CS125	Lecture 10

Network Flows

Suppose that we are given the network in top of Figure 10.1, 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.

This problem can also be reduced to linear programming. We have a nonnegative variable for each edge, representing the flow through this edge. These variables are denoted $f_{SA}, f_{SB}, \ldots$ We have two kinds of constraints: capacity constraints such as $f_{SA} \leq 5$ (a total of 9 such constraints, one for each edge), and flow conservation constraints (one for each node except S and T), such as $f_{AD} + f_{BD} = f_{DC} + f_{DT}$ (a total of 4 such constraints). We wish to maximize $f_{SA} + f_{SB}$, the amount of flow that leaves S, subject to these constraints. It is easy to see that this linear program is equivalent to the max-flow problem. The simplex method would correctly solve it.

In the case of max-flow, it is very instructive to "simulate" the simplex method, to see what effect its various iterations would have on the given network. Simplex would start with the all-zero flow, and would try to improve it. How can it find a small improvement in the flow? Answer: it finds a path from S to T (say, by depth-first search), and moves 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 of simplex (see Figure 10.1).

How would simplex continue? It would 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 10.1.

Next, simplex would again try to find a path from S to T. The path is now $S-A-B-D-T$ (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 10.1.

Next simplex 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. *Simplex then returns 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

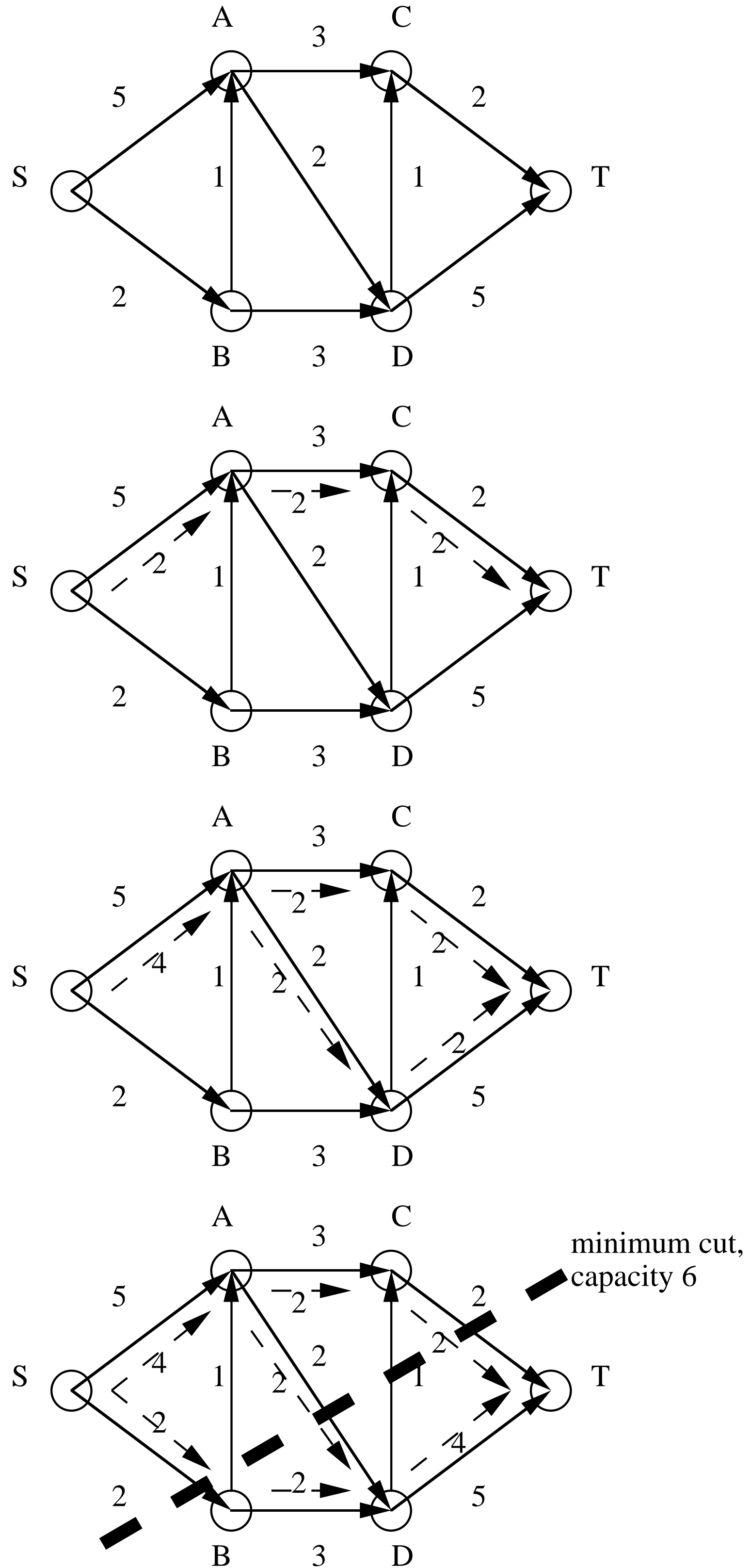

Figure 10.1: Max flow

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 10.2. 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).

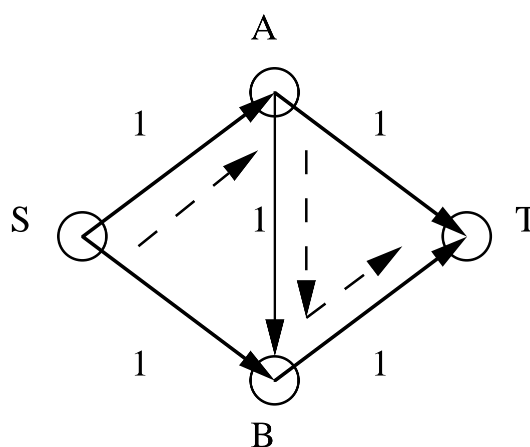

Figure 10.2: Flows may have to be canceled

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 10.1.

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 $O(Ef^*)$, since we may have to do an $O(E)$ depth first search up to f^* times. 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 $O(VE^2)$ time, regardless of whether or not the capacities are integers. We will not prove this here. There are also other algorithms and approaches to the max-flow problem as well that improve on this running time.

To summarize: the max-flow problem can be easily reduced to linear programming and solved by simplex. But it is easier to understand what simplex would do by following its iterations directly on the network. It 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 taking advantage of the algorithm just described.

Duality Again

As it turns out, the max-flow min-cut theorem is a special case of a more general phenomenon called *duality*. Recall duality means that for each maximization problem there is a corresponding minimizations problem with the property that any feasible solution of the min problem is greater than or equal any feasible solution of the max problem. Furthermore, and more importantly, *they have the same optimum*.

Consider the network shown in Figure 10.3, and the corresponding max-flow problem. We know that it can be written as a linear program as follows:

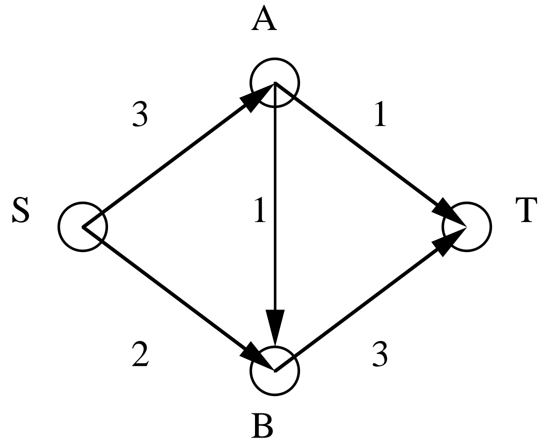

Figure 10.3: A simple max-flow problem

$$
\begin{array}{llllllr}
\max & f_{SA} & +f_{SB} & & & & \\
 & f_{SA} & & & & & \leq 3 \\
 & & f_{SB} & & & & \leq 2 \\
 & & & f_{AB} & & & \leq 1 \\
 & & & & f_{AT} & & \leq 1 \\
 & & & & & f_{BT} & \leq 3 \\
 & f_{SA} & & -f_{AB} & -f_{AT} & & = 0 \\
 & & f_{SA} & +f_{AB} & & -f_{BT} & = 0 \\
 & & & & & & f \geq 0
\end{array}
\qquad P
$$

Consider now the following linear program:

$$
\begin{array}{llllllll}
\min 3y_{SA} & +2y_{SB} & +y_{AB} & +y_{AT} & +3y_{BT} & & & \\
y_{SA} & & & & & +u_A & & \geq 1 \\
 & y_{SB} & & & & & +u_B & \geq 1 \\
 & & y_{AB} & & & -u_A & +u_B & \geq 0 \\
 & & & y_{AT} & & -u_A & & \geq 0 \\
 & & & & y_{BT} & & -u_B & \geq 0 \\
 & & & & & & & y \geq 0
\end{array}
\qquad D
$$

This LP describes the min-cut problem! To see why, suppose that the u_A variable is meant to be 1 if A is in the cut with S, and 0 otherwise, and similarly for B (naturally, by the definition of a cut, S will always be with S in the cut, and T will never be with S). Each of the y variables is to be 1 if the corresponding edge contributes to the cut capacity, and 0 otherwise. Then the constraints make sure that these variables behave exactly as they should. For example, the second constraint states that *if A is not with S, then SA must be added to the cut.* The third one states that *if A is with S and B is not* (this is the only case in which the sum $-u_A + u_B$ becomes -1), *then AB must contribute*

to the cut. And so on. Although the y and u's are free to take values larger than one, they will be "slammed" by the minimization down to 1 or 0.

These two linear programs are in fact, duals of each other. This fact is most easily seen by putting the linear programs in matrix form. The first program, which we call the primal (P), we write as:

max	1	1	0	0	0		
	1	0	0	0	0	$\leq$	3
	0	1	0	0	0	$\leq$	2
	0	0	1	0	0	$\leq$	1
	0	0	0	1	0	$\leq$	1
	0	0	0	0	1	$\leq$	3
	1	0	-1	-1	1	$=$	0
	0	1	1	0	-1	$=$	0
	$\geq$	$\geq$	$\geq$	$\geq$	$\geq$		

Here we have removed the actual variable names, and we have included an additional row at the bottom denoting that all the variables are non-negative. (An unrestricted variable will be denoted by unr.

The second program, which we call the dual (D), we write as:

min	3	2	1	1	3	0	0		
	1	0	0	0	0	1	0	$\geq$	1
	0	1	0	0	0	0	1	$\geq$	1
	0	0	1	0	0	-1	1	$\geq$	0
	0	0	0	1	0	-1	0	$\geq$	0
	0	0	0	0	1	0	-1	$\geq$	0
	$\geq$	$\geq$	$\geq$	$\geq$	$\geq$	unr	unr		

Each variable of P corresponds to a constraint of D, and vice-versa. Equality constraints correspond to unrestricted variables (the u's), and inequality constraints to restricted variables. Minimization becomes maximization. The matrices are transpose of one another, and the roles of right-hand side and objective function are interchanged.

Repeating from the past lecture notes, it is a mechanical process, given an LP, to form its dual. Suppose we start with a maximization problem. Change all inequality constraints into $\leq$ constraints, negating both sides of an equation if necessary. Then

- transpose the coefficient matrix
- invert maximization to minimization
- interchange the roles of the right-hand side and the objective function
- introduce a nonnegative variable for each inequality, and an unrestricted one for each equality
- for each nonnegative variable introduce a $\geq$ constraint, and for each unrestricted variable introduce an equality constraint.

By the max-flow min-cut theorem, the two LP's P and D above have the same optimum. However, as we have stated before, an LP and its dual, as long as both have bounded optimum solutions, will have the same optimum value. The max-flow min-cut theorem could therefore also have been derived by showing that, in general, the max-flow linear program and min-cut linear program are always dual linear programs, so that we must have the max-flow equals the min-cut.

Matching

It is often useful to *compose* reductions. That is, we can reduce a problem A to B, and B to C, and since C we know how to solve, we end up solving A. A good example is the matching problem.

Suppose that the *bipartite* graph shown in Figure 10.4 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 10.4 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 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

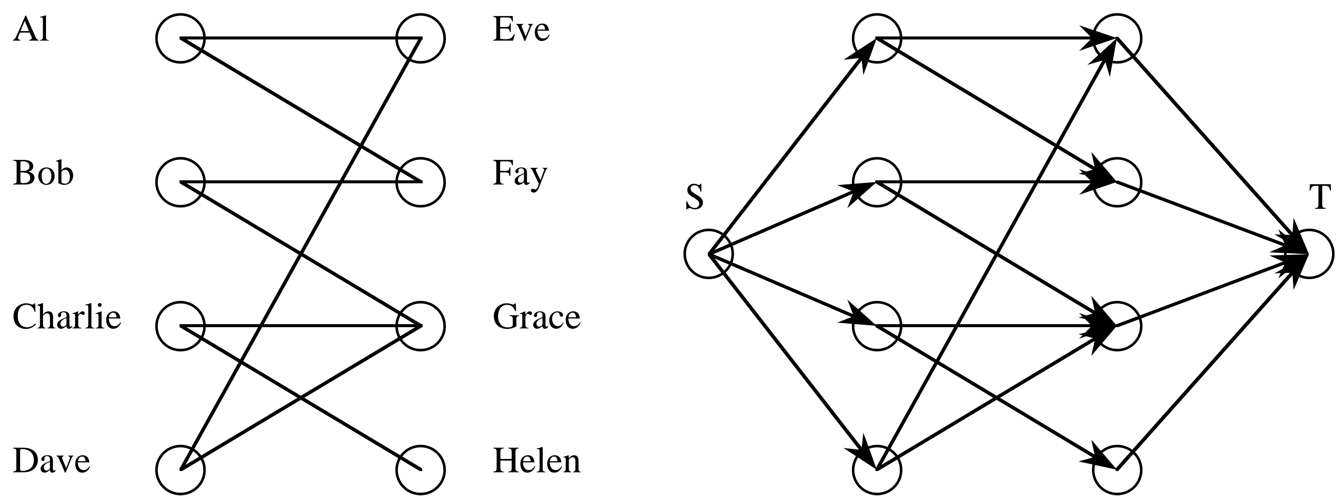

Figure 10.4: Reduction from matching to max-flow (all capacities are 1)

(some or all of) the variables be integers.