子比较大, 所以实际运行中, 解法三的程序要比解法二的程序慢很多.

同时我们也可以发现, 解法一和解法三的程序运行时间相差很小, 这说明 Link-Cut Trees 的切换 Preferred Child 的操作实际上是非常快的. Link-Cut Trees 的最耗时操作是恰好用 Splay Tree 维护路径的操作. 有趣的是, 当使用 Splay Tree 作为维护路径的数据结构之后, Link-Cut Trees 的出现和应用就变得理所应当, 因为它只多花费了很小的代价, 就得到了很大的可扩展性. 这个可扩展性, 正是 Splay Tree 这种特殊的数据结构(其复杂度分析方法的特殊性)带来的. 在 Splay Tree 带来了如此大的可扩展性的同时, 也不可避免地带来了不小的常数因子. 姑且认为, 使用 Splay Tree 维护路径, 就是专为使用 Link-Cut Trees 而设计的(否则就大材小用了).

这就启发着我们去思考, 是否存在另一种针对树的形态不会发生改变的, 用不花费高昂的旋转代价的平衡树来维护路径的, 并且与解法一和解法三最差相同理论时间复杂度的, 实现本题要求的操作的数据结构. 朝着这个方向考虑, 就得到了这里的解法四.

从解法三, 解法四中可以看出, 在解决有些问题的时候, 局部的最优并不一定带来整体的最优, 我们不应该固执地追求局部的最优, 而是要时刻保持全局观念, 把握整体与局部的联系, 从而得到整体的最优. 这便是我写作本文的主要目的.

A 致谢

湖南长郡中学的袁欣颢同学为我完成这篇解题报告提供了很大帮助, 在这里向他表示感谢!

参考文献

[1] Daniel D. Sleator , Robert Endre Tarjan, *A data structure for dynamic trees*, Journal of Computer and System Sciences, v.26 n.3, p.362-391, June 1983

[2] Daniel Dominic Sleator , Robert Endre Tarjan, *Self-adjusting binary trees*, Proceedings of the fifteenth annual ACM symposium on Theory of computing, p.235-245, December 1983

[3] 周戈林, *《Query on a tree（spoj375）》解题报告*