Analysis of Algorithms	**Problem Set 3**
CS 6820 Fall 2014	**Due 5pm, Friday, October 10, 2014**

This problem set has 4 problems with parts of varying difficulty. I have assigned points to each part, with a maximum possible total of 110. For full credit for a grade of A, solve enough parts to collect at least 80 points (OK to solve the easier parts of each problem). A full solution for each problem includes proving that your answer is correct. If your group cannot solve a problem, but can do some parts, or have partial results, write down how far you got, and why are you stuck.

Students may work on homework in groups of up to 2-3 people. Each group may turn in a single solution set that applies to all members of the group. However, students are asked to understand each of their group's solutions well enough to give an impromptu whiteboard presentation of the solution. You may use any fact we proved in class without proving the proof or reference, and may read the relevant chapters of the Kleinberg-Tardos or Kozen books, provided you state them clearly. However, you may **not use other published papers, or the Web to find your answer**.

Solutions can be submitted on CMS in pdf format (only). Please type your solution or write extremely neatly to make it easy to read. If your solution is complex, say more than about half a page, please include a 3-line summary to help us understand the argument.

Please ask any clarifying questions using Piazza, where we will post all answers.

(1) *(30 points)* We defined flow in class using a formulation using flow conservation. An alternate way to define flow is to think about path from s to t, and how much flow they each carry. This problem will show that the two versions are essentially equivalent. Here is how that version would be defined. A path P from s to t can carry flow from s to t. Let $\mathcal{P}$ denote the set of all s-t paths in G. For this problem, we define a *path-flow* by a value $f_P \geq 0$ for each path $P \in \mathcal{P}$, which we think of as the amount of flow the path carries. Now the total value of the flow is

$$v(f) = \sum_{P \in \mathcal{P}} f_P,$$

and the capacity constraint is written as

$$\sum_{P \in \mathcal{P}: e \in P} f_P \leq c_e \text{ for all } e \in E.$$

The problem then is to find a path-flow of values $f_P \geq 0$ for $P \in \mathcal{P}$ that satisfy the capacity constraint and maximize the value $v(f)$.

(a) *(3 points)* Show that for any path-flow f, the following

$$f'(e) = \sum_{P \in \mathcal{P}, e \in P} f_P$$

defines a (traditional) flow of equal value in G.

(b) *(12 points)* Show that for any (traditional) flow f' there is a path-flow f of equal value ($v(f) = v(f')$) such that the flow on each edge e in the f flow is no higher than the flow in f', that is, $\sum_{P \in \mathcal{P}: e \in P} f_P \leq f'(e)$. Is this also true with equal instead of the inequality?

(c) *(7 points)* One trouble with path-flows is that there can be exponentially many s-t paths in G. Given a flow f' give a polynomial time algorithm to create a path-flow f as defined in (b), where your flow uses only polynomial many paths to carry the flow. Your algorithm should output the list of paths P with their value $f_P > 0$, and implicitly define $f_P = 0$ for all paths not on the list.

(d) *(8 points)* Make the algorithm for (c) run in $O(mn)$ time, where m and n denote the number of edges and nodes respectively.

(2) *(20 points)* In class we talked about the preflow/push max-flow algorithm. Here we consider a version that stops early yet finds the min-cut. One of the invariants of the algorithm was that there is no path from s to t in the residual graph, this means that the cut $B = \{v :$ there is a path from v to t in the residual graph$\}$, and $A = V \setminus B$, forms an (s, t) cut, with $s \in A$ and $t \in B$.

(a) *(5 points)* Show that the capacity of the (A, B) cut, $\sum_{e \in (A,B)} c_e$ is always equal to $\sum_{v \in B} e_f(v)$, that is, its the flow delivered to t, plus the amount of excess on the B side of the cut.

(b) *(10 points)* Consider running the maxflow algorithm with limiting all nodes to height at most $n = |V|$. So this version of the algorithm terminates when all nodes with excess have height n. Clearly, at this point, we may not have a valid flow. However, prove that the cut (A, B) defined above is the min-capacity (s, t) cut in the graph.

(c) *(5 points)* Consider the state of the preflow/push algorithm from part (b). Note that the remaining of the algorithm will take the excess at the nodes (all of height n), and push them back to the source. This can be done much faster not using preflow/push. Use part (d) of the previous question to solve this in $O(mn)$ time, where m and n denote the number of edges and nodes respectively. (OK to use (1d), even if you didn't solve the (d) part of the previous question).

(3) *(40 points)* Many of the most powerful applications of flow algorithms are for finding (s, t) cuts, and not flows. In the application we have seen, a serious limitation is that there are only two terminals s and t (which limited us to forground/background segmentation). Here we consider a three terminal version of the problem. You are given an undirected graph $G = (V, E)$ with costs/capacities c_e on the edges, and three nodes $s_1, s_2, s_3 \in V$. A *3-way cut* is a partition of the nodes into three sets (A_1, A_2, A_3) such that $s_i \in A_i$. We say that an edge $e = (v, w)$ is cut by the partition if v and w are not in the same parts of the partition. We will consider two versions of the problem.

(a) *(8 points)* As a useful first step consider a regular min-cut problem with only two terminals s and t. Show that there is an (s,t) min-cut (A,B) that is minimal in the sense that all other (s,t) min-cuts (A',B') satisfy $A \subset A'$. Give a polynomial time algorithm to find such a cut (A,B).

(b) *(8 points)* Suppose we consider the following algorithm for finding a 3-way partition: for each terminal s_i, consider the max-flow problem with s_i as the source, and the two other terminals contracted to a single node t_i as the sink, and let (A_i,B_i) the minimal min-cut in the graph in the sense of (a). Show that the sets A_i must be disjoint. Hint: it seems useful to consider the cuts defined by sets $A_i \setminus A_j$ and $A_j \setminus A_i$ and consider how the the capacities of the pairs of cuts $c(A_i)+c(A_j)$ and $c(A_i \setminus A_j)+c(A_j \setminus A_i)$ are related.

(c) *(8 points)* Consider the following algorithm for finding a low-cost 3-way cut: Find the sets A_i as given in part (a). Output the 3-way cut where A_i is the part assigned to terminal s_i. As defined, this is not a 3-way cut, as there may be nodes in $V \setminus \cup_i A_i$. Show that the total cost of this partition is at most twice the cost of any 3-way cut. (Note that finding the exact minimum capacity 3-way cut is NP-hard, so its interesting to resort to approximations.)

(d) *(6 points)* Note that you can turn this into a 3-way cut, we propose that you assign these nodes to any of the three parts (all of them to the same part). Show that this doesn't increasing the cost of the partition, and in fact, can improve the guarantee claimed by part (c).

(e) *(10 points)* Can you extend parts (a-c) to multiway cuts with k terminals (and not only 3).

(4) *(20 points)* The linear programming (LP) containment problem is the following decision problem. We are given matrices A_0 and A_1 and vectors b_0 and b_1, and we must decide if the relation

$$\{x : A_0 x \leq b_0\} \subseteq \{x : A_1 x \leq b_1\}$$

holds. You should assume that A_0 and b_0 have the same number of rows, that A_1 and b_1 also have the same number of rows, and that A_0 and A_1 have the same number of columns, as these assumptions are needed in order for the notation in the problem definition to even make sense. You may also assume that the entries of A_0;A_1; b_0; b_1 are integers. Give a polynomial-time algorithm to solve the LP containment problem. You can assume that your algorithm has access to a subroutine that solves linear programs of size N in time $t(N)$, where $t(N)$ is polynomial in N. When analyzing the running time of your own algorithm, you may express it in terms of $t(N)$. In other words, don't worry about the precise running time of the subroutine to solve linear programs, just worry about how many times your algorithm calls the subroutine, and how much time it spends on other operations.