CSE 521: Design and Analysis of Algorithms
Assignment #4
January 27, 2005
Due: Wednesday, February 2

Reading Assignment: Kleinberg and Tardos, Network Flow, handout on linear programming, section 11.6 in new book.

Problems:

1. Let $G = (V, E)$ be a graph with n nodes in which each pair of nodes is joined by an edge. There is a positive weight w_{ij} on each edge (i, j); and we will assume these weights satisfy the *triangle inequality* $w_{ik} \leq w_{ij} + w_{jk}$. For a subset $V' \subseteq V$, we will use $G[V']$ to denote the subgraph (with edge weights) induced on the nodes in V'.

   We are given a set $X \subseteq V$ of k *terminals* that must be connected by edges. We say that a *Steiner tree* on X is a set Z so that $X \subseteq Z \subseteq V$, together with a sub-tree T of $G[Z]$. The *weight* of the Steiner tree is the weight of the tree T.

   Show that there is function $f(\cdot)$ and a *polynomial function* $p(\cdot)$ so that the problem of finding a minimum-weight Steiner tree on X can be solved in time $O(f(k) \cdot p(n))$.

   Hint: Here is Yiannis's hint from last year. "In case you're wondering if a Steiner tree can contain vertices not in X, consider the following example.

   "V_1, V_2, $V_3 \in X$, form a triangle connected with edges of weight 7.

   "Consider a vertex V_4 in the middle, not in X, connected with the other 3 using edges of weight 4.

   "Observe the triangle inequality holds.

   "The Steiner tree of minimum weight would be to select the middle edges and not two of the triangle.

   "In fact, since your graph needs to satisfy the triangle inequality, a good idea is to think of edge weights as the distance between the vertices. Therefore, drawing examples on scratch paper could be quite informative and give you ideas about a solution..."

Additional hint: You should think about minimum spanning trees. And as with all dynamic programming problems, you should build up the bigger solution by solving smaller problems.

2. The problem of searching for cycles in graphs arises naturally in financial trading applications. Consider a firm trades shares in n different companies. For each pair $i \neq j$ they maintain a trade ratio r_{ij} meaning that one share of i trades for r_{ij} shares of j. Here we allow the rate r to be fractional, i.e., $r_{ij} = \frac{2}{3}$ means that you can trade 3 shares of i to get a 2 shares of j.

   A *trading cycle* for a sequence of shares $i_1, i_2, \ldots, i_k$ consists of successively trading shares in company i_1 for shares in company i_2, then shares in company i_2 for shares i_3, and so on, finally trading shares in i_k back to shares in company i_1. After such a sequence of trades, one ends up with shares in the same company i_1 that one starts with. Trading around a cycle is usually a bad idea, as you tend to end up with fewer shares than what you started with. But occasionally, for short periods of time, there are opportunities to increase shares. We will call such a cycle an *opportunity cycle*, if trading along the cycle increases the number of shares. This happens exactly if the product of the ratios along the cycle is above 1. In analyzing the state of the market, a firm engaged in trading would like to know if there are any opportunity cycles.

   Give a polynomial time algorithm that finds such an opportunity cycle, if one exists.

3. One way to assess how "well-connected" two nodes in a directed graph are is to look not just at the length of the shortest path between them, but also to count the *number* of shortest paths.

   This turns out to be a problem that can be solved efficiently, subject to some restrictions on the edge costs. Suppose we are given a directed graph $G = (V, E)$, with costs on the edges; the costs may be positive or negative, but every cycle in the graph has strictly positive cost. We are also given two nodes $v, w \in V$. Give an efficient algorithm that computes the number of shortest v-w paths in G. (The algorithm should not list all the paths; just the number suffices.)

4. *Hidden surface removal* is a problem in computer graphics that scarcely needs an introduction — when Woody is standing in front of Buzz you should be able to see Woody but not Buzz; when Buzz is standing in front of Woody, ... well, you get the idea.

   The magic of hidden surface removal is that you can often compute things faster than your intuition suggests. Here's a clean geometric example to illustrate a

basic speed-up that can be achieved. You are given n non-vertical lines in the plane, labeled $L_1, \ldots, L_n$, with the i^{th} line specified by the equation $y = a_i x + b_i$. We will make the assumption that no three of the lines all meet at a single point. We say line L_i is *uppermost* at a given x-coordinate x_0 if its y-coordinate at x_0 is greater than the y-coordinates of all the other lines at x_0: $a_i x_0 + b_i > a_j x_0 + b_j$ for all $j \neq i$. We say line L_i is *visible* if there is some x-coordinate at which it is uppermost — intuitively, some portion of it can be seen if you look down from "$y = \infty$."

Give an algorithm that takes n lines as input, and in $O(n \log n)$ time returns all of the ones that are visible. Figure 1 gives an example.

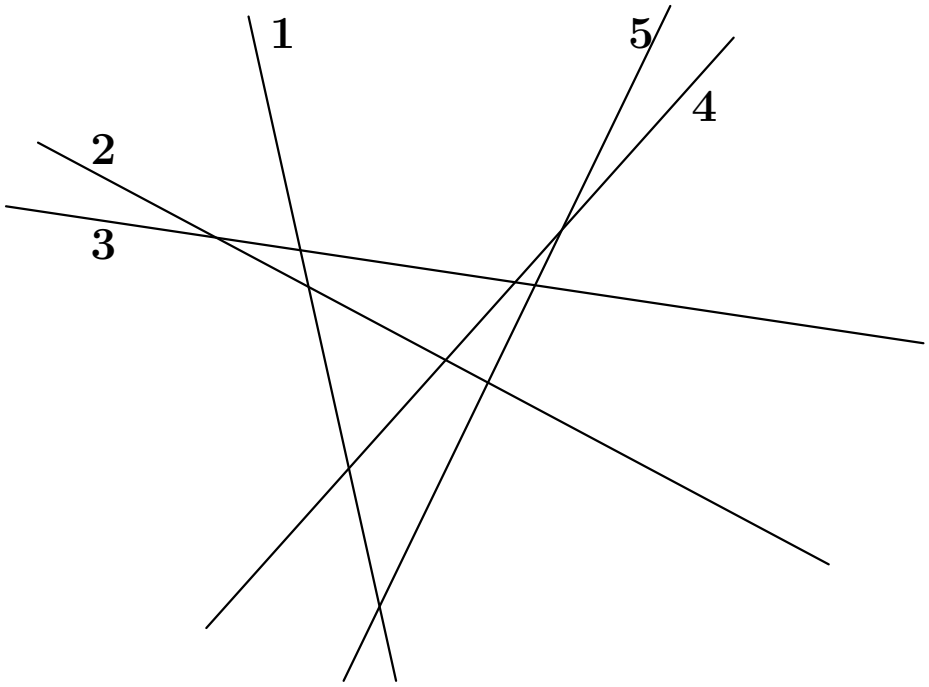

Figure 1: An instance with five lines (labeled "1"–"5" in the figure). All the lines except for "2" are visible.