CS261: Exercise Set #2

For the week of April 6–10, 2015

Instructions:

(1) *Do not turn anything in.*

(2) The course staff is happy to discuss the solutions of these exercises with you in office hours or on Piazza.

(3) While these exercises are certainly not trivial, you should be able to complete them on your own (perhaps after consulting with the course staff or a friend for hints).

Exercise 6

In the *s-t directed edge-disjoint paths problem*, the input is a directed graph $G = (V, E)$, a source vertex s, and a sink vertex t. The goal is to output a maximum-cardinality set of edge-disjoint s-t paths $P_1, \ldots, P_k$. (I.e., P_i and P_j should share no edges for each $i \neq j$, and k should be as large as possible.)

Prove that this problem reduces to the maximum flow problem. That is, given an instance of the disjoint paths problem, show how to (i) produce an instance of the maximum flow problem such that (ii) given a maximum flow to this instance, you can compute an optimal solution to the disjoint paths instance. Your implementations of steps (i) and (ii) should run in linear and polynomial time, respectively. (Can you achieve linear time also for (ii)?) Include a brief proof of correctness.

[Hint: for (ii), make use of your solution to Problem 1 (from Problem Set #1).]

Exercise 7

In the *s-t directed vertex-disjoint paths problem*, the input is a directed graph $G = (V, E)$, a source vertex s, and a sink vertex t. The goal is to output a maximum-cardinality set of internally vertex-disjoint s-t paths $P_1, \ldots, P_k$. (I.e., P_i and P_j should share no vertices other than s and t for each $i \neq j$, and k should be as large as possible.) Give a polynomial-time algorithm for this problem.

[Hint: reduce the problem either directly to the maximum flow problem or to the edge-disjoint version solved in the previous exercise.]

Exercise 8

In the *(undirected) global minimum cut problem*, the input is an undirected graph $G = (V, E)$ with a nonnegative capacity u_e for each edge $e \in E$, and the goal is to identify a cut (A, B) — i.e., a partition of V into non-empty sets A and B — that minimizes the total capacity $\sum_{e \in \delta(S)} u_e$ of the cut edges. (Here, $\delta(A)$ denotes the edges with exactly one endpoint in A.)

Prove that this problem reduces to solving $n-1$ maximum flow problems in undirected graphs.[1] That is, given an instance the global minimum cut problem, show how to (i) produce $n-1$ instances of the maximum flow problem (in undirected graphs) such that (ii) given maximum flows to these $n-1$ instances, you can compute an optimal solution to the global minimum cut instance. Your implementations of steps (i) and (ii) should run in polynomial time. Include a brief proof of correctness.

[1] And hence, by Exercise 2, to solving $n-1$ maximum flow problems in directed graphs.

Exercise 9

In the *(unweighted) bipartite matching problem*, the input is an undirected bipartite graph $G = (A, B, E)$. That is, the vertex set is $A \cup B$, and every edge has one endpoint in each of A and B. A *matching* is a subset $M \subseteq E$ of edges such that no pair of edges shares an endpoint. The goal is to output a maximum-cardinality matching of G.

Prove that the bipartite matching problem reduces to the maximum flow problem. That is, given an instance of bipartite matching, show how to (i) produce an instance of the maximum flow problem such that (ii) given a maximum flow to this instance, you can recover a maximum-cardinality matching of the bipartite matching instance. Your implementations of steps (i) and (ii) should run in linear time. Include a brief proof of correctness.

[Hint: the solution is similar to that of Exercise 1, but with a different setting of the edge capacities.]

Exercise 10

In lecture we proved a bound of $O(n^3)$ on the number of operations needed by the Push-Relabel algorithm (where each iteration, we select the highest vertex with excess to Push or Relabel) before it terminates with a maximum flow. Give an implementation of this algorithm that runs in $O(n^3)$ time.

[Hints: first prove the running time bound assuming that, in each iteration, you can identify the highest vertex with positive excess in $O(1)$ time. The hard part is to maintain the vertices with positive excess in a data structure such that, summed over all of the iterations of the algorithm, only $O(n^3)$ total time is used to identify these vertices. Can you get away with just a collection of buckets (implemented as lists), sorted by height?]