Applications of Range Search

Instructor: Richard Peng

Apr 7, 2021

DISCLAIMER: These notes are not necessarily an accurate representation of what I said during the class. They are mostly what I intend to say, and have not been carefully edited.

Last time we spent most of the lecture talking about a data structure that supports:

Lemma 0.1. We can solve a sequence of $n$ operations of:

1. range add,
2. range sum/min/max,

in $O(n \log n)$ time.

Today we discuss some applications of such data structures in graph algorithms.

1 MinCut that 2-Respect Some Tree

A variant of this structure, that maintains min, is useful for computing the minimum cut that intersects two edges of some tree. It shows up because:

1. 3 vertex connectivity and planarity testing is closely related,
2. global min cut, with some reduction, becomes finding an edge that intersects up to two edges of a tree.

As a warmup, consider finding the minimum cut that intersects one edge of some tree. Think about what happens to the set of edges that cross some cut, as we move the cut around the tree. Say the edge we cut is from some $u$ to its parent $p(u)$, then the weight of the cut is the total weight of the subtree. There are two ways to deal with this:

1. Either maintain a (mergeable) priority queue of all the edges in each subtree, sorted by the depth of their ‘highest point’ (lowest common ancestor).
2. Or do a heavy-light decomposition to partition the trees into paths. Then move up each path, maintaining one priority queue for when the edges exit.

For the 2 edges being cut case, assume the 2 edges being cut belong to different pieces of a centroid decomposition. Then the thing to note is that an edge from the ‘lower’ tree incurs a cost equaling to its weight to everything in the ‘upper’ tree that’s not in the subtree of its other end point. So in the pre-order layout of the tree, this is a range increment of everything outside of a subtree. There is several layers of recursion involved, so the overall runtime comes out to $O(n \log^3 n)$ if done naively. I think

https://arxiv.org/abs/1911.01145

gets this all the way down to $O(m \log n)$. 


2 Dynamically Changing Trees

Things get even more interesting when the underlying tree is changing.

Thorup'00 gave a fully dynamic data structure for maintaining the minimum cut in a graph. Key to it is a data structure that supports the following operations on a graph plus a tree:

1. Insert/remove an edge
2. Link/cut the tree
3. Check if some tree edge is crossed by at most \( c \) edges, for some small value of \( c \).

The goal is something that runs in \( O(\sqrt{n} \cdot poly(c)) \) time per update.

For simplicity, start with the case where the tree is guaranteed to be a path. Then we partition the path into pieces of length \( B \).

Then up to \( n/B \) updates, we only ever cut the pieces apart.

On each piece, note that the edges only go left or right. The key observation is that only the leftmost/rightmost \( c \) edges matter.

**Claim 2.1.** For each pair of pieces, we only need to store the \( c \) leftmost/rightmost edges between them.

Then when we move the pieces around, the number of edges updated is

\[
\left(\frac{n}{B}\right)^2 \cdot c
\]

while the cost of updating a single piece, assuming bounded degree, is just \( O(B) \).

So setting \( B \) to around \( n^{2/3} \) gives a cost of \( n^{2/3}c \) per update.

For general trees, observe that we can always partition a tree into \( O(n/B) \) pieces of size \( B \) so that each piece is incident to at most two portals (vertices involved in other pieces). Call the path between these two portals the ‘highway’ of that piece.

Then a graph edge covers the path from its end point to the highway, no matter how the neighboring piece is assigned. The portion of the path on the highway then behaves analogously to the path case, and we get the same runtime back.