CS261: Problem Set #1

Due by 11:59 PM on Tuesday, January 26, 2016

Instructions:

(1) Form a group of 1-3 students. You should turn in only one write-up for your entire group.

(2) Submission instructions: We are using Gradescope for the homework submissions. Go to www.gradescope.com to either login or create a new account. Use the course code 9B3BEM to register for CS261. Only one group member needs to submit the assignment. When submitting, please remember to add all group member names in Gradescope.

(3) Please type your solutions if possible and we encourage you to use the LaTeX template provided on the course home page.

(4) Write convincingly but not excessively.

(5) Some of these problems are difficult, so your group may not solve them all to completion. In this case, you can write up what you've got (subject to (3), above): partial proofs, lemmas, high-level ideas, counterexamples, and so on.

(6) Except where otherwise noted, you may refer to the course lecture notes *only*. You can also review any relevant materials from your undergraduate algorithms course.

(7) You can discuss the problems verbally at a high level with other groups. And of course, you are encouraged to contact the course staff (via Piazza or office hours) for additional help.

(8) If you discuss solution approaches with anyone outside of your group, you must list their names on the front page of your write-up.

(9) Refer to the course Web page for the late day policy.

Problem 1

This problem explores "path decompositions" of a flow. The input is a flow network (as usual, a directed graph $G = (V, E)$, a source s, a sink t, and a positive integral capacity u_e for each edge), as well as a flow f in G. As always with graphs, m denotes $|E|$ and n denotes $|V|$.

(a) A flow is *acyclic* if the subgraph of directed edges with positive flow contains no directed cycles. Prove that for every flow f, there is an acyclic flow with the same value of f. (In particular, this implies that some maximum flow is acyclic.)

(b) A *path flow* assigns positive values only to the edges of one simple directed path from s to t. Prove that every acyclic flow can be written as the sum of at most m path flows.

(c) Is the Ford-Fulkerson algorithm guaranteed to produce an acyclic maximum flow?

(d) A *cycle flow* assigns positive values only to the edges of one simple directed cycle. Prove that every flow can be written as the sum of at most m path and cycle flows.

(e) Can you compute the decomposition in (d) in $O(mn)$ time?

Problem 2

Consider a directed graph $G = (V, E)$ with source s and sink t for which each edge e has a positive integral capacity u_e. Recall from Lecture #2 that a *blocking flow* in such a network is a flow $\{f_e\}_{e \in E}$ with the property that, for every s-t path P of G, there is at least one edge of P such that $f_e = u_e$. For example, our first (broken) greedy algorithm from Lecture #1 terminates with a blocking flow (which, as we saw, is not necessarily a maximum flow).

Dinic's Algorithm

initialize $f_e = 0$ for all $e \in E$
while there is an s-t path in the current residual network G_f **do**
  construct the layered graph L_f, by computing the residual graph G_f and running breadth-first search (BFS) in G_f starting from s, stopping once the sink t is reached, and retaining only the forward edges[1]
  compute a blocking flow g in G_f
  `// augment the flow` f `using the flow` g
  for all edges (v, w) of G for which the corresponding forward edge of G_f carries flow ($g_{vw} > 0$) **do**
    increase f_e by g_e
  for all edges (v, w) of G for which the corresponding reverse edge of G_f carries flow ($g_{wv} > 0$) **do**
    decrease f_e by g_e

The termination condition implies that the algorithm can only halt with a maximum flow. Exercise Set #1 argues that every iteration of the main loop increases $d(f)$, the length (i.e., number of hops) of a shortest s-t path in G_f, and therefore the algorithm stops after at most n iterations. Its running time is therefore $O(n \cdot BF)$, where BF is the amount of time required to compute a blocking flow in the layered graph L_f. We know that $BF = O(m^2)$ — our first broken greedy algorithm already proves this — but we can do better.

Consider the following algorithm, inspired by depth-first search, for computing a blocking flow in L_f:

A Blocking Flow Algorithm

Initialize. Initialize the flow variables g_e to 0 for all $e \in E$. Initialize the path variable P as the empty path, from s to itself. Go to **Advance.**

Advance. Let v denote the current endpoint of the path P. If there is no edge out of v, go to **Retreat**. Otherwise, append one such edge (v, w) to the path P. If $w \neq t$ then go to **Advance**. If $w = t$ then go to **Augment**.

Retreat. Let v denote the current endpoint of the path P. If $v = s$ then halt. Otherwise, delete v and all of its incident edges from L_f. Remove from P its last edge. Go to **Advance**.

Augment. Let Δ denote the smallest residual capacity of an edge on the path P (which must be an s-t path). Increase g_e by Δ on all edges $e \in P$. Delete newly saturated edges from L_f, and let $e = (v, w)$ denote the first such edge on P. Retain only the subpath of P from s to v. Go to **Advance.**

And now the analysis:

(a) Prove that the running time of the algorithm, suitably implemented, is $O(mn)$. (As always, m denotes $|E|$ and n denotes $|V|$.)

[Hint: How many times can **Retreat** be called? How many times can **Augment** be called? How many times can **Advance** be called before a call to **Retreat** or **Augment**?]

[1] Recall that a forward edge in BFS goes from layer i to layer $(i + 1)$, for some i.

(b) Prove that the algorithm terminates with a blocking flow g in L_f.

[For example, you could argue by contradiction.]

(c) Suppose that every edge of L_f has capacity 1 (cf., Exercise #4). Prove that the algorithm above computes a blocking flow in linear (i.e., $O(m)$) time.

[Hint: can an edge (v, w) be chosen in two different calls to **Advance**?]

Problem 3

In this problem we'll analyze a different augmenting path-based algorithm for the maximum flow problem. Consider a flow network with integral edge capacities. Suppose we modify the Edmonds-Karp algorithm (Lecture #2) so that, instead of choosing a shortest augmenting path in the residual network G_f, it chooses an augmenting path on which it can push the most flow. (That is, it maximizes the minimum residual capacity of an edge in the path.) For example, in the network in Figure 1, this algorithm would push 3 units of flow on the path $s \to v \to w \to t$ in the first iteration. (And 2 units on $s \to w \to v \to t$ in the second iteration.)

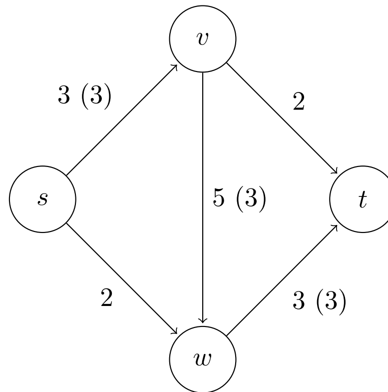

Figure 1: Problem 3. Edges are labeled with their capacities, with flow amounts in parentheses.

(a) Show how to modify Dijkstra's shortest-path algorithm, without affecting its asymptotic running time, so that it computes an s-t path with the maximum-possible minimum residual edge capacity.

(b) Suppose the current flow f has value F and the maximum flow value in G is F^*. Prove that there is an augmenting path in G_f such that every edge has residual capacity at least $(F^* - F)/m$, where $m = |E|$.

[Hint: if Δ is the maximum amount of flow that can be pushed on any s-t path of G_f, consider the set of vertices reachable from s along edges in G_f with residual capacity more than Δ. Relate the residual capacity of this (s, t)-cut to $F^* - F$.]

(c) Prove that this variant of the Edmonds-Karp algorithm terminates within $O(m \log F^*)$ iterations, where F^* is defined as in the previous problem.

[Hint: you might find the inequality $1 - x \leq e^{-x}$ for $x \in [0, 1]$ useful.]

(d) Assume that all edge capacities are integers in $\{1, 2, \ldots, U\}$. Give an upper bound on the running time of your algorithm as a function of $n = |V|$, m, and U. Is this bound polynomial in the input size?

Problem 4

In this problem we'll revisit the special case of *unit-capacity* networks, where every edge has capacity 1 (see also Exercise 4).

(a) Recall the notation $d(f)$ for the length (in hops) of a shortest s-t path in the residual network G_f. Suppose G is a unit-capacity network and f is a flow with value F. Prove that the maximum flow value is at most $F + \frac{m}{d(f)}$.

[Hint: use the layered graph L_f discussed in Problem 2 to identify an s-t cut of the residual graph that has small residual capacity. Then argue along the lines of Problem 3(b).]

(b) Explain how to compute a maximum flow in a unit-capacity network in $O(m^{3/2})$ time.

[Hints: use Dinic's algorithm and Problem 2(c). Also, in light of part (a) of this problem, consider the question: if you know that the value of the current flow f is only c less than the maximum flow value in G, then what's a crude upper bound on the number of additional blocking flows required before you're sure to terminate with a maximum flow?]

Problem 5

(Difficult.) This problem sharpens the analysis of the highest-label push-relabel algorithm (Lecture #3) to improve the running time bound from $O(n^3)$ to $O(n^2\sqrt{m})$.[2] (Replacing an n by a $\sqrt{m}$ is always a good thing.) Recall from the Lecture #3 analysis that it suffices to prove that the number of non-saturating pushes is $O(n^2\sqrt{m})$ (since there are only $O(n^2)$ relabels and $O(nm)$ saturating pushes, anyways).

For convenience, we augment the algorithm with some bookkeeping: each vertex v maintains at most one *successor*, which is a vertex w such that (v, w) has positive residual capacity and $h(v) = h(w)+1$ (i.e., (v, w) goes "downhill"). (If there is no such w, v's successor is NULL.) When a push is called on the vertex v, flow is pushed from v to its successor w. Successors are updated as needed after each saturating push or relabel.[3] For a preflow f and corresponding residual graph G_f, we denote by S_f the subgraph of G_f consisting of the edges $\{(v, w) \in G_f : w \text{ is } v\text{'s successor}\}$.

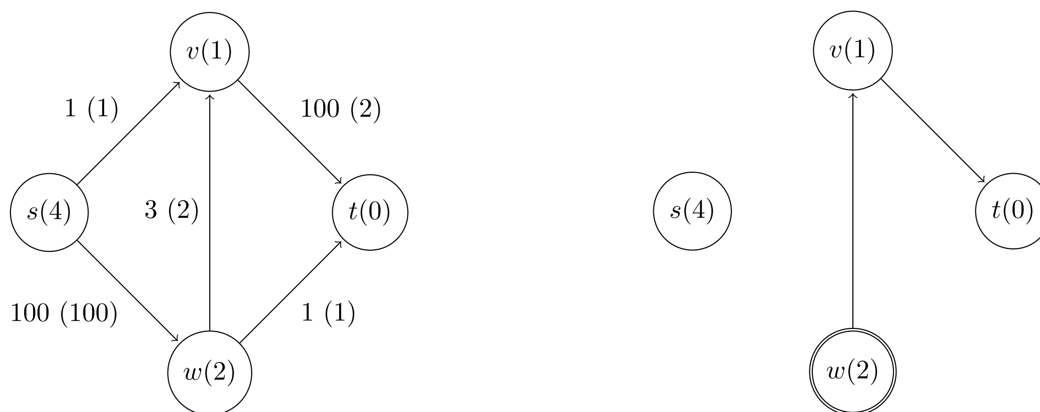

Figure 2: (a) Sample instance of running push-relabel algorithm. As usual, for edges, the flows values are in brackets. For vertices, the bracketed values denote the heights of vertices. (b) S_f for the given preflow in (a). Maximal vertices are denoted by two circles.

(a) Note that every vertex of S_f has out-degree 0 or 1. Prove that S_f is a directed forest, meaning a collection of disjoint directed trees (in each tree, all edges are directed inward toward the root).

(b) Define $D(v)$ as the number of descendants of v in its directed tree (including v itself). Equivalently, $D(v)$ is the number of vertices that can reach v by repeatedly following successor edges. (The $D(v)$'s can change each time the preflow, height function, or successor edges change.)

Prove that the push-relabel algorithm only pushes flow from v to w when $D(w) > D(v)$.

[2]Believe it or not, this is a tight upper bound — the algorithm requires $\Omega(n^2\sqrt{m})$ operations in the worst case.

[3]We leave it as an exercise to think about how to implement this to get an algorithm with overall running time $O(n^2\sqrt{m})$.

(c) Call a vertex with excess *maximal* if none of its descendants have excess. (Every highest vertex with excess is maximal — do you see why? — but the converse need not hold.) For such a vertex, define

$$\phi(v) = \max\{K - D(v) + 1, 0\},$$

where K is a parameter to be chosen in part (i). For the other vertices, define $\phi(v) = 0$. Define

$$\Phi = \sum_{v \in V} \phi(v).$$

Prove that a non-saturating push, from a highest vertex v with positive excess, cannot increase Φ. Moreover, such a push strictly decreases Φ if $D(v) \leq K$.

(d) Prove that changing a vertex's successor from NULL to a non-NULL value cannot increase Φ.

(e) Prove that each relabel increases Φ by at most K.

[Hint: before a relabel at v, v has out-degree 0 in S_f. After the re-label, it has in-degree 0. Can this create new maximal vertices? And how do the different $D(w)$'s change?]

(f) Prove that each saturating push increases Φ by at most K.

(g) A *phase* is a maximal sequence of operations such that the maximum height of a vertex with excess remains unchanged. (The set of such vertices can change.) Prove that there are $O(n^2)$ phases.

(h) Arguing as in Lecture #3 shows that each phase performs at most n non-saturating pushes (why?), but we want to beat the $O(n^3)$ bound. Suppose that a phase performs at least $\frac{2n}{K}$ non-saturating pushes. Show that at least half of these strictly decrease Φ.

[Hint: prove that if a phase does a non-saturating push at both v and w during a phase, then v and w share no descendants during the phase. How many such vertices can there be with more than K descendants?]

(i) Prove a bound of $O(\frac{n^3}{K} + nmK)$ on the total number of non-saturating pushes across all phases. Choose K so that the bound simplifies to $O(n^2\sqrt{m})$.

Problem 6

Suppose we are given an array $A[1..m][1..n]$ of non-negative real numbers. We want to round A to an integer matrix, by replacing each entry x in A with either $\lfloor x \rfloor$ or $\lceil x \rceil$, without changing the sum of entries in any row or column of A. (Assume that all row and column sums of A are integral.) For example:

$$\begin{bmatrix} 1.2 & 3.4 & 2.4 \\ 3.9 & 4 & 2.1 \\ 7.9 & 1.6 & 0.5 \end{bmatrix} \to \begin{bmatrix} 1 & 4 & 2 \\ 4 & 4 & 2 \\ 8 & 1 & 1 \end{bmatrix}$$

(a) Describe and analyze an efficient algorithm that either rounds A in this fashion, or reports correctly that no such rounding is possible.

[Hint: don't solve the problem from scratch, use a reduction instead.]

(b) Prove that such a rounding is guaranteed to exist.