CS261: Problem Set #1

Due by 11:59 PM on Tuesday, April 21, 2015

Instructions:

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

(2) Turn in your solutions by email to `cs261submissions@gmail.com`. Please type your solutions if possible and feel free to use the LaTeX template provided on the course home page.

(3) Write convincingly but not excessively.

(4) 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.

(5) Except where otherwise noted, you may refer to your course notes and the specific supplementary readings listed on the course Web page *only*. You can also review any relevant materials from your undergraduate algorithms course.

(6) 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.

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

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

Problem 1

[ESTIMATED DIFFICULTY LEVEL: Not too bad.] The goal of this problem is to compute a "path decomposition" 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 $|V|$.

The desired output is a collection of s-t paths $P_1, \ldots, P_k$ with nonnegative weights $g_1, \ldots, g_k$ and directed cycles $C_1, \ldots, C_\ell$ with nonnegative weights $h_1, \ldots, h_\ell$ such that:

(P1) For every edge $e \in E$, the total weight on e across all paths and cycles is exactly f_e:

$$\sum_{i\,:\,e \in P_i} g_i + \sum_{j\,:\,e \in C_j} h_j = f_e;$$

and

(P2) the combined number $k + \ell$ of paths and cycles is at most m.

Prove that every flow admits a decomposition with properties (P1) and (P2). Can you compute such a decomposition in $O(mn)$ time?

[Hint: Try a greedy approach, beginning with the paths.]

Problem 2

[ESTIMATED DIFFICULTY LEVEL: A bit difficult.] 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).

Recall also from Lecture #2 Dinic's algorithm:

1. Initialize $f_e = 0$ for every $e \in E$.
2. While there is an s-t path in the residual graph G_f:
   (a) Compute the layered graph L_f, by computing the residual graph G_f and running breadth-first search (BFS) in G_f starting from s, aborting once the sink t is reached, and retaining only the forward edges. (Recall that a forward edge in BFS goes from layer i to layer $(i+1)$, for some i.)
   (b) Compute a blocking flow b in L_f.
   (c) Set $f := f + b$. (You should check that since b is a feasible flow in the residual network G_f, $f + b$ is a feasible flow in the original network G.)
3. Return f.

The termination condition implies that the algorithm can only halt with a maximum flow. We argued in lecture 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:

Initialize. Initialize the flow variables b_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 b_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**?]

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

[For example, you could argue by contradiction.]

(c) Suppose that every edge of L_f has capacity 1. 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

[ESTIMATED DIFFICULTY LEVEL: A bit difficult.] This problem outlines a still-faster algorithm for computing a blocking flow in the layered graph L_f. We use the fact that L_f is a directed acyclic graph. Assume that there is no edge (s, t) in L_f. By the *capacity* of a vertex v, we mean the minimum of the total capacity of its incoming edges and the total capacity of its outgoing edges:

$$u(v) = \min\{\sum_{e \in \delta^-(v)} u_e, \sum_{e \in \delta^+(v)} u_e\}. \tag{1}$$

As usual, $\delta^-(v)$ and $\delta^+(v)$ denote the incoming and outgoing edges of vertex v, respectively.

Here is a sketch of the algorithm:

1. Insert all vertices of L_f other than s and t into a heap, using vertex capacities (1) as keys.
2. Topologically sort the vertices of L_f, so that all edges of L_f go from an earlier vertex to a later vertex in the ordering. (Recall from CS161 that this can always be done for a directed acyclic graph, and it requires only linear time.)
3. Initialize $b_e = 0$ for every $e \in E$.
4. While the heap is non-empty:
   (a) Extract from the heap the vertex v with smallest remaining capacity (in the sense of (1)).
   (b) If v has zero capacity, delete it and all of its incident edges from L_f. Update (i.e., decrease) the remaining capacity of other vertices as appropriate.
   (c) Otherwise, if v has positive capacity Δ:
      i. Push Δ units of flow from v to t. Do this greedily, as follows. First, the outgoing edges of v are saturated in an arbitrary order, leaving at most one partially saturated edge. All saturated edges are deleted from L_f. If there is a partially saturated edge, then its remaining capacity is decreased accordingly. Remaining capacities of vertices are also decreased as appropriate. This operation is then repeated on the first vertex (in the topological ordering) that received flow from this process, and so on all the way to t.
      ii. Analogously, pull Δ units of flow back from v to s, deleting edges and updating remaining capacities appropriately.
      iii. Delete from L_f the vertex v and all of its remaining incident edges.

And now the analysis:

(a) Explain why it is always possible to push Δ units of flow from v to t and to pull Δ units of flow from v back to s.

(b) Give a brief proof that the algorithm terminates with a blocking flow of L_f.

(c) Explain why a suitable implementation of this algorithm runs in $O(n^2 \log n)$ time. Fill in implementation details as needed for your argument. (This gives an $O(n^3 \log n)$-time algorithm for the maximum flow problem.)

[Hints: You might want to review all of the operations supported by a heap. First prove a bound of $O(m \log n)$ for the time needed for all of the operations other than the non-saturating pushes.]

(d) **Optional; do not hand in.** Read about "Fibonacci heaps" — for example, in the CLRS book or on Wikipedia — and explain how to use them to obtain an improved implementation of the algorithm that runs in time $O(n^2)$. (This gives an $O(n^3)$-time algorithm for the maximum flow problem.)

Problem 4

[ESTIMATED DIFFICULTY LEVEL: A bit difficult.] In this problem we'll analyze a different augmenting path-based algorithm for the maximum flow problem, which is based on the idea of *scaling*. In contrast to the Edmonds-Karp algorithm of Lecture #2, rather than looking for short augmenting paths, we look for augmenting paths along which it's possible to send a lot of flow.

Here is the algorithm (as usual, assume that every capacity u_e is a positive integer):

1. Initialize $f_e = 0$ for every edge e.
2. Let Δ be the smallest power of 2 that is at least $\max_{e \in E} u_e$.
   (a) While $\Delta \geq 1$:
      i. While there exists an s-t path P in G_f such that every edge of P has residual capacity at least Δ:
         A. Pick an arbitrary such path P.
         B. Let Δ_P denote the minimum residual capacity of an edge of P.
         C. Augment the flow f by sending Δ_P units of flow on the path P.
      ii. Divide Δ by 2.
3. Return f.

Here is the analysis:

(a) Give a linear-time algorithm that, given G_f and Δ, either computes an s-t path P of G_f such that every edge of P has residual capacity at least Δ, or determines that no such path exists.

(b) Argue that the algorithm in the problem statement terminates with a maximum flow.

(c) Prove that at the conclusion of an iteration of the outer while loop with parameter Δ, the value of the current flow f is within $m\Delta$ of the value of a maximum flow.

[Hint: generalize the proof of correctness of the Ford-Fulkerson algorithm given in Lecture #2.]

(d) Argue that the inner while loop executes $O(m)$ times for each value of Δ.

(e) Conclude that the algorithm in the problem statement solves the maximum flow problem correctly in $O(m^2 \log U)$ time, where $U = \max_{e \in E} u_e$. (Since the running time is logarithmic in U and hence polynomial in the description length of the edge capacities, this is a polynomial-time algorithm. There are some parameter values, such as when $m, U = \Theta(n)$, where this algorithm is faster than any of the other ones that we've studied.)

Problem 5

[ESTIMATED DIFFICULTY LEVEL: Somewhat difficult.] 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 Problems 2 and 3 to identify an s-t cut of the residual graph that has small residual capacity. Then argue as in Problem 4(c).]

(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 6

[ESTIMATED DIFFICULTY LEVEL: Somewhat difficult.] The goal of this problem is to suggest variants of the Push-Relabel algorithm that speed up the practical running time without ruining its worst-case complexity. Recall that the algorithm maintains the invariant (called INV in Lecture #3) that $h(v) \leq h(w)+1$ for all edges (v, w) in the residual graph of the current preflow. We proved in Lecture #3 that if f is a flow (not just a preflow) with this invariant, then it is a maximum flow. Heights were monotone increasing, and the running-time analysis depended on bounding the number of times vertices can increase their heights. Practical experience shows that the algorithm is almost always much faster than suggested by the worst case, and that the practical bottleneck of the algorithm is relabeling vertices (and not the non-saturating pushes that govern the worst-case theoretical bound). The goal of the problems below is to decrease the number of relabelings by increasing heights more aggressively. As usual, assume you have a directed graph G with n vertices, m edges, capacities $\{u_e\}_{e \in E}$, source s, and sink t.

(a) The Push-Relabel algorithm of Lecture #3 starts by setting $f_e = u_e$ on all edges e leaving the source, $f_e = 0$ on all other edges, $h(s) = n$, and $h(v) = 0$ for all other vertices v. Give an $O(m)$-time procedure for initializing vertex heights that is better than the one we constructed in lecture. Your method should set the height of each vertex v to be as high as possible given the initial preflow (subject to INV).

(b) In this part we add a new step, called *gap relabeling*, to the Push-Relabel algorithm. This step will increase the labels of lots of nodes by a potentially large amount. Consider a preflow f and heights h such that INV holds. A *gap* in the heights is an integer $0 < h < n$ so that no vertex has height exactly h. Assume that h is a gap value, and let A be the set of nodes v with heights satisfying $n > h(v) > h$. Gap relabeling is the process of changing the heights of all vertices in A so that they are equal to n. Prove that gap relabeling maintains the invariant INV (and hence this more aggressive Push-Relabel algorithm still terminates with a flow guaranteed to be maximum).

(c) In lecture we proved that $h(v) \leq 2n - 1$ throughout the algorithm. Here we will have a variant that has $h(v) \leq n$ throughout. The idea is that we "freeze" all vertices when they get to height n; that is, vertices at height n are no longer considered eligible for Push and Relabel operations, even if they have positive excess. With this modification, at the end of the algorithm, we have a preflow and height function that satisfy INV, and such that all excess is at vertices with height n. Let B be the set of nodes v so that there is a path from v to t in the residual graph of the current preflow. Let $A = V \setminus B$. Prove that at the end of the algorithm, (A, B) is a minimum-capacity s-t cut.

(d) The algorithm in part (c) computes a minimum s-t cut but fails to find a maximum flow (as it ends with a preflow with non-zero excesses). Give an algorithm that takes the preflow f at the end of the algorithm of part (c) and converts it into a maximum flow in at most $O(mn)$ time.

[Hint: consider vertices with excess, and try to send the excess back to s using only reverse edges in the residual graph. A lemma from Lecture #3 might be useful.]