15-451 Algorithms

Fall 2012

D. Sleator

Link/Cut trees  October 9, 2012

Full paper at: http://www.cs.cmu.edu/~sleator/papers/self-adjusting.pdf

---

The goal of link/cut trees is to represent a forest of trees under these kinds of operations:

| | |
|---|---|
| link(p,q) | p is a tree root, and q is any node in another tree. make p a child of q. |
| cut(p) | p is not a tree root. Delete the edge from p to its parent, thus separating the tree in two |
| path(p) | This just means to do something to the path from p to the root of p's tree. This could be counting its length, finding the minimum cost edge on this path, adding a constant to all the costs on this path, etc. (Or, in this case, it's going to be to toggle whether or not each edge on this path is currently turned on.) The operations supportable are those you can do on a subsequence of nodes in a binary search tree. |
| findroot(p) | return the root of the tree containing p |

All of these operations are supported by link/cut trees in O(log n) time where n is the size of the tree.

I'll use the terminology from our paper.  The "real tree" is the tree that the data structure is trying to represent.  It's the tree that the API presents to the user.  The "virtual tree" is the actual tree represented in the memory of the computer.  (We should probably should have reversed these two terms, to better fit the analogy with the real and virtual images in optics.)  The virtual tree has exactly the same set of nodes as the real tree, they're just linked together in a different way.

The virtual tree has dashed and solid edges.  The connected components of solid edges are binary trees (I'll call them splay trees).  The splay tree is represented with the usual parent, left, and right and child pointers.  The root of each splay tree is linked via a dashed edge to some other node in the virtual tree.

The relationship between the real tree and the virtual tree that represents it can be made clear by explaining how to transform the virtual tree into the real tree.  To do this, convert each splay tree into a path by traversing it in left to right order.  This is now a path of nodes up the real tree.  (In other words, if nodes a and b are in the same splay tree, and b is the successor of a, then a is a child of b in the real tree.)  The rightmost node of this path is linked to the node that is attached to the root of that splay tree by a dashed edge.

Note that the real tree has been partitioned into solid paths (which

always go up the tree from child to parent) and dashed edges. This partitioning is determined by the algorithm and will change with time. It is not under the control of the client.

Every node in the virtual tree has the usual parent, left, and right child pointers. How do we tell if a node is a root of a splay tree? Easy. It's a root if (1) its parent pointer is null or (2) if its parent's left and right children ARN'T it. This is computed by the isroot() function below.

Now that we know how to identify the root, it's easy to implement the splay(p) operation that takes a node p and splays it to the root of its splay tree.

And using splay() we can implement expose(p), which transforms the virtual tree in a way that puts node p at the root of the virtual tree. (The real tree, of course, does not change.) Expose works by doing a sequence of splays and relinking operations, called splices. A splice converts the left solid edge down from a node to dashed and converts one of the dashed edges down from that node to solid. (Picture is really required to explain it. Check out the code, which is quite short.) After expose(p) the path from p to the root of the real tree consists of all the nodes to the right of p in its splay tree. (Actually, the way the code below works, after expose(p), p is the root and also the leftmost node in its splay tree.)

[The expose algorithm is the weakest part of these notes, that must be explained on the blackboard.]

This design does not allow us to walk down the tree along the dashed edges. But walking down is not necessary for the basic operations of expose, link, cut, path, findroot, etc.

Theorem:

A sequence of m link/cut operations on starting from a set of n separate nodes is O(m log n).

Proof:

We need to assign a weight to each node so that we can carry out the analysis using the access lemma for splay trees. Recall from the splay tree analysis that the size of a node is the total weight of all nodes in the subtree rooted there. We will define the weights of the nodes so that the size of a node is the number of nodes in the virtual tree rooted there.

To make this be the case, we simply assign the weight of a node to be 1 (for itself) plus the sizes of all the subtrees connected to it via dashed edges. It's easy to see that with this definition of weight, a splice operation does not change the sizes of any nodes, and thus does not change the potential of the tree.

Recall that the rank of a node is the binary log of the size of the node. Let the potential function be (Sum rank(x)) for all nodes x. Here is the access lemma:

Access Lemma: The amortized number of rotations done when splaying a node x (of rank r(x)) in a tree rooted at t (of rank r(t)) is bounded by 3(r(t) - r(x)) + 1.

We will focus on the cost of the expose() operation. (The other operations are easy to analyze using the bound (derived below) for expose() combined with an analysis of the potential function changes involved.)

Let the cost of expose() be measured as the number of edges on the path from the exposed node to the root of the virtual tree. (This is exactly the same as the number of rotations done in the expose.)

We will prove that with this potential function, the amortized cost of expose() is at most 12m(log n) + 2m.

Let's look at one expose(). Say that there are k splay operations until p gets to the root (see the code below). These splays are called phase1 splays. After this we splay q, this splay is a phase2 splay.

The cost of the phase1 splays telescope, because the starting size of one splay is greater than the ending size of the previous splay. So the cost of the phase1 splays is at most 3(log n) + k.

The phase2 splay costs amortized 3(log n)+1. So the total amortized cost of the expose is 6(log n)+k+1. If we add this together for all m expose() operations we can write:

    total cost <= 6 m (log n) + m + (#splays in phase1)

Where the latter term comes from summing all the "k"s from each

expose() operation.  It remains to bound this term.

Consider a modified analysis of splaying where we do not count the rotation done in the zig case.  In this case the access lemma bound becomes 3(r(t) - r(x)).  The total modified cost of splaying in a sequence of exposes is then at most 6 m (log n).  But the number of phase1 splays is at most the true cost of the phase2 splays.  (This is because if there are k phase1 splays in an expose() then the corresponding phase2 splay does k rotations.)  So the true cost of the phase2 splays is at most m more than the modified cost we defined in this paragraph (there are at most m zigs that have to be accounted for).  Thus:

(#splays in phase1) <= 6 m (log n) + m.

Putting this together gives us:

total cost <= 12 m (log n) + 2m

We should note in conclusion that the initial potential is 0 (all the nodes are separate, so the sizes are all 1).  And the final potential is positive.

QED.

The code below is customized to support the operations needed for this specific problem (http://www.codeforces.com/contest/117/problem/E). Each node in the virtual tree stores the information about the edge from that node in the real tree to its parent in the real tree.  A

node that is a root of its real tree does not correspond to any edge.
We capture this by giving each node an individual weight my_s.  my_s
is 1 for all nodes except a node that is the root of the real tree.
The s field maintains the sum of all the my_s fields in the subtree of
the splay tree rooted here.

Each node also has a boolean value flip.  Toggling this bit implicitly
changes the on/off state of all the edges in the subtree (of the splay
tree) rooted here.  Each node has my_flip, which tells if the current
edge is flipped or not.  And each node has an "on" value which keeps
the total weight of all the 'on'' edges in the subtree rooted here.

Using this representation it's very simple to flip the state of all
the edges from a node p to the root of the real tree.  To see how this is
done look at the code for toggle(p).

It should be fairly easy to modify the code below to support various
different operations.  Most of the work should be in setting up the node
class, and adjusting normalize() and update().

```
class Node {
    int s, my_s, on, id;
    boolean flip, my_flip;
    Node l, r, p;

    Node (int c, int i) {
        id = i;
        s = my_s = c;
```

```
        on = 0;
        l = r = p = null;
        flip = my_flip = false;
    }

    boolean isroot() {
        return p==null || (p.l != this && p.r != this);
    }

    /* If this node is flipped, we unflip it, and push the change
       down the tree, so that it represents the same thing. */
    void normalize() {
        if (flip) {
            flip = false;
            on = s-on;
            my_flip = !my_flip;
            if (l != null) l.flip = !l.flip;
            if (r != null) r.flip = !r.flip;
        }
    }

    /* The tree structure has changed in the vicinity of this node
       (for example, if this node is linked to a different left
       child in a rotation).  This function fixes up the data fields
       in the node to maintain invariants. */
    void update() {
        s = my_s;
        on = (my_flip)?my_s:0;
        if (l != null) {
```

```
            s += l.s;
            if (l.flip) on += l.s-l.on; else on += l.on;
        }
        if (r != null) {
            s += r.s;
            if (r.flip) on += r.s-r.on; else on += r.on;
        }
    }
}

class LinkCut {
    static void rotR (Node p) {
        Node q = p.p;
        Node r = q.p;
        q.normalize();
        p.normalize();
        if ((q.l=p.r) != null) q.l.p = q;
        p.r = q;
        q.p = p;
        if ((p.p=r) != null) {
            if (r.l == q) r.l = p;
            else if (r.r == q) r.r = p;
        }
        q.update();
    }

    static void rotL (Node p) {
        Node q = p.p;
        Node r = q.p;
```

```
        q.normalize();
        p.normalize();
        if ((q.r=p.l) != null) q.r.p = q;
        p.l = q;
        q.p = p;
        if ((p.p=r) != null) {
            if (r.l == q) r.l = p;
            else if (r.r == q) r.r = p;
        }
        q.update();
    }

    static void splay(Node p) {
        while (!p.isroot()) {
            Node q = p.p;
            if (q.isroot()) {
                if (q.l == p) rotR(p); else rotL(p);
            } else {
                Node r = q.p;
                if (r.l == q) {
                    if (q.l == p) {rotR(q); rotR(p);}
                    else {rotL(p); rotR(p);}
                } else {
                    if (q.r == p) {rotL(q); rotL(p);}
                    else {rotR(p); rotL(p);}
                }
            }
        }
        p.normalize(); // only useful if p was already a root.
```

```
        p.update();    // only useful if p was not already a root
    }

    /* This makes node q the root of the virtual tree, and also q is the
       leftmost node in its splay tree */
    static void expose(Node q) {
        Node r = null;
        for (Node p=q; p != null; p=p.p) {
            splay(p);
            p.l = r;
            p.update();
            r = p;
        };
        splay(q);
    }

    /* assuming p and q are nodes in different trees and
       that p is a root of its tree, this links p to q */
    static void link(Node p, Node q) throws MyException {
        expose(p);
        if (p.r != null) throw new MyException("non-root link");
        p.p = q;
    }

    /* Toggle all the edges on the path from p to the root
       return the count after - count before */
    static int toggle(Node p) {
        expose(p);
        int before = p.on;
```

```
        p.flip = !p.flip;
        p.normalize();
        int after = p.on;
        return after - before;
    }

    /* this returns the id of the node that is the root of the tree containin
g p */
    static int rootid(Node p) {
        expose(p);
        while(p.r != null) p = p.r;
        splay(p);
        return p.id;
    }
}
```