CS261: Problem Set #1*

Due by 11:59 PM on Tue, Jan 28, 2020

Instructions:

(1) We encourage you to pair up with another student in working on this homework. You should turn in only one write-up for your group.

(2) Submission instructions: We will use Gradescope for the homework submissions. Go to www.gradescope.com to either login or create a new account. Use the course code 9GZK2R 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) You must use LaTeX, LyX, Microsoft Word, or a similar editor to typeset your write-up.

(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 materials listed on the course web page *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 Edmonds-Karp algorithm guaranteed to produce an acyclic maximum flow? Prove or show a counterexample.

(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?

*We thank Tim Roughgarden for permission to include problems from his Winter 2016 edition of CS 261.

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 L_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. 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. (This was actually suggested by a student in class.) 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.)

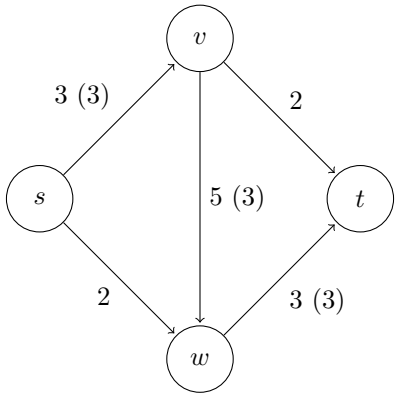

Figure 1: Problem 3. Edges are labeled with their capacities, with flow amounts in parentheses.

(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 $1 - x \leq e^{-x}$ 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

Suppose we are given an array $A[1..m][1..n]$ of non-negative real numbers, such that all row and column sums of A are integral. 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. For example:

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

(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.

Problem 5

A *vertex cover* of an undirected graph (V, E) is a subset $S \subseteq V$ such that, for every edge $e \in E$, at least one of e's endpoints lies in S.[2]

(a) Prove that in every graph, the minimum size of a vertex cover is at least the size of a maximum matching.

(b) Give a non-bipartite graph in which the minimum size of a vertex cover is strictly bigger than the size of a maximum matching.

(c) Prove that the problem of computing a minimum-cardinality vertex cover can be solved in polynomial time in bipartite graphs.[3]

[Hint: reduction to maximum flow.]

(d) Prove that in every bipartite graph, the minimum size of a vertex cover equals the size of a maximum matching.

Problem 6

In the *minimum-cost flow problem*, the input is a directed graph $G = (V, E)$, a source $s \in V$, a sink $t \in V$, a target flow value d, and a capacity $u_e \geq 0$ and cost $c_e \in \mathbb{R}$ for each edge $e \in E$. The goal is to compute a flow $\{f_e\}_{e \in E}$ sending d units from s to t with the minimum-possible cost $\sum_{e \in E} c_e f_e$. (If there is no such flow, the algorithm should correctly report this fact.)

Given a min-cost flow instance and a feasible flow f with value d, the corresponding *residual network* G_f is defined as follows. The vertex set remains V. For every edge $(v, w) \in E$ with $f_{vw} < u_{vw}$, there is an edge (v, w) in G_f with cost c_e and residual capacity $u_e - f_e$. For every edge $(v, w) \in E$ with $f_{vw} > 0$, there is a reverse edge (w, v) in G_f with the cost $-c_e$ and residual capacity f_e.

A *negative cycle* of G_f is a directed cycle C of G_f such that the sum of the edge costs in C is negative. (E.g., $v \rightarrow w \rightarrow x \rightarrow y \rightarrow v$, with $c_{vw} = 2$, $c_{wx} = -1$, $c_{xy} = 3$, and $c_{yv} = -5$.)

(a) Prove that if the residual network G_f of a flow f has a negative cycle, then f is not a minimum-cost flow.

(b) Prove that if the residual network G_f of a flow f has no negative cycles, then f is a minimum-cost flow.

[Hint: look to the proof of the minimum-cost bipartite matching optimality conditions for inspiration.]

[2] Yes, the problem is confusingly named.

[3] In general graphs, the problem turns out to be NP-hard (you don't need to prove this).

(c) Give a polynomial-time algorithm that, given a residual network G_f, either returns a negative cycle or correctly reports that no negative cycle exists.

[Hint: feel free to use an algorithm from CS161. Be clear about which properties of the algorithm you're using.]

(d) Assume that all edge costs and capacities are integers with magnitude at most M. Give an algorithm that is guaranteed to terminate with a minimum-cost flow and has running time polynomial in $n = |V|$, $m = |E|$, and M.[4]

[Hint: what would the analog of Ford-Fulkerson be?]

[4] Thus this algorithm is only "pseudo-polynomial." A polynomial algorithm would run in time polynomial in n, m, and $\log M$. Such algorithms can be derived for the minimum-cost flow problem using additional ideas.