CS124	Lecture 17

Network Flows

Suppose that we are given the network in top of Figure 17.1, where the numbers indicate capacities, that is, the amount of flow that can go through the edge in unit time. We wish to find the maximum amount of flow that can go through this network, from S to T.

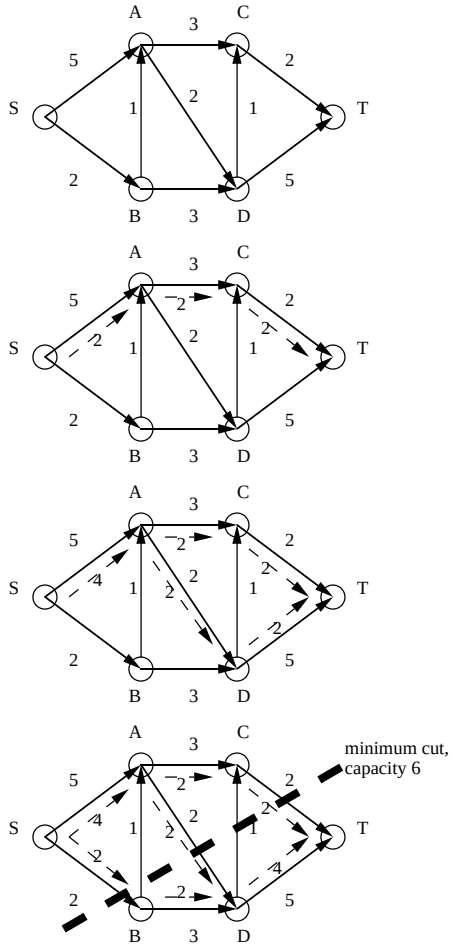

Figure 17.1: Max flow

More formally, we are given as input a directed graph $G = (V,E)$. Every edge e has a nonengative capacity c_e, which in this class we will assume are also integral. We imagine that two vertices $S \neq T$ in V are special: S is called the *source*, and t is referred to as the *sink*. One should imagine that we have some network of pipes, where each edge allows water to flow in one direction and has a capacity of the number of, say, liters of water per second that

can flow through that pipe (this is that pipe's capacity). Some source of water directly sprouts out from S, and we imagine that source can sprout water at an arbitrarily fast rate. Meanwhile the sink vertex T is the drain. A *flow* is then a vector $f = (f_1, \ldots, f_{|E|})$ where f_e is the rate of water flowing through pipe e. A valid flow is any such vector satisfying the following two conditions:

1. For any edge e, $0 \leq f_e \leq c_e$
2. For any vertex $u \neq s,t$,

$$\sum_{\substack{e \in E \\ e=(u,\cdot)}} f_e = \sum_{\substack{e \in E \\ e=(\cdot,u)}} f_e.$$

That is, flow is conserved: the flow *into* any non-source/non-sink vertex equals the flow *out* of that vertex.

For any flow vector f, its *value* $val(f)$ is the total amount of flow leaving S (which is also the total amount of flow entering t by flow conservation). That is,

$$val(f) = \sum_{\substack{e \in E \\ e=(S,\cdot)}} f_e$$

Let us now develop an algorithm to compute a maximum flow in G. This algorithm is known as the *Ford-Fulkerson* algorithm. We first start with the all-zero flow. How can we find a small improvement in the flow? Answer: we find a path from S to T (say, by depth-first search) and move flow along this path of total value equal to the *minimum* capacity of an edge on the path (it can obviously do no better). This is the first iteration (see Figure 17.1).

How would we continue? We could look for another path from S to T. Since this time we already partially (or totally) use some of the edges, we should do depth-first search on the edges that have some *residual capacity*, above and beyond the flow they already carry. Thus, the edge CT would be ignored, as if it were not there. The depth-first search would now find the path $S-A-D-T$, and augment the flow by two more units, as shown in Figure 17.1.

Next, we would again try to find a path from S to T. The path is now $S-B-D-T$ (the edges $C-T$ and $A-D$ are full are are therefore ignored), and we augment the flow as shown in the bottom of Figure 17.1.

Next we would again try to find a path. But since edges $A-D$, $C-T$, and $S-B$ are full, they must be ignored, and therefore depth-first search would fail to find a path, after marking the nodes S,A,C as reachable from S. We could then return the flow shown, of value 6, as maximum.

How can we be sure that it is the maximum? Notice that these reachable nodes define a *cut* (a set of nodes containing S but not T), and the *capacity* of this cut (the sum of the capacities of the edges going out of this set) is

6, the same as the max-flow value. (It must be the same, since this flow passes through this cut.) The existence of this cut establishes that the flow is optimum!

There is a complication that we have swept under the rug so far: when we do depth-first search looking for a path, we use not only the edges that are not completely full, but we must also traverse *in the opposite direction* all edges that already have some non-zero flow. This would have the effect of canceling some flow; canceling may be necessary to achieve optimality, see Figure 17.2. In this figure the only way to augment the current flow is via the path $S-B-A-T$, which traverses the edge $A-B$ in the reverse direction (a legal traversal, since $A-B$ is carrying non-zero flow).

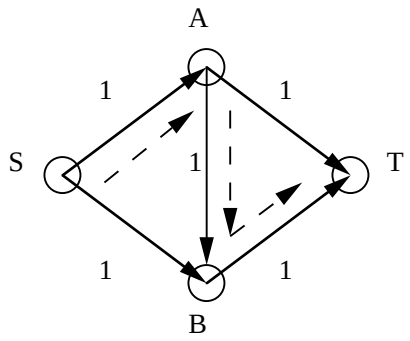

Figure 17.2: Flows may have to be canceled

In general, a path from the source to the sink along which we can increase the flow is called an *augmenting path*. We can look for an augmenting path by doing for example a depth first search along the *residual network*, which we now describe. For an edge (u,v), let $c(u,v)$ be its capacity, and let $f(u,v)$ be the flow across the edge. Note that we adopt the following convention: if 4 units flow from u to v, then $f(u,v)=4$, and $f(v,u)=-4$. That is, we interpret the fact that we could reverse the flow across an edge as being equivalent to a "negative flow". Then the *residual capacity* of an edge (u,v) is just

$$c(u,v)-f(u,v).$$

The residual network has the same vertices as the original graph; the edges of the residual network consist of all weighted edges with strictly positive residual capacity. The idea is then if we find a path from the source to the sink in the residual network, we have an augmenting path to increase the flow in the original network. As an exercise, you may want to consider the residual network at each step in Figure 17.1.

Suppose we look for a path in the residual network using depth first search. In the case where the capacities are integers, we will always be able to push an integral amount of flow along an augmenting path. Hence, if the

maximum flow is f^*, the total time to find the maximum flow is $O((m+n)f^*)$, since we may have to do an $O(m+n)$ depth first search up to f^* times (recall we use n to denote $|V|$ and $m = |E|$). This is not so great.

Note that we do not have to do a depth-first search to find an augmenting path in the residual network. In fact, using a breadth-first search each time yields an algorithm that provably runs in $O(nm^2)$ time, regardless of whether or not the capacities are integers. We will not prove this here. Finding augmenting paths using breadth-first search is known as the Hopcroft-Karp algorithm or Dinic's algorithm (the two groups discovered it independently during the Cold War). There are also other algorithms and approaches to the max-flow problem as well that improve on this running time.

To summarize: we repeatedly finds a path from S to T along edges that are not yet full (have non-zero residual capacity), and also along any reverse edges with non-zero flow. If an $S-T$ path is found, we augment the flow along this path, and repeat. When a path cannot be found, the set of nodes reachable from S defines a cut of capacity equal to the max-flow. Thus, *the value of the maximum flow is always equal to the capacity of the minimum cut*. This is the important *max-flow min-cut theorem*. One direction (that max-flow$\leq$min-cut) is easy (think about it: *any* cut is larger than *any* flow); the other direction is proved by the algorithm just described.

Note that we have already seen an algorithm for the minimum cut problem: Karger's algorithm (which was Monte Carlo). However the problems are slightly different. Karger's algorithm tries to find the *global* minimum cut in the graph; that is to say, there aren't two special source and sink nodes that must be separated from each other. Meanwhile, the maximum flow algorithm above finds a minimum $S-T$ cut.

Exercise: Show how to use any $S-T$ minimum cut algorithm $\mathcal{A}$ in a black box way to obtain a global minimum cut. Your reduction should only make $n-1$ different calls to $\mathcal{A}$ (giving different (S,T) pairs each time as input).

Matching

We know show a reduction: that is, we will describe another graph problem, *matching*, which reduces to maximum flow. That is, it can be solved using any maximum flow algorithm as a black box.

Suppose that the *bipartite* graph shown in Figure 17.3 records the compatibility relation between four boys and four girls. We seek a maximum matching, that is, a set of edges that is as large as possible, and in which no two edges share a node. For example, in Figure 17.3 there is a *complete* matching (a matching that involves all nodes).

To reduce this problem to max-flow, we create a new source and a new sink, connect the source with all boys and all girls with the sinks, and direct all edges of the original bipartite graph from the boys to the girls. All edges

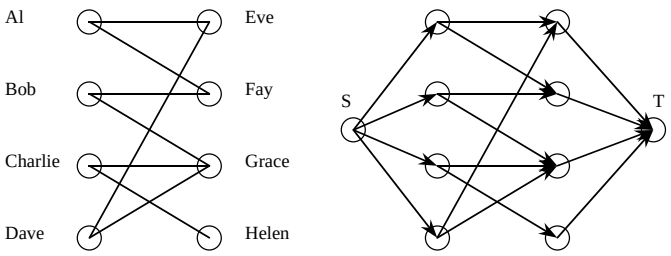

Figure 17.3: Reduction from matching to max-flow (all capacities are 1)

have capacity one. It is easy to see that the maximum flow in this network corresponds to the maximum matching.

Well, the situation is slightly more complicated than was stated above: what is easy to see is that the optimum *integer-valued* flow corresponds to the optimum matching. We would be at a loss interpreting as a matching a flow that ships .7 units along the edge Al-Eve! Fortunately, what the algorithm in the previous section establishes is that *if the capacities are integers, then the maximum flow is integer.* This is because we only deal with integers throughout the algorithm. Hence *integrality comes for free in the max-flow problem*.

Unfortunately, max-flow is about the only problem for which integrality comes for free. It is a very difficult problem to find the optimum solution (or *any* solution) of a general linear program with the additional constraint that (some or all of) the variables be integers. We will see why in forthcoming lectures.