CS 125 - 6

Tiffany (Haotian) Wu

October 2014

Problem 1

Note: before performing any of these algorithms, we must first calculate the residual network, which takes $O(|E|)$ time. Running the DFS algorithms then incorporates this residual network.

Also, the given algorithms assume integer flows. Similar algorithms would work for non-integer flows.

(a) Since max flow is equal to min cut, then by adding 1 to the capacity of any edge we increase the min cut by at most 1. Thus, the only question is whether or not we can push one more unit of flow through the graph from source to sink.

To do this, we note that since the max flow algorithm we went over in class is greedy, we need only to do a DFS on the graph whereby we say a vertex is "reachable" if there is a way we can reach it using the leftover capacity (including the newly updated capacity). We really need only to check to see if a edge has nonzero leftover capacity at each edge we touch, which is a constant amount of work per edge. At the end, we can increment the flows at each edge included in this path, which is $O(|V|)$ amount of work.

If we are indeed able to reach the sink from the source, then the changes we've made so far reflect that, and the new max flow is one more than the old max flow.

Since DFS takes $O(|V| + |E|)$ time to run, this algorithm suffices.

(b) By the same token, decrementing one of the edge capacities decreases the min cut by at most 1, so we need only check whether or not this updated capacity prevents 1 unit of flow from being transported.

To check this, we can first check to see if the edge (v, w) we updated is at full capacity. If it wasn't at full capacity, then obviously it doesn't matter if we decrement the edge capacity. So we would return the same max flow.

If it was at full capacity, we can pick a path with edges having nonzero flows from source to sink through edge (v, w) and decrement all the flows by 1 and update the residual capacities to reflect the changes. Any path will do, since max flow algorithm is greedy. Then to see if we can perhaps propagate one more unit of flow through some other channel, we can do a DFS (similar to part a) from source s to sink t. Since the max flow algorithm is greedy, if there exists some other channel, we should be able to find it at this state.

To find the path going through edge (u, v), you can simply follow edges with nonzero flow backwards from u to s, and follow edges "forwards" with nonzero flow from v to t. If at any point there is a choice between many viable directional edges with endpoint w in the path we're building, you can choose arbitrarily, as the max flow algorithm is greedy (so there is no choice that would 'cut off' flow when there is flow possible). That is:

current vertex = u;
while current vertex != s:
—- for all edges (w,current):
——— if $flow_{(w,current)} > 1$:
————— $flow_{(w,current)} - -$;
————— break;

and the path finder from v to t is similar.

These two algorithms take $O(|V| + |E|)$ time.

After this, we run the DFS from part (a), for an overall runtime of $O(|V| + |E|)$. If we are able to reach t from s, then we have max flow of the new graph is the same as max flow of the old graph. If we are unable to, then the max flow of the new graph is equal to max −1.

Problem 2

(a) The constraints are the same as for max flow problems, except that flow in equals flow out, and we must maximize flow into the sink t , because flow out from the source s may undergo many changes.

We make a variable $x_{(u,v)}$ for the flow through each edge (u, v), and let capacity be represented by constants $k_{(u,v)}$. The linear program is as follows:

$$\begin{aligned} \max & \sum x_{(u,t)} \\ & x_{(u,v)} \geq 0 && \text{for each edge } (u,v) \\ & x_{(u,v)} \leq k_{(u,v)} \\ & \frac{1}{2}\sum x_{(u,v)} = \sum x_{(v,w)} && \text{for each vertex } v \end{aligned}$$

(b) Using the same variables as defined in part (a), we first run a linear program to determine the max flow, and call that constant m.
Then we run another linear program with the added constraint that the flow in must be equal to the max flow, and minimize the costs.

The first linear program is as follows:

$$\begin{aligned} \max & m = \sum x_{(u,t)} \\ & x_{(u,v)} \geq 0 && \text{for each edge } (u,v) \\ & x_{(u,v)} \leq k_{(u,v)} \\ & \sum x_{(u,v)} = \sum x_{(v,w)} && \text{for each vertex } v \end{aligned}$$

and the second linear program is as follows:

$$\begin{aligned} \min & \sum c_{(u,v)} x_{(u,v)} \\ & x_{(u,v)} \geq 0 && \text{for each edge } (u,v) \\ & x_{(u,v)} \leq k_{(u,v)} \\ & \sum x_{(u,v)} = \sum x_{(v,w)} && \text{for each vertex } v \\ & \sum x_{(u,t)} = m \end{aligned}$$

Problem 3

Add two vertices s and t. Also add edges of weight 1 from s to each element of U, and edges of weight 1 from each element of V to s.

Our problem them becomes a max flow problem. If the max flow is $n = |U| = |V|$, then that means we are able to push n units of flow from s to t.

Why does max flow $= n$ suffice?

Lemma: max flow $= n \Rightarrow$ exists a bijection between U and V using the existing edges in the graph.

Proof: If we can prove that the map from U to V is both injective and surjective, then we have a bijection.

Surj. Since flow to t is equal to the sum of the weights of the incoming edges, and each incoming edge is from V and of capacity 1, then flow is equal to the number of vertices in V that receive at least 1 unit of flow. That means each v is hit.

Inj. But since there is maximally one unit of flow is being supplied to each of the vertices in U, then the only way of shipping 1 unit of flow from U to s is through the set V, which means each vertex in U pushes one unit of flow to some vertex V (in the max flow situation), and that this map from U to V is injective - since each vertex in V can only convey 1 unit of flow each, then that means no two vertices $u_1, u_2 \in U$ can push their flow to the same vertex $v \in V$ - otherwise, some vertex in V is not hit, and we get $flow < n$. Thus, there's an injection.

Thus, there must be a bijection. □.

Now all that remains is to prove that the max flow is equal to n. That means the min cut is equal to n, which means we must prove that for any partition of $U \cup V$ into sets S and T should yield at least n edges from S to T.

Pf: Clearly, the minimum n is achieved if we select the partitions

$$S = \{s\},\ T = \{t\} \cup U \cup V$$

or

$$S = \{s\} \cup U \cup V,\ T = \{t\}$$

The only remaining cases are if S includes $x \geq 1$ vertices from U and $n - y \geq 1$ vertices from V. We have two cases:
$y \leq x$: The number of edges from S to the $n - x$ remaining vertices in U is $n - x$. The number of edges from S to the remaining vertices in V is

Problem 4

(a) Suppose, given the game matrix, we found the row player's optimal strategy is $(a, 1-a)$. Then we can reverse engineer what x was.

Since we've already maximized the minimum with value 0, then 0 must be the minimum of the weighted column sums. So we have that 0 is less than or equal to both of them:

$$xa - 2(1-a) \geq 0 \text{ and } -3a + 4(1-a) \geq 0$$

so solving for a in both inequalities, we get:

$$\begin{aligned} -3a + 4(1-a) &\geq 0 \\ -3a + 4 - 4a &\geq 0 \\ 7a &\leq 4 \\ a &\leq \frac{4}{7} \end{aligned}$$

and

$$\begin{aligned} xa - 2(1-a) &\geq 0 \\ xa - 2 + 2a &\geq 0 \\ (x+2)a &\geq 2 \qquad \text{assuming x+2 is positive} \\ a &\leq \frac{2}{x+2} \end{aligned}$$

Since this occurs at the optimum, and we found the optimum using the simplex method of examining the vertices, this must be a vertex, and thus, the boundaries of the inequalities must be equal.

Thus, $\frac{2}{x+2} = \frac{4}{7} \Rightarrow x = 1.5$.

(b) The payout matrix is then $\begin{pmatrix} 1.5 & -3 \\ -2 & 4 \end{pmatrix}$, which yields the row player linear program

$$\begin{array}{llll} \max z & & & \\ z & -1.5x_1 & +2x_2 & \leq 0 \\ z & +3x_1 & -4x_2 & \leq 0 \\ & x_1 & +x_2 & = 1 \end{array}$$

Substituting $x_2 = 1 - x_1$, we get

$$\begin{aligned} z - 1.5x_1 + 2(1-x_1) &\leq 0 \\ z &\leq 3.5x_1 - 2 \end{aligned}$$

and

$$\begin{aligned} z + 3x_1 - 4(1-x_1) &\leq 0 \\ z &\leq 7x_1 - 4 \end{aligned}$$

Which, at their intersection (simplex, look at vertex), we have that $x_1 = \frac{4}{7}$ with value 0.

We solve similarly for the column player:

The column player linear program is

$$\begin{array}{lrrl} \min w & & & \\ w & -1.5y_1 & +3y_2 & \geq 0 \\ w & +2y_1 & -4y_2 & \geq 0 \\ & y_1 & +y_2 & = 1 \end{array}$$

and solving, we get

$$\begin{aligned} w - 1.5y_1 + 3(1 - y_1) &\geq 0 \\ w &\geq 4.5y_1 - 3 \end{aligned}$$

and

$$\begin{aligned} w + 2y_1 - 4(1 - y_1) &\geq 0 \\ w &\geq 4 - 6y_1 \end{aligned}$$

which comes out to $y_1 = 2/3$ at the intersection, yielding a value $w = 0$.

Thus, the row player's optimal strategy is $(4/7, 3/7)$, and the column player's optimal strategy is $(2/3, 1/3)$.