

Disclaimer: *These are rough notes, with some exercises.*

This presentation borrows bits from Demaines lecture notes on link-cut trees.

7.1 General Flows again.

Question: Recall blocking flows in general. Recall the depth first search algorithm. What was its runtime? Why?

Depth first search which took $O(nm)$ time. Charge work to pops. At most n pops per augmentation, so it is $O(nm)$.

Question: Do we have to pop the stack completely?

No. We could just pop back to the saturated arc.

Question: Does this help?

Maybe not. We still may pop $\Theta(n)$ nodes.

Question: But what if we left evidence that we had been here before?

Say, leave a pointer to the node we had been exploring before the augmentation.

Question: What would we be leaving?

We had a path, an augmentation cuts it at a point, leaving two paths.

Then we start exploring again and hit a previous path. Jump to its root, and continue exploring. Find another augmentation, and cut it, and so on. We leave out-pointers sprinkled through the graph. Since we have at most one out-pointer per node, we have trees sprinkled throughout the graph.

Trees sprinkled throughout the graph.

Question: In a new depth first search, if we hit a tree, where should we go, again?

We should jump to its root, and explore a new edge there.

Question: When do we change a pointer for a node? How often do we do so?

Only when an arc is saturated. $O(m)$ times.

Question: When do we join trees?

When we change out-pointers.

Question: What is slow?

Finding the root of a tree. Could have to chase pointers to the root.

Question: We are talking about a data structure here. What are its operations?

Link (u, v) to join a tree. Cut (u, v) to cut away a subtree. Findroot to find the root of the tree.

Question: Finally, when we find the sink, what must we do?

We need to augment flow along the source sink path. Reduce the capacities of each of the arcs, and cut the saturated arcs.

Question: Uh oh. We need to pay attention to the capacities. Shall we (re)think the data structure operations?

Sure. We should have the following operations.

Link (e, c) adds link e with associated capacity c to set of pointers.

Cut (e) removes e from set of out-pointers, returns current capacity in tree.

Find-Bottleneck (u) Finds minimum capacity link from u to “root” of its tree.

Find-Root (u) Finds “root” u of its tree.

Decrease-capacity (u, δ) Decreases capacity of all links along path to root.

Question: What is the total number of operations one needs for maximum flow?

$O(nm)$. $O(m)$ for each blocking flow since push is link, pop is cut, augment is implemented by finding-bottleneck edge amount, reducing residual capacity of all arcs on path using decrease-capacity, and cutting bottleneck edges.

The actual flow values of an arc is the difference between the original capacity and the residual capacity at the end of the blocking flow computation.

Question: How do we implement the data structure?

Link is easy, cut is easy. What about find-root, we can hop along the path until we reach the root. Oops. That may take $\Theta(n)$.

Question: What should we do? How about if it were just a path not a tree?

Well. We should be able to do this with some sort of ordered balanced binary tree where the leaves correspond to the edges in the path with cut and join operations.

Uh-oh. Should we call this tree the “ds-tree” (data structure tree) just to confuse ourselves less.

For example, find-root, is the head of the first edge in the order.

Question: What does the path from an edge e to the beginning of the path correspond to in the ds-tree?

Notice that the as we go to the root, the union of the leaves in the left sub ds-trees (and the original leaf) correspond to the path to the beginning of the path.

Question: How could we implement reduce capacity?

Well, we want to reduce the capacity of all the edges to the beginning of the path.

Question: How do we use the “ds-trees”?

Well, we could put a “difference” on the root of each sub ds-tree signalling that we have reduced the capacities of all edges in the subtree.

Question: In somewhat more detail?

That is, maintain for each internal node a “Delta” value. Then, when reduce-capacity is called on a node (equivalently, edge), we traverse up the tree and increase the Delta value on each left child. This means for all edges in sub ds-tree the capacity is thought to be reduced by delta. Inductively, the capacity of an edge is its capacity reduced by the delta values of all its ancestors. (To find the capacity of an edge, one could envision traversing the ds-tree to the root and reducing the original capacities by all the delta that we encounter.)

Question: How would we find the bottleneck capacity for some subpath?

In addition, one can maintain a heap ordering of the partially reduced capacities as one traverses up to the root. That is, we can maintain the invariant that the root of a sub ds-tree knows the minimum of the capacities reduced by deltas in the subtree.

Question: How can this be maintained?

As we reduce-capacities, we increase the Delta value on the left sub ds-tree. But this does not change the relative order of the edges there. So, the minimum of that tree can be adjusted and compared to the minimum from the right as we move up.

Question: So, how do we find the bottleneck value from an edge?

Thus, to find the bottleneck value. We traverse up the tree as follows. We initialize the minimum to be the value of the edge at a leaf, reduce by the current Delta, go up the tree one level and take the minimum of current minimum and the left child, and continuing.

Question: How do we maintain trees?

We can decompose a tree into paths as follows. Choose the root to leaf path that always goes toward the heaviest subtree. This yields a forest and continue. The maximum number of paths that one needs to interact with is $O(\log n)$ since each change of path doubles the size of the subtree. This yields on $O(\log^2 n)$ per operation bound.

Question: The heaviest path may change. How do we deal with maintaining this dynamically?

Make each find-bottleneck (link, cut or reduce-capacity) join paths along path (breaking off other branches.) (That is, forget about the heaviest path.) We can use amortized analysis to say that on average there are $O(\log n)$ switches as follows.

A light edge switch is the event where one joins an edge in the process above where there a factor of two fewer nodes in the subtree below the edge. There are at most $\log n$ light edge switches since the root contains at most n nodes, and each light edge reduces the number of nodes by a factor of two.

The total number of heavy edge switches over all operations is bounded by the total number of light edge switches plus an upper bound on the number of times any edge becomes heavy.

Exercise 1: Show that the total number of times edges become heavy is upper bounded by number of operations time $O(\log n)$.

Thus the total number of path switches is $O(m \log n)$.

Question: What is the total time?

We get $O(\log n)$ time per path switch so, over $O(m)$ operations, our total time becomes $O(\log^2 n)$. That is, a blocking flow can be computed using these data structures in this time.

Exercise 2: If any tree that is being stored is of depth d , can you

give a better bound on the complexity above.

Question: Can one do better?

Yes. $O(\log n)$ per operation. Using splay trees and fancier arguments. In particular, the $O(\log n)$ cost per path switch can be less if the place where the path switch is near the root in the splay tree. If on the other hand one is far from the splay tree root, one makes big progress towards the root of the dynamic tree in this operation.

Indeed, one can get to $O(nm \log(n^2/m))$ time using a different scheme called push-relabel invented by Goldberg (and developed in conjunction with Tarjan.)

7.2 Scaling algorithms.

Question: Those data structures are pretty fancy. Are there simpler $O(nm)$ ish algorithms?

Sure. Well, we will show $O(m^2)$, which is around the same for sparse graphs, i.e., $m = O(n)$.

Question: We did “shortest” paths before. What might be another thing we could do?

Fattest residual path.

Question: How do we analyze this?

Perhaps, say we get a decent fraction of the remaining flow.

Question: Well, assume that we have a flow in the residual network of value f . Does there exist a path of high(ish) value? How would you show this?

Well, one could show that the flow is the sum of a small number of flows, each of which consists of a path.

Question: And?

Well, given the flow. One can find an $s - t$ path of positive flow edges. Remove the path by decreasing the corresponding flow values, one of which to zero. This will yield at most m paths.

Question: So, the “fattest” of these paths, must have value at least?

At least f/m .

Question: So, in m iterations, we are done?

No. In the next iteration, we may route f'/m , where f' is the residual flow left (the difference between the current flow and the optimal.)

Question: What can we say?

That the flow remaining reduces by a factor of at least $(1 - 1/m)$ with each augmentation.

Question: How many augmentations before the fattest path algorithm reduces the flow by $1/2$?

$O(m)$.

Question: What is the total running time?

$O(m \log n C(m + n \log n))$, $O(m + n \log n)$ time to find fattest path, and $O(m \log n C)$ iterations to get down to one unit of flow remaining. One doesn't need to find the fattest, but do so within a factor of two.

Exercise 3: Give as fast an algorithm as you can for the fattest path problem.

Exercise 4: Give as fast an algorithm as you can for the problem of finding a path whose fatness is within a factor of 2 for optimal.

This can easily be done in time $O(m)$. So, we get $O(m^2 \log nC)$.