CSE 521: Design and Analysis of Algorithms
Assignment #5
February 2, 2005
Due: Wednesday, February 9

Reading Assignment: Kleinberg and Tardos, Network Flow, handout on linear programming

Problems:

1. Your friends have written a very fast piece of maximum flow code based on repeatedly finding augmenting paths as in Section 6.2. However, after you've looked at a bit of output from it, you realize that it's not always finding a flow of *maximum* value. The bug turns out to be pretty easy to find; your friends hadn't really gotten into the whole backward-edge thing when writing the code, and so their implementation builds a variant of the residual graph that *only includes the forward edges*. In other words, it searches for s-t paths in a graph $\tilde{G}_f$ consisting only of edges e for which $f(e) < c_e$, and it terminates when there is no augmenting path consisting entirely of such edges. We'll call this the "forward-edge-only" flow algorithm. (Note that we do not try to prescribe how this algorithm chooses its forward-edge paths; it may choose them in any fashion it wants, provided that it only terminates when no forward-edge path exists any more.)

   It's hard to convince your friends they need to re-implement the code; in addition to its blazing speed, they claim in fact that it never returns a flow whose value is less than a fixed fraction of optimal. Do you believe this? The crux of their claim can be made precise in the following statement.

   > *There is an absolute constant $b > 1$ (independent of the particular input flow network), so that on every instance of the maximum flow problem, the forward-edge-only algorithm is guaranteed to find a flow of value at least $1/b$ times the maximum flow value (regardless of how it chooses its forward-edge paths).*

   Decide whether you think this statement is true or false, and give a proof of either the statement or its negation.

2. Suppose you and your friend Alanis live, together with $n - 2$ other people, at a popular off-campus co-operative apartment, The Upson Collective. Over the

next n nights, each of you is supposed to cook dinner for the co-op exactly once, so that someone cooks on each of the nights.

Of course, everyone has scheduling conflicts with some of the nights (e.g. prelims, concerts, etc.) — so deciding who should cook on which night becomes a tricky task. For concreteness, let's label the people

$$\{p_1, \ldots, p_n\},$$

the nights

$$\{d_1, \ldots, d_n\};$$

and for person p_i, there's a set of nights $S_i \subset \{d_1, \ldots, d_n\}$ when they are *not* able to cook.

A *feasible dinner schedule* is an assignment of each person in the co-op to a different night, so that each person cooks on exactly one night, there is someone cooking on each night, and if p_i cooks on night d_j, then $d_j \notin S_i$.

(a) Describe a bipartite graph G so that G has a perfect matching if and only if there is a feasible dinner schedule for the co-op.

(b) Anyway, your friend Alanis takes on the task of trying to construct a feasible dinner schedule. After great effort, she constructs what she claims is a feasible schedule, and heads off to class for the day.

Unfortunately, when you look at the schedule she created, you notice a big problem. $n - 2$ of the people at the co-op are assigned to different nights on which they are available: no problem there. But for the other two people — p_i and p_j — and the other two days — d_k and d_ℓ — you discover that she has accidentally assigned both p_i and p_j to cook on night d_k, and assigned no one to cook on night d_ℓ.

You want to fix Alanis's mistake, but without having to re-compute everything from scratch. Show that it's possible, using her "almost correct" schedule, to decide in only $O(n^2)$ time whether there exists a feasible dinner schedule for the co-op. (If one exists, you should also output it.)

3. Suppose you're looking at a flow network G with source s and sink t, and you want to be able to express something like the following intuitive notion: some nodes are clearly on the "source side" of the main bottlenecks; some nodes are clearly on the "sink side" of the main bottlenecks; and some nodes are in the middle. However, G can have many minimum cuts, so we have to be careful in how we try making this idea precise.

   Here's one way to divide the nodes of G into three categories of this sort.

- We say a node v is *upstream* if for all minimum s-t cuts (A, B), we have $v \in A$ — that is, v lies on the source side of every minimum cut.
- We say a node v is *downstream* if for all minimum s-t cuts (A, B), we have $v \in B$ — that is, v lies on the sink side of every minimum cut.
- We say a node v is *central* if it is neither upstream nor downstream; there is at least one minimum s-t cut (A, B) for which $v \in A$, and at least one minimum s-t cut (A', B') for which $v \in B'$.

Give an algorithm that takes a flow network G, and classifies each of its nodes as being upstream, downstream, or central. The running time of your algorithm should be within a constant factor of the time required to compute a *single* maximum flow.

4. Some of your friends have recently graduated and started a small company called WebExodus, which they are currently running out of their parents' garages in Santa Clara. They're in the process of porting all their software from an old system to a new, revved-up system; and they're facing the following problem.

   They have a collection of n software applications, $\{1, 2, \ldots, n\}$, running on their old system; and they'd like to port some of these to the new system. If they move application i to the new system, they expect a net (monetary) benefit of $b_i \geq 0$. The different software applications interact with one another; if applications i and j have extensive interaction, then the company will incur an expense if they move one of i or j to the new system but not both — let's denote this expense by $x_{ij} \geq 0$.

   So if the situation were really this simple, your friends would just port all n applications, achieving a total benefit of $\sum_i b_i$. Unfortunately, there's a problem ...

   Due to small but fundamental incompatibilities between the two systems, there's no way to port application 1 to the new system; it will have to remain on the old system. Nevertheless, it might still pay off to port some of the other applications, accruing the associated benefit and incurring the expense of the interaction between applications on different systems.

   So this is the question they pose to you: which of the remaining applications, if any, should be moved? Give a polynomial-time algorithm to find a set $S \subseteq \{2, 3, \ldots, n\}$ for which the sum of the benefits minus the expenses of moving the applications in S to the new system is maximized.