CS 224: Advanced Algorithms Fall 2014

Lecture 23 — 11/18/2014

Prof. Jelani Nelson *Scribe: Jeffrey Yan*

1 Overview

- Finish link-cut analysis (will show $O(\log^2 n)$).
- Min cost max flow.

2 Last time

- `access`(v) makes v the root vertex of root tree of aux trees.
- `cut`(v): `access`(v), v.left.parent = none, v.left = none
- `link`(v, w): `access`(v), `access`(w), v.left = w, w.parent = v.

(number of splays during access) = 1 + PCC, where PCC is the number of preferred child changes in this access call. Therefore, runtme of m operations is at most cost(splay) $\times$ $(m + \text{PCC}) \leq O(\log n) \times (m + \text{PCC})$.

Before we bound PCC...

3 Heavy-light decomposition

Definition 1. *Heavy light decomposition*

For vertex v, define size(v) to be the size of its subtree. We say (u, v) is "light" if size(u) $\leq \frac{size(v)}{2}$ and otherwise we way it's "heavy".

Thus each edge is one of 4 types $\{\text{light}, \text{heavy}\} \times \{\text{preferred}, \text{notpreferred}\}$.

If (u, v) (v is parent(v)) used to be preferred, but now (w, v) is, then we say (u, v) was "destroyed" and (w, v) was "created."

Claim 2. $\text{PCC} \leq 2\,\text{LPCC} + m + n$, *where LPCC stands for light preferred child creation.*

Proof. Say (u, v) is destroyed and (w, v) is created. If (w, v) is light, it's counted on both sides. If (w, v) is heavy, we have two cases.

- Case 1: u = null. Either this is first access in v's subtree (happens at most n times), or last access in v's subtree was to v. Charge this to the last access on v (happens at most m times).

- Case 2: $u \neq$ null. We destroyed preferred light edge (u, v). Charge to its creation (hence factor of 2 in the bound).

□

Now let's bound LPCC across access/link/cut.

- access: on any root-to-v path in rep. tree, number of light edges $\leq \log(n)$ and light vs heavy doesn't change.
- link(v, w): We first do access(v) and access(w). Then all edges along preferred child path to w got heavier, which we aren't counting (if they're light their creations were counted in access). Other edges hanging off this path got lighter, but they aren't preferred.
- cut(v): cuts off v from its parent w in rep tree. This time edges hanging off preferred path to w got heavier, so we stil don't count these. However, there are some LPCC's along this path, but at most $\log(n)$ of them. Therefore, in total, LPCC"s across m operations is $O(m \log(n)$.

Therefore the total time is $O(m \log^2(n) + n \log(n))$. For operations like add and subtracting flow, we use augmented BST's to achieve the same time bound.

4 Min cost max flow

New problem: Each edge also has a cost $c(e)$ (possibly negative). Capacities are in $\{1, \ldots, U\}$. Costs are in $\{1, \ldots, C\}$. Goal is to find max $s-t$ flow f which minimizes $\text{cost}(f) = \sum f_e c(e)$.

Note: in residual graph, reverse of edge e has cost $-c(e)$.

Related problem: Min cost circulation, where a circulation is a flow such that there is zero net flow through all vertices.

Claim 3. *MCMF and MCC efficiently reduce to each other.*

Proof. To get MCMF $\Longrightarrow$ MCC, create isolated source and sink.

To get MCC $\Longrightarrow$ MCMF, max flow f in G, and then add on the MCC of the residual graph. This is valid because the difference of any two flows with the same value (e.g. any two max flows) is a circulation. □

Proposed algorithm for MCC: while there exists a negative cost cycle in residual graph, augment along that cycle.

We can find negative cost cycles in time

- $O(mn)$ by Bellman-Ford.
- $O(m\sqrt{n}\log(C))$ by [3]

Proof. Proof of correctness: Say we have circulation f stricly worse than MCC f^*. Note that $f^* - f$ is a negative cost circulation that is feasible in the residual graph. Since circulations decompose into cycles, we can augment by a negative cycle. Thus, once the algorithm terminates, we have found an MCC. □

Runtime with Bellman Ford: $O(mn(mUC))$. There are at most $2mUC$ iterations (each iteration decreases cost by at least one, and the initial cost is at most mUC and can never be less than $-mUC$. The per-iteration cost is $O(mn)$ using Bellman Ford.

Definition 4. *A price function $p : V \to \mathbb{R}$ induces a reduced cost function $c_p(u, v) = c(u, v) + p(u) - p(v)$.*

Claim 5. *f is MCC iff there exists price function p in G_f such that all reduced costs are nonnegative.*

Proof. Suppose there exists such a p. Then all cycles are nonnegative because price functions cancel along cycle.

Suppose G_f has no negative cycles. Introduce new vertex s with 0-cost edges to all v. Define $p(u) = d(s, u)$. Triangle inequality implies that $c_p \geq 0$. □

How can we get a faster algorithm for MCMF?

Simple case: no negative cost edges, and all capacities are 1.

Shortest Augmenting Paths (SAP) algorithm: Find shortest $s - t$ path in residual graph (by cost), and augment along this path. Repeat until we have found a max flow. The runtime of this is $O(nm + n^2 \log(n))$, because there are at most n iterations, with $O(m + n \log(n))$ per iteration using Dijkstra wtih Fibonacci heaps.

Claim 6. *There will never be negative reduced costs in the residual graph.*

Proof. Set prices as $p(v) = d(s, v)$ in residual graph. Any edge we augment along is in a shortest path, so its reduced cost is 0 (so is its reversed edge). Therefore, we never add negative reduced cost edges to the residual graph, and thus the residual graph never has a negative cost cycle. □

Recall a max flow is an MCMF when the residual graph has no negative cycles, so this algorithm is correct. The running time is $O((m + n \log n)n)$ (at most n iterations since we push one more unit of flow each iteration and the out-capcaity of s is at most n; $O(m + n \log n)$ per iteration via Dijkstra's algorithm using Fibonacci heaps.

4.1 Min cost circulation using SAP

Here again we will assume all capacities are 1, but we will not assume that there are no negative cost cycles (else the zero flow is the MCC!). How can we use SAP in this scenario?

First, define a flow f which fully saturates all negative cost edges. That is, if edge e has cost $c(e) < 0$, then set $f_e = -1$; else set $f_e = 0$. Now in the residual graph all edge costs are nonnegative (and all

positive capacities are still 1). Unfortunately, there is no reason to expect this f to be a circulation, which is what remains to be fixed.

Note that given our current flow is f, some vertices v have flow *excess*, i.e. the outflow of v minus the inflow of v is some positive number $g(v) > 0$. Meanwhile some vertices might have flow *deficit*, i.e. $g(v) < 0$. We need to fix this by sending the excess flow to the deficits (and we want to do this in the cheapest way possible). This can be done by, modifying the residual graph G_f to create a new graph G'_f. We do this by defining a new source vertex s and sink vertex t. For all v with $g(v) > 0$, we create an edge (s, v) in G'_f with 0 cost and capacity $g(v)$. For all v with $g(v) < 0$, we create an edge (v, t) with 0 cost and capacity $-g(v)$. Now we wish to find a min cost max flow from s to t in G'_f. Clearly the max flow value saturates all out-edges from s. This is because one valid flow in G'_f is to simply saturate all out-edges from s, in-edges to t, and also add $-f$ (i.e. send flow back on the edges we fully saturated in the first place). Therefore, a max flow in G'_f added to f is indeed a circulation in G. The min cost such max flow then provides an MCC.

4.2 Min cost max flow using SAP

What about unit capacity MCMF but when we aren't promised that there are no negative cost cycles? Recall the reduction from MCMF to MCC in the proof of Claim 3. We can perform this reduction (the result of which still has unit capacities) then run the MCC algorithm from the last subsection.

On problem set 8, problem 4, you will design a scaling algorithm that can handle general capacities in $\{1, \ldots, U\}$. The running time is $O(m(m + n \log n) \log U)$.

Recently Lee and Sidford showed an interior-point based algorithm achieving time $\tilde{O}(m\sqrt{n} \log^2 U)$ (note MCMF can be formulated as an LP). We say $g = \tilde{O}(f)$ when $g = O(f \cdot \log^c f)$ for some constant c. A technically incomparable algorithm (though slower unless U is very large) achieves time $O(nm \log \log U \log(nC))$ by scaling both in terms of capacity and costs [1]. The fastest strongly polynomial algorithm is due to [5] and takes time $O(m \log n(m + n \log n))$. See the encyclopedia article on min cost max flow for more references and history on the problem [2].

References

[1] Ravindra K. Ahuja, Andrew V. Goldberg, James B. Orlin, Robert Endre Tarjan. Finding minimum-cost flows by double scaling. *Math. Program.* 53, 243–266, 1992.

[2] Ravindra K. Ahuja, Thomas L. Magnanti, James B. Orlin. Minimum Cost Flow Problem. *Encyclopedia of Optimization*, pages 2095–2108, 2009.

[3] Andrew Goldberg. Scaling Algorithms for the Shortest Paths Problem. *SIAM J. Comput.*, 24(3), 494-504.

[4] Yin Tat Lee, Aaron Sidford. Following the Path of Least Resistance : An $\tilde{O}(m\sqrt{n})$ Algorithm for the Minimum Cost Flow Problem. *CoRR* abs/1312.6713, 2013.

[5] James B. Orlin. A faster strongly polynomial minimum cost flow algorithm. *Operations Research*, 41(2), 338-350, 1993.