6.851: Advanced Data Structures Spring 2010

Lecture 13 – Thursday, March 18, 2010

Prof. Erik Demaine *Scribe: David Charlton*

1 Overview

This lecture covers two methods of decomposing trees into smaller subtrees: Separator Decomposition and ART / leaf-trimming-decomposition. The latter is applied to solve the marked ancestor problem and the decremental connectivity problem on trees.

2 Tree decompositions

This course covers several ways of decomposing trees:

- Preferred paths: Rearranges a balanced tree to have a less balanced but nicer structure. We saw this in Tango trees, and will use it again when we cover link-cut trees.
- Heavy-light: Decomposes any tree (including an unbalanced one) into paths so that every leaf-to-root path hits only $\log n$ "heavy paths" (this means we can compress the heavy paths to get a balanced tree). This will also be used in link-cut trees.
- Separator decomposition: Rearranges any tree into a balanced tree by recursively splitting it into subtrees of roughly equal size.
- ART/leaf-trimming decomposition: Split the tree into "bottom" subtrees having $\leq \lg n$ leaves and a "top" subtree consisting of the full tree with the bottom trees removed.

This lecture covers the latter two decompositions, with some applications and open problems.

3 Separator theorem on trees

A typical separator theorem on graphs would give a small set of vertices that splits the graph into balanced components. In the case of trees, we can do even better, as shown by this classic result of Camille Jordan [Jor69]:

Theorem 1 (Jordan, 1869). *Any tree on n vertices has a vertex whose removal leaves connected components of size at most $n/2$.*

Proof. Consider any vertex v. If v does not divide the tree into components of size $\leq n/2$, then it is adjacent to a node of size $> n/2$, so take one step into that subtree and recurse (see Figure 1, where the recursion proceeds from v to u).

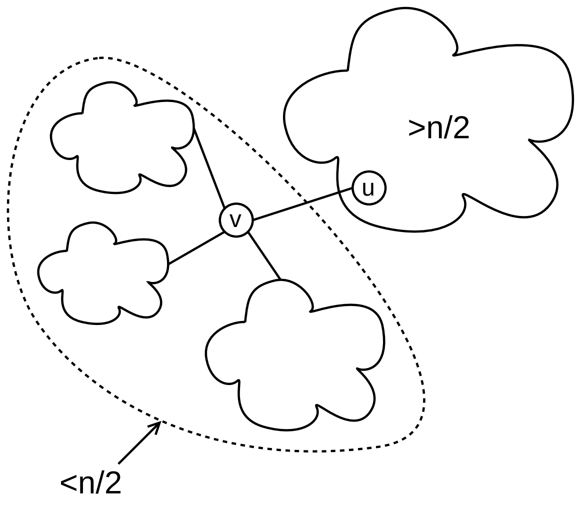

Figure 1: A wrong choice of v results in an adjacent subtree having more than $n/2$ vertices. We change v to u, the root of the largest subtree adjacent to v, and recurse.

This process will never cross the same edge twice, since a single tree edge can't have $> n/2$ nodes on both sides, so we will find the desired separator vertex in at most n iterations (in fact, at most $n/2$, since we can't start at distance more than $n/2$ from the target).

$\square$

3.1 Separator decomposition

The separator decomposition is defined recursively: given an input tree, apply Theorem 1 to get a balanced cut, and make the selected node v the root of a new tree. Remove v from the tree and recurse on the components to get the child subtrees. The resulting decomposition is a new tree on the same vertices with height at most $\log n$. The decomposition will generally have different topology than the input: see the example in Figure 2.

The separator decomposition tree can be constructed in $O(n \lg n)$ time, since we can find a separator vertex on $O(n)$ time. It's an open question whether there is a faster algorithm to construct this decomposition.

3.2 Application: finding cats in trees

We describe a somewhat silly but useful application of separator trees by Aronov et al. [ABD$^+$06]. Suppose we have a tree with a distinguished vertex, known as the *cat*. We also have with us a

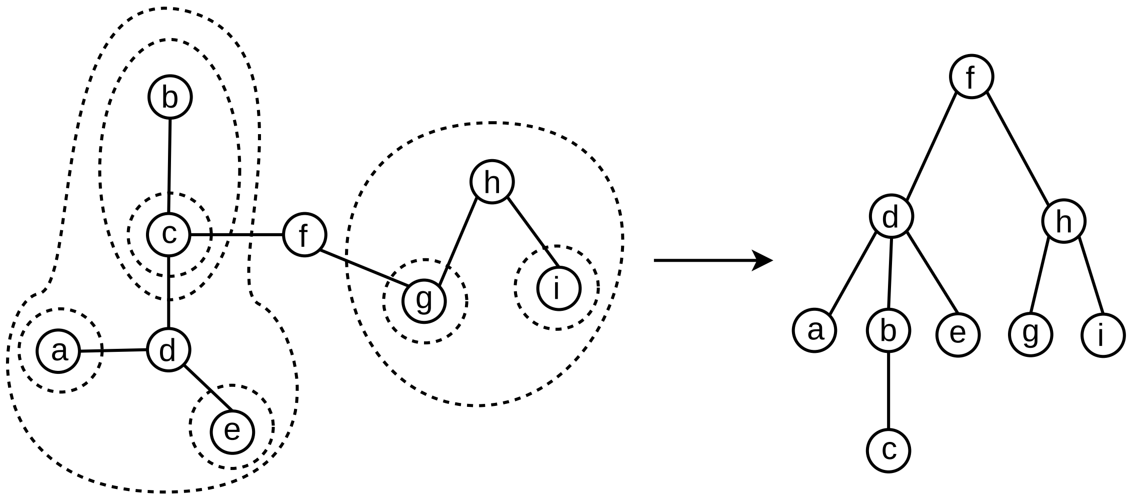

Figure 2: An example of separator decomposition.

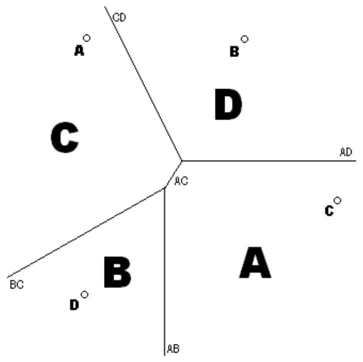

Figure 3: An example of farthest-point Voronoi diagram.

person who is allergic to cats, who can tell us, given any edge, in which direction the cat lies. Our goal is to find the cat with a small number of queries, so their allergies don't flame up too much.

More precisely, we have an oracle that, given an edge in the tree, tells us which of the adjacent subtrees the cat is in. If the tree has bounded degree, we can order our queries to do a binary search on the separator decomposition to find the cat in $O(\log n)$ queries.

This technique can be applied to keeping a synchronized copy of a file hierarchy: given two copies of the hierarchy, we want to quickly find which files have been modified. Here the modified file plays the role of the cat, and the file hierarchy is the tree.

The separator decomposition has also been applied to the problem of dynamic farthest point queries: maintain a collection of points in the plane so that answering "which of the stored points is farthest from my query point?" questions is fast. The usual solution in the static case is to build a "farthest points" Voronoi diagram (see Figure 3). Then it is pretty easy to get logarithmic time. The dynamic case requires the observation that even though the geometry of the Voronoi diagram can change completely with every change, the topology of the Voronoi tree is mostly the same, and so the separator decomposition can help find the target region even though the geometry has changed.

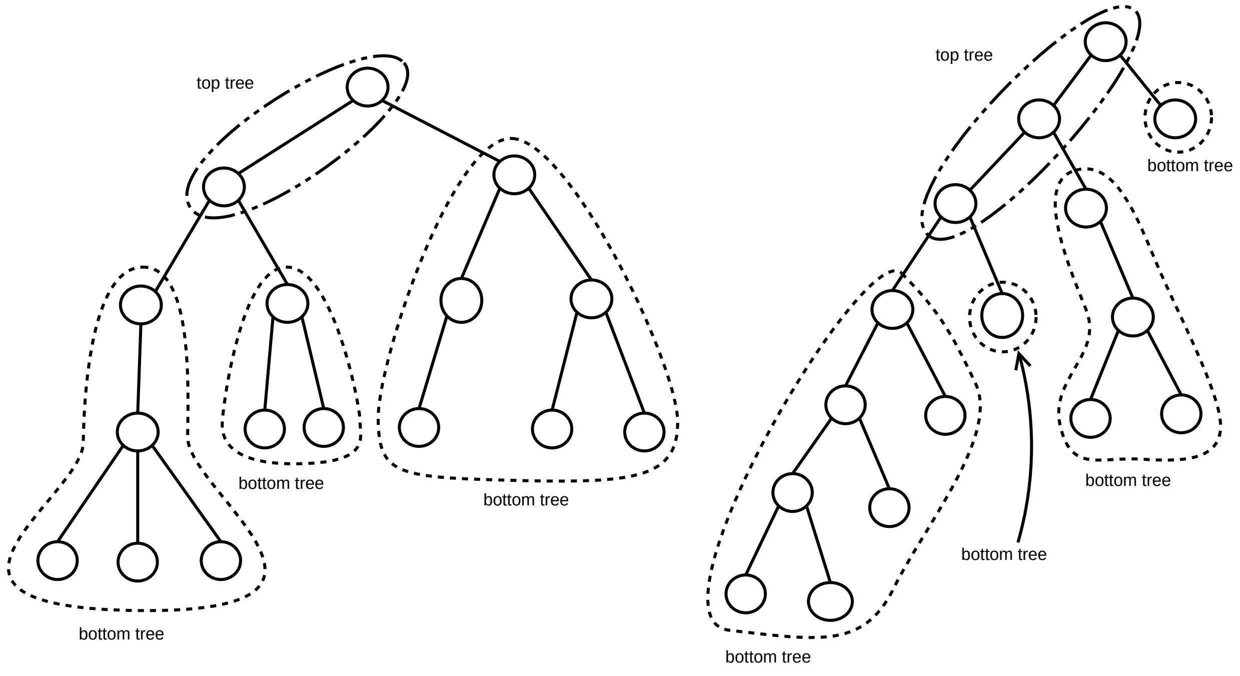

Figure 4: Some decompositions of trees with sixteen nodes.

It's an interesting question whether there are any other simple applications of the separator decomposition.

4 ART decomposition

We now present the main topic of this lecture, the ART decomposition developed by Alstrup, Husfeldt, and Rauhe in [AHR98]. It is defined the by the following rules:

- For every maximally high node whose subtree contains no more than $\lg n$ *leaves*, we designate the tree rooted at this node a *bottom tree*.
- Nodes not in a bottom tree make up the *top tree*.

See Figure 4 for an example decomposition. This is similar to the decomposition used for y-fast trees in Lecture 9, except that in y-fast trees the bottom trees contained $\log n$ *nodes* rather than *leaves* as in the ART decomposition. This means that an ART bottom tree could potentially be linear size (e.g. if it contains a long path). Limiting the number of leaves does, however, imply that a bottom tree has a logarithmic number of *branching* nodes.

Proposition 2. *The top tree has at most $\frac{n}{\lg n}$ leaves.*

Proof. Consider a leaf of the top tree. All of its children are roots of bottom trees. The total number of leaves in these bottom trees must be at least $\lg n$: Otherwise, the leaf of the top tree would itself be part of a bottom tree. Thus, we can associate each leaf of the top tree with at least $\lg n$ leaves of the original tree. □

This proposition shows that the "complexity" of the top tree (measured in leaves, or equivalently in branching nodes) is lower than the original tree by a logarithmic factor. (Note that this would not have been true if we limited bottom trees to $O(\lg n)$ nodes rather than leaves.)

An an optional step, recurse on the top tree to produce a hierarchy of top and bottom trees. This decomposition is powerful with and without recursion, and we will do one example of each.

5 Marked ancestor problem

In the marked ancestor problem, we are given a static rooted tree (with no condition on degree) and must support the following dynamic operations:

- mark(v): mark node v.
- unmark(v): unmark node v.
- lowest-marked(v): return the lowest (nearest) marked ancestor of v, if one exists.

Using Euler trees we can perform these operations in logarithmic time, but we want to do better than that. The optimal bound (of which we will only show the upper bound) is $\theta(\lg n/\lg\lg n)$ query and $O(\lg\lg n)$ update. This is a harder problem than simple predecessor, though we will use van Emde Boas trees in the solution.

One motivation for this problem is from object-oriented programming in a dynamic language. The tree is a class diagram, and the marked nodes are classes that implement a specified method. Given a class, we want to efficiently decide which version of the method to call. (The case of multiple inheritance, where the class hierarchy diagram is a DAG instead of a tree, is still open.)

We solve marked ancestor with a recursive ART decomposition on the input tree. Our algorithm for search will look in the target bottom tree for a marked ancestor, and recurse on the top tree if none is found. Since there are $O(\lg n/\lg\lg n)$ levels of recursion before the size of the subtrees becomes constant, this means we need to handle each bottom tree in constant time.

Recall the key property of bottom trees: we know that each of them has at most $\lg n$ branching nodes connected by paths. On a path, marked predecessor becomes precisely the predecessor problem, so we will replace each multi-node path within the bottom tree by a "compressed path" – a vEB tree that will answer our query in constant time – and otherwise treat it as a single node. This leaves at most $O(\lg n) = O(w)$ nodes, so we can store a bit mask for all of them in constant space. Let each node in the bottom tree be mapped to a bit index (ordered by increasing depth). Each node stores a bit mask specifying which nodes are its ancestors (these can be preprocessed since the tree is static), and the whole tree stores a bit mask specifying which nodes are marked (including which compressed paths contain at least one mark), so the most recent marked ancestor can be read from the masks with a least-significant-bit operation. The time taken is a constant-time bit operation plus a vEB query, giving $\lg n/\lg\lg n$ over all levels of recursion, with update operations being either a bit flip or a $\lg\lg n$ vEB update.

That was easy. Let's do a harder one.

6 Decremental connectivity (in trees)

Another application of the ART decomposition is the decremental connectivity problem in trees. In this problem, we are given an initial tree (or forest), and are required to support two operations:

- remove(e): remove edge e;
- connected(u, v): checks whether vertices u and v are in the same connected of the forest.

We will achieve $O(n)$ total time for all remove operations (at most $n - 1$ of them), and $O(1)$ worst-case time for every query. For this application, it suffices to decompose the tree once without recursing (which wouldn't work anyway: recursion to non-constant depth can't help us attain constant time).

First, we explain a $O(\lg n)$ solution: assign a unique ID to each component, and explicitly store that ID in every node of that component. When an edge is removed, check which resulting subtree is smaller (solvable by running DFS in parallel on each subtree and terminating when one finishes), and reassign the unique id of that component by modifying each of its vertices (this ensures we perform the update on the component having the lower cost).

Consider the size of the component containing an updated node's ID: it changes at most $\lg n$ times, because the size of the component that contains it decreases by a factor of at least two with every update. This can happen at most $\lg n$ times, giving us $O(\lg n)$ amortized time over all updates, with constant query time.

We'd like to get constant (amortized) query time as well, so we'll start with a simpler problem: getting constant time for paths instead of trees.

6.1 Dealing with a compressed path

We now consider a linear chain of k nodes, and we show how to support all edge deletions in this chain in $O(k)$ time, while answering queries in $O(1)$ time. The main idea is that we partition the path into chunks of $\lg n$ consecutive vertices. The state of each chunk can be packed in a word, and we can support updates and queries in constant time.

We now consider an abbreviated path formed by the first and last vertices in each chunk. This has $O(k/\lg n)$ vertices. We maintain for each vertex the ID of its connected component; in the beginning, all vertices have the same component. There are two occasions in which we update the abbreviated path: either when the edge between two chunks is removed, or when the first edge inside a chunk is removed. All edges inside the chunk are abbreviated by one edge, so removing the first edge from the chunk removes the abbreviated edge.

Removing an edge of the abbreviated path splits a connected component into two components, so we need to change the component number of all the vertices in one component. We choose to update vertices in the smaller component. Thus, each vertex's component number can change at most $\lg\left(\frac{k}{\lg n}\right) = O(\lg k)$ times, because one change means that the size of the component the vertex belongs to is reduced to at most half. Thus, the total work in changing component numbers is $O\left(\frac{k}{\lg n} \times \lg k\right) = O(k)$.

We now have to implement queries and specify how to recognize the smaller component when a split occurs. The idea is that for each component ID we hold the first and last vertex of the component. When these change but the ID remains the same, it only takes $O(1)$ time to update.

6.2 The rest of the algorithm

Now we're going to use the ART decomposition, with only one level: we have bottom trees and a single top tree. Since the top tree has $O(n/\lg n)$ branching nodes, we can apply the path compression above to bring its total size down to $O(n/\lg n)$, which means we can solve queries within the top tree using the $\lg n$ amortized approach described above and still get amortized constant time.

This means we just need to worry about bottom trees. For this case, we will use the same solution as the marked ancestor algorithm: the "marked" nodes are now the nodes which are roots of their subtree (i.e. have the edge to their original parent deleted), and we can find the root of any node's subtree in constant time, so we are done.

References

[ABD$^+$06] Boris Aronov, Prosenjit Bose, Erik D. Demaine, Joachim Gudmundsson, John Iacono, Stefan Langerman, and Michiel Smid, *Data structures for halfplane proximity queries and incremental Voronoi diagrams*, LATIN 2006: Theoretical informatics, Lecture Notes in Comput. Sci., vol. 3887, Springer, Berlin, 2006, pp. 80–92.

[AHR98] Stephen Alstrup, Thore Husfeldt, and Theis Rauhe, *Marked ancestor problems*, IEEE Symposium on Foundations of Computer Science, 1998, pp. 534–544.

[Jor69] Camille Jordan, *Sur les assemblages de lignes*, Journal für reine und angewandte Mathematik **70** (1869), 185–190.