6.854 Advanced Algorithms

Lecture 7: September 30, 1999 Lecturer: David Karger

Scribes: Daniel Barkalow, Sara Picket, Ilya Shlyakhter

7.1 Dynamic Graph Connectivity

With a static graph, in order to determine if two nodes are connected by any series of edges, we could use a breadth-first search. The problem becomes more difficult, however, if we allow edges to be dynamically inserted and deleted from the graph, but still wish to be able to make queries about connectivity. We could simply redo the breadth-first search for every query, but if there are limited operations or limited structures, we can perform updates in such a way as to make querying much more efficient.

Dynamic graph connectivity is the the standard example of dynamic graph data structures, and there's a lot of history and previous work for this problem. There are many sub-linear time algorithms from the 1980s.

- Frederickson – $O(\sqrt{n})$
- 1995 – Henzinger and King: $O(\log^3 n)$ with randomization
- 1997? – Holm, de Lichtenberg, and Thorup: $O(\log^2 n)$. Similar to Henzinger and King's algorithm, but with the randomization removed. It's a rather subtle algorithm.

All of those running times are amortized – an algorithm with a good worst case running time has not yet been developed.

7.1.1 Forests of Trees

What can we do if a graph is always a forest of trees and we only allow some operations?

Forests with Insertion Only

Use disjoint sets to store the tree connectivity. Adding an edge between trees is a union; to check connectivity, see if the two nodes are in the same set. This is very fast, and runs in $O(m\alpha(n))$ time over m operations, for $O(\alpha(n))$ amortized time per operation, where α is the inverse Ackerman function.

Forests with Deletion Only

Label each node with the component it's in. This label can be anything, as long as it's unique for each component. Query is then $O(1)$.

To do a deletion, maintain the information by relabeling the smaller tree. Find the smaller tree by a depth-first search on both sub-trees a nd stop when you've only hit the end of the smaller. This takes only as long as relabeling the tree.

Each time relabeling takes place, the component size is halved, so each node can only be relabeled $O(\log n)$ times. Therefore, the total time for the deletes is $O(n \log n)$ if all of the edges are deleted.

7.1.2 Non-forests

Insert doesn't care about whether it's a tree. Extra edges in a component can just be ignored.

For deletion of edges, still label nodes with component names, but also maintain a spanning forest. If an edge to be deleted isn't in the spanning forest, ignore it. If it is, relabel the smaller half of the component. Then try to find a replacement edge which goes between in the smaller half and the other half. If we can find the replacement edge, then we have to un-relabel the smaller half of the component.

Analysis: On a failed search, all failed edges have both endpoints in the half we relabeled, so their component size has been halved. Therefore, there can be at most $O(\log n)$ failures per edge.

On a successful search, nothing halves. So it's still $O(m)$ time, where m is the number of edges in the graph. *But*, if we could somehow conserve the work we did on the smaller halve, we could improve performance by not redoing work. This will require a real algorithmic tweak, not just a change in our analysis.

7.1.3 Forests, with Insertion and Deletion

Use the idea of union find: root the trees at arbitrary nodes, and do a "find" to identify the containing component by going up from the nodes towards the root(s) and testing if the two nodes have a common root.

This takes time $O(d)$, where d is the depth of the tree, which may be $O(n)$. We need some method of making the tree shallow via encoding, since we can't modify the tree:

- Sleator and Tarjan: compressed long paths in trees
- Tarjan and Vishkin: represent tree as a list with Euler tour trees

7.2 Euler Tour Trees

We want a representation of an arbitrary tree as a linear data structure, so we can use our usual techniques on it (i.e., $O(\log n)$ operations). To do this, we define an Euler tour as follows:

Definition 1 *An Euler tour on a tree T is a sequence of nodes of T such that for each edge in T, the vertices at the ends of the edge appear adjacent once in each order, and no other vertices appear adjacent. The root of the tour is the first (and last) vertex listed.*

Notice that the last vertex in the sequence is always a repeat of the first; if this occurrence is ignored, several things become simpler. Each vertex appears a number of times equal to its degree. Changing the root is simply rotation. However, we keep both the first and last occurrences in the following discussion.

Every tree has at least one Euler tour with any given vertex as the root. One can be found by starting at the root and traversing any unused edge whenever there is such an edge, and traversing the only edge which has been used once when there are no unused edges, unless all edges have been traversed twice, at which point the tour is finished. (Note that it is always possible to conduct an Euler tour because trees are planar.)

Every tour defines a unique tree, which can be found by accumulating the edges corresponding to adjacent vertices in the tour.

It is possible to change the root of a tour by removing the final vertex, splitting the sequence before some occurrence of the new root, reversing the two halves, joining the two pieces, and duplicating the first vertex at the end.

To connect two trees represented by Euler tours, change the roots of the two trees to be the two endpoints of the edge to be added, and then append the tours, duplicating the first vertex at the end of the tour.

For example, if you connect the trees represented by **babcb** and **ded** with an edge from **b** to **d**, you get **babcbdedb**. (Figure 7.1)

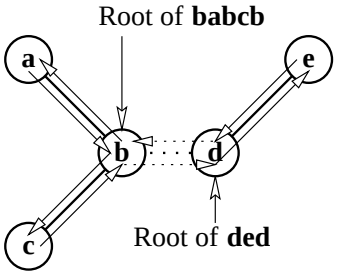

Figure 7.1: Adding an edge from b in babcb to d in ded

To remove an edge, split the tour at the two places corresponding to the edge. One of the new trees is the middle section. The other tree is formed by removing the first vertex in the last section and appending the first and last sections.

For example, if you take the tree represented by **abdedbcba** and split it by removing the edge from **b** to **d**, you get **ded** and **abcba**. (Figure 7.2)

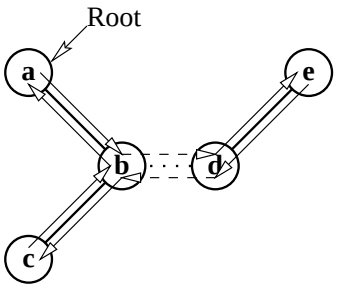

Figure 7.2: Deleting the edge from b to d from abdedbcba

The fastest way to represent an Euler tour is with a splay tree. Since splay trees support $O(\log n)$ split and join, adding and deleting edges are $O(\log n)$. In order to find significant places quickly, keep pointers from the vertices to nodes in the splay tree which represent them, and from the edges to both pairs of nodes surrounding the edge. (Figure 7.3)

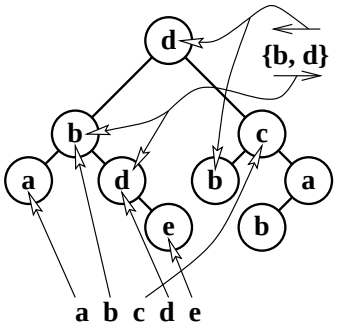

Figure 7.3: References from the graph to the splay tree