Network Flows, and Linear Programming Duality

1. Network Flows

Suppose that we are given the network of the Figure (top), 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.

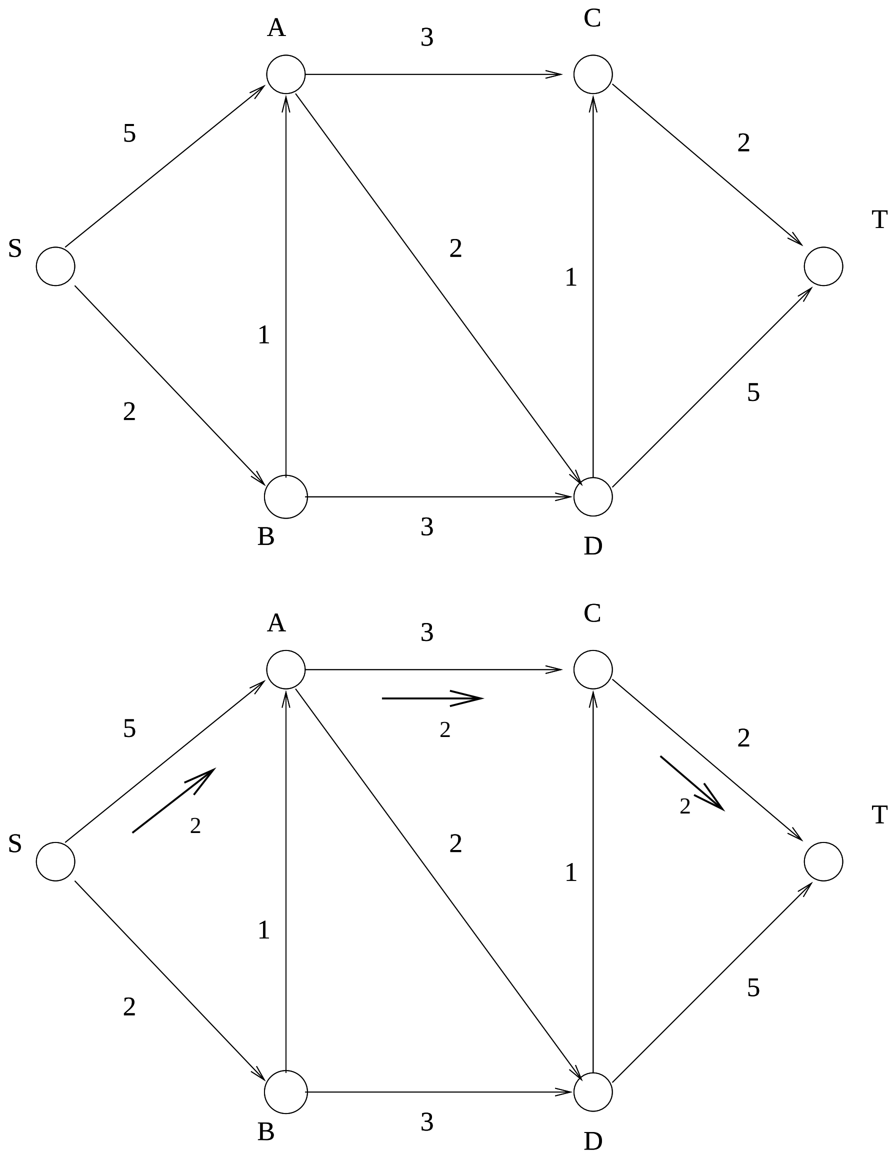

Figure 1: Max flow

More on the simplex algorithm:

So far you have only seen a vague description of the simplex algorithm: as an algorithm that walks from vertex to vertex of the convex polytope that forms the feasible region (satisfies all the constraints of the given linear program). Moreover, the algorithm performs the 'hill climbing' heuristic on these vertices — from its current vertex it moves to a vertex with a better value of the linear function that the linear program asks to optimize. What prevented us from describing the simplex method in more detail are the complications of the linear algebra necessary to describe how to move from one vertex to another in n dimensional Euclidean space $\mathcal{R}^n$. In this section, we will describe what the simplex algorithm does in the special case of network flows. It is quite remarkable that even though the simplex algorithm works purely by relying on the linear algebra of the given problem, when it is interpreted in the concrete example of *the network flow problem*, it can be described in terms of repeated depth-first searches from the source vertex.

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 the bottom of Figure 3).

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 the top of Figure 4.

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 4.

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 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 5. 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).

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 the algorithm just described.

6. Duality As it turns out, the max-flow min-cut theorem is a special case of a more general

phenomenon called *duality.* Basically, duality means that a maximization and a minimization problem have the property that any feasible solution of the min problem is greater than or equal any feasible solution of the max problem (see Figure). Furthermore, and more importantly, *they have teh same optimum.* We will see this in greater detail in the next lecture.

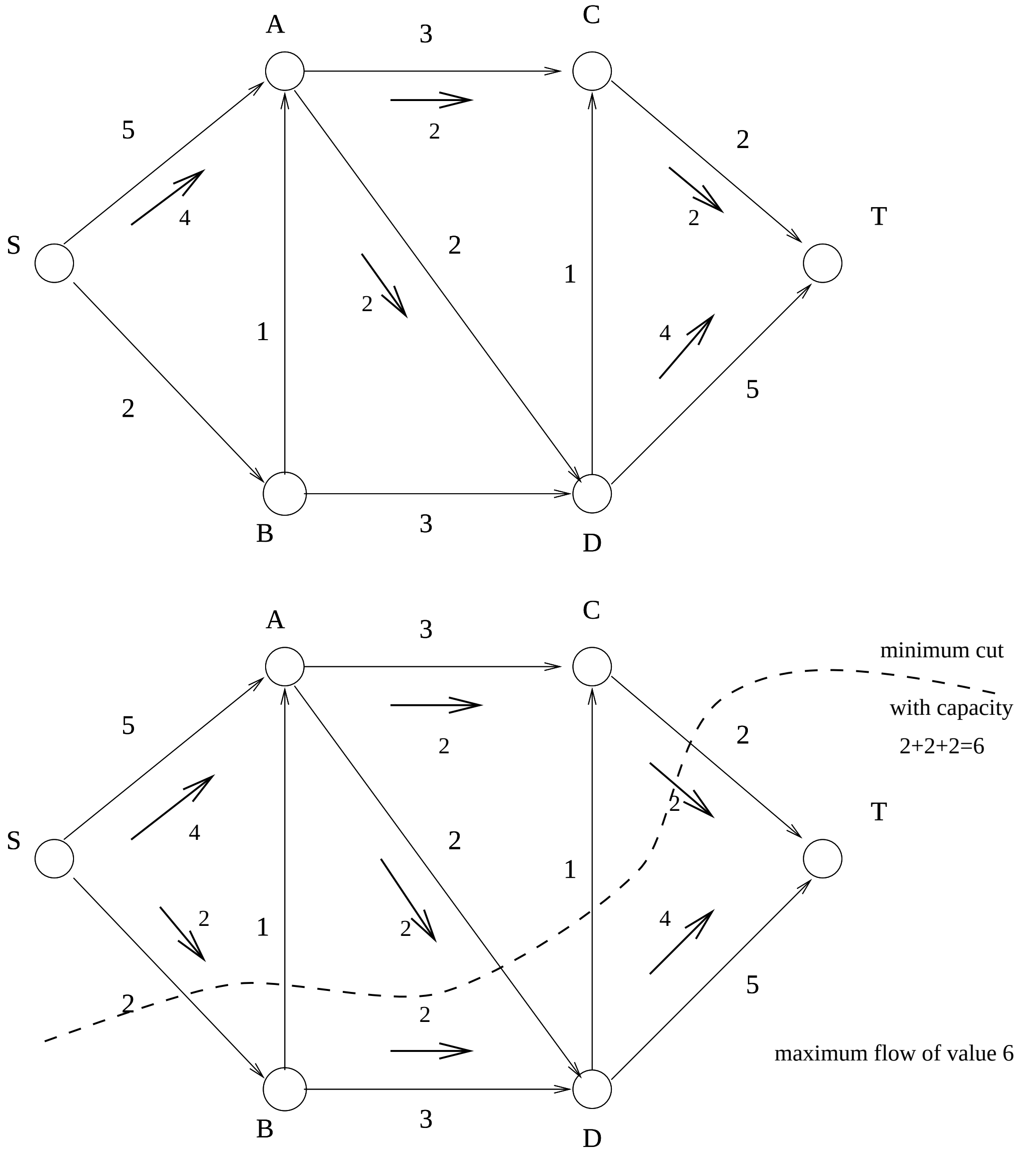

Figure 2: Max flow (continued)

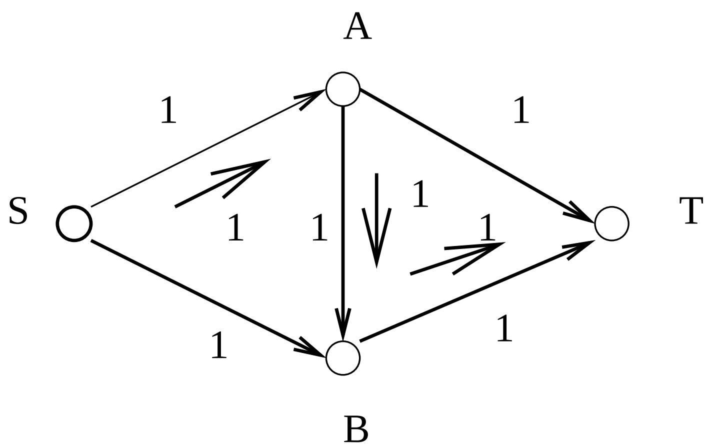

Figure 3: Flows may have to be canceled