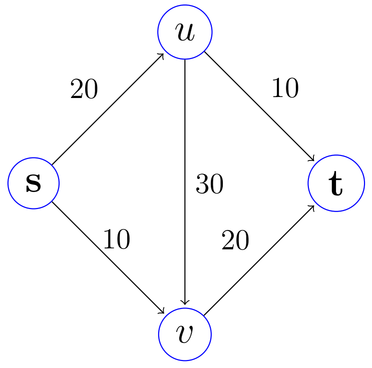
CS 124 Section #9 **Network Flows and Two Player Games** 4/3/17

1 Network Flow

The intuition behind network flow is to think of the edges as pipes and the weights on the edges as the capacity of the pipes per unit of time. The question we are trying to ask is: *how many units of water can we send from the source to the sink per unit of time?*



1.1 Ford-Fulkerson Algorithm

The Ford-Fulkerson algorithm for finding the maximum flow in a graph is based on the idea of residual flows. We know that it is not correct to simply choose paths from s to t and fill them up until we cannot find any more paths from s to t with positive capacity remaining. This would be the greedy algorithm and is not correct.

Instead, what the Ford-Fulkerson algorithm keeps track of the *residual graph*, which starts off with the same vertices, edges, and capacities as the original graph. At each step of the algorithm, we find a path from s to t in the original graph and increase the flow along those edges in the original graph. Then, in the residual graph, we take the path from s to t that we just found, reverse it, and add edges along that path from t to s with the flow that we just found. We keep on doing this until there are no paths from s to t in the residual graph.

Exercise 1. Run Ford Fulkerson on the above graph to find the max flow between s and t.

Solution
We can choose any path to begin our algorithm with, but in order to get the most flow possible at the beginning, I will choose the path $s \to u \to v \to t$. This gives a flow of 20, and changes $s \to u$ to 0, $u \to v$ to 10, and $v \to t$ to 0. In addition, we add in the residual edges: $u \to s$ of 20, $v \to u$ of 10, and $t \to v$ of 20. We add these paths in the residual graph in order to indicate that if we wanted to, we could *push back* flow along those edges, if it results in a more beneficial result for another path. Pushing back flow means re-routing flow through another path.

In the example above, the next step would be to take the path $s \to v \to u \to t$ which has weight 10. By doing this, we essentially have taken 10 out of the 20 units of flow from the original $s \to u \to v \to t$

path and re-routed it through $s \to u \to v$ and the new $s \to v \to u \to t$ path that we have found has turned into 10 units down $s \to v \to 10$.

1.2 Run-time

- Ford-Fulkerson has a run-time of $\boxed{O(Ef^*)}$ because each time we augment a path, we increase the total flow, so at worst the number of times we find an augmenting path is $O(f^*)$, where f^* is the value of the max flow. We can find an augmenting path in $O(E)$ with DFS.
- If we use BFS instead of find our augmenting paths, then can be proved that the run-time is $\boxed{O(VE^2)}$, which does not depend on f^* anymore. This is commonly known as the *Edmond-Karp* algorithm, an implementation of Ford-Fulkerson.

1.3 Max Flow Min Cut

Recall: A cut if a partitioning of the vertices V into S and $V - S$. The weight of the cut is the sum of all the edge weights crossing S and $V - S$. In the context of max flow, the minimum cuts we refer to are *minimum s-t cuts*, which means that s and t must be separated in the cut.

- Min Cut $\geq$ Max Flow
  Reason: If I gave you a particular cut, you know that all the units of flow must at one point pass between S and $V - S$. Thus, if the weight of the cut is c, then at most c units of flow could pass between s and t. The minimum $s - t$ cut is a bottle neck for the flow.
- Min Cut $\leq$ Max Flow
  Reason: At the end of Ford-Fulkerson, I can look at all the nodes that can be reached from s and all the nodes that can be reached from t in the residual graph. There must not be a path from s to t or else the algorithm has not terminated. Therefore, this partitions the vertices into S, those that can be reached from s in this final graph, and $V - S$, those that can reach t, but not s. This is some cut of the graph and the weight of the cut is the max flow, so in particular the minimum cut can be no heavier than this particular cut.
- **Therefore, the min $s - t$ cut and max-flow are equal.**

If all capacities are integers, then there is an integer solution to max-flow (also via Ford-Fulkerson). There may also exist non-integer solutions. Note that the **value** of the max flow is unique, but the **specific amounts of flow** on the edges are not.

1.4 As a Linear Program

We can easily formulate network flow as a linear program. We define the variables f_{uv} to be the amount of flow sent along the edge (u, v). We are trying to maximize the amount of flow coming out of the sink (or going into the source) subject to the following conditions:

- **Capacity**: $f_{uv} \leq c_{uv}$ where c_{uv} is the capacity of the edge (u, v).
- **Conservation**: For every vertex w besides $w = s$ and $w = t$, we must have that

$$\sum_{(u,w)} f_{uw} - \sum_{(w,v)} f_{wv} = 0$$

.

- **Nonnegativitiy**: $f_{uv} \geq 0$

2 Applications of Flow Problems

Exercise 2. In the transportation problem, our goal is to figure out how to ship commodities between a bunch of sources and sinks in the cheapest possible way. We are given a bipartite graph where every vertex is either a source $\{s_1, s_2, \ldots, s_k\}$ or a sink $\{t_1, t_2, \ldots, t_\ell\}$. Edges of the graph only go between s_i and t_j. For each source s_i, there is an associated weight a_i pounds of the commodity that vertex can supply, and for each sink t_j, there is an weight b_j pounds of the commodity that vertex demands such that $\sum_{i=1}^{k} a_i = \sum_{j=1}^{\ell} b_j = N$. For each edge (s_i, t_j) there is an associated cost $c_{ij} \geq 0$, which represents the cost per pound of shipping this commodity along the route from s_i to t_j. The goal is to satisfy all the t_j's demand for this good in the cheapest possible way.

(a) Show how to formulate this question as a linear program.

(b) Suppose some of the roads from s_i to t_j are not so well paved, so there is also a capacity C_{ij} of goods that can even travel between s_i and s_j. How would you modify your linear program to change this?

Solution

(a) Our goal is to minimize the cost of shipping all the goods. Let f_{ij} be the flow Therefore, the objective equation is:

$$\text{minimize} \sum_{i=1}^{k} \sum_{j=1}^{\ell} c_{ij} f_{ij}$$

One of the constraints we have is that for each t_j, we must have

$$\sum_{i=1}^{k} f_{ij} \geq b_j$$

because sink t_j needs as least b_j units of goods. Note that it is equivalent to use $=$ and $\geq$. We also have the constraint that each of the sources s_i cannot supply more of the good than the amount they have in stock. This gives the constraint:

$$\sum_{j=1}^{\ell} f_{ij} \leq a_i$$

. Finally, we have that $f_{ij} \geq 0$.

(b) We would add the constraint $f_{ij} \leq C_{ij}$ for each pair i, j.

Exercise 3. Suppose you are given a $u \times v$ matrix A with real entries. Find an efficient algorithm to either construct a $u \times v$ matrix B with integer entries such that the sum of the entries in each row/column of A is equal to the sum of the entries of each row/column of B, or output that this is impossible.

Here's an example:

$$A = \begin{bmatrix} 0.8 & 0.8 & 0.4 \\ 0.2 & 0.2 & 0.6 \end{bmatrix} B = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

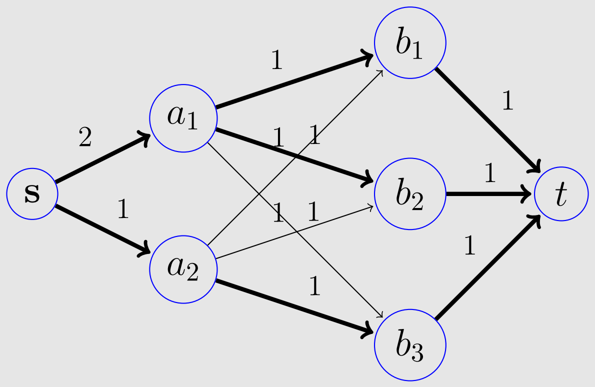
Solution
If any of the row/column sums for A are not integers, this is clearly impossible. This may be checked in $O(uv)$ time. Otherwise, we claim this is always possible.

Firstly, we may reduce to the case when entries in A are in $[0, 1]$ (subtracting/adding an integer to any particular entry doesn't change the property that the row/column sums are integers). This takes time $O(uv)$.

Construct a graph G, which has a source vertex s, a sink vertex t, vertices $a_1, ..., a_u$ corresponding to each row, and vertices $b_1, ..., b_v$ corresponding to each column. Connect s to each a_i with the corresponding row sum, and connect each b_j to t with the corresponding column sum. Connect each a_i to every b_j with capacity 1.

Note that the flow that saturates all edges out of s and into t, and sends flow A_{ij} from edge a_i to b_j is a max-flow. Indeed, its value is the sum of the entries in the matrix A, which is the same value as the cut that separates s from the remainder of the graph (and the cut that separates t from the remainder of the graph).

Run Edmonds-Karp on this graph. Since the capacities are integers, this will result in an integer max-flow. If there is a flow of 1 from a_i to b_j, set $B_{ij} = 1$ and otherwise set $B_{ij} = 0$. Since this is a maximum flow, it saturates the edges out of s and into t (these cuts have the same value as the max-flow by the above paragraph). This implies B has the same row and column sums as A.



Our graph has $O(u + v)$ vertices and $O(uv)$ edges, so running Edmonds-Karp takes time $O(u^2v^2(u + v))$.

Exercise 4. (Kleinberg and Tardos Chapter 7)
Network flow issues come up in dealing with natural disasters and other crises, since major unexpected events often require the movement and evacuation of large numbers of people in a short amount of time.

Consider the following scenario. Due to large-scale flooding in a region, paramedics have identified a set of n injured people distributed across the region who need to be rushed to hospitals. There are k hospitals in the region, and each of the n people needs to be brought to a hospital that is within a half-hour's driving time of their current location (so different people will have different options for hospitals, depending on where they are right now).

At the same time, one doesn't want to overload any one of the hospitals by sending too many patients its way. The paramedics are in touch by cell phone, and they want to collectively work out whether they can choose a hospital for each of the injured people in such a way that the load on the hospitals is balanced: Each hospital receives at most $\lceil n/k \rceil$ people.

Give a polynomial-time algorithm that takes the given information about the people's locations and determines whether this is possible.

Solution
We build the following flow network. There is a node v_i for each patient i, a node w_j for each hospital j and an edge (v_i, w_j) with capacity 1 if patient i is within a half hour drive of hospital j. We then connect a super-source s to each of the patient nodes by an edge of capacity 1, and we connect each of the hospital nodes to a super-sink t by an edge of capacity $\lceil n/k \rceil$.

We claim that there is a feasible way to send all patients to hospitals if an donly if there is an $s - t$ flow of value n. If there is a feasible way to send patients, then we send one unit of flow from s to t along each of the paths $s \to v_i \to w_j \to t$. This does not violate the capacity constraints, especially on edges (w_j, t) due to the load constraint. This means that no hospital can be overloaded. Conversely, if there is a flow of value n, then there is one with integer values, so that corresponds to some path for assignment of hospitals. We send patient i to hospital j if the edge (v_i, w_j) caries one unit of flow, and we observe that the capacity condition ensures that no hospital is overloaded.

The running time of this algorithm is the time required to solve an max-flow problem with $O(n + k)$ nodes and $O(nk)$ edges. With Ford Fulkerson, this can be $O(nk \cdot n)$ because $f^* = n$. and using Edmond-Karp, we get $O((n + k)^2 \cdot nk)$.

3 Two Player Games

A two player game is usually denoted by a matrix of real values. Usually, the numbers represent payoffs to the row player. If the matrix has n rows and m columns, then the row player has n different possible moves and the column player has m different moves. At each round, both players choose a move and the corresponding value is the payoff to the row player. The question we want to ask is: what is each player's best strategy?

Main Ideas:

- Our strategy must be randomized. If it is deterministic, then if our opponent found out our algorithm for determining the next move, they could always play the move that best counteracts our move at each iteration.
- Given a particular random strategy $(y_1 \dots y_n)$ for the row player, we know that for each move y_i, there exists a move from the column player that is best and should always be played. This is called a pure strategy for the column player. *Given my move proportions, the worst case scenario for the row player occurs when the column player plays one of his m pure strategies.*
- For the row player, we analyze the **best worst possible strategy**, i.e the strategy whose worst possible outcome is as best as possible. For the column player, we analyze the opposite – the **worst possible best strategy** for the row player. These two linear programs are *dual* to each other.

Exercise 5. Consider the following two player game, where the entries denote the payoffs of the row player:

$$\begin{pmatrix} 3 & -1 & 2 \\ 1 & 2 & -2 \end{pmatrix}$$

Write the row player's maximization problem as an LP. Then write the dual LP, which is the column player's minimization problem.

Solution
The LP solved by the row player, is

variables	y_1	y_2	$\tilde{z}$		
constraints	1	1		$=$	1
	-3	-1	1	$\leq$	0
	1	-2	1	$\leq$	0
	-2	2	1	$\leq$	0
	$\geq$	$\geq$	unr		
objective			1		max

and the column player's dual is:

variables	z	x_1	x_2	x_3		
constraints	-1	3	-1	2	$\leq$	0
	-1	1	2	-2	$\leq$	0
		1	1	1	$=$	1
	unr	$\geq$	$\geq$	$\geq$		
objective	1					min

Exercise 6. Now find the equilibrium of this game.

Solution
The column player can eliminate the left column, so this becomes a 2×2 game. This can be done because x_3 strictly dominates x_1 in the game (remember that small number is better for the column player) The row player's strategy is $(4/7, 3/7)$ with payoff $2/7$, the column player's strategy is $(0, 4/7, 3/7)$ with payoff $-2/7$. Note that we expect the payoff to be the same always, but the strategies can be arbitrary.