Self-Adjusting Top Tree

杨嘉成 林伟鸿 谭博文

April 29, 2016

Abstract

Self-adjusting Top Tree（简称 Top Tree）由 Robert E. Tarjan [1] 发明，这是一种能够在均摊 $O(n \log n)$ 的时间复杂度内支持完全动态树的所有操作的数据结构 [2]，从而从根本上解决了完全动态树问题。而完全动态树问题在许多算法设计时都会很有作用，例如最大流问题就可以使用 Top Tree 进行优化。本文将从 Top Tree 的原理出发，讲述 Top Tree 的实现以及一些应用，并且给出 Top Tree 的复杂度证明。

Contents

1 Basic idea

Top Tree 的基本想法是想将一颗树通过两种操作将这颗树压缩成一条边，这两种操作分别是 Rake 和 Compress，Rake 是指将度数为 1 的节点合并到与其相邻的子树上来，而 Compress 操作是指如果存在边 (u, v) 和边 (v, w)，那么现在我可以将这条边压缩成 (u, w)。

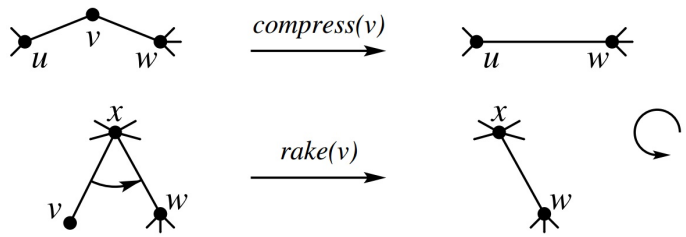


图 1: Rake 和 Compress 操作

通过这两种操作，我们就可以将一颗树压缩成一条边，而正是在这样的压缩过程中，我们可以对树的形态和信息进行维护，而 Top Tree 就是用来维护这样一种 Rake 和 Compress 序列（以下简称 RC 序列）的数据结构。

2 Construction and structure

Top Tree 是以 Link Cut Tree 为基础而实现的，但是 Link Cut Tree 并不支持子树操作，而 Top Tree 通过添加 Rake 节点让子树操作成为了可能，在这一段中，我们会先对 Link Cut Tree 进行介绍，然后再对 Top Tree 的构造方法和结构进行介绍。

2.1 Link Cut Tree

LCT 是 link-cut-tree 的缩写，这是一种支持链操作以及删边加边的动态树。其储存方式很简单，我们把树边分成两种，第一种称为“重边”，第二种称为“轻边”。

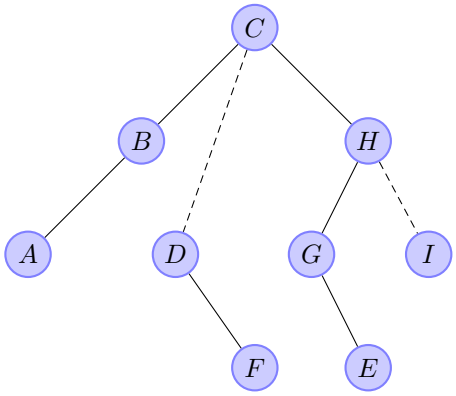


图 2: Link Cut Tree 的一个例子

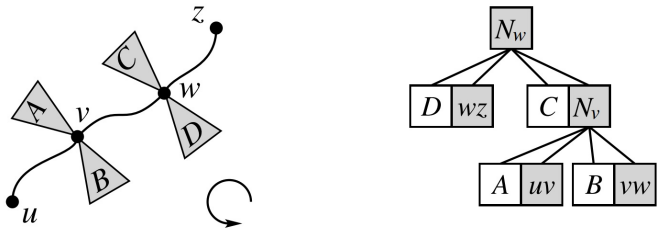


图 3: Top Tree 的结构

如上图，C 为该树的根，那么实线为“重边”、虚线为“轻边”。

我们规定，这些“重边”在树上组成的链必须只有两个端点，且这两个端点必须是子孙关系（也就是说与每个点相邻的“重边”最多只能有 2 条）。

这样，我们发现，树被分成了一堆重链和一堆轻边，且每个点最多只会被一条重链覆盖。然后 LCT 的思想是用 splay 来维护每条重链 (关键字为该点的深度)，利用 access(x) 操作将 x 点到根路径上的点放进同一条重链中（断开他们与其他链的重边连接）这个核心操作，完成动态树的加边删边以及链操作这些基本功能，通过势能分析可以证明这个操作平均是 $O(n \log n)$ 的复杂度。

2.2 Top Tree

2.2.1 Basic Structure

Top Tree 的思想是在 LCT 的基础上进行一定的扩展。我们依然将树按轻重边剖分，然后可以看出，重边组成了一些链，我们仍然用 splay 来维护这些链，只不过，与 LCT 不同，在 Top Tree 中 splay 维护一条链时，不是存链上的点，而是存链上的边，而且为了保证算法总体复杂度，我们需要建立一些额外 splay 树节点（其实可以表示成树上的节点），使得表示边的节点都“沉”到叶子上去（splay 关键字为该边到根的距离）。

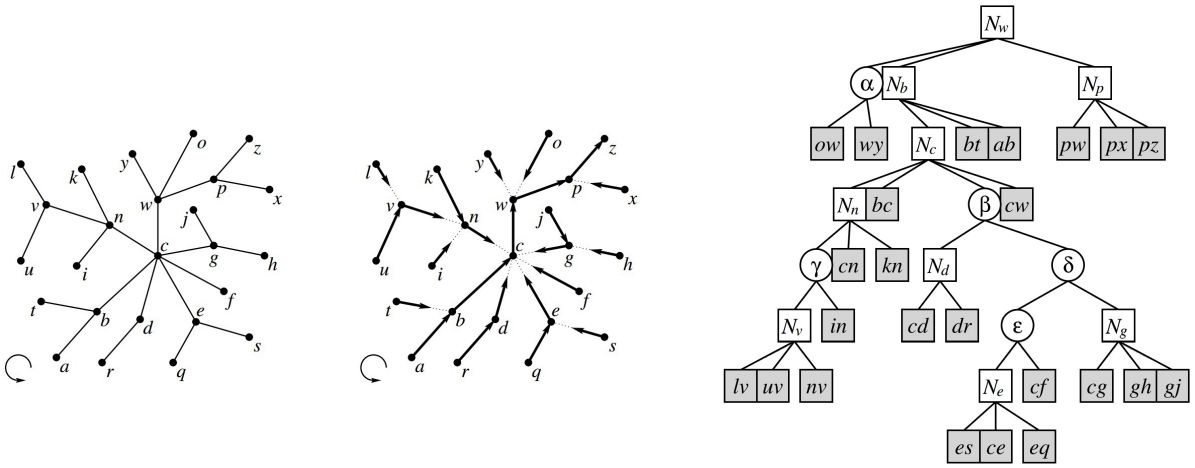
然后，LCT 算法不考虑虚边，因为它只打算维护链的信息，但 Top Tree 不同，Top Tree 在维护链信息的同时，还需要维护子树信息，因此，我们得把虚边也进行一些维护。

在 Top Tree 中，按照链剖分，一个点所出去的虚边会分成最多两个部分。(这里我们可以考虑一下边表存边的时候，一个点去儿子的边最多只有 1 条是重边，那么按顺序遍历边的时候，该点出去的虚边自然而然的可能会被这条重边分割成两个部分)。我们认为两个这些虚边被分成“一左一右”两个集合，然后我们将这些虚边也变成一个树形结构。也就是说，我们把每个点的虚边也建成一棵树（后面称为虚树），依然加入一些虚节点使得所有虚边沉到这棵树的叶子节点去，然后代表虚边的叶子节点，如果该虚边所去儿子在某条重链上，那么直接用这个儿子所在重链的 splay 的根来表示，否则直接用该虚边表示。　这张图可以稍微说明一下树的结构，N_x 代表 x 节点，然后左边的树可以表示成右边的 Top Tree（假设 z 是根）。可以看出，每个点的第 2、4 个儿子在这里表示的是 splay 上的儿子，我们按照重链将 w、v 的虚边儿子集合分成了 2 个部分，我们把这两个子树集分别用该点的虚树的左右子树来存储。

2.2.2 Details

首先，对于每个节点，我们记录四个儿子（准确地说，可以看成两棵二叉树），前两个儿子表示这个点所在的链 splay 上的儿子，后两个表示这个点的虚树的左右儿子。然后虚树中表示虚边的点在叶子上，并且如果这条虚边所去儿子在某条重链上，这个点就是这条虚边出去的儿子所在重链的 splay 的根（也就是另一个树中的节点，它也有最多 4 个儿子，定义和前面说的点一样），否则直接用这条虚边来表示。然后除了虚树上的虚节点，每个树中的点都是类似的结构，也就是可以看成 LCT 和虚树相互交错的形态。

下图为根据这个定义建出一棵 Top Tree 的例子。



(a) 一棵树的一种分割方案

(b) 左图对应的 Top Tree

上面的图中，靠上的图分成左右两个部分，左边为这棵树的形态，右边为轻重边的剖分情况（z 是这棵树的根节点）。下面的图就是根据上面的剖分情况建出来的一棵 Top Tree，可以看出 N_w 表示 w 点，是根节点所在重链的上的一个点，其 splay 上的左右儿子节点是 N_b 和 N_p，因为 w 在这种剖分下并没有虚边，故没有虚树的两个儿子，然后 N_b 和 N_p 也具有类似的子结构。

考虑 N_c，其重链 splay 上的两个儿子代表 bc,cw 两条边，然后其根据重链把虚边分成了两个集合，其虚树左儿子集合只有一条虚边 nc，因此左儿子直接就是叶节点，并且用此时 n 所在重链的 splay 的根 N_n 表示，而 N_n 具有一样的儿子结构。虚树右儿子有不止一条虚边，因此在右边虚子树中添加了三个虚节点，将这 4 条虚边存在虚树的叶子上，依然用虚边所去的儿子所在的 splay 的根来表示，这些叶子节点（表示树中的一个节点，或者当虚边所去儿子不在任何一条重链上是，就是用这条虚边来表示）也具有一样的儿子结构。

综上，我们可以看出 Top Tree 是在 LCT 的基础上，转而维护边集，splay 中内节点存的是树中的节点，叶子节点表示重链上的边。对于每个点，其在重链的 splay 上，然后它自己也有一棵虚树，表示其出去的虚边，并且我们增加了一些虚点使得这棵虚树的所有表示虚边的节点都是叶子节点，每条虚边有两种可能的表示，第一种是这条虚边所去的儿子不在任何一条重链上，那么我们就用这条虚边表示，第二种就是该虚边所去的儿子在某条重链上，那么我们用这个儿子所在的重链的 splay 的根节点表示该虚边。总的来说，Top Tree 可以看成 LCT 和虚树相互交错联系的形态。

3 Implementation

敝队的实现方法与 tarjan 论文中稍有不同，论文中的 Top Tree 基于 RC Tree，RC Tree 的信息是集中在边上维护的，我们的程序实现基于 Tarjan 之前发明的一种数据结构——Link Cut Tree（简称 lct），而 lct 的信息是集中在点上维护的。在此基础上改良使之同样能够实现维护子树信息的功能。Link Cut T ree 是一种处理动态树问题的有力武器，但一大缺陷是无法高效地维护子树信息。由于实现方法基于 LCT 的改良。建议读者能够先熟练掌握 LCT。

3.1 Link Cut Tree

考虑子树操作和子树查询，若使用传统的 lct，做法可以是先把询问点所在的链进行修改或结果加入答案，然后沿着虚边走到更为底端的链上继续操作。显然，这样的复杂度是我们不可以接受的。但这个方法给了我们一些启示，要

统计子树信息，就要想办法减小遍历虚边对复杂度的贡献。而我们可以在 lct 上的每一个点上都维护一个平衡树，保存子树信息，使询问的时候更快地得到答案。我们称这样每个点上的平衡树为 AAA Tree（有关 AAA Tree 的操作将在后面介绍）。

有关 LCT 的操作均与原 LCT 类似，在执行子树操作和子树询问时，我们要做 access(u)，可以把 u 和 u 的子树之间的边全部变成虚边，这样就可以简化为给一棵 AAA Tree 打标记。link、cut、链操作以及换根操作和普通的 LCT 类似，需要特殊处理的是子树操作，我们的处理方式是将子树 access 到根，这样就保证当前询问的节点到根一定是根路径，并且当前节点没有重儿子。因而询问或操作的子树节点一定是 AAA Tree 中的点再加上当前节点。

3.2 AAA Tree

AAA Tree 的主要操作有两个：add 和 delete，分别表示将一个节点插入另一个节点的 AAA Tree 和从一个节点的 AAA Tree 中删除一个节点。这部分其实是非常简单的，就是普通的平衡树的插入和删除。在 LCT 的核心操作 access 中，需要注意在断开一条重链时，要把其中一个点加入另一个点的 AAA Tree 中，添加一条重边时要在一个点 u 中执行 delete(v) 操作。同时请注意在 AAA Tree 中我们仍然需要下传子树标记并且维护子树的和。

在我们的实现中，AAA Tree 是使用 Treap 来实现的，因而效率比单独用 splay 的效率要高一些。

3.3 Tags

我们要维护两种标记，分别是链标记和子树形标记，链标记只对一颗链 splay 有效，而子树标记对整个子树都有效。实现上，我们可以将这两种标记分优先级，子树标记是最高级，他可以改变在这个点以下的所有链标记，而链标记作用于 $data$。建议为了实现上的方便使用先导型的标记，即打了标记表示这个点等待操作，并且在实现时请不要使用后导型标记，即修改完之后再打标记，因为这样会造成 AAA Tree 和 LCT 的连环修改从而可能增加额外的复杂度。

3.4 Framework

我们写了 6 个大类，其中的继承和组合关系如下：

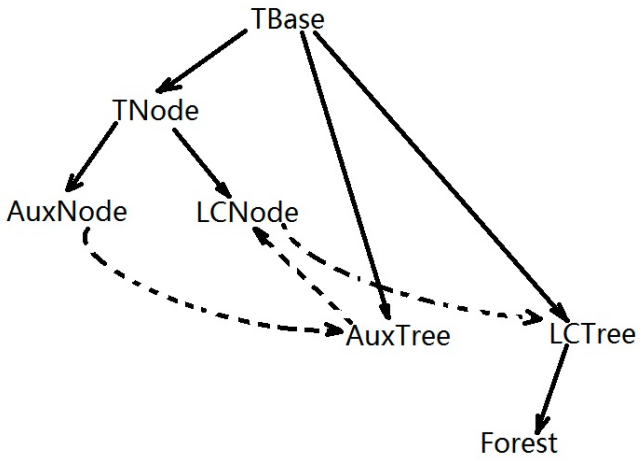


图 4: 代码框架示意图

这几个类的作用如下：

- TBase: 用来做所有内的基类，其它类均继承自这个类；
- TNode: 节点的基类，继承自 TBase 内部存放了两个 functor 的静态成员；
- AuxNode: AAA Tree 节点类，继承自 TNode 内部有 update、pushTagTree 等函数；
- AuxTree：AAA Tree 类，继承自 TBase，用来存储维护虚边信息；
- LCTNode：LCT 节点类，继承自 TNode，内部有 update、pushTagTree、pushTagChain 等函数；
- LCTree：Link Cut Tree 类，继承自 TBase，用来维护实边信息和形态，并且提供用户操作端口；
- Forest：主类，提供用户接口，并且内嵌一个节点类（作用类似迭代器）。

在实现过程中，我们采用了类模板的设计方法，我们传入三个类模板参数 T、A、M，其中 T 表示数据类型，A 表示一个 T 和 T 的加法仿函数（Functor），M 表示一个 size_t 类型的和一个 T 类型的仿函数，这样的实现方法可以支持维护大部分数据，美中不足的是，这样的结构并不能处理多个标记的问题，若需要处理，则需要至少传入 6 个类模板参数，分别是 T、D、A、G、O、M 分别是数据类型，标记类型、T 和 T 的加法、T 和 D 的加法、D 和 D 的加法、size_t 和 D 的乘法。

References

[1] Robert E. Tarjan and Renato F. Werneck. Self-adjusting top trees. *SODA '05 Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, Session 1A(813-822), 2005.

[2] Wikipedia, https://en.wikipedia.org/wiki/Top_tree. *Top Tree*.