

维护森林连通性——动态树

华东师大二附中

陈首元

本文将介绍一种数据结构，称为动态树，它能够维护一个带权的森林，并支持 `link` 操作，用途是将两棵树合并。支持 `cut` 操作，用途是删除一条边，是一棵树分为两棵。在网络优化中的用途十分广泛。

[动态树的基本操作]

`Parent(v)`: 返回 v 的父亲节点，如果是根返回 `null`。

`Root(v)`: 返回包含节点 v 的树的根。

`Cost(v)`: 返回边 $(v, \text{parent}(v))$ 的费用，假定 v 不是根。

`Mincost(v)`: 返回从 `root(v)` 到 v 的路径上权最小的边。

`Update(v, x: real)`: 使从 `root(v)` 到 v 路径上的边的费用 $+x$ 。

`Link(v, w, x: real)`: 将以 v 为根的树连接到节点 w 上， (v, w) 的费用为 x 。

`Cut(v)`: 从树中删除 $(v, \text{parent}(v))$ ，分为两棵树。

`Evert(v)`: 翻转，将 v 设为根，并将 v 到 `root(v)` 上所有边反向。

1、通过两次 `update` 可以修改一条边的费用。

2、`Mincost` 可以改为 `Maxcost`

假设初始情况下有 n 个单独的点，接下来要执行 m 步上述的操作。

显然，通过保存 `dparent(v)`, `dcost(v)`，分别记录 v 的父节点，与边的费用，可以实现朴素算法，在 $O(1)$ 实现 `parent, cost, link, cut` 而 `root, mincost, evert, update` 的复杂度与树的深度有关，最坏情况下是 $O(n)$ 。

本文的算法，并不直接对整棵树进行操作，通过这种算法， m 步操作可以在 $O(m \log N)$ 内实现。

将树中的边分成实边虚边两种，从每个顶点出发，最多有 1 条实边连向它的子节点。一个路径包括一些自底向上连通的实边。剩下的边都是虚边，通过记录 `dparent(v)`, `dcost(v)`，可以将所有的虚边都保存下来，虚边可以在一定条件下转化为实边并保存。如果 `tail(P)` 是树根，那么 `dparent(P)=null`。

通过对完全由实边组成的路径进行操作，就能够实现动态树操作了。这里假定已经实现了以下的一些路径操作，先说明这些路径操作的功能，再介绍如何通过这些路径操作实现前面所说的基本操作，最后再讨论实现这些路径操作的方法。

[路径结构的基本操作]

`Path(v)`: 返回包含 v 的路径（每个路径有一个标志）

`Head(p)`: 返回路径 p 的首节点

`Tail(p)`: 返回路径 p 的尾节点

`Before(v)`: 返回路径中 v 的前驱节点

After(v): 返回路径中 v 的后继节点

Pcost(v): 返回边 $(v, \text{after}(v))$ 的费用

Pmincost(p): 返回 p 中费用最小的边

Pupdate(p, x:real): 将 p 中每条边的费用 $+x$

Reverse(p): 将 p 中的每条边反向

Concatenate(p, q, x:real): 添加边 $(\text{tail}(p), \text{head}(q))$ 费用为 x , 将路径 p, q 合并

Split(v): 通过删除与 v 相连的边, 将路径 $\text{path}(v)$ 分为三部分, 返回 p, q, x, y ,

p 是 $\text{head}(\text{path}(v))$ 到 $\text{before}(v)$, q 是 $\text{after}(v)$ 到 $\text{tail}(\text{path}(v))$ 。

x 是 $(\text{before}(v), v)$ 的费用, y 是 $(v, \text{after}(v))$ 的费用。

如果 v 是头节点, 那么 p 是 null , 如果 v 是尾节点那么 q 是 null 。

通过下面两种操作, 我们就能维护树中的实边虚边, 并实现动态树的基本操作。

Splice(p: path): 作用是将路径 p 向更靠近根的方向增长。

实现方法: 把虚边 $(\text{tail}(P), \text{dParent}(p))$ 变为实边, 为了维护实边的性质, 将原来从 $\text{dParent}(P)$ 中连出的边设为虚边。

下面是伪代码

Function splice(p: Path);

Begin

$U := \text{dparent}(\text{tail}(P));$

$[q, r, x, y] := \text{split}(u);$

 If $q \neq \text{nil}$ Then Begin

$\text{dparent}(\text{tail}(q)) := v;$

$\text{dcost}(\text{tail}(q)) := x;$

 End;

$P := \text{concatenate}(p, \text{path}(P), \text{dcost}(\text{tail}(P)));$

 If $r = \text{nil}$ Then return p

 Else return $\text{concatenate}(p, r, y);$

End;

Expose(v: vertex): 作用是从 v 到 $\text{root}(v)$ 中所有边设为实边。

实现方法: 不断调用 splice 直到根为止。

Function expose(v: vertex);

Begin

$[q, r, x, y] := \text{split}(v);$

 If $q \neq \text{nil}$ Then Begin

$\text{Dparent}(\text{tail}(q)) := v;$

$\text{Dcost}(\text{tail}(q)) := x;$

 End;

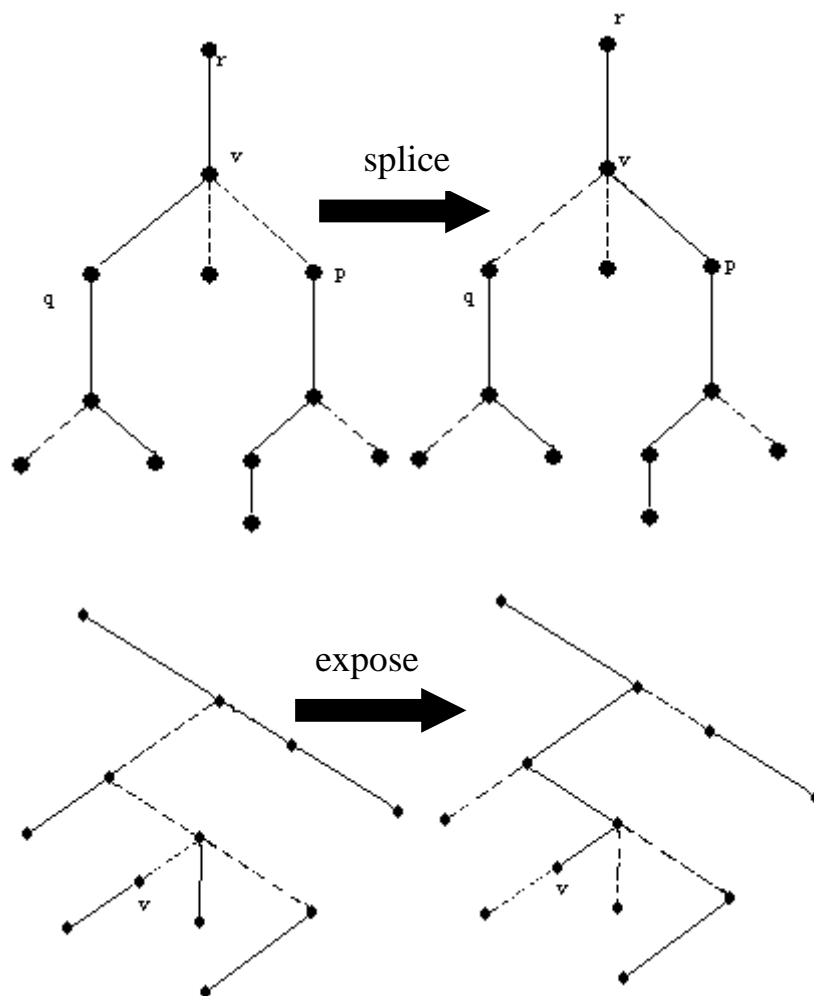
 If $r = \text{nil}$ Then $p := \text{path}(V)$

 Else $p := \text{concatenate}(\text{path}(V), r, y);$

 While $\text{dparent}(v) \neq \text{nil}$ do $p := \text{splice}(p);$

 Return $p;$

End;



通过上面 2 个操作，和路径的基本操作，我们就能把 8 个动态树基本操作实现了。

Function Parent(v :vertex)

Begin

 If $v = \text{tail}(\text{path}(v))$ Then Return Dparent(v)

 Else Return after(v)

End;

Function cost(v :vertex)

Begin

 If $v = \text{tail}(\text{path}(v))$ Return dcost(v)

 Else Return Pcost(v)

End;

Function root(v :vertex);

Begin

 Return tail(expose(v));

End;

Function Mincost(v :vertex);

Begin

 Return Pmincost(expose(v));

End;

Procedure Update(v:vertex;x:real);

Begin

 Pupdate(expose(v),x);

End;

Procedure Link(v,w:vertex;x:real);

Begin

 Concatenate(path(v),expose(w),x);

End;

Function Cut(v:vertex);

Var

 p,q:path;

 x,y:real;

Begin

 Expose(v);

 [p,q,x,y]:=split(v);

 Dparent(v):=nil;

 Return y;

End;

Procedure Evert(v);

Begin

 Reverse(expose(v));

 Dparent(v):=nil;

End;

至此，我们已把树的问题转化为链的问题了。下面介绍如何实现这些路径操作。

[路径结构的实现]

可以用伸展树存放路径，路径上的点以它们离头节点的距离大小为权值组成伸展树。

reverse 操作：

对每个节点保存 reverse 标志，如果为真，表示以这个节点为根的子树是需要翻转的，即对这棵子树的每个子节点的左右儿子互换。在访问一个节点 v 的时候，如果 reverse 标志为真那么交换左右子节点，并将 reverse 标志设为假，再把 reverse 标志传递到子节点（取反）执行 reverse 操作时只需改变根节点的 reverse 值

pupdate 操作：

对每个子节点 v 保存 2 个量，ecost, change。cost 表示(v,after(v))的权值，change 代表了以 v 为根的子树中边权值的修正量。在访问一个节点 v 的时候，令 ecost=ecost+change，

再将 `change` 设为 0，最后让 `change` 值传递到子节点（加到子节点的 `change` 值上）

另外对每个节点都保存 `emin`，表示以此节点为根的子树中的最小权值。

执行 `update` 操作时只需修改根节点的 `change` 值

`Path(v)`: 先沿父指针向上找到根节点并返回，然后沿此路径向下并维护 `reverse, change, ecost` 等值

`Tail(p)`: 返回最右子孙

`Head(p)`: 返回最左子孙

`Before, After` 与伸展树中的操作一样

（`path, tail, head` 的时间复杂度与 `splay(v)` 相同，不会影响以后的分析）

`Concatenate(p, q, x)`: 令 `r=tail(p)` 首先 `splay(r)`，再令 `path(q)` 作为 `r` 的右儿子，访问 `head(path(q))`，并修改它的 `ecost` 值为 `x`。最后修改 `r` 的 `emin` 值，使它符合定义。

`Split(v)`: 首先 `splay(v)`，维护 `change, reverse` 的值，删掉与其相联的两条边（如果没有返回 `null`），返回值为：`[v 的左儿子, v 的右儿子, tail(v 左儿子)的 ecost, head(v 右儿子)的 ecost]`。

（伸展树操作：`Splay(v)`，树通过旋转将 `v` 节点调整至树根，旋转时维护 `emin`）

在每次 `split` 和 `concatenate` 操作之后，`splay` 最左子孙，使它成为根。

[复杂度分析]

1. 每一个动态树操作都需要用到 1 次 `expose()`，首先分析 `expose()` 的次数。

对于边 $v \rightarrow w$ （动态树中，不是伸展树中），如果 v 的子孙 w 的子孙 2 ，称为 A 类边，否则成为 B 类边。

显然每个点最多连出一条 A 类边，树中每条路径上最多有 $O(\log_2 N)$ 条 B 类边。

令 p 为 B 类边中的虚边数。

执行一次 `expose`，可能执行许多 `splice` 操作，每一次 `splice` 操作：

添加一条 B 类虚边进入路径： $p+1$

这种情况最多发生 $O(\log_2 N)$ 次

添加一条 A 类虚边进入路径： $p-1$

可能发生许多次，但代价是 p ， p 是保持非负的。

由上面分析可以看出平均每次 `expose` 操作要执行 $O(\log_2 N)$ 次路径操作。（1）

2. 每次 `splay` 的均摊操作复杂度是 $O(\log N)$ ，一个简单的结果是每次动态树操作 $O(\log^2 N)$ 。

（如果使用 AVL，红黑树等平衡二叉树来实现路径结构的话复杂度就是 $O(\log^2 N)$ ，但伸展树的均摊复杂度比这个结果少了 $\log N$ ）

下面更进一步分析均摊时间复杂度。

先介绍一下均摊复杂度的定义：

对整棵伸展树定义一个势 p ，一个操作的均摊复杂度 $a=t+p'-p$ ，其中 t 为实际操作

时间 p 为操作前的势， p' 为操作之后的势。那么 m 步操作的时间复杂度 $\sum t(i)$ 为：

$$\sum_{i=1}^m t(i) = \sum_{i=1}^m (a_i + p_{i-1} - p_i) = \sum_{i=1}^m a_i + p_0 - p_m$$

定义 $s(i)$ 为在动态树中以 i 为根子树中的节点数。定义 $r(i)$ 为 $\log_2(s(i))$ 。定义一棵动态树的势为这棵动态树中所有节点的 $r(i)$ 和。

伸展树有一个的性质：每一次 `splay` 操作的均摊复杂度不超过： $3(r(t)-r(x))+1$ （ t 为这棵

伸展树的树根), 而这个性质用在这个问题里也是正确的 (在 $r(i)$ 的定义改变之后)。下面简单地证明如下。

证明: 用 $r'(x)$ 表示操作以后的顶点 x , $r(x)$ 表示操作以前的顶点 x 。



1. 单旋转

$$\begin{aligned}
 &1 + r'(x) + r'(y) - r(x) - r(y) \\
 &\leq 1 + r'(x) - r(x) \qquad (r(y) \geq r'(y)) \\
 &\leq 1 + 3(r'(x) - r(x)) \qquad (r'(x) \geq r(x))
 \end{aligned}$$



2. 双旋转 (顶点 x 不是根, 且顶点 x 与 x 的父节点使他们父节点的左儿子)

$$\begin{aligned}
 &2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) \\
 &= 2 + r'(y) + r'(z) - r(x) - r(y) \qquad (r'(x) = r(z)) \\
 &\leq 2 + r'(x) + r'(z) - 2r(x) \qquad (r'(x) \geq r'(y)) \text{ and } (r(y) \geq r(x))
 \end{aligned}$$

由对数函数的性质, 对于任意一对 $x, y > 0$ $x + y \leq 1$, 满足 $\log(x) + \log(y) \leq -2$ 。

$$r(x) + r'(z) - 2r'(x) = \log(s(x)/s'(x)) + \log(s'(z)/s'(x)) \leq -2$$

$$\text{即 } 2r'(x) - r(x) - r'(z) \geq 2$$

$$\text{由此可得 } 3(r'(x) - r(x)) - (2 + r'(x) + r'(z) - 2r(x)) = 2r'(x) - r(x) - r'(z) - 2 \geq 0$$

3. 另一种情况的双旋转, 分析与前一种情况类似, 省略。

Splay 操作是通过多次旋转, 将这些旋转的复杂度叠加就得到最多 $3(r(t) - r(x)) + 1$, 于是得证。

Expose(v)操作把从 v 到动态树树根的路径上的所有虚边消除, 并合并路径上的伸展树, 由前面的结论, 路径的合并操作均摊复杂度不超过 $3 * (r(\text{tail}) - r(\text{head})) + 1$, 叠加后可得到 expose(v)复杂度不超过 $3(r(\text{root}) - r(v)) + k$, (k 是从 v 到树根路径上虚边个数), 由前面的结论 (1), 可知平均 k 是对数级别的, 而 $r(\text{root}) - r(v)$ 也是对数级别的, 所以 **Expose(v)均摊复杂度为 $O(\log N)$** 。这也是所有动态树操作的时间界。

Link-cut 操作会改变动态树的势, 但仍然是对数级别的。

[应用]

1、最近公共祖先

询问 v, w 的最近公共祖先, 首先执行 expose(v), 再执行 expose(w), 当执行 expose(w)

时，记录最近的一个再上次 `expose` 中被访问的点，这个点就是最近公共祖先。

每次询问需要 $O(\log N)$ 时间。

2、集合的合并与分离

可以支持 `Union` 和 `Find` 操作，还能支持以某种方式分离，而每个集合操作的时间复杂度为 $O(\log N)$

3、最大流

动态树可以用来优化最短路径增广算法，使每次增广的复杂度降为 $O(m \log N)$ ，并使总复杂度为 $O(mn \log N)$ 。

4、最小生成树

动态树在最小生成树问题中有许多应用。

比如，最小生成树的增量算法、度限制生成树。还有其他许多种变形。

[实现]

直接使用前面的算法，空间复杂度会比较高，下面介绍一种本质相同的算法，空间复杂度会降低。而且编程也略微简单一些。

同样的，并不保存整棵树，而保存“虚拟树”。虚拟树中的顶点与动态树中的是一一对应的。虚拟树中的每个顶点最多连出两条实边，其余为虚边，分别成为左儿子和右儿子，其他顶点称为中间顶点。完全由实边组成的子树称为实树。对每个顶点记录它的父节点 $p(x)$ ，左儿子和右儿子 $left(x)$ ， $right(x)$ 。记每个顶点的费用为 $cost(x)$ ，其子树中最小费用为 $mincost(x)$ 。为了实现 `reverse`，对每个节点保存一个 `reverse` 的标志。为了实现 `update` 函数，每个节点保存 $dcost, dmin$ 两个量：

如果 x 是一颗实树的根： $dcost(x) = cost(x)$

否则 $dcost(x) = cost(x) - cost(p(x))$

$dmin(x) = cost(x) - mincost(x)$

注意 $dmin$ 是非负的。

实现 `expose` 需要两种操作：一种是 `splay`，将一棵实树（也是二叉树），从某个节点伸展使这个节点成为这个实树的根。第二种是 `splice`，将某个节点的一个中间节点变为左儿子，左儿子变为中间节点。

执行 `expose(v)` 操作需要分为三步。首先沿 v 往上到虚拟树的树根，对沿路的各实树进行 `splay`，执行完这个步后从 v 到树根的路径上只有虚边。再沿 v 往上到虚拟树的树根，对沿路各节点执行 `splice`，将从 v 到虚拟树树根路径上的边变为虚边。这时 v 和树根在一个实树中了。这时再执行 `splay(v)`，把 v 调整为树根。

有了 `expose(v)`，动态树的 8 个功能就可以实现了。

可以看出，这个算法和原算法的本质是相同的，所以它们的复杂度也是一样的。即均摊每步 $O(\log N)$ 。下面简单说说 `splay` 操作时 $dcost$ 和 $dmin$ 的维护。

对于 `splay`，假设需要旋转的边连接节点 v 与节点 v 的父节点 w ，令 a, b 为旋转前 v 的左右儿子。 b, c 为旋转后 w 的左右儿子。 $dcost$ 与 $dmin$ 的变化如下：

$$dcost'(v) = dcost(v) + dcost(w)$$

$$dcost'(w) = -dcost(v)$$

$$dcost'(b) = dcost(v) + dcost(b)$$

$$dmin'(w) = \max(0, dmin(b) - dcost'(b), dmin(c) - dcost(c))$$

$$dmin'(v) = \max(0, dmin(a) - dcost(a), dmin'(w) - dcost'(w))$$

其他节点的量不变。

对于 splice, 假设 v 是 w 的新的左儿子, w 原来的左儿子为 u

$$\text{dcost}'(v) = \text{dcost}(v) - \text{dcost}(w)$$

$$\text{dcost}'(u) = \text{dcost}(u) + \text{dcost}(w)$$

$$\text{dmin}'(w) = \max(0, \text{dmin}(v) - \text{dcost}'(v), \text{dmin}(\text{right}(v)) - \text{dcost}(\text{right}(v)))$$

其他节点的量不变。

[参考文献]

- [1] Sleator and Tarjan A data structure for dynamic trees.
- [2] Sleator and Tarjan Self adjusting binary search tree.
- [3] Georgiads, Tarjan, Werneck Design of Data Structure for Mergeable Trees.
- [4] 拉文德拉 K.阿胡亚 托马斯 L.马南提 詹姆斯 B.沃林 网络流理论、算法与应用 机械工业出版社