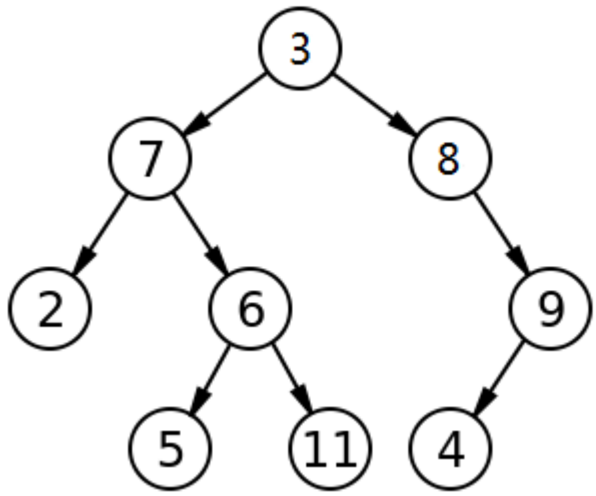
基于欧拉环游树的动态树实现

冼臧越洋 张哲恺 胥拿云 龙思杉

欧拉环游树(ETT)：

DFS 序：



（图 1）

DFS 序就是按先序遍历的方式遍历一棵树，例如图（1）的 DFS 序为：

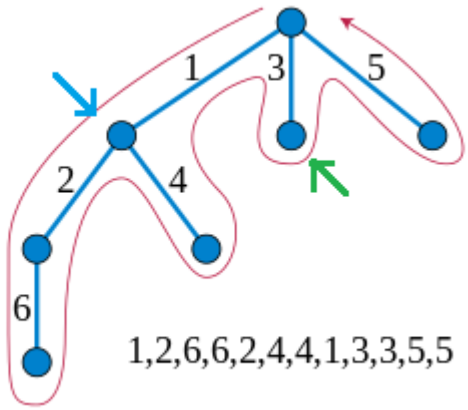
3 7 2 6 8 11 5 9 4

可以发现，任意一颗子树的所有点均在一段连续的区间上，这样就可以方便的维护子树的信息。利用维护子树大小甚至可以支持换父亲的操作。

然而它不能很好的支持换根操作，例如在上图中我们把 9 换成根，那么 DFS 序应为：

9 4 8 3 7 2 6 5 11

它和原来的 DFS 序关系不是很明显（其实是有的），为了能够很好的支持换根，我们掏出了 ETT：



图（2）

维护边的 ETT 比较自然，所以我们先用边的 ETT 做例子。对于图（2）的树，我们将每条无向边拆成两条有向边，然后以 DFS 遍历树：

1 2 6 6 2 4 4 1 3 3 5 5

这称为欧拉环游序，显然依旧有一颗子树的边在一段连续的区间上这个性质。

同时我们发现这样做其实是把树变成的一个环，那么对于换根操作无非就是将这个环的起点挪一挪，例如我们将根转为蓝箭头指向的点，那么欧拉环游序变为：

1 3 3 5 5 | 1 2 6 6 2 4 4

注意到从“|”处交换两段序列的位置既得原序列，反之亦然。这样我们就很好的支持了换根。

它也能支持换父亲操作，例如把蓝箭头所指的子树切下来（就是把父亲换成空），我们只需找到它的两条出入边（就是 1），然后从原序列切下来即可（此时边 1 被销毁）。序列变为：

2 6 6 2 4 4

3 3 5 5

如果要把它连到绿箭头所指的点上，我们只要找到这个点的入边，然后插在这条边的后面即可（需要加条边，譬如 7）。

3 7 (2 6 6 2 4 4) 7 3 5 5

这里的 7 是由于加了条边先夹在了 3 中间，（）实际找到的边是 7。

同时我们也能做维护点的 ETT（只是有点奇怪），例如对于图（1），它的点 ETT 序列为：

3 7 2 2 7 6 5 5 6 11 11 6 7 3 8 9 4 4 9 8 (3)

其实它和边的 ETT 没有什么区别，无非是边变成了用两个点描述而已，一个是起点一个是终点。如果把这个序列首尾相连形成环就能发现它就是边的 ETT。

当然它也能换根换父亲，具体不在详细介绍。

如果我们不用换根只有换父亲，那么点 ETT 序列可以简化成：

3 7 2 2 6 5 5 11 11 6 7 8 9 4 4 9 8 3 (*)

任意一点为根的子树即为它在序列对应的两点之间的部分，相对于上一个序列它简化的多然而换根不能。

由于对 ETT 序列的操作就是区间切割，区间插入，区间修改，所以我们使用 splay 来维护它。所以对任意一个操作它的复杂度为 O(logN)。

Link-cut-tree

它不是我们的重点，所以就不细讲了（反正你们都懂）。我们知道它可以维护链上的信息，并且 preferred child 的切换次数是均摊 O(logN)的。

ETT+LCT

如果我们想同时维护链操作和子树操作怎么办？显然不是每条链在 ETT 上都是连续的，用 LCT 维护子树听着更蛋疼（然而貌似 top-tree 就是这么干的）。难道链用 LCT，子树用 ETT？然而并不能把信息合并，所以这样做是不行的。

但注意到链其实就是一个特定的 DFS 序，或者说点人话，ETT 中最左边和最右边的一段其实就是一条链，例如(*)中最左边的 3 7 2 和最右边的 4 9 8 3。这就给了我们一个启发：如果我们能保证每个点的 preferred child 都是第一个遍历的点（或者形象点，最左边的儿子），那么链上的点也能在 ETT 上形成连续的区间，也就能高效地修改了。

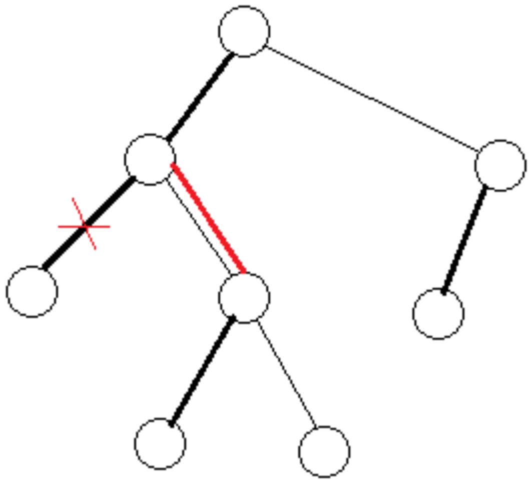
对于不用换根的树来说，这比较的简单。每次 LCT 的 access 中 preferred child 变化的时候（例如现在 7 的 preferred child 变为了 6），在 ETT 中把 preferred child 对应的子树切下来移到它新父亲的右边即可，对于例子来说就

是：

3 7 2 2 ｜ 6 5 5 11 11 6 ｜ 7 8 9 4 4 9 8 3

-> 3 7 ｜ 6 5 5 11 11 6 ｜ 2 2 7 8 9 4 4 9 8 3

这样 3 7 6 就变成了一段连续区间了。



证明十分简单，只要原先每个 preferred child 是在最左边，那么将每个变化的 preferred child 移到最左边那么性质依旧满足。

例如上图，链上其他重边已经是最左边的边了，只要把红边移到最左边即可。因为 preferred child 变化次数均摊 O(logN)，每次区间操作是 O(logN)的，所以复杂度为 O(log^{2}N)的。

算法局限：对于换根的问题

LCT 和 ETT 都支持换根，看起来这个算法能轻松愉悦支持换根吧。然而根本不是这样的好不啦。LCT 换根后新根到旧根的链需要翻转，如果我们暴力的在 ETT 上做是什么样的呢？例如 1-2-3-4-5 的树，根为 1，现在转成 5：

|

6-7-8

先 access(5) ETT 序列:1 2 3 4 5 5 4 3 6 7 8 8 7 6 2 1

然后愉快的翻转，正常来说 ETT 序列应该是:

5 4 3 2 1 1 2 6 7 8 8 7 6 3 4 5

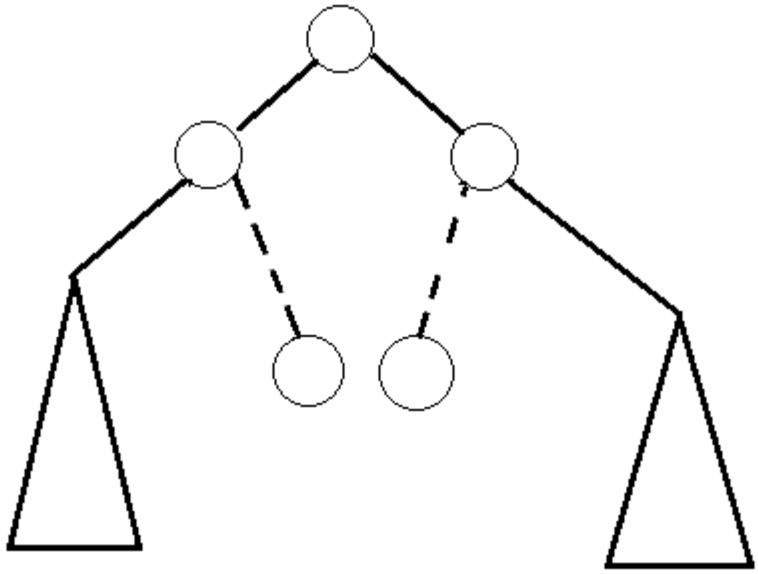
或者是

5 4 3 6 7 8 8 7 6 2 1 1 2 3 4 5

那么这是怎么做的呢，把 5 切出来，把 4 切出来放到 5 里，把 3 切出来放到 4 里。。。。。。这个复杂度就变成了链长度*log(N)，瞬间爆炸。

算法可能的改进：

问题显然出在了 pushdown 翻转标记的时候，那么我们在 pushdown 的时候同时改变 ETT 的形态就可以了。例如



这是一颗辅助树，当根的反转标记下传时，我们若能知道这棵树左子树最右边的点 A，根 B，右子树最左边的点 C，那么在 ETT 序列中：

..........ABC........CBA........

把 C 切出来，把 B 插进 C 右边，把 A 插在 B 右边，把 C 插回去，就完成了形态的同步。

这样要在辅助树上多维护一些信息，或者暴力的去做，或者用 ZZK 的 SPLAY 的迭代器的前驱后继，复杂度大概要再多个 logN（算上常数快赶上 O(N)了）

另一种思路：

如图所示，图中的实边表示 LCT 中的实边，此时 ETT 的边序列为

1,2,4,4,2,3,5,5,6,6,3,1,7,8,8,9,9,7

考虑将点 A 提成根，首先在 LCT 中 access 点 A，则 ETT 序列变为

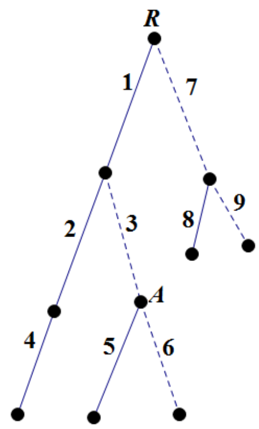
1,3,5,5,6,6,3,2,4,4,2,1,7,8,8,9,9,7

然后在 ETT 上提根，变为

3,2,4,4,2,1,7,8,8,9,9,7,1,3,5,5,6,6

如果我们把两个 3 之间的序列翻转，则变成

3,1,7,9,9,8,8,7,1,2,4,4,2,3,5,5,6,6



不难发现，原来的 A 到 R 这条链上的边在 ETT 上依然是连续的，同时这条链上其它子树的序列全部被翻转。

既然被翻转了，那么如果能在需要的时候将这些序列翻转回来就可以了，于是我们得到一种新的思路，提根时将新的根和旧的根之间的点上打上标记，access 时如果发现上面的点上有标记，就将其除了 preferred child 以外的点的子树翻转，同时去掉标记即可。利用 Splay 可以实现链上打标记和区间翻转。所需要做的是在每次切换 preferred child 时额外花 O(logN)的时间进行区间翻转。操作的次数上面提到是均摊 O(logN)的，这样的时间复杂度依然在 O(Nlog^2N)

实践表明这样的做法是可行的（见 github 上的 experiment 分支），但是运行速度并不理想，在应对 10000 个点的数据时甚至比暴力慢许多。