CS 270: Combinatorial Algorithms and Data Structures Spring 2023

Lecture 6 — February 2, 2023

Prof. Jelani Nelson *Scribe: Kelvin Lee, Sanjay Subramanian*

1 Link-cut Tree Analysis Wrap-up

Recall from last lecture that with link-cut trees, we can perform two actions: `link(v,w,x)` and `cut(v)`.

- `link(v,w,x)`: `access(v)`, `access(w)`, `v.parent = w`, `w.left = v`.
- `cut(v)`: `access(v)`, `v.left.parent = NULL`, `v.left = NULL`.

Also recall that the total time to perform m operations is $O(m + \underbrace{\text{cost}(splay)}_{O(\log n)} \cdot PCC)$ where PCC is the number of preferred child changes. Now we want to bound PCC using the Heavy-light decomposition technique introduced from previous lecture.

1.1 Heavy-light Decomposition

Recall the Heavy-light decomposition, which partitions the edges into two groups: heavy or light.

Definition 1.1 (Heavy/Light)**.** Suppose we have a represented tree, say v has child w, then (v, w) is *light* if $\text{size}(w) \leq \text{size}(v)/2$. We call it *heavy* otherwise.

Claim 1.2. $PCC \leq 2 \cdot LPCC + n + m$.

Proof. Note that $PCC = LPCC + HPCC$. Then our goal is to show that $HPCC \leq LPCC + n + m$. Suppose (v, w) is a light preferred child destruction and (v, z) is a heavy preferred child creation. We consider two cases:

1. (Case 1: `w = NULL`) This could happen for two reasons:
   - the last access in v's subtree was v. This happens at most m times since where m is the number of operations.
   - this is the first ever access in v's subtree. This happens at most n times.
2. (Case 2: `w != NULL`) This happens at most $LPCC$ times because we charge the creation of (v, z) to the destruction of (v, w), and each light preferred child can only be destructed once.

□

Claim 1.3. $LPCC = O(m \log n)$.

Proof.

1. `access`: if we look at any path from the root to v in the represented tree, the number of light edges is at most $\log n$ and so $LPCC \leq O(\log n)$.

2. `link`: adding the edge doesn't change preferredness in the tree, it only changes heaviness. We can only have HPCC as a result of linking, so there's no effect on LPCC.

Since LPCC $\leq O(\log n)$, the total LPCC across the m operations would be $O(m \log n)$. □

2 Min-cost Max-Flow

The Min-cost Max-Flow (MCMF) problem has the same setup as max flow except that now edges have costs in the set $\{-C, ..., C\}$.
Goal: Find max flow of minimum cost.

$$cost(f) = \sum_{e \in E} f_e \cdot c_e$$

Chen et al. [1] recently gave an almost-linear time algorithm for this problem: $m^{o(1)} \lg(C) \lg(U)$.
A related problem is Min. Cost Circulation (MCC).

Definition 2.1 (Circulation)**.** A circulation is a flow $f : E \to \mathbb{R}$ such that $val(f) = 0$.

The goal of the Min. Cost Circulation problem is to find a circulation of minimum cost.

Note that when constructing the residual graph of a flow, the cost of the reverse edges that are added should be the negative of the cost of the corresponding forward edge.

Theorem 2.2. *Given an algorithm for either Min. Cost Max. Flow or Min. Cost Circulation, we can solve the other problem with at most an extra linear factor in the running time.*

Proof. We will first show that if we have an algorithm for MCMF, then we can obtain an algorithm for MCC. Given the graph G, we will add a source s and a sink t with no edges incident on them, and apply the MCMF algorithm. The resulting flow must be a min. cost circulation in the original graph G.

Next, we will show that if we have an MCC algorithm, then we can obtain an algorithm for MCMF. Given the graph G with source s and sink t, add an edge from t to s with capacity ∞ and cost $-(m+1)C$. We can find the min-cost max-flow by first finding a max flow and then using the MCC algorithm to find a min circulation in the residual graph. □

Note that any circulation decomposes into a sum of at most m cycles (similar to the fact that a flow decomposes into a sum of at most m paths). Each cycle in min. cost. circulation has a cost. What can we say about each cycle's cost? *Each cycle's cost must be negative.* The reason is that if a cycle's cost weren't negative, we could remove it from the circulation and doing so would not reduce the cost of the circulation.
Algorithm: The argument above motivates the following algorithm for MCC:

1. Findg max. flow f
2. Compute residual graph for f
3. If there is no negative cost cycle, then terminate.
4. If there is a negative cost cycle, augment by that cycle and return to step 2.

Note that Step 4 in the algorithm does not change the flow because it adds a cycle. To implement Step 4, we can use Bellman-Ford or the BNW algorithm presented earlier in this class.
To bound the running time of the algorithm, note that the number of iterations can be at most $2mUC$. To see why, we first observe that the flow on each edge in the max. flow f can be at most U and the cost on each edge can be at most C, so $cost(f) \leq m \cdot U \cdot C$. Each time we execute Step 4, the cost must decrease by at least 1. The minimum possible cost of a flow occurs if all edges have maximum flow value and the lowest (most negative) cost; that value is $-mUC$. Thus, Step 4 can occur at most $mUC - (-mUC) = 2mUC$ times.

We would like to design an algorithm for MCMF that does not have a polynomial dependence on U and C. Problem Set 2 will involve this problem, but we will now see how this can be achieved in some special cases.
Recall the concept of a price function: $\phi : V \to \mathbb{R}$. The reduced cost of edge $e = (v, w)$ is $c_\phi(e) := c(e) + \phi(v) - \phi(w)$.

Theorem 2.3. *All edges have nonnegative reduced costs if and only if the costs of all cycles are nonnegative.*

Proof. First, suppose that all edges have nonnegative reduced costs. The reduced cost of each cycle is the sum of the reduced costs of the edges in the cycle, so the reduced cost of each cycle must be nonnegative. The reduced cost of each cycle is also equal to the cost of the cycle (due to the telescoping sum). Thus, the reduced cost of each cycle must be nonnegative.

Now, suppose that all cycles have nonnegative costs. Create a dummy vertex α and add edges from α to all other vertices, each of which has a capacity of 0. Now for each vertex v, define $\phi(v) = d(\alpha, v)$, which is the shortest path from α to v. (This shortest path is well-defined because there are no cycles with negative costs.) The reduced costs of all edges will be nonnegative under this price function. □

- **Special Case 1:** Suppose that all costs are nonnegative to start off and all edges have unit capacity ($\forall e$, $u_e = 1$).

  Consider the following algorithm.

  1. $\forall e \in E$, $f(e) := 0$.
  2. $\forall v \in V$, $\phi(v) = 0$
  3. Find shortest augmenting path (SAP) from s to t using c_ϕ.
  4. Augment along path SAP
  5. $\forall v \in V$, $\phi(v) := \phi(v) + d_\phi(s, v)$.

  Theorem 2.4. *This algorithm will never introduce a negative cost.*

  Proof. Consider an SAP used by the algorithm in some iteration. Let (v, w) be any edge on this path. The algorithm will add $d_\phi(s, v)$ to $\phi(v)$. The algorithm will add $d_\phi(s, w)$ to $\phi(w)$. Thus, the total amount that will be added to $c_\phi(v, w)$ is $d_\phi(s, v) - d_\phi(s, w)$. Now note that since (v, w) is part of a shortest s-t path under the edge weight function given by c_ϕ, $d_\phi(s, w) = d_\phi(s, v) + c_\phi(v, w)$. Therefore, the amount added to $c_\phi(v, w)$ is $d_\phi(s, v) - d_\phi(s, w) = -c_\phi(v, w)$. So the new value of $c_\phi(v, w)$ will be 0. Thus in the residual graph, the newly added

reverse edges will not have negative reduced cost: they will have zero cost! All other edges remain with nonzero reduced cost by the triangle inequality ($\phi(v) = d(s, v)$ is always a feasible price function, as we saw in SSSSP). □

- **Special Case 2:** $\forall e \in E$, $u_e = 1$, but some costs can be negative.

  Consider the following algorithm. Let G be the input graph. First, find a max. flow. Then we need to find a min. cost circulation in the residual graph. To this end, first fully saturate the edges in G that have negative cost. Note that by saturating these edges, we could cause some vertices to have a deficit of incoming flow and some vertices to have a surplus of incoming flow. Now consider the residual graph. Create a "super" source s' with an edge to all vertices that have a deficit of incoming flow. Create a "super" sink t' with an edge to all vertices that have a surplus of incoming flow. All of the added edges should have 0 cost. Now we can use the algorithm described in the first special case to find a min. cost max flow from s' to t'.

References

[1] Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, Sushant Sachdeva. Maximum Flow and Minimum-Cost Flow in Almost-Linear Time. *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 612-623, 2022.