Link-Cut-Tree

wcz[1]

December 31, 2017

[1]Contact me:aiyoupass@outlook.com

Contents

1 Dynamic Tree Problems[2]

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.

维护一个包含 N 个点的森林, 并且支持形态和权值信息的操作.

(1). 形态信息

(a). link(u,v) – 添加边 (u,v).

(b). cut(u,v) – 删除边 (u,v).

(c). find(u) – 找到 u 所在的树.

(2). 权值信息

(a). 路径操作: 对一条简单路径上的所有对象进行操作.

(b). 树操作: 对一棵树内的所有对象进行操作.

现有数据结构,

- Euler Tour Trees[1]
- ST-Trees[2]
- Top-Trees[3]

这几种数据结构都存在一定的局限性, 因此动态树问题并没有被完全解决.

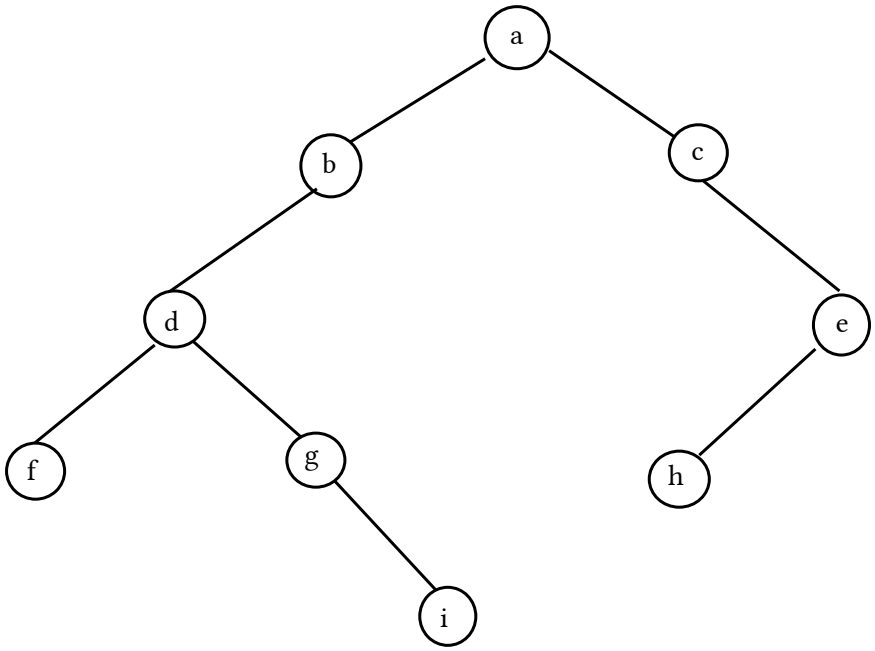
在信息学奥赛中, 我们常常会遇到动态树的简化问题. 我们涉及的操作只有对树形态的操作和对于路径的操作. 因此就有一种解决动态树问题的数据结构

$$Link - cut - Trees[4]$$

[1]不支持路径操作
[2]不支持树权操作
[3]常数过大
[4]由 Sleator 和 Tarjan 发明

2 Link-Cut-Trees[3]

Link-Cut-Trees, 简称 LCT, 它对上述操作的均摊时间不超过 $O(\log n)$.
它所操作的对象是森树, 能实现对于树的合并与分离.



Former Tree

这是一颗已经构建好的树 (森林)，我们用 Link-Cut-Trees 来维护它.

2.1 Build

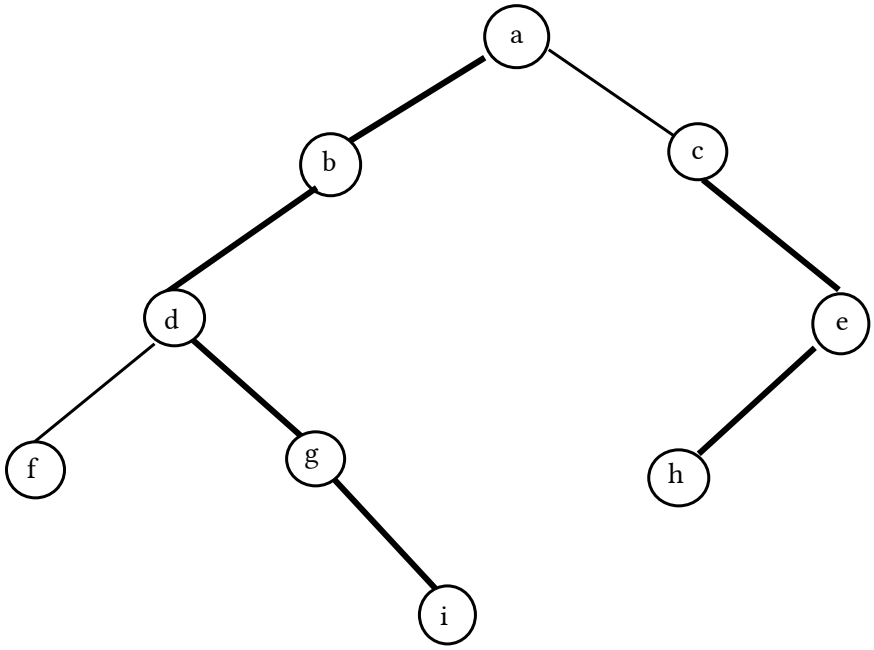
对于一个节点进行访问的操作称为 $Access$;

称我们要表示的这棵树为 Former Tree;

Preferred Child , 如果对于节点 u 所在的子树中, 节点 v 为最后访问过的点, 则称节点 v 为节点 u 的 Preferred Child;

Preferred Edge , 每个点到 Preferred Child 的边称为 Preferred Edge.

preferred Path , 由 preferred Edge 构成的不可再延伸的路径称为 Preferred Path.

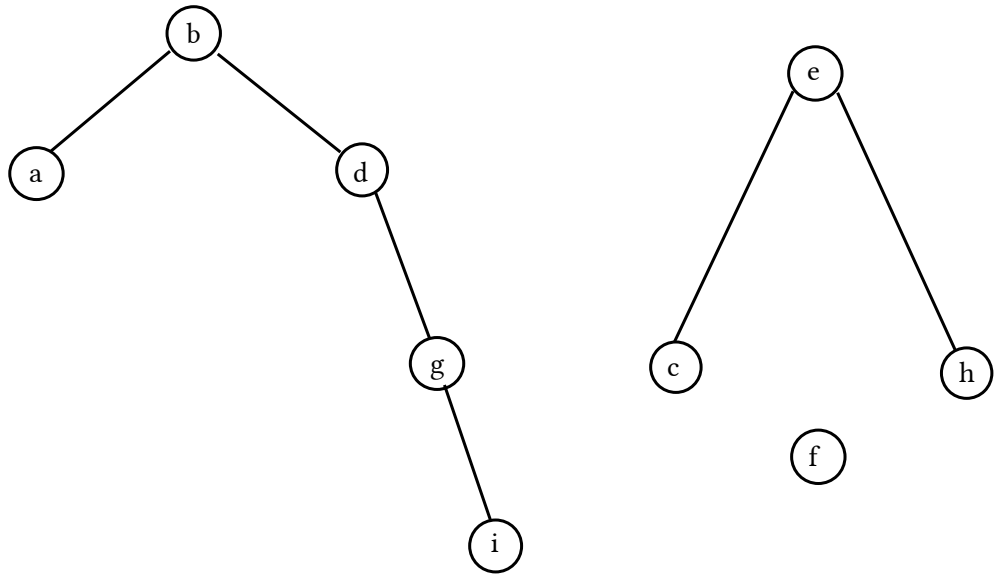


此时 a 为最后一次 Access 的点

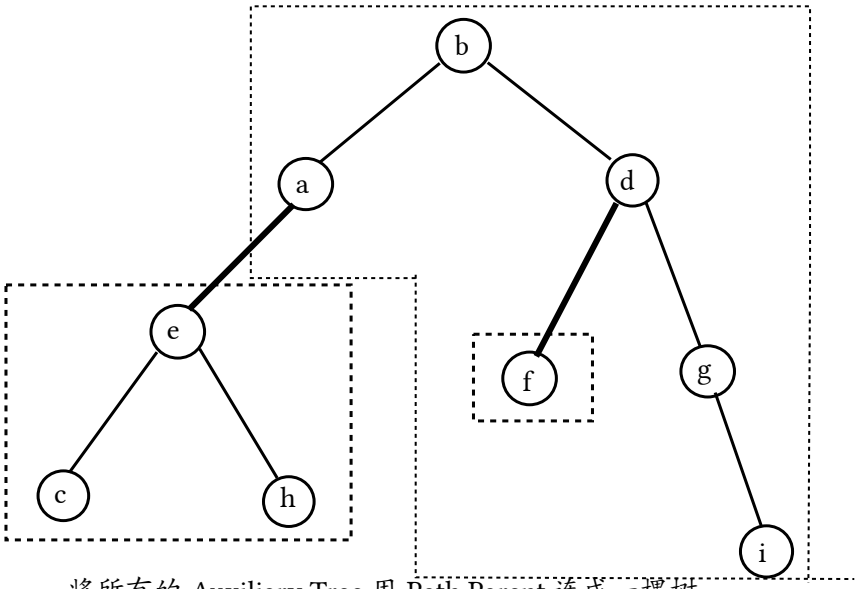
假若上图就是对 $FormerTree$ 标记了 $PreferredEdge$ 的图.

Auxiliary Tree , 在每一条 Preferred Path 中, 以路径上点的深度为关键字, 用 Splay Tree 来维护它, 就是 Auxiliary Tree;
在 Auxiliary Tree 中, 每个点的左子树中的点, 都在 Preferrd Path 中这个点的上方; 右子树中的点都在 Prefreed Path 中这个点的下方.
Path parents , 用 Path parents 来表示其 Auxiliary Tree 对应的 Preferrd Path 中最高的节点;
因为 Former Tree 可以用若干条 Prefreed Edge 来表示, 用每个 Auxiliary Tree 表示一条 preferrd Edge , 那么我们最终所构建的就是一颗用 Parents Edge 将所有 Auxiliary Tree 连接起来的树.

因为 Auxiliary Tree 可以维护 Former Tree 的信息, 因此在实际操作中, 只需要维护 Auxiliary Tree.



对三条 Preferrd Edge 建立 Auxiliary Tree



将所有的 Auxiliary Tree 用 Path Parent 连成一棵树

2.2 Access

Access 操作是 Link-Cut Trees 的所有操作的基础. 假设调用了 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 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.

在对节点 v 进行一次 Access 操作后, 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 Auxiliary Tree 中的右子树 (对应着 v 的 Preferred Child 之下的 Preferred Path) 从 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.

这是一个递归操作的过程. 最终目的还是完成对 preferrd Edge 的修改.

2.3 Is root

在 Auxiliary Tree 中, 如果一个点是 root , 那么他的 Parent 为 NULL.

2.4 Find root

寻找点 v 所在 Auxiliary Tree 的根节点, 先将 v 旋转到根, 然后寻找其 Auxiliary Tree 中最左边的点.

2.5 Be root

注意将节点 v 进行 Access 操作之后 v 只是变成了 Auxiliary Tree 的 root , 但在 Former Tree 中仍然不是 root , 因为 v 还有左子树. 所以 Be root 操作的意义在于将节点 v 变成 Former Tree 中的 root .

首先进行 Access(v) 和 Splay(v) 操作, 然后将 v 到 root 路径上点的深度反转, 我们面对翻转的处理方法是先打标记然后在维护 Splay 时处理.

2.6 Split

先访问 v, 然后把 v 旋转到 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.

2.7 Merge

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 .

2.8 Code

```
bool isroot(int x){
        return c[fa[x]][0]!=x&&c[fa[x]][1]!=x;
}

void beroot(int x){
        Access(x);
        Splay(x);
        rev[x]^=1;
}

void pushdown(int k){
        int l=c[k][0],r=c[k][1];
        if(rev[k]){
                rev[k]^=1;rev[l]^=1;rev[r]^=1;
                swap(c[k][0],c[k][1]);
        }
}

void rotate(int x){
        int y=fa[x],z=fa[y],l,r;
        if(c[y][0]==x)l=0;
        else l=1;r=l^1;
        if(!isroot(y))
        if(c[z][0]==y)c[z][0]=x;
        else c[z][1]=x;
        fa[x]=z;fa[y]=x;fa[c[x][r]]=y;
        c[y][l]=c[x][r];c[x][r]=y;
}

void splay(int x){
        top=0;q[++top]=x;
        for(int i=x;!isroot(i);i=fa[i])
        q[++top]=fa[i];
```

```
        for(int i=top;i;i--)pushdown(st[i]);
        while(!isroot(x)){
                int y=fa[x],z=fa[y];
                if(!isroot(y)){
                        if(c[y][0]==x^c[z][0]==y)
                                rotate(x);
                        else rotate(y);
                }
                rotate(x);
        }
}
void access(int x){
        for(int t=0;x;t=x,x=fa[x])
                splay(x),c[x][1]=t;
}

int find(int x){
        access(x);splay(x);
        while(c[x][0])
        x=c[x][0];
        return x;
}
void merge(int x,int y){
        beroot(x);access(y);splay(y);
        if(c[y][0]==x)
        c[y][0]=fa[x]=0;
}
void split(int x,int y){
        beroot(x);fa[x]=y;
}
```

3 Path operation*[1]

针对的是对于路径的修改和查询. 对于节点 u 和 v 之间路径的操作, 我们首先 Be root(u), 然后 Access(v), Splay(v). 然后发现 u,v 之间的路径在 Auxiliary Tree 上都位于 v 的左子树. 然后就进行各种维护就可以了.

3.1 Example 1

Bzoj 2002 弹飞绵羊

需要维护树的分离与合并.

如何建模? 我们考虑用 LCT,

我们注意到如果在 u 位置能到达　v 位置那么可以连一条边 (u,v) , 设 T 为空点即到达此点时已经跳出, 则查询从 u 几次跳出我们可以查询在 Auxiliary Tree 中 u,t 之间路径的长度即为答案. 那么根据我们之前所提到的如何维护路径, 我们只需要 Be root(t) , 然后 Access(u) , Splay(u) 答案即为 size(t->left).

对于另一种操作即改变一个点所到达的点的位置, 我们只需要切开原来的边并且重新连一条边即可.

4 Others

4.1 Connect

对于树上路径信息的维护与修改, 一般会用树链剖分来做. 树链剖分是将树划分成若干条路径并用线段树等数据结构来维护, 我们可以看到树链剖分与 Link-Cut-Tree 的联系与不同, 都是将树转化成若干条链的方式, 对于树链剖分, 它只能维护静态的树的信息, 却不能对树的形态进行改变; 而 Link-Cut-Tree 能动态的改变树的形态, 也能维护树的路径信息. 例如, 给出一棵树, 进行以下操作.

- 求 u 子树和;
- 将 u 子树加上一个数;
- 求 u,v 之间路径和;
- 将 u,v 之间每条边加上一个数;

像这种对树上路径进行查询修改的话我们可以用树链剖分做.

再例如, 给出一棵树, 进行以下操作.

- 改变 u,v 边权.
- 查询 u,v 路径最大值.

解法一　Lint-Cut-Tree

无疑问, 这道题可以用 LCT 做, 这两种操作都是 LCT 所支持的基础操作, 但是毫无疑问, 因为这道题目没有涉及到更改树形态信息. 而 LCT 依据 Auxiliary Tree 实现, 不得不考虑 Splay 巨大的常数.

解法二　树链剖分

直接用线段树来维护路径最大值.

其他解法 [3]

References

[1] PoPoQQQ. Link-cut-tree.

[2] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. pages 114--122, 1981.

[3] Yang Zhe. Some research on qtree solution.