**《水管局长》解题报告**

南京外国语学校 贾志鹏

**一、摘要**

核心算法思想：维护最小生成树

主要数据结构：伸展树

其它辅助知识：*Link-Cut Trees*

时间复杂度：*O(MlogM + (N+Q)\*logN)*

空间复杂度：*O(N + M + Q)*

**二、题目大意：**

给出一个无向带权图（没有重边），支持两个操作：

1. 给出顶点a和b，确定w的最小值，使得存在一条a到b的路径，这条路径上边的权值都不超过w。

2．删除一条当前存在的边。

**三、算法分析：**

对于操作一，可以这么思考：将边按照权值从小到大加入，直到顶点a和b连通为止。可以发现，这个和经典的Kruskal算法的执行过程很相似，因此我们猜想这个w是原图最小生成树中a和b路径上边权的最大值。

设原图G的最小生成树为T。利用反证法，假设这个命题不成立，那么在G中就存在一条a和b之间路径P，这条路径上边权的最大值w’小于T中a、b路径上边权的最大值w。由于w’<w，因此P中必然存在非T中的边，且权值为w的边不在P中。然后对于P中的非树边，必然存在一条边(u, v)，使得T中u、v的路径上包含了权值为w的边。然后可知(u, v)的权值小于w，因此我们删除T中u、v路径上权值最大的边，也就是权值为w的边，再插入(u, v)，所得到的还是原图的生成树，且边的权值和小于T，与T为最小生成树矛盾，所以猜想得证。

证明了上述命题，我们可以发现只需要维护当前无向图的最小生成树就可以了。但是还有删除一条边的操作，这个不是很好解决。

这时候我们可以换一种思路。由于题目没有要求算法是在线的，因此我们可以倒着处理询问，这样删除边的操作就变成了加入边的操作。对于当前最小生成树T，如果加入一条边(u, v)，可以发现就形成了一个环，并且这个环包括u和v，就如下图所示：

为了维护**最小生成树**，显然我们要去掉新产生的环上权值最大的边。这个环由两部分组成：原来树中u到v的路径、边(u, v)。假如(u, v)的权值大于树中u到v路径上边的最大权值，那么我们无需插入(u, v)，否则删除u到v路径上权值最大的边并且插入(u, v)边。

有了以上分析，现在的问题就很明了了。维护一棵树，支持以下三个操作：

1. 加入一条边
2. 删除一条边
3. 询问两个结点路径上边权值的最大值

对于这个问题，我下面提出两种解决方法。

【算法一】

最容易想到的算法显然是朴素算法。首先我们先把最初的最小生成树从无根树变成有根树。我们记录每个结点的父结点和指向父结点边的权值（根结点的父亲为null）。对于询问操作，假设询问u和v路径上边权的最大值，我们先顺着u向上访问，直到到达根结点，这个过程中我们记录下每一个访问过的结点。然后再顺着v向上访问，这时第一个出现在刚才u的访问序列中的结点就是u和v的LCA。这个过程中我们就可以找到u和v路径上边权的最大值。对于删除边的操作，显然很好处理。假设要删除边(u, v)，且u是v的父结点，那么我们将v的父结点赋为null就可以了。

下面来看加入边的操作。假设我们要加入边(u, v)，且u和v在两个子树中。如果v是其所在子树的根结点，那么只需要将v的父结点赋为u就可以了。但是v不一定是其所在子树的根结点，这时我们需要改变一些边端点的父子关系，使得v变成根结点。这个过程如下图所示（箭头指向子结点）：

实际上我们就是把v到根路径上的边反向，这样v就变成了这棵子树的根，然后我们再把v的父结点赋为u就可以了。

可以看出这个算法删除边操作的复杂度为O(1)，询问和加入边操作的复杂度均为O(n)。但是实际上询问加边操作的复杂度都和当前处理结点到根的距离（我们定义为结点的深度）有关。我个人感觉对于一个n个结点的有根树，其结点深度平均值的期望值为O(logN)，不过我无法证明这一点，但是通过后面的测试可以看出这个算法在随机数据中的效率还是非常高的。

由于一开始求解最小生成树的时间复杂度为O(MlogM)[[1]](#footnote-1)，因此算法一的总时间复杂度为O(MlogM + QN)，但是期望的时间复杂度为O(MlogM + QlogN)，空间复杂度为O(N + M + Q)。

【算法二】

对于官方给出的测试数据，算法一已经可以在规定的时间内出解，但是这并不意味着不存在时间复杂度更低的算法。

可以发现现在主要的问题就是对一棵树进行维护，这是经典的**动态树问题**。动态树问题公认的时间复杂度下界为每次操作O(logN)。这个题我们只需要回答路径上的信息，可以通过Link-Cut Trees来维护整个树，使得前文中要求的操作都做到平摊O(logN)的时间复杂度。

Link-Cut Trees是由Sleator和Tarjan提出的一种动态树问题的解决方案，可以处理对树上路径的操作，其核心思想就是**树链剖分**，然后用**伸展树**维护树链。

先介绍一下树链剖分的概念。对于一棵有根树，其中某些边是**重边**，其余的边是**轻边**，且满足任何一个结点指向子结点的边中最多只有一条重边。然后我们将连续的重边连成路径，称作**重路径**，也就是所谓的**树链**（单独结点也算一条树链）。假如剖分方式如下图所示：

那么就产生了4条树链：{v1, v4, v8}、{v3, v7}、{v2, v6, v9}和{v5}。可以发现，树链之间通过轻边连接，对于一条树链，其最上方的结点向上的边必然是一条轻边，我们将这个结点的父结点称为这条树链的**Path Parent**。对于这些树链，如果我们能够高效地维护，那么整个问题也就能够高效地完成。

在Link-Cut Trees中，每一条树链我们使用伸展树来维护。对于一个伸展树，其中序遍历就表示一条树链（中序遍历中靠前的结点在树中靠上），并且这个伸展树的根结点处记录了这条树链的Path Parent。Link-Cut Trees的核心操作就是**Expose操作**。Expose(x)就是将结点x到根的路径变成一条重路径（x指向其子结点的边均为轻边）。这个操作实际上不是非常复杂，伪代码如下：

|  |
| --- |
| Expose(x)  u ← x  v ← null  while u ≠ null do  Splay(u)  t ← RightChild[u]  Pre[t] ← null  PathParent[t] ← u  RightChild[u] ← v  Pre[v] ← u  v ← u  u ← PathParent[u]  end while |

通过一些复杂的分析[[2]](#footnote-2)，可以证明Expose操作的平摊复杂度为O(logN)[[3]](#footnote-3)。有了Expose操作，前文中提到的操作现在都可以实现了。

先来看询问操作。设每个结点在树中与其父亲边的权值为UpWeight，那么我们在伸展树中就要维护一个子树中结点UpWeight的最大值。假如我们现在要询问结点u和v路径上边权的最大值，那么我们先Expose(u)，然后再Expose(v)。在Expose(v)的过程中，必然会出现一个结点x，使得x和根结点在同一个树链中，那么显然x就是u和v在树中的LCA。实际情况如下图所示：

我们执行完Splay(x)后，可以发现u到v的路径被分成了两部分：u到x和v到x。u到x的路径对应的是x右子结点为根的伸展树，v到x对应的是当前v所在的伸展树。这样我们就解决了询问操作。

接着来看删除边和插入边的操作。删除边的操作非常好实现，伪代码如下：

|  |
| --- |
| Cut(u, v)  Expose(u)  Splay(v)  PathParent[v] ← null |

接着是插入边的操作。假设我们要插入边(u, v)，并且u和v处在不同的子树，v是其所在子树的根结点，那么这个操作实现也非常简单：

|  |
| --- |
| Link(u, v)  Splay(v)  PathParent[v] ← u |

但是现在的问题是v不一定是其所在子树的根结点。根据算法一，可以发现将v变成根的过程实际上就是将v到根的路径取反。因此我们只需要在伸展树上加入一个表示翻转的标记就行了[[4]](#footnote-4)。不过这里在实现的时候需要注意一些问题。可以发现如果一条树链被取反了，那么每个结点的UpWeight都会改变，并且是变成原来树链中这个结点向下的边。因此我们还需要对于每个结点在伸展树中维护DownWeight，同时也需要对子树维护DownWeight的最大值。

至此，问题已经得到了完美的解决，我们得到了一个时间复杂度为O(MlogM + (N+Q)\*logN)的算法，并且空间复杂度和算法一一样为O(N + M + Q)。

**四、心得体会：**

深入思考问题，看清问题本质，再使用数据结构高效地维护。

**五、附录：**

【程序测试】

评测平台：

Core i5 430M @ 2.27GHz / 2GB

Windows 7 32bit

编译器：

Pascal：fpc 2.0.4、dcc32 v14.0

C++：g++ 3.4.2 MinGW

测试结果：

一、比较算法一和算法二。随机数据，M为N的10倍，Q为100000，操作1和操作2次数比为7:3，N从10000到100000递增。两个程序均为C++程序且开启O2优化。

二、比较C++程序和Pascal程序。构造数据，M为N的10倍，Q为100000，N从40000到100000递增，操作一和操作二比例7:3。程序一为C++代码，使用g++编译，开启O2优化；程序二为Pascal代码，使用Delphi v6.0编译，编译开关默认；程序三为Pascal代码，使用fpc编译，开启O2优化，所有函数和过程inline。三个程序均为算法二的实现。

三、比较C++程序和Pascal程序。数据、程序都和二一样，程序一为C++代码，使用g++编译；程序二为Pascal代码，使用fpc编译。均不开启任何优化。

【参考资料】

1. 杨哲，07年国家集训队作业《QTREE解法的一些研究》
2. 陈首元，06年国际集训队论文《维护森林连通性——动态树》
3. <http://compgeom.cs.uiuc.edu/~jeffe/teaching/data-structures/notes/07-linkcut.pdf>

1. 这里使用的是并查集优化的Kruskal算法。当然我们也可以利用斐波那契堆优化Prim算法，时间复杂度为O(NlogN + M) [↑](#footnote-ref-1)
2. 详细的分析过程可参见附录中的参考文献 [↑](#footnote-ref-2)
3. 严格地说应该是在同一棵树中连续执行M次Expose操作的总复杂度为O((N+M)\*logN) [↑](#footnote-ref-3)
4. 关于给伸展树加入标记的技巧可以参见NOI2005的《维护数列》 [↑](#footnote-ref-4)