CS 170 HW 9

Due **2020-03-23, at 10:00 pm**

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, you must explicitly 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) The problem is known as feasible circulation. 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 circulation, and otherwise we answer "no," G does not have a feasible circulation

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

($\Rightarrow$) Suppose that there is a feasible circulation f in G. The value of this circulation (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 circulation.
- ($v \in T$) The flow out of v to t^* matches the demand coming into v from the circulation.

- ($v \in V \backslash (S \cup T)$) 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 circulation for G.

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 perfect matching 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 Faster Maximum Flow

In the class, we see that the Ford-Fulkerson algorithm computes the maximum flow in $O(mF)$ time, where F is the max flow value. The run-time can be very high for large F. In this question, we explore a polynomial-time algorithm whose run-time does not the depend on the flow value. Recall that Ford-Fulkerson provides a general recipe of designing max flow algorithm, based on the idea of *augmenting path* in residual graph:

Algorithm 1: Ford-Fulkerson

while *there exists an augmenting path in* G_f **do**
- Find an arbitrary augmenting path P from s to t;
- Augment flow f along P;
- Update G_f

This problem asks you to consider a specific implementation of the algorithm above, where each iteration we find the augmenting path with the smallest number of edges and augment along it.

Algorithm 2: Fast Max Flow

while *there exists an augmenting path in* G_f **do**
- Find the augmenting path P from s to t with the smallest number of edges;
- Augment flow f along P;
- Update G_f

We first show that each iteration can be implemented efficiently. Then we analyze the number of iterations required to terminate. Throughout, we define the shortest path from u to v as the path from u to v with the smallest number of edges (instead of to the smallest sum of edge capacities). Consequently, we define distance $d(u, v)$ from u to v as the number of edges in the shortest path from u to v.

(a) Show that given G_f, the augmenting path P from s to t with the smallest number of edges can be found in $O(m + n)$ time.
Hint: Use the most basic graph algorithm you know.

(b) Show that the distance from s to v in G_f never decreases throughout the algorithm, for any v (including t).
Hint: Consider what augmentation does to the "forward edges" going from u to u', where $d(s, u') = d(s, u) + 1$.

(c) Show that with every m flow augmentations, the distance from s to t must increase by (at least) 1.
Hint: On an augmenting path in the residual graph, call an edge bottleneck if its capacity is the smallest. Observe that the bottleneck edges are removed by each flow augmentation.

(d) Conclude by proving that the total number of flow augmentations this algorithm performs is at most $O(mn)$. Analyze the total run-time of the algorithm.
Hint: You may observe that the distance from s to t can increase at most $O(n)$ times throughout the algorithm. If you use this fact, explain why it holds.

Solution: *Bibliographical note*: This algorithm is due to Jack Edmonds and Richard Karp in 1972. In the same paper, they also considered the "fattest augmenting path" algorithm, namely, pushing along the augmenting path that allows the most increase in the flow value. The last excercise of the textbook asks you to analyze it.

(a) Recall that in unweighted graph, BFS suffices to find the shortest path, and it runs in time $O(m+n)$. Specifically, run BFS starting from s. The level where t appears in the BFS tree is the distance from s to t, and the shortest path is simply the corresponding tree path.

(b) We consider what a flow augmentation does to the edges in G_f. Consider the residual graph before the augmentation. Define the *rth layer* of the graph as $L_r = \{v : d(s,v) = r\}$, the set of nodes of distance r from s. We call an edge (u,v) forward if $u \in L_r$ and $v \in L_{r+1}$ for some r, and backward otherwise. Observe that no edge can skip a layer (check!).

Since we push along the path found by BFS, an elementray property of BFS is that it contains only forward edges. Therefore, the augmentation would create reverse edges going from L_{r+1} to L_r. In particular, it does not create any shortcut edge that skips at least a layer—namely, edge going from L_r to $L_{r'}$, for $r' \geq r+2$. Therefore, $d(s,v)$ does not decrease.

(c) Note that each flow augmentation removes at least one bottleneck edge. Whenever the distance from s to t does not increase, we would always push along a path consisting only of forward edges, and thus remove one of them each round. Since all bottleneck edges are forward edges, there are at most m of them in G_f, so we can remove for at most m times. Hence, after m augmentations starting with graph G_f, either t becomes disconnected and we are done, or we would have to use a backward edge in G_f in the next augmenting path, implying that the distance from s to t would then increase.

(d) Before the algorithm terminates, s,t are connected in G_f; that is, $d(s,t) < \infty$. (If it's not the case, we know that there is no more augmenting path from s to t, and the algorithm would terminate by simply outputting the current flow f.) Note that the maximum distance is $n-1$, since there are n nodes in total. Therefore, before the algorithm terminates, $d(s,t)$ can only increment (by 1) for $O(n)$ times. Each incrementation may take at most $O(m)$ augmentations by (c), and each augmentation costs $O(m)$ time via BFS, as shown in (a). Hence, the total runtime of the algorithm is $O(m^2 n)$.