6.851: Advanced Data Structures Spring 2007

Lecture 18 — Monday, April 23, 2007

Prof. Erik Demaine *Scribes: Pramook Khungurn, Boris Alexeev, Galen Pickard*

1 Overview

Throughout this course, we have seen a couple of techniques for dividing a tree into smaller substructures: the preferred path and heavy-light decompositions. In this lecture, we discuss two new techniques of decomposing trees into smaller subtrees: the 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

By the end of this course, we will have described four different tree decompositions:

- preferred paths: used in Tango trees and link-cut trees, known since the 1980s
- heavy-light: also used in the analysis of link-cut trees
- separator decomposition: split the tree into roughly equally-sized pieces and recurse
- ART/leaf-trimming decomposition: we trim (groups of) $\lg n$ nodes from the tree

We have discussed the first two in the past, and we will discuss the last two today, with the primary focus being on the ART decomposition. All of them are useful tools to have in your toolbag when designing data structures, especially on unbalanced trees. Leaf trimming is even sometimes useful in dynamic programming.

3 Separator theorem on trees

The separator theorem is based on the following classic, 19th century result of Camille Jordan [Jor69]:

Theorem 1 (Jordan, 1869)**.** *Any tree on n vertices has a vertex whose removal disconnects the tree into components of size at most $n/2$, half of the size of the original tree.*

Proof. We present an algorithm that finds such a vertex. Begin with any vertex v. If v does not satisfy the desired property, there exists a component (after v's removal) with more than $n/2$ vertices in it. Walk one step into that component and repeat.

We need to show that this algorithm terminates, which we can do by showing that it never cycles. But it's clear that we never go back to any old vertices because the component containing them

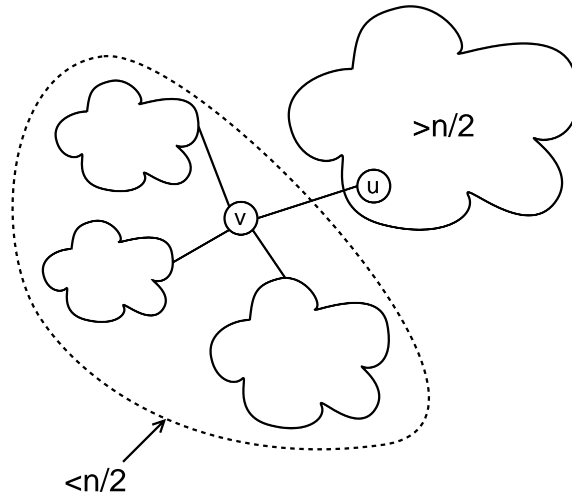

Figure 1: In the proof of Jordan's theorem, a wrong choice of v results in one component having more than $n/2$ vertices. We change v to u, the vertex in the large component adjacent to v, and continue the process.

must have less than $n/2$ vertices. Indeed, we assumed that the rest of the tree contained at more than $n/2$ vertices; see Figure 1. Since the tree is finite and we never revisit vertices, this algorithm terminates on a vertex with the desired property.

□

3.1 Separator decomposition

Using the theorem recursively, we get a tree decomposition. Given an input tree, apply the theorem to get a cut vertex. Make that the root of the new tree, remove that vertex, and recurse on the components to get child subtrees. The output is another tree on the same vertices, except this time with logarithmic depth. An example is given in Figure 2.

The separator decomposition tree can be constructed in $O(n \lg n)$ time, as we can find a separator vertex on $O(n)$ time. Whether the tree can be constructed faster is open.

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

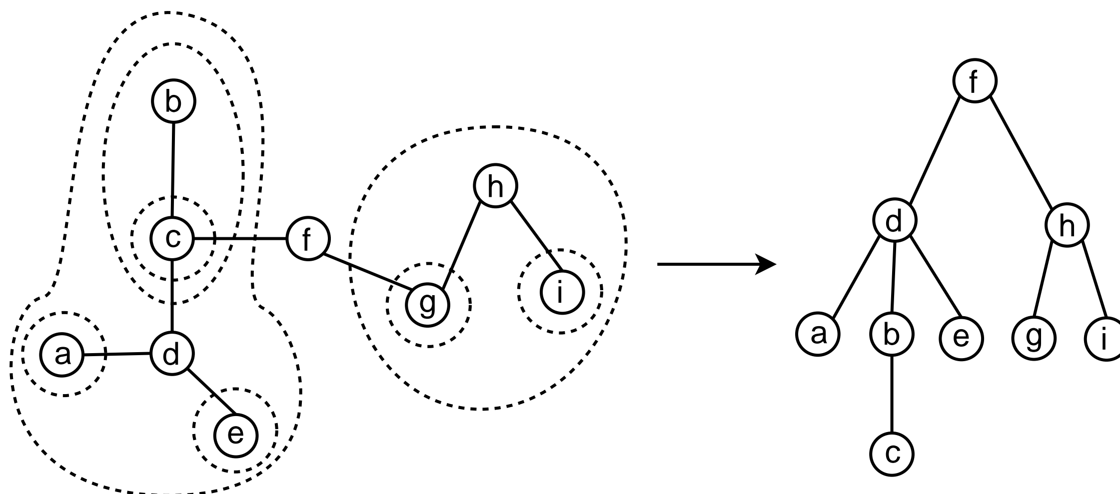

Figure 2: An example of separator decomposition.

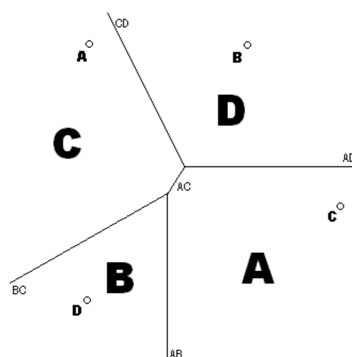

Figure 3: An example of farthest-point Voronoi diagram.

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

More precisely, our oracle here, given an edge (u, v) in the tree, tells us which subtree contains the cat if the edge were removed. This is like binary search, except on a tree instead of a line. It is clear that we can find the cat in $O(\lg n)$ queries to the oracle, assuming the tree has bounded degree.

This has been further 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.

(As a technical note, you may have to use link-cut trees somewhere along the way in the dynamic solution. Also, Professor Demaine is looking for other simple applications of the separator decomposition.)

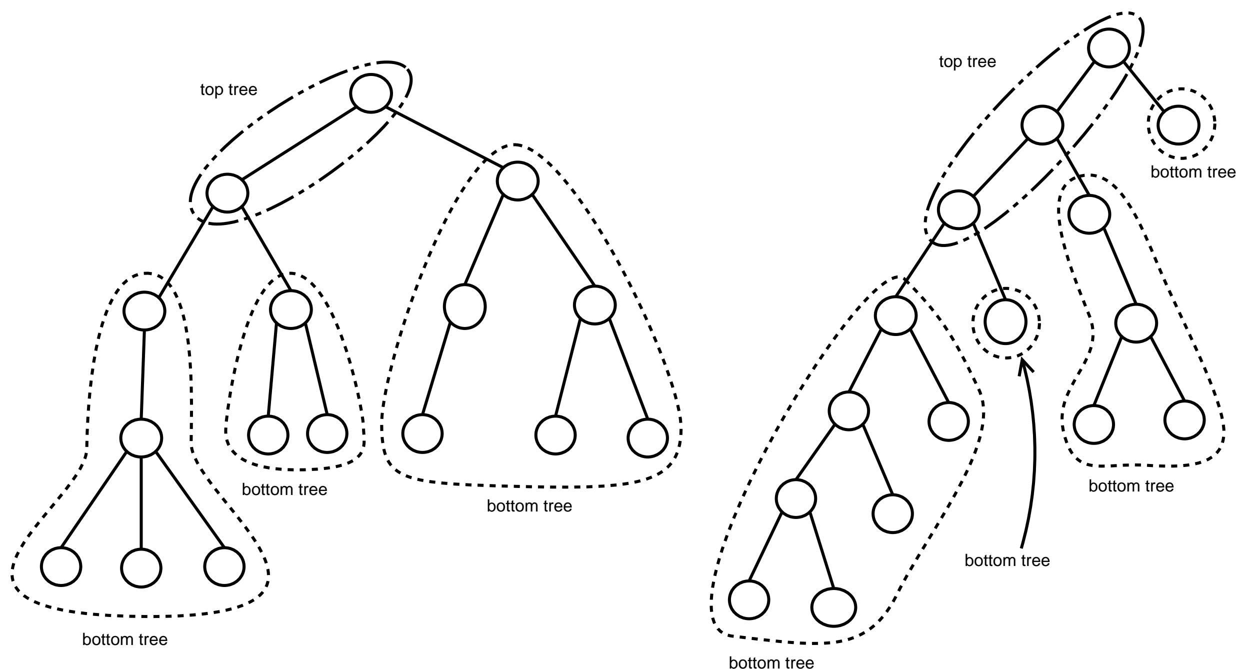

Figure 4: Some decompositions of trees with sixteen nodes.

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. Notice the similarity to the decomposition from Lecture 9, where we had a similar thing except for the word "*nodes*" instead of "*leaves*" above.

Observe that bounding the number of leaves in a bottom tree does not imply any bound on the number of nodes in the bottom tree. However, it does imply that each bottom tree contains 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. We must have that the sum of the number of leaves in these bottom trees must be at least $\lg n$. Otherwise, the leaf of the top tree itself would have been merged with the subtrees to form a higher bottom tree. Thus, we can associate a leaf of the top tree with at least $\lg n$ leaves of the original tree. $\square$

This proposition shows that the "complexity" of the top tree (measured in leaves, or equivalently in branching nodes) decreases by a logarithmic factor. Observe that this crucial property would

not have been true if we had limited bottom trees to $O(\lg n)$ nodes, not leaves.

An an optional step, recurse on the top tree. This idea 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) at preprocessing time. We would then like to 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 link-cut trees or Euler trees, we can do this 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 different from and harder than predecessor, at least for some values of w. The update looks like predecessor, and indeed, it's from van Emde Boas.

The original motivation for this problem is from object-oriented programming in a dynamic language. The tree is a class diagram, and the marked nodes are which classes have explicit implementations of some method. Given an instance of some class, we want to find out which method to call. (Multiple inheritance makes this harder and probably hasn't been studied.)

We use a recursive ART decomposition to solve the problem. Note that there are $O(\lg n / \lg \lg n)$ recursions before the size of the subtrees becomes constant. Thus, we want to step constant time per recursive step. Our solution is to look into a bottom tree, and try to find the nearest marked ancestor. If none is found, recurse in the top tree.

To facilitate this, we divide each bottom tree into nonbranching paths. Because there are at most $\lg n$ leaves, there are at most $O(\lg n)$ nonbranching paths. Each bottom tree maintains a bitvector of which nonbranching paths have marked nodes. They are generally unordered, except by the depth of the top node. This is static, so we can preprocess the paths. (The bits, however, may change.)

With a little more preprocessing, we can get a bitmask of ancestor paths for each node. This allows us to, in constant time, determine the nearest ancestor path with a marked node. Unfortunately, this only gives the ancestor path. We further use a predecessor structure on each nonbranching path. This allows us to find, in time $O(\lg \lg n)$ that we only use once, the nearest marked ancestor *in the path* the query node is in, if there is one.

To facilitate update, each node also stores the level at which it's in a bottom tree. The resulting update time is $O(\lg \lg n)$.

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; recursion is not necessary (nor effective, because we can't even afford $O(\lg n/\lg\lg n)$ time anymore).

First, we explain a $O(\lg n)$ solution: explicitly maintain a component ID for every node. On remove(u, v), run a DFS in parallel from u and v and stop when one of these depth-first searches completes. That will give us the smaller component in order the size of the smaller component time. Update all the labels in that component.

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 two each time. This can happen at most $\lg n$ times, giving us $O(\lg n)$ amortized time.

We'd like to get constant, 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, we only take $O(1)$ time to update (because the ID remains the same).

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. We also have a black box which will solve our problem for paths.

When querying two nodes, we find their LCA and test whether there is any deleted edge on the path from either node to the LCA. If both nodes are in the same bottom tree, this is internal to the bottom tree. Otherwise, we first test the path between each node and the root of its bottom tree, and then perform the same query in the top tree.

In bottom trees, we use the same data structure as in the marked ancestor problem, except that compressed paths use the solution from above. Thus, all operations run in constant time.

To deal with the top tree, we also compress paths as above. Then, the compressed top tree has $O(n/\lg n)$ nodes. Then, we can afford $O(\lg n)$ time per node, while still getting a linear time bound overall. Each node maintains an ID of its connected component. When an edge is deleted, a component gets split into two. We traverse both components in parallel, until we exhaust the smaller one. Then, we remark the nodes in the smaller component. The time is linear in the size of the smaller component. Note that each individual node can be in the smaller component at most $\lg n$ times, so we have a cost per node of $O(\lg n)$.

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.