Massachusetts Institute of Technology
6.854J/18.415J: Advanced Algorithms
David Karger

Handout 11
Wednesday, October 10, 2012

Problem Set 6

Due: Wednesday, October 17, 2012.

Collaboration policy: collaboration is *strongly encouraged.* However, remember that

1. You must write up your own solutions, independently.
2. You must record the name of every collaborator.
3. You must actually participate in solving all the problems. This is difficult in very large groups, so you should keep your collaboration groups limited to 3 or 4 people in a given week.
4. **No bibles. This includes solutions posted to problems in previous years.**

Problem 1. (a) We create a graph to represent this problem as follows. Make a source s and a sink t. Make a node v_p for each prefrosh p. Make a node v_i for each student i. Make a node v_s, v_g, and v_d for each suite s, floor g, and dorm d. Make the following edges, all of capacity 1:

- From s to each prefrosh v_p, of cost 0
- From each v_p to each v_i with cost equal to $-f(p,i)$, where $f(p,i)$ is the suitability factor for prefrosh p and student i. If $f(p,i) = -\infty$, then do not draw this edge.
- From each v_i to v_s, where student i lives in suite s of cost 0.
- From each v_s to v_g, where suite s lies on floor g of cost 0.
- From each v_g to v_d, where floor g is in dorm d of cost 0.
- From each v_d to t of cost 0.

Finally, we place some capacities on vertices. (We can place capacities on vertices, because, as we saw in class, we can replace each vertex v with two vertices v_{in} and v_{out} and an edge of the appropriate capacity in between them.) We place:

- a capacity of size 1 on each student i
- a capacity of size m_s on suite s
- a capacity of size M_g on floor g
- a capacity of size $\mathcal{M}_d$ on dorm d

Thus, if a feasible solution exists, it is a a maximum flow through this network from s to t, not violating any of the capacities. Furthermore, we can maximize the sum of the suitability by minimizing the cost on this flow.

(b) We modify the graph above as follows: for each student i and the suite s that student lives in, add an edge (in addition to the one already there) of infinite capacity and cost W, where W is some large number (larger than the sum of the costs of all other edges). Thus, we can now overfill the suite limits, at a large cost. If the cost of a maximum flow is $aW - b$, then the total exceedance of suite capacities is a, and the total suitability is b. Now, b never exceeds W, so minimizing the cost of the maximum flow means first minimizing the total exceedance, and then maximizing the suitability.

Problem 2. No, Dinic's normal blocking-flow algorithm does not compute a min-cost max-flow in a graph with unit capacities and where each edge has cost 1. Note that in a graph with unit capacities, the min-cost max-flow is the flow of maximum value that puts flow on the least number of edges.

Recall that Dinic's algorithm finds a flow in each iteration that blocks off the shortest augmenting paths, thus increasing the distance to the sink by at least 1 in each iteration. The difficulty is that the first path it takes, while it may use the least number of edges, may force the next shortest augmenting path to use many edges.

As an example, consider the following graph. s and t are separated by a path of n edges. Let a and b be the first and second nodes on this path, and c the last node on this path. Construct a path from s to c with n edges. Construct a path from a to t with n edges. Lastly, add an edge from b to the first node along the path (call it d) from a to t.

Dinic's algorithm will first find the $s - t$ path of n edges, then it will find a path that goes from s through c, then b, then d, through to t. There will be no $s - t$ path, and the number of edges used is $2n + 3$.

However, if we route flow from s to c to t, and from s to a to d to t, we will only use $2n + 2$ edges.

Problem 3. **(a)** Given a graph G and its min-cost circulation f, we would like to re-optimize the solution after increasing/decreasing the cost of one edge by 1. We compute the residual graph G_f and the reduced costs using the Bellman-Ford algorithm. The reduced edge costs are always non-negative and no negative cost cycles exist in G_f. If an edge's reduced cost is strictly positive, the flow on the edge is 0.

Increasing the cost of an edge $e = (u, v)$ in G_f can potentially increase the minimum cost. If the reduced cost of the edge is strictly greater than 0, the prices are valid after the cost of e is incremented. Therefore, reduced-cost optimality is also satisfied. If the reduced cost is 0, we need to re-optimize the solution.

We would like to push as little flow as possible on the edge e. Edge e may have some flow $f(e)$ on it if its reduced cost is 0. We perform a max-flow on the subgraph G_f without the edge e constrained by a maximum of $f(e)$ at the source (done by adding an edge (s', s) with capacity $f(e)$, only on the edges of reduced cost 0. Let f' be this max-flow. If $|f'|$ is $f(e)$, we send no flow across e and our solution is optimal. Otherwise, we send flow $f(e) - |f'|$ through edge e. This is optimal since there is no negative cycle induced by edge e. If there does exist one, it contradicts the optimality of f'.

Decreasing the cost of an edge e can potentially reduce the minimum cost. If the reduced cost of e is strictly greater than 0, the prices are valid even after the decrement and reduced-cost optimality is satisfied. If the reduced cost of e is 0, we need to re-optimize the solution.

We would like to send as much flow across e as possible. As in the previous section, we compute a max-flow from u to v limiting the flow value by $u(e)-f(e)$. Even if all of $u(e) - f(e)$ units are not shipped, the solution is optimal due to the absence of a negative cost cycle.

Our algorithm takes one Bellman-Ford computation, and one maximum flow computation. The $O(mn \log n)$ time taken by max-flow dominates the running time.

(b) We can design a cost-scaling using the above re-optimization routine. In a phase of the scaling algorithm, we shift a bit to the costs on the edges. At the beginning of the kth phase, we have a min-cost flow on the graph with the first k bits of the cost. We double the costs. Notice that doubling the cost does not change the min-cost flow. Then, for each edge e with $(k + 1)$st bit set to 1, we increment/decrement the cost based on the sign of cost $c(e)$. Each phase involves up to m unit changes in edge costs. There are $O(\log C)$ phases. So the running time of the algorithm is $O(m^2 n \log n \log C)$.

Problem 4. For reference, the linear program is as follows:

$$\begin{aligned} \text{minimize } & cx \\ \text{s.t. } & x_1 + x_2 \geq 1 \\ & x_1 + 2x_2 \leq 3 \\ & x_1 \geq 0 \\ & x_2 \geq 0 \\ & x_3 \geq 0 \end{aligned}$$

(a) For $c = (-1, 0, 0)$, this minimizes the quantity $-x_1$, so it maximizes x_1. We have $x_1 + 2x_2 \leq 3$ with $x_2 \geq 0$, so the most x_1 can be is 3. This sets x_2 to 0, and all the other constraints are satisfied if $x_3 \geq 0$. Thus we have the optimal value of -3 and the set of possible solutions $\{(3, 0, x_3) : x_3 \geq 0\}$.

(b) For $c = (0, 1, 0)$, this minimizes the quantity x_2. Given that $x_2 \geq 0$, we set x_2 to 0. The rest of the constraints force $1 \leq x_1 \leq 3$ and $x_3 \geq 0$. Thus we have the optimal value of 0 and the set of possible solutions $\{(x_1, 0, x_3) : 1 \leq x_1 \leq 3, x_3 \geq 0\}$.

(c) For $c = (0, 0, -1)$, this minimizes the quantity x_3, so it maximizes x_3. Since the only constraint on x_3 is that it be nonnegative, the solution is unbounded and has an optimal value of $-\infty$. The set of possible solutions is $\{(x_1, x_2, \infty) : x_1 + x_2 \geq 1, x_1 + 2x_2 \leq 3, x_1 \geq 0, x_2 \geq 0\}$.

Problem 5. **(a)** Let x_i be the variable that denotes the amount of money we wish to trade with client i. We want to maximize the amount of yen we make, so we want to maximize the sum of the x_i's that trade from anything to yen, which we denote by y, minus the amount of yen we trade away. Making sure we optimize for the final amount of yen is important because otherwise we would simply optimize for a maximum amount of yen we could achieve over time, which could mean trading away all the yen we got into pesos and trading it back into yen again, increasing the LP's optimal value but not actually getting more yen. We denote by d the dollar currency. When we write $i = (a_i, b_i)$, we mean the client i that trades currency a_i for b_i. We can write that as:

$$\max \left(\sum_{i=(a_i,y)} r_i x_i - \sum_{i=(y,a_i)} y \right)$$

We subject the variables x_i to a few constraints. First, we have that we cannot trade more than x_i dollars with client i:

$$0 \leq x_i \leq u_i$$

Lastly, we cannot trade more of a currency than we have for every currency c:

$$\sum_{i=(a_i,c)} r_i x_i \geq \sum_{i=(a,b_i)} x_i$$

Also, we cannot trade more than D dollars, as that is our initial amount of money, plus whatever we trade into dollars:

$$\sum_{i=(d,a_i)} x_i \leq D + \sum_{i=(a_i,d)} r_i x_i$$

(b) We can consider the solution to the LP as a flow on a graph with currencies as nodes and clients as edges. The flow conservation rules are not the same as that of max-flow due to the multiplicative factor.

We can decompose the flow into paths. Given any path in the graph starting from s, we compute the currency that is effectively traded by the path. This computation involves finding the minimum of $f(e_i)/\prod_{j<i} r(e_j)$. We will end up with a path and cycle decomposition. The paths can be executed one after another till all operations are performed. Cycles do not earn money due to the "no profit by arbitrage" condition, so we can ignore them. Thus no borrowing is necessary.

(c) Paths that do not end in yen can be ignored. Thus we can construct a flow based on the paths we extracted. This simplified flow will have the same value and will end the day with only dollars and yen. Alternatively, we can modify our linear program to constrain that for all types of currencies not dollars or yen, we have to trade away exactly the amount that we get from other curriences. This is still feasible (we just don't trade the extra dollars that would have become the useless amounts of currency at the end), and it is still optimal because those amounts of currency do not affect the optimum value of the LP.

Problem 6. Let $E = \{(i,j) \mid \text{a packet needs to be routed from } i \text{ to } j\}$. Here, $m = |E|$.

(a) Observe that the $2n$ equality constraints are all tight, but only $2n-1$ of them are independent (because the sum of all the row sums equals the sum of all the column sums). But we are working in an m-dimensional space, so $m-2n+1$ of the inequality constraints must be tight. In other words, $m-2n+1$ of the x_{ij} must equal zero.

(b) If $(i,j) \notin E$, then $M_{ij} = 0$. This leaves m potentially nonzero entries. If at least two of these are nonzero on every row, then the total number of zero entries is at most $n^2 - 2n$, of which we have already accounted for $n^2 - m$. This leaves at most $m - 2n$ zero entries $x_{ij} = M_{ij}$ such that $ij \in E$.

But by (a), there are at least $m-2n+1$ of the x_{ij} are zero. This is a contradiction, so some row a must have at most one nonzero entry.

The entire row cannot be zero (because of the row sum property), so there must be exactly one nonzero entry $x_{ab} = 1$. Because the entries of the matrix are nonnegative, all the other entries in column b must be zero as well.

(c) Consider the matrix M' obtained by deleting row a and column b. The sum of each row and column of this matrix is still 1, because we have only deleted a zero entry. Therefore, M' is doubly stochastic.

If R_a and C_b represent the positions in row a and column b respectively, then let $p = |E \cap (R_a \cup C_b)|$. In going from M to M', we lost p variables. We also lost 2 equality constraints (for the row and the column) and $p-1$ tight inequality constraints. However, these $p+1$ constraints are not linearly independent, because if we set all but one entry in a row or a column, the remaining entry is determined by the sum constraint. Therefore, we have taken away at most p linearly independent constraints, so that M' is a vertex for the resulting LP.

By the induction hypothesis, M' has integer entries, so that M has integer entries.

For the base case, note that for $n = 1$, the only doubly stochastic matrix is (1), which satisfies the given condition. So our claim is true by induction.

A one-zero matrix corresponds to a matching between row i and column j whenever $x_{ij} = 1$. Because any feasible point of the demand polytope can be written as a convex combination of vertices, our switch can always decompose any demand into a convex combination of matchings, in which the sum of the weights of the individual matchings is $\leq \lambda$. So as long as it can deliver matchings at rate λ, it can deliver the specified traffic.

Problem 7.