6.854J / 18.415J Advanced Algorithms
Fall 2008

18.415/6.854 Advanced Algorithms September 29, 2008

Lecture 7 - Dynamic Trees

Lecturer: Michel X. Goemans

1 Overview

In this lecture, we discuss dynamic trees, a sophisticated data structure introduced by Sleator and Tarjan. Dynamic trees allow to provide the fastest worst-case running times for many network flow algorithms. In particular, it will allow us to efficiently perform the Cancel operation in the Cancel and Tighten algorithm. Dynamic trees build upon splay trees, which we introduced in the previous lecture.

Dynamic trees manage a set of node-disjoint (not necessarily binary) **rooted trees**. With each node v is associated a cost. In our use of dynamic trees, the cost will be coming from the edge $(p(v), v)$, where $p(v)$ denotes the parent of v; the cost of the root in that case will be set arbitrarily large (larger than the cost of any other node), say $+\infty$.

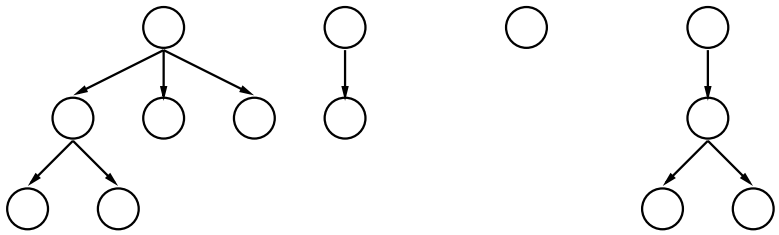

Figure 1: Example of Dynamic Tree.

Dynamic trees will support the following operations:

- MAKE-TREE(V): Creates a tree with a single node v, whose cost is $+\infty$.
- FIND-ROOT(V): Finds and returns the root of the tree containing the node v.
- FIND-COST(V): Returns the cost of node v. (This may sound like a trivial operation, but in fact there is real work to be done, because we will not explicitly maintain the costs of all nodes.)
- FIND-MIN(V): Finds and returns the ancestor of w of v with minimum cost. Ties go to the node closest to the root.
- ADD-COST(V, X): Adds x to the cost of all nodes w on the path from FIND-ROOT(v) to v.
- CUT(V): Breaks the rooted tree in two by removing the link to v from its parent. The node v is now the root of a new tree, and its cost is set to $+\infty$.
- LINK(V, W, X): Assumes that (1) w is a root, and (2) v and w are not in the same tree, i.e. FIND-ROOT(v) $\neq w$. Combines two trees by adding an edge (v, w), i.e. $p(w) = v$. Sets the cost of w equal to x.

We will later show that all of these operations run in $O(\log n)$ amortized time.

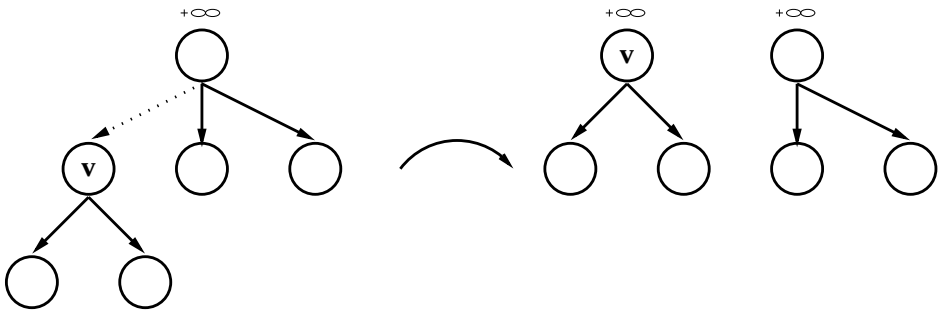

Figure 2: CUT(v) operation.

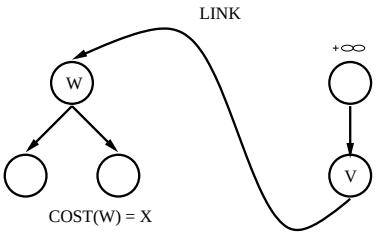

Figure 3: LINK(v, w, x) operation.

Theorem 1 *The total running time of any sequence of m dynamic tree operations is $O((m + n) \log n)$, where n is the number of nodes.*

We defer the proof of this theorem until the next lecture.

2 Implementation of Cancel with dynamic trees

Recall the setting for the Cancel step in the algorithm Cancel and Tighten for the minimum cost flow problem. We have a circulation f and node potentials p in an instance defined on graph G. Recall that an edge (v, w) is admissible if $c_p(v, w) < 0$, and the admissible graph (V, E_a), is the subgraph of E_f (the residual graph corresponding to our circulation) containing only the admissible edges. Our aim is to repeatedly find a cycle in the admissible graph and saturate it. Each time we do this, all of the saturated edges disappear from the graph. Also recall that no edges are added to the admissible graph during this process, because any new edge in the residual graph must have positive reduced cost and are therefore is not admissible.

We represent the problem with dynamic trees, where the nodes in the dynamic trees correspond to nodes in G and the edges of the dynamic trees are a subset of the admissible edges. We maintain two (disjoint) sets of admissible edges: those which are currently in the dynamic tree, and those which still need to be considered. The cost of a node v will correspond to the residual capacity $u_f(p(v), v)$ of the edge $(p(v), v)$, unless v is a root node, in which case it will have cost $+\infty$. We will also mark some of the roots (denoted graphically with a $(*)$) to indicate that we dealt with them and concluded they can't be part of any cycle. For the edges not in the dynamic tree, we also maintain the flow value. (We don't need to maintain the flow explicitly for the edges in the trees, since we can recover the flow from the edge capacities in G and the residual capacity.)

To summarize, we begin with a set of n singleton trees. All of the edges start out in the remaining pool. In each iteration, we try to find an admissible edge leading to the root r of one of the dynamic trees. If we fail to find such an edge, this implies there are no admissible cycles which include r,

and so we mark it and remove it from consideration. Suppose, on the other hand, that we do find an edge (w, r) leading into the root. If w is in a different tree, we join the two trees by adding an edge connecting w and r. On the other hand, if w and r are part of the same tree, it means we have found a cycle. In this case, we push flow along the cycle and remove the saturated edges from the data structure.

In more detail, we keep repeating the following procedure as long as there still exist unmarked roots:

- ▷ Choose an unmarked root r.
- ▷ Among admissible edges, try to find one which leads to r.
- ▷ CASE 1: there is no such $(v, r) \in E_a$.
  - ▷ Mark r, since we know it cannot possibly be part of a cycle.
  - ▷ Cut all the children v of r.
  - ▷ Set

$$\begin{aligned} f(r,v) &\leftarrow u(r,v) - u_f(r,v) \\ &= u(r,v) - \textsc{find-cost}(v) \end{aligned}$$

- ▷ CASE 2: there is an admissible edge (w, r) from a different tree, i.e. $\textsc{find-root}(w) \neq r$.
  - ▷ Link the two trees: $\textsc{link}(w, r, u(w, r) - f(w, r))$
- ▷ CASE 3: there is an admissible edge (w, r) from the same tree, i.e. $\textsc{find-root}(w) = r$.
  - ▷ We've found a cycle, so push flow along the cycle. The amount we can push is

$$\delta = \min(u(w,r) - f(w,r), \textsc{find-cost}(\textsc{find-min}(w)))$$

  - ▷ $\textsc{add-cost}(w, -\delta)$
  - ▷ Increase $f(w, r)$ by δ
  - ▷ If $f(w, r) = u(w, r)$, then (u, r) is inadmissible, so we get rid of it.
  - ▷ While $\textsc{find-cost}(\textsc{find-min}(w)) = 0$:
    - ▷ $z \leftarrow \textsc{find-min}(w)$
    - ▷ $f(p(z), z) \leftarrow u(p(z), z)$
    - ▷ $\textsc{cut}(z)$

The last while loop is to delete all the edges that became inadmissible along the path from r to w.

2.1 Running time

In a cancel step, we end up cancelling at most $O(m)$ cycles, where m is the number of edges. In addition, each edge gets saturated at most once (if it does, it becomes inadmissible); therefore the number of $\textsc{cut}(z)$ and $\textsc{find-min}(w)$ over all cases 3 is $O(m)$. Thus the total number of dynamic tree (and also other arithmetic or control) operations is at most $O(m)$. Hence, by Theorem 1, the running time of each Cancel operation is $O((m + n) \log n) = O(m \log n)$. The overall running time of Cancel-and-Tighten is therefore $O(m^2 n \log^2 n)$ (strongly polynomial running time bound) or $O(mn \log n \log(nC))$.

3 Dynamic trees implementation

We now turn to the implementation of dynamic trees. Here we present the definitions; we will cover the running time analysis in the next lecture. The dynamic trees data structure is a collection of rooted trees. We decompose each rooted tree into a set of node-disjoint (directed) paths, as shown in Figure 4. Each node is in precisely one path (possibly containing that node only). We will refer

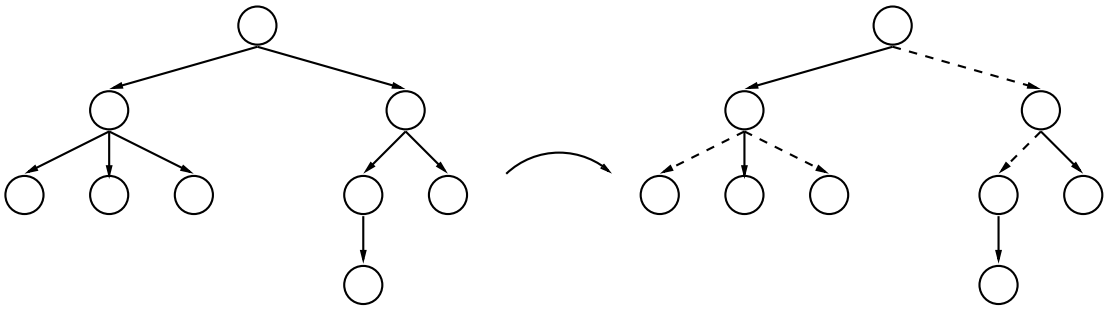

Figure 4: Decomposition of rooted tree.

to the edges on these paths as **solid edges**, and we will refer to the remaining edges as **dashed edges**, or **middle edges**. Each path is directed from its *tail* (highest in the tree) to its *head* lowest in the tree).

There are many possible ways to partition a tree into solid paths. For instance, if we are given a solid edge and a dashed edge which are both children of a single parent, we can swap the solid and dashed edges. This follows from the basic observation that, for any middle edge (v, w), w is the tail of a solid path. This operation is known as **splicing** as shown in Figure 5.

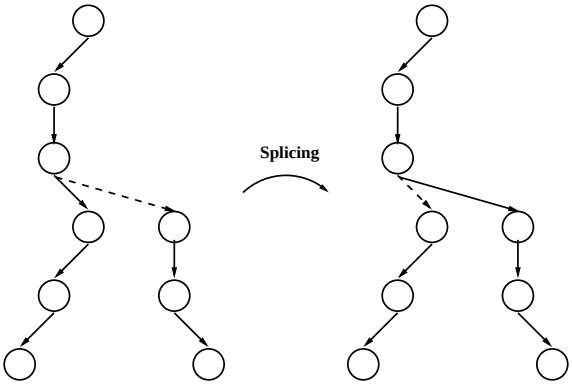

Figure 5: Splicing in the rooted tree.

In a dynamic tree, each solid path is represented by a splay tree, where the nodes are sorted in increasing order from the *head* to the *tail*, as shown in Figure 6. In other words, the node with smallest key is the head (the lowest in the tree), and the node with largest key is the tail (the highest in the tree)

In addition, we will maintain links between the different splay trees. The root of each splay tree is attached to the parent of the tail of the path in the rooted tree, as shown in Figure 7. For example, the edge (e, f) in the original rooted tree becomes the edge (e, i) linking e to the root i of the splay tree corresponding to the solid path $f \rightarrow i$. The entire data structure — with the splay trees corresponding to the same rooted tree being connected to each other — forms what is called a *virtual tree*. Any given node of the virtual tree may have at most one left child and at most one right child (of a splay tree), as well as any number of children attached by dashed edges. Children attached by dashed edges are known as **middle children**, and we draw them in between the left and right children.

Notice that we can reconstruct the rooted tree from the virtual tree. Each splay tree corresponds to a solid path from the node of lowest key to the node of highest key. In addition, for any middle

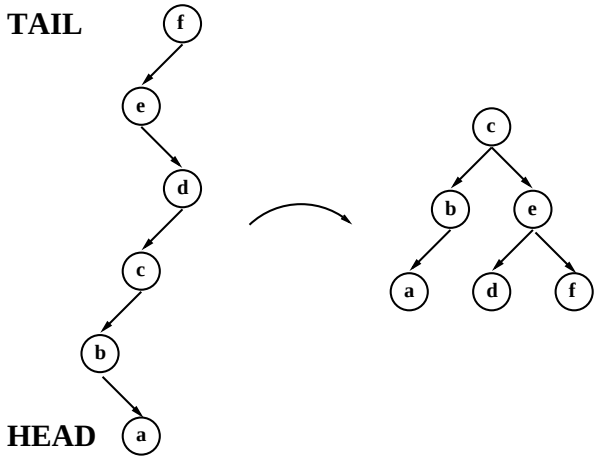

Figure 6: Representation of solid path from head to tail in BST (Splay Tree).

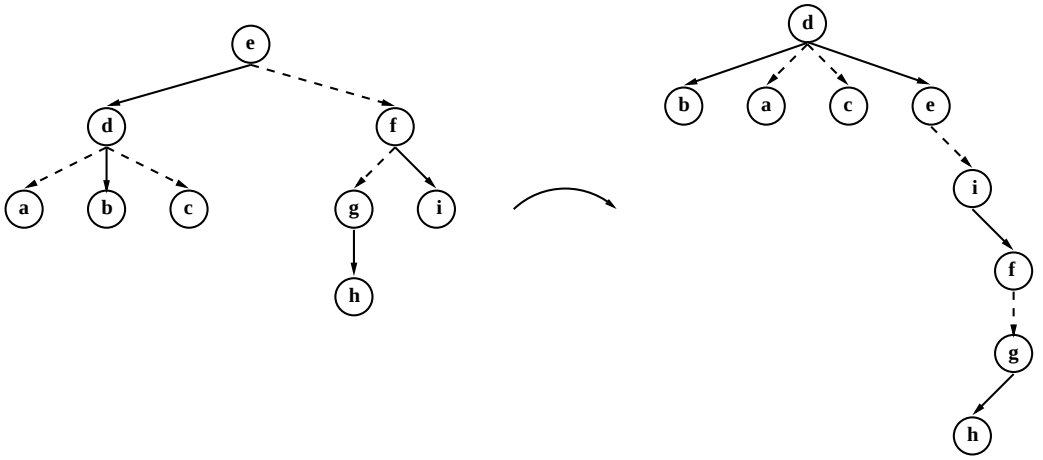

Figure 7: Rooted tree on the left and corresponding virtual tree on the right.

edge, we get an edge of the original rooted tree; for example, to (e, i) in the virtual tree, corresponds the edge (e, f) in the original tree where f is the node with highest key in the splay tree in which i resides.

Note that there are many different ways to represent rooted trees as virtual trees, and we can modify virtual trees in various ways which don't affect the rooted trees.

In particular, we define the EXPOSE(v) operation, which brings a given node v to the root of the virtual tree. This operation involves three main steps:

1. Make sure that the path from v to the root only uses *roots* of splay trees. This can be done by performing splay operations whenever we enter a new splay tree.

2. Make sure that the path from v to the root consists entirely of solid edges. We can ensure this through repeated splicing.

3. Do the splay operation to bring v to the top of the resulting splay tree. This is justified since v is now in the same splay tree as the root of the original rooted tree.