6.897: Advanced Data Structures Spring 2005

Lecture 16 — April 5, 2005

Prof. Erik Demaine *Scribe: Pramook Khungurn*

1 Overview

We have seen a couple of techniques for dividing a tree into smaller substructures. Tango trees and link-cut trees decompose represented trees into preferred paths. In the analysis of link-cut trees, we also decomposed represented trees into light paths and heavy paths.

In this lecture, we discuss two techniques of decomposing trees into smaller subtrees: separator decomposition and microtree/macrotree decomposition. The latter is applied to solve the marked ancestor problem, and the decremental connectivity problem in trees.

2 Separator Decomposition

Separator decomposition is based on the following classic result by Jordan [1].

Theorem 1 (Jordan, 1869). *Given a tree on n vertices, there exists a vertex whose removal partitions the tree into components, each with at most $n/2$ vertices.*

Proof. Our goal is to pick a vertex v with the desired property. Consider the following process. We start by picking an arbitrary vertex v from the tree. If v partitions the tree into components of sizes at most $n/2$, then we are done. Otherwise, there exists one component with more than $n/2$ vertices in it. Let u be the vertex in that component adjacent to v as in Figure 1. We change v to u, and repeat. Note that if there is a component with more than $n/2$ vertices, we can continue the process. Moreover, we don't go back to any old vertices because the component containing them must have less than $n/2$ vertices. Since there are finitely many vertices in the tree, and we visit each vertex at most once, the process must stop. The vertex at which the process stops is the desired vertex. □

With the theorem, we can recursively decompose a given tree (or, rather, construct a new tree representing the decomposition), as follows. First, we find the separator vertex of the tree. Then, we make the separator vertex as the root of the decomposition tree. Next, we recursively construct trees representing the decompositions of the components, and make the roots of these trees children of the separator vertex. An example is given in Figure 2.

Clearly, the new tree has depth $O(\lg n)$, and it 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.

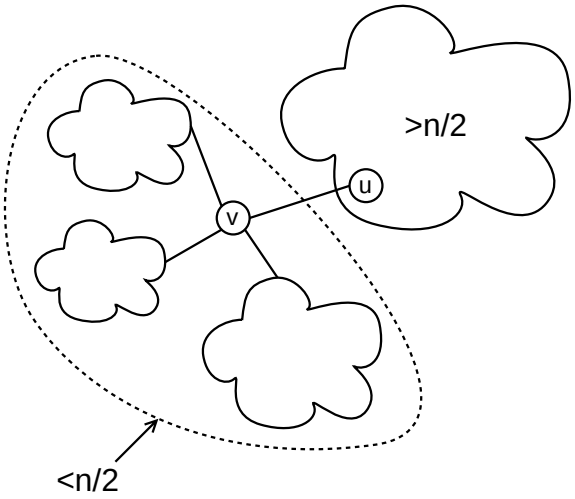

Figure 1: In the proof of Jordan's theorem, a wrong choice of v results in one component having more than $n/2$ vertices. We change v to u, the vertex in the large component adjacent to v, and continue the process.

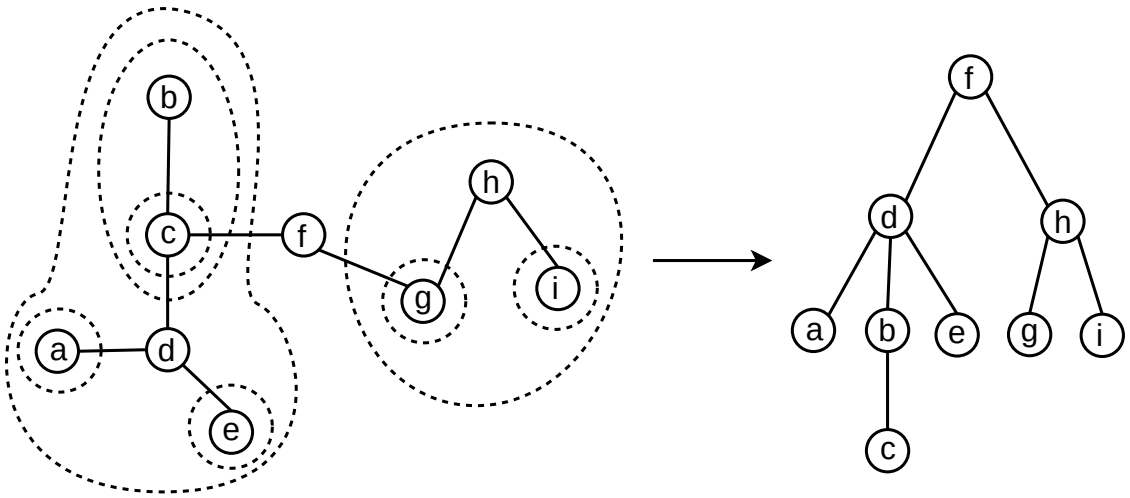

Figure 2: An example of separator decomposition.

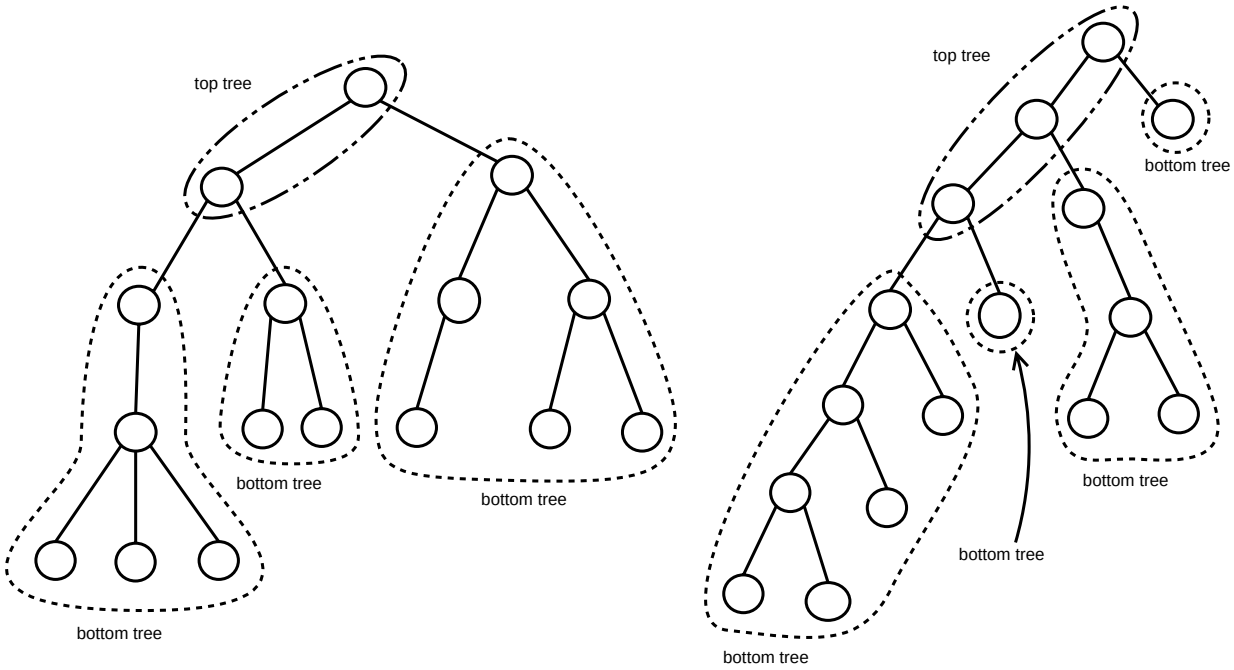

Figure 3: Some decompositions of trees with sixteen nodes.

3 Microtree / Macrotree Decomposition

This is a tree decomposition developed by Alstrup, Husfeldt and Rauhe in [2]. It is defined 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 as a *microtree*.
- Nodes not in any microtree form a macrotree.

Figure 3 gives some examples of this decomposition. Observe that bounding the number of leaves in a microtree does not imply any bound on the number of nodes in the microtree. However, it does imply that each microtree 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 macrotree. All of its children are roots of microtrees. We must have that the sum of the number of leaves in these microtrees must be at least $\lg n$. Otherwise, the leaf of the macrotree itself would have been merged with the subtrees to form a higher microtree. Thus, we can associate a leaf of the macrotree with at least $\lg n$ leaves of the original tree. □

This proposition shows that the "complexity" of the macrotree (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 microtrees to $O(\lg n)$ nodes, not leaves.

4 The Marked Ancestor Problem

In the marked ancestor problem, we are given a static tree at preprocessing time. We would then like to support the following operations:

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

Alstrup, Husfeldt, and Rauhe [2] gave tight bounds and tradeoffs for this problem. This lectures covers the upper bound, while the lower bound will be proven in the next lecture. Specifically, we will show how to support queries in $O\left(\frac{\lg n}{\lg \lg n}\right)$ time, and updates in $O(\lg \lg n)$ time.

Before going into the algorithm, let us consider a simpler problem. Suppose the given tree is a long non-branching chain. Then, we can associate with each node its depth, and maintain the depths of marked nodes in a van Emde Boas predecessor data structure. Finding the lowest marked ancestor of v is therefore equivalent to finding the predecessor of the depth of v. Hence, each operation in this case takes $O(\lg \lg n)$ time.

The main idea of the algorithm is to decompose the tree into microtrees and a macrotree, and then recursively apply this decomposition on the macrotree. When querying, we start in a microtree. If we find that the lowest ancestor is not in the microtree, then we find the lowest ancestor in the macrotree, which is itself decomposed recursively. With each recursion, the number of leaves in the top tree reduces by a factor of $\lg n$, so there are $O\left(\frac{\lg n}{\lg \lg n}\right)$ recursions.

5 Dealing with Microtrees

We conceptually compress each chain of nonbranching nodes into an edge. For such a chain, we maintain a predecessor data structure as mentioned above. Now we have to deal with a compressed tree of $O(\lg n)$ nodes. We maintain the invariant that a node v in the compressed tree is marked if and only if there exists a real marked node in the compressed path above v in the microtree. This node could be v itself, as the lowest node on the path.

Assuming that $w \geq \lg n$, we can perform $mark$, $unmark$, and $firstmarked$ in a compressed tree in constant time. We sort vertices by depth, so that vertex 1 is the root, vertex 2 is one of its children, and so on. Then, we keep a bit vector m of size at most $\lg n$ in a word, such that bit i is 1 if and only if vertex i is marked. Hence, $mark$ and $unmark$ are just changing the appropriate bit, which takes constant time.

To support $firstmarked$, for each vertex v, we keep a precomputed bit vector a_v such that bit i is 1 if and only if i is an ancestor of j. To find the lowest marked ancestor of j, we just compute $m\ AND\ a_j$, and look at the least significant set bit. This can be computed in constant time using bit tricks.

Now we return to the (uncompressed) microtree. To mark or unmark a node, we first update its compressed path in $O(\lg \lg n)$ time. Then, we update the bottommost node on the path in the

compressed tree, which takes constant time.

To query the lowest marked ancestor of v, we first look at the predecessor data structure of the compressed path v is in. If the minimum depth is less than the depth of v, we find the predessor, and output it as the answer. If not, we look at compressed tree, and find a lowest marked ancestor. If there is no such node, then we report that the node has no marked ancestor in the microtree. Otherwise, we answer with the lowest node in the compressed path. Hence, querying also takes $O(\lg \lg n)$ time. But observe that if the node does not have a marked ancestor in the tree, it only takes $O(1)$ time.

6 The Marked-Ancestor Solution

We now switch back to the original marked-ancestor problem. An update in our original problem only needs to update the microtree where the node fits. Thus, updates run in $O(\lg \lg n)$ time.

To query the lowest ancestor of a node, we start with the microtree the node is in. If is has no marked ancestor, we recurse in the macrotree, starting with the root of the microtree. There are $O(\frac{\lg n}{\lg \lg n})$ recursions, as explained above. Note that each recursion except that last one takes constant time. Indeed, a query in the microtree only takes $O(\lg \lg n)$ time if we are querying a predecessor structure corresponding to a compressed path. But this only happens when we know there is a marked ancestor in the compressed path (by comparing against the minimum depth in the path), so the $O(\lg \lg n)$ cost is additive. So queries take $O(\frac{\lg n}{\lg \lg n})$ time.

7 Decremental Connectivity in Trees

Another application of the microtree / macrotree 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.

7.1 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).

7.2 The Rest of the Algorithm

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 microtree, this is internal to the microtree. Otherwise, we first test the path between each node and the root of its microtree, and then perform the same query in the macrotree.

In microtrees, 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 macrotrees, we also compress paths as above. Then, the compressed macrotree 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

[1] C. Jordan, *Sur les assemblages de lignes.* J. Reine Angew. Math. 70, 185-190.

[2] S. Alstrup, T. Husfeldt, T. Rauhe, *Marked ancestor problem*, 39$^{\text{th}}$ FOCS, 534-543, 1998.