Design and Analysis of Algorithms — CSCI 5454
Lecture 10 — 21 February 2011
Prof. Aaron Clauset

1 Maximum Flows and Minimum Cuts

A classic result in graph theory is known as the max-flow/min-cut theorem. The basic idea is that if we have a weighted graph G and pair of vertices s, t, which represent the *source* and *target*, how much flow can we push through the network, if the flow is produced at s and consumed at t? If we assume that individual nodes cannot store flow and that otherwise flow is *conserved*, i.e., not created or destroyed anywhere except at s and t, the maximum amount of flow is given by the minimum weight cut of the network. That is, the max-flow problem asks what's the largest amount of stuff we can move from s to t through G, while the min-cut problem asks what's the minimum amount of damage necessary to separate s and t.

When is finding a max flow or min cut useful? Aside from the obvious applications to flows of commodities (or weapons, or oil, or information, or whatever) through distribution networks, we can also use max flow to solve problems that we can *reduce* to a max flow problem. For instance, if we have a weighted bipartite graph, we can use a max flow algorithm to find a maximum matching on this graph, i.e., a collection of edges such that no two edges share a vertex. We can solve this problem by reducing it to a max-flow problem: let G be the given (undirected) bipartite graph with vertex sets U and V, such that every edge $(u, v) \in E$ has $u \in U$ and $v \in V$. We create a new directed graph G' by orienting every edge from U to V, adding two new vertices s and t, adding edges from s to every vertex in U and from every vertex in V to t. Finally, we assign every edge in G' a capacity of 1. Because any matching on G can be transformed into a max flow on G', a solution on G' corresponds to a solution on G (do you see how?).

Before we explore algorithms for finding the min-cut / max-flow, we need to define what cuts and flows are.[1]

[1]As an interesting point of history, it turns out that the max-flow / min-cut problem was relevant during the Cold War: in the mid-1950s, two researchers in the US Air Force published a classified report studying the railroad network of the Soviet Union and Eastern Europe. The network had 44 vertices and 105 edges, with each vertex representing a geographic region and edges representing railway links between them. Through laborious trial-and-error, they determined the maximum amount of material that could be transported from Russia to Europe as well as the cheapest way to disrupt that flow (by blowing up train tracks). Their results were declassified in 1999. (See A. Schrijver's "On the history of combinatorial optimization (till 1960).")

1.1 Flows

Recall that under our model, flow is *conserved*, that is, for each vertex (except s and t), the flux into a vertex v must equal the flux out of v; mathematically, we say

$$\sum_u f(u \to v) = \sum_w f(v \to w)$$

where f is a function that maps edges into real values ($f : E \to \mathbb{R}_{\geq 0}$) and we assume that $f(u \to v) = 0$ if $(u, v) \notin E$.

Flow, however, doesn't make any sense on a real network unless edges also have *capacities*, that is, we have another function c that gives the maximum flow possible, as a real value, on a particular edge ($c : E \to \mathbb{R}_{\geq 0}$). And, naturally, we assume that $c(u, v) = 0$ if $(u, v) \notin E$.

A flow is called *feasible* with respect to c if $f(e) \leq c(e)$ for every edge $e \in E$. Typically, capacities are fixed and we only consider feasible flows with respect to c.

We say that a flow *saturates* an edge e if $f(e) = c(e)$ and a flow *avoids* an edge if $f(e) = 0$.

The maximum flow problem is then to compute a feasible (s, t)-flow on G, given c such that $\sum_{e \in E} f(e)$ is as large as possible. (Because s and t are not required to respect the conservation property, the *value* of the flow, if we impose the conservation requirement on all other vertices, is equal to the net flow out of s or the net flow into t, which must be equal. Do you see why?)

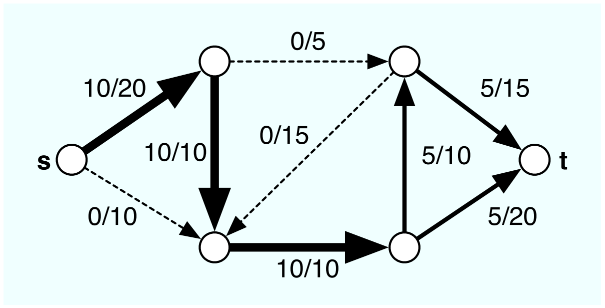

Figure 1: An example network showing an (s, t)-flow f of value $|f| = 10$. In each edge label, the numerator is the flow on the edge and the denominator is the capacity. Dashed edges are avoided by f.

1.2 Cuts

A *cut* on G is a bipartitioning of the vertices V into subsets S and T (meaning $S \cup T = V$ and $S \cap T =$) with $s \in S$ and $t \in T$.

If we're given some cut and a capacity function c (as above), the *capacity* of a cut is the sum of the capacities of the edges in the cut, i.e., the edges that start in S and end in T:

$$||S,T|| = \sum_{v \in S} \sum_{w \in T} c(v \to w)$$

Note that the definition is asymmetric: the capacity of a cut $||S,T||$ is not necessarily equal to the capacity of the cut $||T,S||$.

The minimum cut problem is to compute an (s,t)-cut whose capacity is as small as possible. (From the practical standpoint, the edges in this cut are the minimal ones we would need to remove to disconnect s and t.)

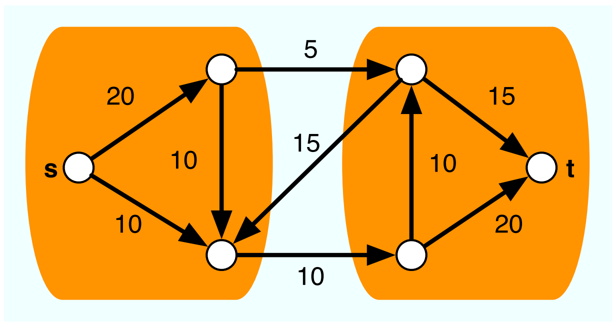

Figure 2: An example network showing an (s,t)-cut with capacity $||S,T|| = 15$. Edge labels give the capacity. (In this case $||S,T|| = ||T,S||$.)

1.3 Max-Flow Min-Cut Theorem

It's not hard to prove that the value of any feasible (s,t)-flow is at most the capacity of any (s,t)-cut. In fact, an even stronger statement holds, which we call the equality condition: the value of a flow $|f| = ||S,T||$ if and only if the flow f saturates every edge from S to T and avoids every edge from T to S. (Do you see why avoiding edges from T to S is important?) Moreover, for a flow f and a cut (S,T) that satisfies this equality condition, f must be a maximum flow and (S,T) must be a minimum cut.

For any weighted directed graph, there is always a flow f and a cut (S,T) that satisfy the equality condition. Or, more precisely:

Max-Flow Min-Cut Theorem: The value of the maximum flow equals the capacity of the minimum cut.

Proof: For a fixed graph G, vertices s and t and capacity function c, let us derive a *reduced graph* G' such that for any pair of vertices u and v, either $c(u \to v) = 0$ or $c(v \to u) = 0$ (that is, if an edge appears in G', then its reversal does not). This can be done easily by taking each pair of non-reduced vertices $\{(u,v),(v,u)\}$ and replacing them with a small subgraph $\{(u,v),(v,x),(x,u)\}$, which inserts a new vertex x into one of the two offending edges, thereby eliminating the redundancy. The max flow and min cut of G' will be the same as G (do you see why?).

Now, suppose we have some feasible flow f. To make it easier to keep track both of the capacity used by f and the unused capacity that still remains in the graph, it will be convenient to define a *residual capacity* function that maps pairs of vertices to real values, $c_f : V \times V \to \mathbb{R}$:

$$c_f(u \to v) = \begin{cases} c(u \to v) - f(u \to v) & \text{if } (u,v) \in E \\ f(u \to v) & \text{if } (v,u) \in E \\ 0 & \text{otherwise} \ . \end{cases}$$

The first case corresponds to edges $u \to v$ that currently carry some flow in f; c_f assigns them a weight equal to their unused capacity. The second case assigns a weight to the edge $v \to u$ equal to the flow on the edge $u \to v$. The third case assigns 0 weight to all other edges. Because $0 \leq f(u \to v) \leq c(u \to v)$, residual capacities are always non-negative.

This residual capacity function is useful because it lets us define a *residual graph* $G_f = (V, E_f)$, where E_f is the set of edges whose residual capacity is positive, i.e., the set of edges can are not saturated. For example, see Figure 3. In the residual graph, a "back" edge $v \to u$ stores the flow in the other direction, along the edge $u \to v$; this lets us store the unused capacity of $u \to v$ as a "forward" edge. Basically, the residual graph is a fancy kind of graph data structure that stores used and unused capacity in a way that makes it easy to find ways to increase the total flow.

To see why, suppose there's no path from s to t in the residual graph G_f. Let S be the set of all vertices that are reachable from s in G_f, and let $T = V - S$ (the rest of the graph). The partition (S,T) is then an (s,t)-cut, and for every pair of vertices $u \in S$, $v \in T$, we have

$$c_f(u \to v) = \Big(c(u \to v) - f(u \to v)\Big) + f(v \to u) = 0$$

which implies that each of these edges from S to T is saturated (i.e., $c(u \to v) - f(u \to v) = 0$) and each edge from T to S is avoided (i.e., $f(v \to u) = 0$). Thus, f is a maximum flow and (S,T)

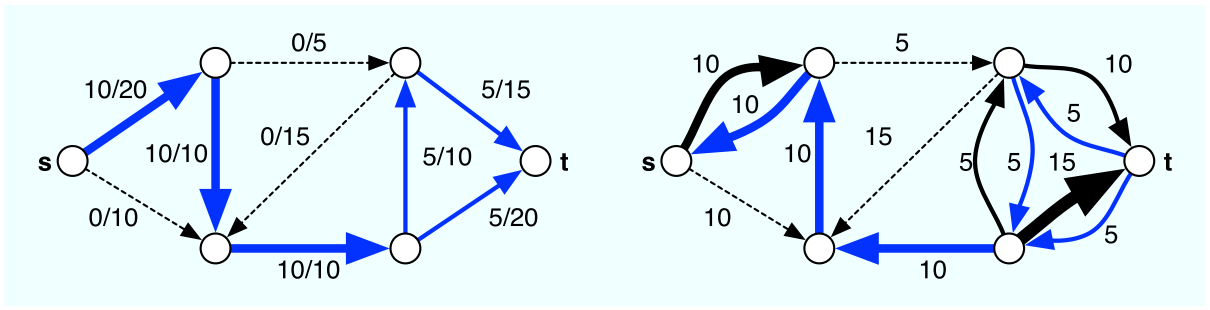

Figure 3: A flow f on our example weighted graph G and the corresponding residual graph G_f. The blue edges represent the flow edges; in G, they point forward, while in G_f, they point backward, representing used capacity. Black edges in G_f represent unused capacity along edges currently carrying flow. Dashed edges carried no flow in f and thus point forward in G_f.

is a minimum cut.

But, what if there is a path $\sigma_{st} = \{s = v_0 \to v_1 \to \cdots \to v_r = t\}$ in the residual graph G_f, i.e., a path from s to t. We call σ_{st} an *augmenting path*. On this path, $F = \min_i c_f(v_i \to v_{i+1})$ is the maximum amount of residual flow in G_f, and we can improve our current flow f by pushing this additional amount of flow through G. We can update f to include this flow in the following way: let f' be the updated flow, and let

$$f'(u \to v) = \begin{cases} f(u \to v) + F & \text{if } (u,v) \in \sigma_{st} \\ f(u \to v) - F & \text{if } (v,u) \in \sigma_{st} \\ f(u \to v) & \text{otherwise} \end{cases} .$$

In the first case, there is unused capacity on an existing edge, and we can increase the flow from s to t by using F of that capacity. In the second case, we move some flow off that edge (because we move it elsewhere). Finally, if an edge is not on the augmenting path, then we leave it alone.

Let's quickly prove that f' is feasible with respect to the original capacities c, which simply requires us to verify that $0 \leq f' \leq c$ for each edge. Since we only modify edges in the augmenting path itself, we must only check whether we have violated any of their capacities. If the forward edge $(u,v) \in \sigma_{st}$, then we increase its flow and $f'(u \to v) > f(u \to v) \geq 0$ and thus

$$\begin{aligned} f'(u \to v) &= f(u \to v) + F \\ &\leq f(u \to v) + c_f(u \to v) \\ &= f(u \to v) + c(u \to v) - f(u \to v) \\ &= c(u \to v) \end{aligned}$$

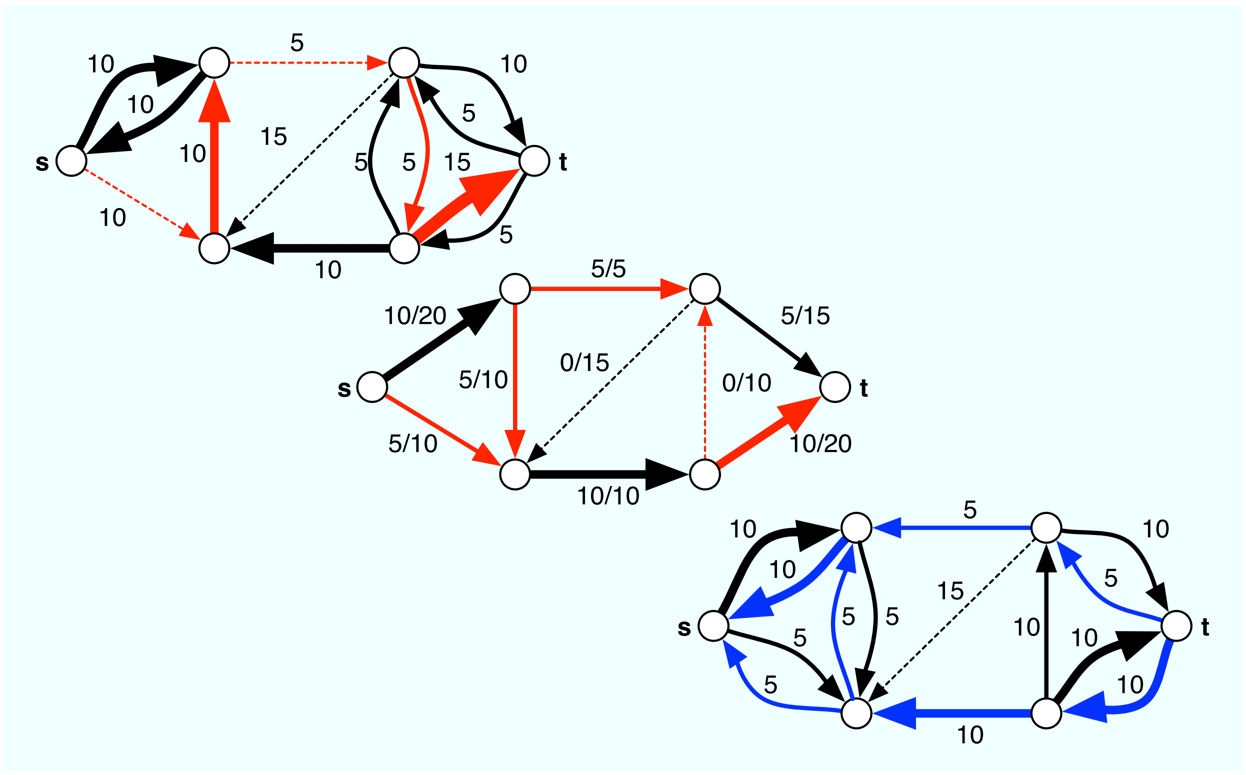

Figure 4: (upper) An augmenting path σ_{st} (shown in red) in G_f with additional flux $F = 5$, (middle) the corresponding augmented flow f' on G, and (lower) the residual graph $G_{f'}$ after the update. In the final residual graph, the fluxes of f' are shown in blue and residual capacities in black; note that now there exists no additional σ_{st}, implying that f' is a max flow. (There are other max flow solutions to this graph, including one that uses only 6 edges. Do you see it?)

On the other hand, if the backward edge $(v, u) \in \sigma_{st}$, then we decrease its flow and $f'(u \to v) < f(u \to v) \leq c(u \to v)$ and thus

$$\begin{aligned} f'(u \to v) &= f(u \to v) - F \\ &\geq f(u \to v) - c_f(u \to v) \\ &= f(u \to v) - f(u \to v) \\ &= 0 \end{aligned}$$

Finally, since $|f'| = |f| + F > 0$, f was not a maximum flow. $\square$

1.4 Ford-Fulkerson algorithm

Most max-flow min-cut algorithms use the observation about augmenting paths to construct a solution to the problem.

The Ford-Fulkerson algorithm does exactly this: it starts with $f = 0$ and then repeatedly finds any augmenting path in the residual graph, which repeatedly increases the flow, until no such path can be found.

If all the edge capacities are integers, then each augmentation step increases $|f|$ by a positive integer, and the algorithm halts after $|f^*|$ iterations, where $|f^*|$ is the maximum flow. Each iteration of this algorithm requires $O(E)$ time, which accounts both for creating the residual graph G_f and for running a search-tree algorithm (it doesn't matter what kind) to find some augmenting path. Thus, Ford-Fulkerson takes $O(E|f^*|)$ time.

It further holds that Ford-Fulkerson halts if the edge capacities are rational numbers because we can convert any set of rational numbers into a set of integers by multiplying by a sufficiently large integer (do you see why? what integer would do the trick?). However, if any edge capacity is irrational, then Ford-Fulkerson fails to halt because the algorithm can always find an augmenting path with smaller and small additional flux F, and worse, this infinite sequence of augmentations may not even converge on the maximum flow.[2] This point is not entirely academic: because modern computers represent rational numbers using floating point structures, a careless implementation of Ford-Fulkerson could enter into an infinite loop simply because of a round-off error.

1.5 Edmond-Karp (1)

We can be smarter about converging on the maximum flow if we're smarter about how we choose which augmenting path to use, if there's more than one. One approach, called Edmonds-Karp, is to by greedy about it, i.e., to choose the augmenting path with the largest bottleneck value, which can be summarized by saying that, when given a choice, we should always choose the fattest open pipe to fill.

The maximum bottleneck (s, t)-path in a directed graph can be computed in $O(E \log V)$ time using Prim's MST algorithm or Dijkstra's SSSP algorithm: start growing a directed spanning tree S rooted at s by repeatedly finding and adding to S the highest-capacity edge within the cut (S, T) (i.e., among the edges that leave S) until the tree contains a path from s to t.

[2]Can you construct an example of such pathological behavior? Uri Zwick identified one such graph with 6 nodes and 9 edges where six edges have some large integer capacity X, two have capacity 1 and one has capacity $\phi_- = (\sqrt{5} - 1)/2 = 0.61803....$

The analysis of this algorithm is similar to that of Ford-Fulkerson, and for graphs with integer capacities, it runs in $O(E^2 \log E \log |f^*|)$ time.

1.6 Edmond-Karp (2)

There's another algorithm due to Edmond-Karp (which was independently discovered by Dinit), which says to choose the augmenting path with the fewest edges, i.e., to find the shortest open pipe to fill. This version, which I won't go into here,[3] runs in time $O(VE^2)$, which can be improved to $O(V^2E)$ using an insight due to Dinit.

2 For Next Time

1. Guest lecture by Prof. Sankaranarayanan on Wednesday, February 23
2. Reminder: solutions to PS2 due Tuesday via email by 11:59pm

[3]Briefly: this is like asking for the shortest augmenting path, which means running breadth-first search on the residual graph. Surprisingly, the running time of this approach can be shown to be independent of the edge capacities, which can be a desirable property.