18.415/6.854 Advanced Algorithms	November 5, 2001
Lecture 15	
Lecturer: Michel X. Goemans	*Scribe: Timothy Danford*

Last time we saw that all the operations on dynamic trees are expressed in terms of expose, which was itself expressed in terms of splice and splay. We'll do the running time analysis of dynamic trees in a manner very similar to the way we did the analysis of splay trees.

1 Analysis of Dynamic Trees

- We'll maintain a credit invariant, the *rank* of the node.

  $s(x) \;=\;$ number of descendants of x in the virtual tree.

  $r(x) \;=\; \log_2(s(x))$

  The credit invariant: every node maintains $r(x)$ units of credit.

- Now, let's look at splicing:
  - We switch a middle child with a left child, but this has no impact on the credit invariance.
  - Splicing occurs only at Step 2 of expose, and therefore can be charged to Step 3 of expose. (i.e., we charge the splicing of Step 2 to the splaying of Step 3).

- Therefore, the cost of our expose operations is the cost of the splaying in Step 1 and the cost of the splaying in Step 3 (everything else we said about splay trees also applies to virtual trees).

1.1 Splaying

We showed that $\mathsf{splay}(v)$ had the amortized cost of $3(r(\mathsf{root}) - r(v)) + 1 = O(\log n)$ since $0 \leq r(v) \leq \log n$ for any v in a tree of size n.

We can double this with no problem, so the amortized cost of $\mathsf{expose}(v)$ is $O(\log n)$. Therefore, any sequence of m expose operations has an amortized cost of $O(m \log n)$. This implies that the cost is $O(m \log n) + O(n \log n) = O((m+n)\log n)$.

find-cost, find-root, find-min, add-cost: no change in the tree, so no change in the credit invariant.

cut: adds an edge and removes another. Therefore, we gain relative to the credit invariant.

link: we add a subtree. When we do this, we've exposed two vertices w and v, so they are the roots of their respective trees. Then when we link the two trees, the credit invariant increases by $\log n$ at only *one* node (i.e., at v if we're linking w to v) and by $O(\log n)$ overall.

Bottom Line: Any sequence of m dynamic tree operations will take $O((m+n)\log n)$ time.

2 Using Dynamic Trees to implement Cancel & Tighten efficiently

- We have a directed graph G, a flow f, and a residual capacity u_f. We want to repeatedly find cycles and push as much flow along those cycles as possible. Some of the edges of the graph will be maintained in the dynamic tree and for those we will maintain their residual capacity u_f; for the edges not in the dynamic tree, we will maintain their actual flow. Thus, whenever an edge is removed from the dynamic tree data structure, we need to compute its flow by subtracting the residual capacity u_f from the capacity u; conversely when we add an edge to the tree.

- We start with all the trees as singletons from the graph.
- We always try to find an admissible arc entering the root of one of the (dynamic) trees.
- If we find an admissible arc between two trees (and entering the root of one of them), we connect them with link. Observe that the resulting tree is still a tree directed away from its root.
- If, on the other hand, the admissible arc connects a node to the root r of its tree, then we
  - have found a cycle to cancel consisting of the arc and the path from the root r to the other endpoint of the arc,
  - compute the amount δ by which we can cancel the cycle (which is the minimum of the residual capcaities along the cycle),
  - increase the flow on the admissible arc by δ (the arc is not added to the tree)
  - decrease the residual capacity by δ along the path from root to the other endpoint of the arc (the call to $\mathsf{addcost}$).
  - remove all the edges that have been saturated
- When there are no admissible edges entering r, there are no more admissible cycles passing through r and we remove all the arcs in the tree leaving r. This means that for every arc (r, v) where v is a child of r in the tree, we compute its flow from its residual capacity and remove the arc from the tree; we then remove r from consideration my marking it.

2.1 Cancel & Tighten Algorithm, with Dynamic Trees

$\forall v : \mathsf{unmark}(v), \mathsf{make}-\mathsf{tree}(v)$
while $\exists$ unmarked v, do

- $r \leftarrow \mathsf{findroot}(v)$
- if $\exists$ admissible arc (w, r) then
  - if $\mathsf{findroot}(w) \neq r$ then
    - $\mathsf{link}(w, r, u_f(w, r))$
  - else
    - $\delta = \min(u_f(w, r), \mathsf{findcost}(\mathsf{findroot}(w)))$
    - $f(w, r) \leftarrow f(w, r) + \delta$
    - if $f(w, r) = u(w, r)$, then (w, r) is inadmissible
    - $\mathsf{addcost}(w, -\delta)$
    - while $\mathsf{findcost}(\mathsf{findmin}(w)) = 0$ do
      - $z \leftarrow \mathsf{findmin}(w)$
      - $f(\mathsf{parent}(z), z) \leftarrow u(\mathsf{parent}(z), z)$
      - $\mathsf{cut}(z)$
- else (there are no admissible arcs (w, r))
  - $\mathsf{mark}(r)$
  - for each child z of v
    - $f(r, z) = u(r, z) - \mathsf{findcost}(z)$
    - $\mathsf{cut}(z)$

2.2 Correctness & Running Time

Correctness should be obvious, as should running time, which works out as follows:

$$\left.\begin{array}{rl} \text{\# of cancel/tighten steps} & \left\langle \begin{array}{l} O(n\log nC) \\ O(mn\log n) \end{array}\right. \\ \text{time per cancel step} & O(m) \\ \text{tighten step} & O((m+n)\log n) \end{array}\right\} \left\langle \begin{array}{c} O(m^2 n\log n) \\ \text{or} \\ O(mn\log n\log nC) \end{array}\right.$$