AAA 树报告

孔冰玉，苏雨峰，张嘉恒，徐晓骏

2016 年 4 月 26 日

1 功能介绍

AAA 树是一种在 link-cut tree(LCT) 的基础上进行改进后的数据结构，可以用于解决相关的动态树问题。我们的 AAA 树可以在不超过 $\mathcal{O}(\log^2 n)$ 的均摊时间内实现以下操作：

- 合并两棵树。
- 将一棵树分为两棵。
- 令树中的一个节点成为该树的根。
- 在一棵树的一条链上增加一个数或全部赋为一个数。
- 查询一条链上所有节点的权值和。
- 查询一棵子树上所有节点的权值和。
- 查询一棵树上两个节点的最近公共祖先 (LCA)。

2 原理说明

2.1 Link-Cut Tree

2.1.1 链的划分

首先介绍 link-cut tree，这是一个维护若干棵有根树组成的树林的数据结构，可以实现树的分割、合并、以及一条链上的相关操作。在一棵树中给出如下定义：

1. 定义操作 $access(u)$，即访问一个节点。

2. 设节点 u 的儿子节点为 $v_1, v_2, \ldots v_n$，且子树 u 中最后被访问的节点在子树 v_i 上，则称 v_i 为 u 的 *preferred child*，u 到 v_i 的边称为 *preferred edge*。若子树 u 中最后被访问的节点为 u 本身，则 u 没有 preferred child.

3. 由 preferred edge 组成的不可再延伸的路径称为 *preferred path.*

可以看出，preferred path 将整棵树分解成了若干条链，每个节点在且仅在一条链上，且每次 access 操作都会改变链的情况（如图 1）所示。因此，只要记录每条链的信息与链之间的关系，就能维护这棵树。

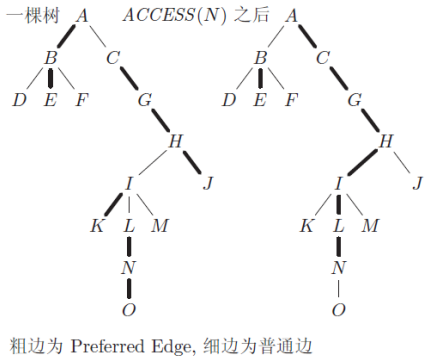

图 1: 树上的 preferred path 示意图。

2.1.2 链信息的维护

为了让链上的操作有较快的时间效率，为每条链建立一棵平衡树来维护信息，在此使用 splay 树。平衡树中的节点以其深度作为关键字维护，即：一个节点在 splay 中左子树中的点在原树中都在该节点的上方。而每条链 (即每棵 splay) 上都会记录其该链在原树中的父节点，称为 *path parent*. (如图 2)。这样，一棵 link-cut tree 即构建完成了。

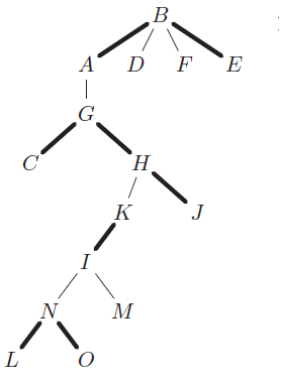

图 2: $access(N)$ 之前的树对应的 link-cut tree.

2.1.3 access 操作

假设操作为 $access(v)$, 则直接递归更新从 v 到根节点的所有 splay。首先，删除节点 v 的 preferred child，即：将节点 v 旋转到 splay 的根，其右子树 (如果存在) 提取成一棵新的 splay，设置该 splay 的 path parent 为 v；其次，递归更新 v 直到其 preferred path 包含根节点：若该 splay 的 path parent 为 u, 则将 u 旋转至其 splay 的根，将 v 设置为 u 的右子树，并将其原来的右子树提取成新的 splay，并递归更新 u。

2.1.4 link 操作

设 $link(u, v)$ 为将 v 节点所在的子树连接至 u 节点上，则操作步骤为：首先 $access(v)$，其次将 v 所属的 splay 的 path parent 设置为 u，然后再次执行 $access(v)$.

2.1.5 cut 操作

设 $cut(u)$ 为将以 u 为根节点的子树从原树中提取成一棵新的树，操作步骤为：首先 $access(v)$, 其次将 v 旋转至该 splay 的根，并将该节点的左子树提取成新的 splay。

2.1.6 换根操作

设 $evert(u)$ 将 u 换为其所在的树的根。操作步骤为：首先 $access(u)$，其次将 u 所在的 splay 进行 reverse 操作 (即把整个 splay 的左右方向颠倒)。这样，由于根节点到 u 的这条链 (splay) 的方向颠倒了，因此 u 就被换到了根节点。

2.1.7 链修改

设 $modify(u, v)$ 为对树上从 u 节点到 v 节点的链进行某种操作 (如：增加某个值，或覆盖为某个值)，则具体实现方法为：首先将 u 换到根节点 (即 $evert(u)$)，其次 $access(v)$，然后对 v 所在的 splay 进行修改操作。

2.1.8 链查询

与链修改操作一样，对于查询链上的和操作 $query(u, v)$，先 $evert(u)$，再 $access(v)$，然后查询 v 所在的 splay 即可。

2.1.9 最近公共祖先

设操作 $LCA(u, v)$ 为查询 u 与 v 最近公共祖先，具体操作为：首先 $access(u)$，然后在 $access(v)$ 的过程中，查找" 最后一次查找到的 path parent"，此即为两节点的最近公共祖先。

2.2 AAA Tree

传统的 link-cut tree 能够较好地解决树的合并、分割与链上的操作。然而，在树上还有另一类十分重要的操作：关于子树的操作。为了实现对一个节点所在的子树的相关操作 (在我们的程序中实现了子树上所有节点值的和的统计)，我们需要在 LCT 的基础上进行改进，这就是 AAA 树。

2.2.1 相关定义

为了方便，称一个节点向孩子连出的边中的那条 preferred edge 为*实边*，称其他边为*虚边*。显然，一个节点 u 的实边连出的点与 u 在同一棵 splay 上，而其虚边连出的节点都各自在不同的 splay 上，且这些 splay 的 path parent 均为 u。如图 3所示。由于主要考虑的是子树信息，因此图中将一条 preferred path 接近于水平放置。

由于 preferred path 由一组实边构成，也成 preferred path 为一条*实链*。

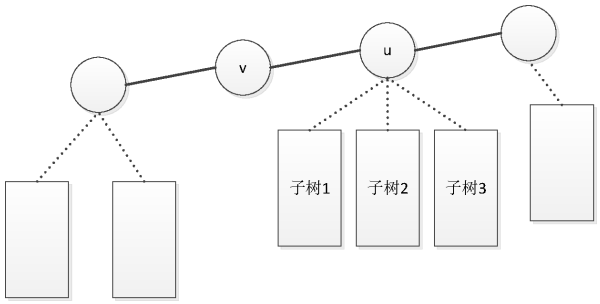

图 3: 一条实链上的构造。

2.2.2 虚边信息的维护

可以看到，子树 u 上的信息 (节点值的和，记为 *treeSum*) 可以由子树 v 上的信息加上 u 连出的虚边上所有子树信息之和得到。因此，在每个节点上还要记录所有从这条链上的点连出的虚边对应的子树的信息的和 (不包括这条链)。一个最简单的方法是，对于每个点暴力记下其连出去的所有虚边，就可以直接枚举这些虚边对应的子树，用这些子树信息维护该节点的 treeSum 值。但是这样的做法时间效率低下，因此要在原 LCT 上进行修改。

我们的做法是，将实链一个点的连出所有的虚边连到的点建立一个 treap 并将 treap 的根节点记录在实链上的节点上。同时 treap 的每个节点还指向了这条虚链连接的实链 splay 的根。如图 4，其中的 treap 节点的意思是这些子树通过 treap 建立联系，省略了内部构造，而不是指单纯地建立一个 treap 节点。

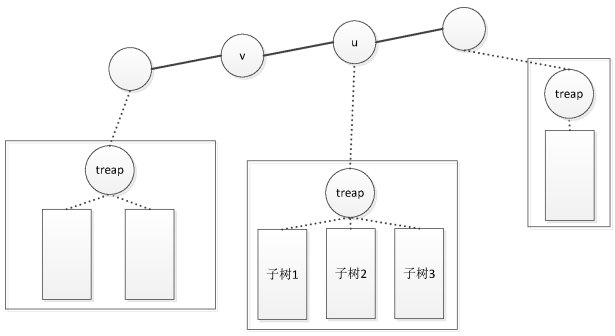

图 4: 建立 treap 后一条实链的构造图。

2.2.3 Add 操作与 Del 操作

在这样的结构下，我们可以实现两种操作：

一、add 操作，也就是将两条被虚边连接的实链首尾相接。我们只需要将两颗 splay merge 一下。再将第二条链的深度最小的点，从第一条链的深度最大的点的 treap 中删除 (因为这个时候虚边变成了实边，所以不再是该点连出去的虚边)。

二、Del 操作，也就是将一条实链变成两条由虚边连接的实链。我们只需要将一颗 splay split 成为两颗 splay，并且将深度度大的那条链中的深度最小的点，加入到另外一条链深度最大的点的虚树 treap 中 (因为有一条实边变成了虚边)。

2.2.4 AAA 树中的 Access 操作

AAA 树的 access 操作在每次递归中会做如下内容：

1. 找到现在的链的父亲 (直接查找该 splay 的 path parent 即可)。
2. 将一条实边变成虚边，即 add 操作。
3. 将某条虚边变成实边，即 del 操作。

其他内容与 LCT 上的操作类似。

2.2.5 子树查询

对于操作 $treequery(u)$，即询问 u 节点下的子树中所有节点的权值和，我们只需要找到现在 u 的原树中的父亲将其 access 到根，这样我们就可以得到一个深度最低点为 x 的实链。将这颗 splay 的实链权值和加上所有虚边子树的权值和 (即每个节点指向的 treap 的权值和) 就是子树信息了。

3 时空复杂度证明

3.1 Link-Cut Trees 的时间复杂度分析

我们经过对代码的分析可以得知，除了 $access$ 操作之外的其他操作，其均摊时间复杂度至多为 $\mathcal{O}(logn)$. 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 $PreferredChild$ 的次数是均摊 $\mathcal{O}(logn)$ 的；第二部分我们证明一次 $access$ 中所有的 $Splay$ 操作的总时间是均摊 $\wr(logn)$。

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

轻重边路径剖分式一种对任意树均适用的分析技巧，它将树的边分为两类，一条边要么是重的，要么是轻的。
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 (u,v) 为重边，称 v 到它的其他儿子 w 的边 (v,w) 为轻边 (即使 $size(w)=size(u)$)。如果一个点到儿子，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

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

证明 假设 $size(u)$ $size(v)/2$, 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 $light - depth(v)$ 为从 v 到根经过的轻边个数，则有

引理 3.1

$$light - depth(v) \leq lgn$$

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，最初的子树的店的个数为 n，v 的子树的点数至少为 1，所以至多经过了 lgn 条轻边。

3.1.2 Preferred Child 变化次数的均摊 $\mathcal{O}(logn)$ 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 做轻重边路径剖分.

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 lgn 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 $O(logn)$ 的.

综上, Preferred Child 的变化次数是均摊 $\mathcal{O}(logn)$ 的.

3.1.3 Splay 操作总时间的均摊 $\mathcal{O}(logn)$ 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 赋一个正权 $w(u)$, 令 $s(u)$ 为 u 的子树中所有结点的权和. 定义势能函数 $\Phi = \sum_u lgs(u)$. 可以证明:

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

$$\hat{cost} \leq 3(lgs'(v) - lgs(v))$$

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

$$\hat{cost} \leq 2(lgs'(v) - lgs(v)) =\leq 3(lgs'(v) - lgs(v))$$

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

$$\hat{cost} \leq lgs'(v) - lgs(v) + 1 \leq 3(lgs'(v) - lgs(v)) + 1$$

引理的证明不是很复杂，但是在这里我们就不证明了。由上述三个引理可得：

$$\hat{cost} \leq 3(lgs'(v) - lgs(v)) + 1$$

在本问题中, 由于一次 ACCESS 操作会执行多次 (均摊 $\mathcal{O}(logn)$ 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 $v_0 = v$, $v_1 = path - parent[v_0]$,...,$v_k = path - parent[v_{k-1}]$ 进行了 SPLAY 操作, 那么 ACCESS 操作的均摊时间

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

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

4 具体实现方式和细节

4.1 AAA 树的实现

在 AAA 树中，对一个原树的节点分别建立了一个 splayNode 与一个 treapNode，并且两个节点指针间有互相的映射关系。此外，splayNode 还有一个节点指向其所连出的所有虚链构成的 treap 的根节点。这样，AAA 树的框架便构建完毕了。

4.2 splay 中各类信息的维护与统计

在 LCT 的原理说明中，我们提到需要对 splay 进行翻转操作、值修改操作与值覆盖操作，并且需要维护 splay 中子树 (即原先的链) 的上所有节点的和。具体的做法是：在每个节点上分别维护三种标记 (翻转标记、修改标记、覆盖标记) 代表当前子树需要被进行这些操作和两个信息 (子树大小、子树节点和)。并且实现” 标记下放”(将一个节点的各类标记下传给它的左右孩子) 和” 信息上传”(将一个节点的各类信息更新为其左右孩子信息的累加)。这样，修改操作与查询操作就可以 $\mathcal{O}(1)$ 时间完成，而每次 splay 操作时，只要在旋转前将途经的节点进行标记下放，并在旋转后再进行信息上传，即可在 $\mathcal{O}(\log n)$ 时间内完成标记与信息的更新操作。

此外，对于子树信息进行维护，因此还需对 splay 节点维护一个”splay 中自己与右子树上 (即原链上的自己与下方节点) 所有虚边连接的子树的权值和” 信息 (即 treeSum，

见 2.2.2)。在信息上传时只需将右子树的所有权值和加上本节点的 treap 中所有的权值和合并即可。这个信息不会包含这条实链上的节点权值。因此，一棵子树的信息等于链上的权值和加上链连出的子树的信息之和。