CS 170 HW 9

Due **2021-04-06, at 10:00 pm**

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, write "none".

2 Flow vs LP

You play a middleman in a market of n suppliers and m purchasers. The i-th supplier can supply up to $s[i]$ products, and the j-th purchaser would like to buy up to $b[j]$ products.

However, due to legislation, supplier i can only sell to a purchaser j if they are situated at most 1000 miles apart. Assume that you're given a list L of all the pairs (i, j) such that supplier i is within 1000 miles of purchaser j. Given $n, m, s[1..n], b[1..m], L$ as input, your job is to compute the maximum number of products that can be sold. The run-time of your algorithm must be polynomial in n and m.

For part (a) and (b), assume the product is divisible, that is, it's OK to sell a fraction of a product.

(a) Show how to solve this problem, using a network flow algorithm as a subroutine. Describe the graph and explain why the output from the network flow algorithm gives a valid solution to this problem.

(b) Formulate this as a linear program. Explain why this correctly solves the problem, and the LP can be solved in polynomial time.

(c) Now let's assume you *cannot* sell a fraction of a product. In other words, the number of products sold by each supplier to each purchaser must be an integer. Which formulation would be better, network flow or linear programming? Explain your answer.

Solution:

(a) *Algorithm:* It is similar to the standard matching to flow reduction. We create a bipartite graph with $n + m + 2$ nodes. Label two of the nodes as a "source" and a "sink." Label n nodes as suppliers, and m nodes as purchasers. Now, we will create the following edges:

- Create an edge from the source to supplier i with capacity $s[i]$.
- For each pair (i, j) in L, create an edge from supplier i to purchaser j with infinite capacity.
- Create an edge from purchaser j to the sink with capacity $b[j]$. We then plug this graph into our network flow solver, and take the size of the max flow as the number of dollars we can make.

Proof of correctness: We claim that the value of the max flow is precisely the maximum amount transactions we can make. To show this, we can show that a strategy of selling products corresponds exactly to a flow in this graph and vice versa.

For any flow, let $x_{i,j}$ be the amount of flow that goes from the node of supplier i to the node of purchaser j. Then, we claim a product selling strategy will sell exactly $x_{i,j}$ products from supplier i to purchaser j. This is a feasible strategy, since the flow going out of the node of supplier i is already bounded by $s[i]$ by the capacity of the edge from the source to that node, and similarly for the purchasers. Similarly, for the other direction, one can observe that any feasible selling strategy leads to a feasible flow of the same value.

(b) We define a variable $x_{i,j}$, denoting the amount of products we take from supplier i and sell to purchaser j. Then, we have the following linear program

$$\begin{aligned}
\max & \sum_{i=1}^{n}\sum_{j=1}^{m} x_{i,j} \\
& \sum_{i=1}^{n} x_{i,j'} \leq b[j'], \text{ for all } j' \in [1,m] \\
& \sum_{j=1}^{n} x_{i',j} \leq s[i'], \text{ for all } i' \in [1,n] \\
& x_{i,j} = 0, \text{ for all } (i,j) \notin L \\
& x_{i,j} \geq 0, \text{ for all } (i,j)
\end{aligned}$$

The linear program has nm variables and $O(nm)$ linear inequalities, so it can be solved in time polynomial in n and m.

(c) Network flow is better. Linear programming is not guaranteed to find an integer solution (not even if one exists), so the approach in part (b) might yield a solution that would involve selling fractional products. In contrast, since all the edge capacities in our graph in part (a) are integers, the Ford-Fulkerson algorithm for max flow will find an integer solution. Thus, max flow is the better choice, because there are algorithms for that formulation that will let us find an integer solution.

3 Feasible Routing

In this problem, we explore a question called *feasible routing*. Given a directed graph G with edge capacities, there are a collection of supply nodes and a collection of demand nodes. The supply nodes want to ship out flow, while the demand nodes want to receive flow. The question is whether there exists a flow that satisfies all supply and demand.

Formally, we are given a capacitated directed graph, and each node is associated with a *demand value*, d_v. We say that v is a supply node if it has a negative demand value (namely, flow out $>$ flow in), and a demand node, it has a positive demand value (namely, flow in $>$ flow out). A node can be neither demand or supply node, in which case $d_v = 0$. Let $c(u, v)$ be the capacity of the directed edge (u, v). Define a *feasible routing* as a flow that satisfies

- Capacity constraint: for each $(u, v) \in E$, $0 \leq f(u, v) \leq c(u, v)$.
- Supply and demand constraint: for each vertex v, $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$.

Here, $f^{\text{in}}(v), f^{\text{out}}(v)$ are the sum of incoming flow and outgoing flow at node v.

Note that this is a feasibility problem, and the answer is simply yes or no, whereas the max flow is an optimization problem, where the answer is a number (max flow value).

(a) Let S denote the supply nodes and T the demand nodes. Define the total demand as $\sum_{u \in T} d_u$ and total supply as $\sum_{u \in S} -d_u$. Is there a feasible routing if total demand does not equal total supply? Explain your answer.

(b) Provide a polynomial-time algorithm to determine whether there is a feasible routing, given the graph, edge capacities and node demand values. Analyze its run-time and prove correctness.
Hint: reduce to max flow.

Solution:

(a) No. For any flow, we have $\sum_v f^{\text{in}}(v) = \sum_v f^{\text{out}}(v)$ (check!). This implies that $\sum_v d_v = 0$, by the supply and demand constraint.

(b) By (a), we may assume that total supply equals total demand (since if not, we can simply answer no immediately.) Now we perform the following reduction.

- Create a new network $G' = (V', E')$ that has the same vertices and edges as G.
- Add to V' a super-source s^* and a super-sink t^*
- For each supply node $v \in S$, we add a new edge (s^*, v) of capacity $-d_v$
- For each demand node $u \in T$, we add a new edge (u, t^*) of capacity d_v

We then invoke any max-flow algorithm on G'. Recalling that D denotes the total demand, we check whether the value of the maximum flow equals D. If so, we answer yes, G has a feasible routing, and otherwise we answer no, G does not have a feasible routing

To prove the correctness of the reduction, we need to show there is a feasible routing in G if and only if G' has an s^*-t^* flow of value D.

($\Rightarrow$) Suppose that there is a feasible routing f in G. The value of this routing (the net flow coming out of all supply nodes) is clearly D. We can create a flow f' of value D in G', by saturating all the edges coming out of s^* and all the edges coming into t^*. We claim that this is a valid flow for G'. Clearly it satisfies all the capacity constraints. To see that it satisfies the flow balance constraints observe that for each vertex $v \in V$, we have one of three cases:

- ($v \in S$) The flow into v from s^* matches the supply coming out of v from the routing.
- ($v \in T$) The flow out of v to t^* matches the demand coming into v from the routing.
- ($v \in V \backslash (S \cup T)$) We have $d_v = 0$, which means that it satisfies flow conservation by the supply/demand constraints.

($\Leftarrow$) Conversely, suppose that we have a flow f' of value D in G'. It must be that each edge leaving s^* and each edge entering t^* is saturated. Therefore, by the flow conservation of f', all the supply nodes and all the demand nodes have achieve their desired supply/demand constraints. All the other nodes satisfy their supply/demand constraints because by the flow conservation of f' the incoming flow equals the outgoing flow. Hence, the resulting flow by ignoring the edges incident to s^*, t^* is a feasible routing for G.

All we do in this algorithm is take linear time to construct the graph G', and then run any max-flow algorithm. Since max-flow can be solved in polynomial time, our algorithm as a whole is also polynomial time.

4 Applications of Max-Flow Min-Cut

Review the statement of max-flow min-cut theorem and prove the following two statements.

(a) Let $G = (L \cup R, E)$ be a unweighted bipartite graph. Then G has a L-perfect matching (a matching with size $|L|$) if and only if, for every set $X \subseteq L$, X is connected to at least $|X|$ vertices in R. You must prove both directions.
Hint: Use the max-flow min-cut theorem.

(b) Let G be an unweighted directed graph and $s, t \in V$ be two distinct vertices. Then the maximum number of edge-disjoint s-t paths equals the minimum number of edges whose removal disconnects t from s (*i.e.*, no directed path from s to t after the removal).
Hint: show how to decompose a flow of value k into k disjoint paths, and how to transform any set of k edge-disjoint paths into a flow of value k.

Solution:

(a) The proposition is known as Hall's theorem. On one direction, assume G has a perfect matching, and consider a subset $X \subseteq L$. Every vertex in X is matched to distinct vertices in R, so in particular the neighborhood of X is of size at least $|X|$, since it contains the vertices matched to vertices in X.

On the other direction, assume that every subset $X \subseteq L$ is connected to at least $|X|$ vertices in R. Add two vertices s and t, and connect s to every vertex in L, and t to every vertex in R. Let each edge have capacity one. We will lower bound the size of any cut separating s and t. Let C be any cut, and let $L = X \cup Y$, where X is on the same side of the cut as s, and Y is on the other side. There is an edge from s to each vertex in Y, contributing at least $|Y|$ to the value of the cut. Now there are at least $|X|$ vertices in R that are connected to vertices in X. Each of these vertices is also connected to t, so regardless of which side of the cut they fall on, each vertex contributes one edge cut (either the edge to t, or the edge to a vertex in X, which is on the same side as s). Thus the cut has value at least $|X| + |Y| = |L|$, and by the max-flow min-cut theorem, this implies that the max-flow has value at least $|L|$, which implies that there must be a perfect matching.

(b) The proposition is known as Menger's theorem. By max-flow min-cut theorem, we only need to show that the max flow value from s to t equals the maximum number of edge-disjoint s-t paths.

If we give each edge capacity 1, then the maxflow from s to t assigns a flow of either 0 or 1 to every edge (using, say, Ford-Fulkerson). Let F be the set of saturated edges; each has flow value of 1. Then extracting the edge-disjoint s-t paths from the flow can be done algorithmically. Follow any directed path in F from s to t (via DFS), remove that path from F, and recurse. Each iteration, we decrease the flow value by exactly 1 and find 1 edge-disjoint s-t path.

Conversely, we can transform any collection of k edge-disjoint paths into a flow by pushing one unit of flow along each path from s to t; the value of the resulting flow is exactly k.

5 Restoring the Balance!

We are given a network $G = (V, E)$ whose edges have integer capacities c_e, and a maximum flow f from node s to node t. Explicitly, f is given to us in the representation of integer flows along every edge e, (f_e).
However, we find out that one of the capacity values of G was wrong: for edge (u, v), we used c_{uv} whereas it really should have been $c_{uv} - 1$. This is unfortunate because the flow f uses that particular edge at full capacity: $f_{uv} = c_{uv}$. We could run Ford Fulkerson from scratch, but there's a faster way.

Describe an algorithm to fix the max-flow for this network in $O(|V| + |E|)$ time. Also give a proof of correctness and runtime justification.

Solution:

Main Idea: Since we know the flows along every edge, we can construct the residual graph in $O(|V| + |E|)$ as there are $2|E|$ edges in the residual graph. The original edges from G in the residual will have weight $c_{uv} - f_{uv}$ and the back edges will have weight f_{uv}.

In this residual graph with the original capacity for (u, v), use DFS to find a path from t to v and from u to s. Join these paths by adding the edge (v, u) in between them to get a path p. Update f by pushing 1 unit of flow from t to s on p (i.e. for each $(i, j) \in p$, reduce f_{ji} by 1, and increase f_{ij} by 1 for all edges except (u, v)). Finally, use DFS to see if the residual graph of the resulting graph, has an $s-t$ path. If so, push 1 unit of flow on this path.

Proof of Correctness: Consider the residual graph of G. If (u, v) is at capacity, then there is a flow of at least 1 unit going from v to t, and since f_e are integral there is a path from v to t with at least one unit of flow moving through it. So the residual graph will have a path from t to v. Similarly, there will be a path from u to s in the residual graph, so the algorithm can always find p correctly.

Note that the size of the maximum flow in the new network will be either the same or 1 less than the previous maximum flow (because changing one edge can change the capacity of the min-cut by at most 1). In the former case, after pushing flow from t to s on p, it is possible to push more flow from s to t, so there must be an $s - t$ path in the new residual graph, so the Ford Fulkerson algorithm will find and push 1 unit of flow because all capacities and flow values are integral. Thus the algorithm finds the new max flow correctly. In the latter case, we will already have the max flow after pushing flow backwards on p, so our algorithm is still correct.

Runtime: The runtime is $O(|V| + |E|)$ because the algorithm just calls DFS on the residual thrice, and it takes $O(|V| + |E|)$ to construct the residual.

Note: for grading, it must be explained how you get the residual graph as it is not given you in the problem statement.

6 Zero-Sum Games

Alice and Bob are playing a zero-sum game whose payoff matrix is shown below. The ij^{th} entry of the matrix shows the payoff that Alice receives if she plays strategy i and Bob plays strategy j. Alice is the row player and is trying to maximize her payoff, and Bob is the column player trying to minimize Alice's payoff.

Alice \Bob	1	2
1	4	1
2	2	5

Now we will write a linear program to find a strategy that maximizes Alice's payoff. Let the variables of the linear program be x_1, x_2 and p, where x_i is the probability that Alice plays row i and p denotes Alice's payoff.

(a) Write the linear program for choosing Alice's strategy to maximize her payoff.

(b) Write a linear program from Bob's perspective trying to minimizing Alice's payoff. Let the variables of the linear program be y_1, y_2 and p, where y_i is the probability that Bob plays strategy i and p denotes Alice's payoff.

(c) As covered in lecture, Bob's linear program and Alice's are dual to each other. How can you see that this is the case for the LPs you have written here? Either take the dual mechanically or make a concise argument. (Hint: You may want to transform the linear programs into a form where it is easy to take the dual, if they are not already in such a form).

(d) What is the optimal solution and what is the value of the game?

Solution:

(a)

$$\begin{aligned} \max\, & p \\ p &\leq 4x_1 + 2x_2 \\ p &\leq x_1 + 5x_2 \\ x_i &\geq 0 \\ x_1 + x_2 &= 1 \end{aligned}$$

(b)

$$\begin{aligned} \min p & \\ p &\geq 4y_1 + y_2 \\ p &\geq 2y_1 + 5y_2 \\ y_i &\geq 0 \\ y_1 + y_2 &= 1 \end{aligned}$$

(c) To make our lives easier, we will make a slight adjustment to Alice's linear program: we will replace $x_1 + x_2 = 1$ with $x_1 + x_2 \leq 1$. This works because decreasing x_1 and x_2 can only decrease p, so our new constraint will hold with equality at the optimum. We will also write Alice's linear program in a more manageable form where all variables appear (possibly with zero coefficients) in the objective function and all constraints, and the right side of every constraint is a constant, like so:

$$\begin{aligned} \max 0x_1 + 0x_2 + 1p & \\ -4x_1 - 2x_2 + 1p &\leq 0 \\ -1x_1 - 5x_2 + 1p &\leq 0 \\ 1x_1 + 1x_2 + 1p &\leq 1 \\ x_i &\geq 0 \end{aligned}$$

With Alice's linear program in this more familiar form, we can now take the dual mechanically. We will call our dual variables y_1, y_2, and p:

$$\begin{aligned} \min 0y_1 + 0y_2 + 1p & \\ -4y_1 - 1y_2 + 1p &\geq 0 \\ -2y_1 - 5x_2 + 1p &\geq 0 \\ 1y_1 + 1y_2 + 0p &\geq 1 \\ y_i &\geq 0 \end{aligned}$$

Applying the same transformations to Bob's linear program as we did to Alice's, we can see that it is exactly equal to the program we have just found. Alice's and Bob's linear programs are dual to each other. This should make some sense: Once both players have announced probabilistic strategies, the expected payoff of the game is fixed. The action of whichever player goes first is to place a bound (in Alice's case, the maximum possible lower bound, and in Bob's case, the minimum possible upper bound) on the payoff achievable by the other player.

(d)

$$\begin{aligned} x_1 &= \frac{1}{2} \\ x_2 &= \frac{1}{2} \\ p &= 3 \end{aligned}$$