CSE 521: Design and Analysis of Algorithms — Winter 2006
Problem Set #3 — Instructor: Venkatesan Guruswami
Due on **February 9, 2006** (Thursday) in class.

Instructions: Same as for Problem Set 1.

Note that this problem set has a shorter deadline than usual. All the more reason to begin work on the problem set early!

Readings: Kleinberg and Tardos, Chapter 7.

Each problem is worth 10 points unless noted otherwise.

1. Chapter 7, Problem 27 (Fair carpooling)

   Some of your friends with jobs out West decide they really need some extra time each day to sit in front of their laptops, and the morning commute from Woodside to Palo Alto seems like the only option. So they decide to carpool to work.

   Unfortunately, they all hate to drive, so they want to make sure that any carpool arrangement they agree upon is fair, and doesn't overload any individual with too much driving. Some sort of simple round-robin scheme is out, because none of them goes to work every day, and so the subset of them in the car varies from day to day.

   Here's one way to define *fairness*. Let the people be labeled $S = \{p_1, \ldots, p_k\}$. We say that the *total driving obligation* of p_j over a set of days is the expected number of times that p_j would have driven, had a driver been chosen uniformly at random from among the people going to work each day. More concretely, suppose the carpool plan lasts for d days, and on the i^{th} day a subset $S_i \subseteq S$ of the people go to work. Then the above definition of the total driving obligation Δ_j for p_j can be written as $\Delta_j = \sum_{i:p_j \in S_i} \frac{1}{|S_i|}$. Ideally, we'd like to require that p_j drives at most Δ_j times; unfortunately, Δ_j may not be an integer.

   So let's say that a *driving schedule* is a choice of a driver for each day — i.e. a sequence $p_{i_1}, p_{i_2}, \ldots, p_{i_d}$ with $p_{i_t} \in S_t$ — and that a *fair driving schedule* is one in which each p_j is chosen as the driver on at most $\lceil \Delta_j \rceil$ days.

   (a) Prove that for any sequence of sets $S_1, \ldots, S_d$, there exists a fair driving schedule.

   (b) Give an algorithm to compute a fair driving schedule with running time polynomial in k and d.

   (c) One could expect k to be a much smaller parameter than d (e.g. perhaps $k = 5$ and $d = 365$). So it could be worth reducing the dependence of the running time on d even at the expense of a much worse dependence on k. Give an algorithm to compute a fair driving schedule whose running time has the form $O(f(k) \cdot d)$, where $f(\cdot)$ can be an arbitrary function.

2. Chapter 7, Problem 17 (Diagnosing links causing a network failure)

   You've been called in to help some network administrators diagnose the extent of a failure in their network. The network is designed to carry traffic from a designated source node s to a designated target node t, so we will model it as a directed graph $G = (V, E)$, in which the capacity of each edge is 1, and in which each node lies on at least one path from s to t.

Now, when everything is running smoothly in the network, the maximum s-t flow in G has value k. However, the current situation - and the reason you're here - is that an attacker has destroyed some of the edges in the network, so that there is now no path from s to t using the remaining (surviving) edges. For reasons that we won't go into here, they believe the attacker has destroyed only k edges, the minimum number needed to separate s from t (i.e. the size of a minimum s-t cut); and we'll assume they're correct in believing this.

The network administrators are running a monitoring pool on node s, which has the following behavior: if you issue the command $ping(v)$, for a given node v, it will tell you whether there is currently a path from s to v. (So $pint(t)$ reports that no path currently exists; on the other hand, $ping(s)$ always reports a path from s to itself.) Since it's not practical to go out and inspect every edge of the network, they'd like to determine the extent of the failure using this monitoring tool, through judicious use of the *ping* command.

So here's the problem you face: give an algorithm that issues a sequence of ping commands to various nodes in the network, and then reports the *full* set of nodes that are not currently reachable from s. You could do this by pinging every node in the network, of course, but you'd like to do it using many fewer pings (given the assumption that only k edges have been deleted). In issuing this sequence, your algorithm is allowed to decide which node to ping next based on the outcome of earlier ping operations.

Give an algorithm that accomplishes this task using only $O(k \log n)$ pings.

3. Let $G = (V, E)$ be an undirected graph. For a subset S of vertices, denote by $E(S)$ the set of edges with both vertices in S, i.e., $E(S) = \{(u, v) \in E \mid u, v \in S\}$. Now define the density of S to be

$$\rho(S) \stackrel{\text{def}}{=} \frac{|E(S)|}{|S|} .$$

If the presence between an edge between a pair of vertices is an indication of friendship or similarity, then the density of a subset is one measure of how "tightly-knit" that subset is or its quality as a single "cluster". Finding dense clusters therefore arises as a natural goal in a few settings.

   (a) For a positive rational number α, give a polynomial time algorithm that can find a subset S that maximizes $|E(S)| - \alpha|S|$.
   (Hint: Set up a flow network such that the minimum cut reveals the set S that maximizes $|E(S)| - \alpha|S|$; specifically, other than the source and sink, the minimum cut has S on one side and $V \setminus S$ on the other side for some S maximizing $|E(S)| - \alpha|S|$.)

   (b) Give an algorithm to compute a subset $S \subseteq V$ with maximum density $\rho(S)$ using $O(\log n)$ maximum flow computations, and therefore in polynomial time.

4. In this exercise, the goal is to develop a method to find augmenting paths that always computes the maximum flow using a number of augmentations that is independent of the edge capacities and depends only on the number of vertices and edges in the graph. This yields what is called a *strongly polynomial time* algorithm for computing maximum flow. The method is a very natural one: at each step, we perform a BFS on the residual graph from the source s till we reach t (if we never reach t, we stop the algorithm and output the flow), and augment flow along the s-t path in the BFS tree. In other words, we augment flow along the shortest path possible in each iteration. Let us call such an augmentation the BFS-augmentation.

(a) Let f be a s-t flow on a network $G = (V, E)$ with source s and sink t. Suppose we perform a BFS-augmentation on f to compute a new flow f'. Prove that, for each $v \in V - \{s, t\}$, the shortest path distance $d_{f'}(s, v)$ from s to v in the residual graph $G_{f'}$ is not smaller than the shortest path distance $d_f(s, v)$ from s to v in G_f. (Here, we measure distance in residual graphs in terms of the number of hops, i.e., each edge in the residual graph has unit distance.)

(b) Suppose we run the Ford-Fulkerson algorithm performing a BFS-augmentation at each augmentation step. Let $u, v \in V - \{s, t\}$. Prove that if the edge (u, v) is the bottleneck link on the chosen augmenting path in G_f when augmenting flow f and it is again, at a later point, the bottleneck link when augmenting flow f', then $d_{f'}(s, u) \geq d_f(s, u) + 2$.

(c) Prove that the version of the Ford-Fulkerson algorithm where we perform a BFS-augmentation at each step terminates and outputs a maximum flow after only $O(|V||E|)$ augmentations, independent of the edge capacities.