CME 305: Discrete Mathematics and Algorithms
Instructor: Professor Amin Saberi (saberi@stanford.edu)
January 22, 2009

Lecture 5: Network Flow

One of the key concepts, in areas as diverse as communications, many interesting, important, and seemingly unrelated problems can be viewed as network flow problems.

Definition: A **Network** N is a set containing:

- a directed graph $G(V, E)$
- a source vertex $S \in V$ and a sink $t \in V$
- a capacity function $c : E \mapsto \mathbb{R}^+$

where $\mathbb{R}^+$ is the set of non-negative real numbers.

Definition: A **Flow** f on a network N is any function $f : E \mapsto \mathbb{R}^+$. A **Feasible Flow** is a flow f that satisfies two conditions:

- **Edge capacity limits:**
$$\forall\ e \in E,\ 0 \leq f(e) \leq c(e)$$
- **Conservation of flow:**
$$\forall\ v \in V/\{s, t\}, \sum_{e\ leaving\ v} f(e) \ = \sum_{e\ entering\ v} f(e)$$

A network flow could be thought of as model of packet routing in computer networks, getting from a to b in traffic/congestion grids, as a supply chain problem, as water flowing through pipes, or electricity flow in a circuit. Additionally, many seemingly unrelated problems, as we shall see, can either be formulated as network flow problems or contain network flow as a subproblem.

Definition: A **value** of a flow f is defined as

$$v(f) \ \equiv \sum_{e\ leaving\ s} f(e)$$

Since s and t are the only nodes that do not conserve flow, the value of f can be equivalently stated as the amount of flow entering t.

Proposition 1 *For any feasible flow,*

$$v(f) = \sum_{e \text{ leaving } s} f(e) = \sum_{e \text{ entering } t} f(e)$$

Proof: This follows directly from conservation of flow.

$$\begin{aligned} v(f) &= \sum_{e \text{ leaving } s} f(e) \\ &= \sum_{e \text{ leaving } s} f(e) - \sum_{v \in V/\{s,t\}} \left[\sum_{e \text{ entering } v} f(e) - \sum_{e \text{ leaving } v} f(e) \right] \\ &= \sum_{e \text{ entering } t} f(e), \end{aligned}$$

where the last line is due to the fact that in the second line, every edge appears once in a *leaving* term and once in an *entering* term, except those edges entering t. ■

Definition: An **s-t cut** $cut(A, B)$ is a partition of V into subset A and B such that $s \in A$ and $t \in B$. We define the **cut value** $c(A, B)$ to be the sum of capacities of all edges going from set A to set B.

$$c(A, B) = \sum_{\substack{e \text{ leaving } A, \\ \text{entering } B}} c(e)$$

Remark: Using a proof similar to Proposition 1, it is easy to show that for any $cut(A, B)$

$$v(f) = \sum_{\substack{e \text{ leaving } A, \\ \text{entering } B}} f(e) - \sum_{\substack{e \text{ leaving } B, \\ \text{entering } A}} f(e)\ .$$

The **Max-Flow Problem** asks, given a network, find a feasible flow with the maximum possible value.

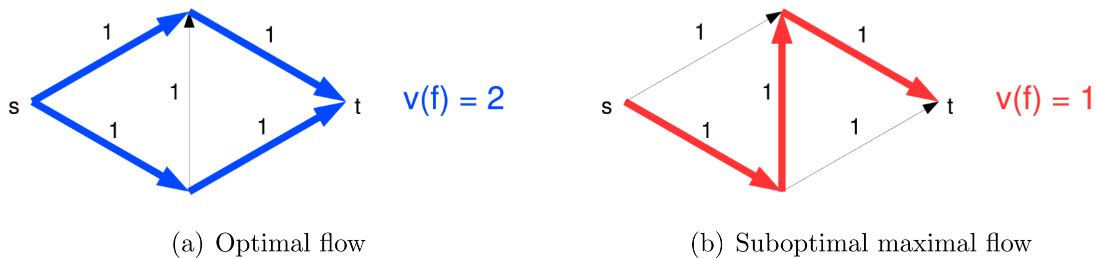
How would we go about designing an algorithm to maximize the flow in a network? One of the first thing to think about when designing a combinatorial algorithm is how a basic "greedy" algorithm would perform.

Although a good starting point, such algorithm doesn't produce a maximum flow in general. A simple counterexample can be seen in the figure 1.

In Figure 1, the greedy algorithm made a bad choice for the first unit of flow to push through. There are no remaining unsaturated s-t paths in the network, but clearly we have not found

Algorithm 1 First tentative algorithm (greedy)

Initialize $f(e) = 0$ for all $e \in E$.
repeat
  Find an unsaturated path P between s and t.
  Augment $f(e)$ for each edges $e \in P$.
until No more unsaturated $s - t$ paths



(a) Optimal flow

(b) Suboptimal maximal flow

Figure 1: Two potential outcomes of the greedy algorithm. a) The optimal flow is achieved. b) no more flow can be greedily pushed through the network.

the maximum flow. Although it may seem trivial to fix in this example, in more complicated networks it is not obvious how to avoid making the wrong path choices so as to not block the optimal flow.

The first rigorous method for solving general max-flow problems is the *Ford-Fulkerson* algorithm. We first define a residual network in network N with respect to a flow f.

Definition: A **residual network** $R(N, f)$ is a network on the original vertex set V and with up to two edges for every edge $e \in E$ constructed as follows:

- if $f(e) < c(e)$ place edge with capacity $c'(e) = c(e) - f(e)$ in the same direction as e.
- if $f(e) > 0$ place edge with capacity $c'(e) = f(e)$ in the opposite direction of e.

The advantage of the residual network R is that any path P from s to t in R gives a path along which we can increase flow. Augmenting the flow f then becomes a connectivity problem, which can be solved using search algorithms using for example depth first search (DFS) or breadth first search (BFS). Building the residual network and augmenting along an s-t path forms the core of *Ford-Fulkerson*.

Algorithm 2 Ford-Fulkerson, 1956

Start with $f(e) = 0$, $\forall e \in E$.
while there is a path P from s to t in $R(N, f)$ **do**
  send a flow of value $\min_{e \in P} c(e)$ in R along P.
  augment f in N using the above flow.
  rebuild the residual network $R(N, f)$.
end while
Output f.

Lemma 1 *If Ford-Fulkerson terminates, it outputs a maximum flow.*

Proof: Let A be the set of vertices reachable from source s in the residual network. Let $B = V/A$ and let f^* be the output of the algorithm. All edges in $cut(A, B)$ are saturated in f^*, i.e. $\forall e$ from A to B, $f^*(e) = c(e)$, and $\forall e$ from B to A, $f^*(e) = 0$.

Therefore, $v(f^*) = c(A, B)$. ■

In the next lecture we will show that if all edge capacities are intergral the Ford-Fulkerson algorithm terminates. As an excercise try to construct a cases in which the algorithm takes a very long time to terminate or does not terminate at all.