CS124	Lecture 16

An introductory example

Suppose that a company that produces three products wishes to decide the level of production of each so as to maximize profits. Let x_1 be the amount of Product 1 produced in a month, x_2 that of Product 2, and x_3 that of Product 3. Each unit of Product 1 yields a profit of 100, each unit of Product 2 a profit of 600, and each unit of Product 3 a profit of 1400. There are limitations on x_1, x_2, and x_3 (besides the obvious one, that $x_1, x_2, x_3 \geq 0$). First, x_1 cannot be more than 200, and x_2 cannot be more than 300, presumably because of supply limitations. Also, the sum of the three must be, because of labor constraints, at most 400. Finally, it turns out that Products 2 and 3 use the same piece of equipment, with Product 3 using three times as much, and hence we have another constraint $x_2 + 3x_3 \leq 600$. What are the best levels of production?

We represent the situation by a *linear program*, as follows:

$$\begin{aligned} \max 100x_1 + 600x_2 + 1400x_3 & \\ x_1 &\leq 200 \\ x_2 &\leq 300 \\ x_1 + x_2 + x_3 &\leq 400 \\ x_2 + 3x_3 &\leq 600 \\ x_1, x_2, x_3 &\geq 0 \end{aligned}$$

The set of all *feasible* solutions of this linear program (that is, all vectors in 3-d space that satisfy all constraints) is precisely the polyhedron shown in Figure 16.1.

We wish to maximize the linear function $100x_1 + 600x_2 + 1400x_3$ over all points of this polyhedron. Geometrically, the linear equation $100x_1 + 600x_2 + 1400x_3 = c$ can be represented by a plane parallel to the one determined by the equation $100x_1 + 600x_2 + 1400x_3 = 0$. This means that we want to find the plane of this type that touches the polyhedron and is as far towards the positive orthant as possible. Obviously, the optimum solution will be a vertex (or the optimum solution will not be unique, but a vertex will do). Of course, two other possibilities with linear programming are that (a) the optimum solution may be infinity, or (b) that there may be no feasible solution at all.

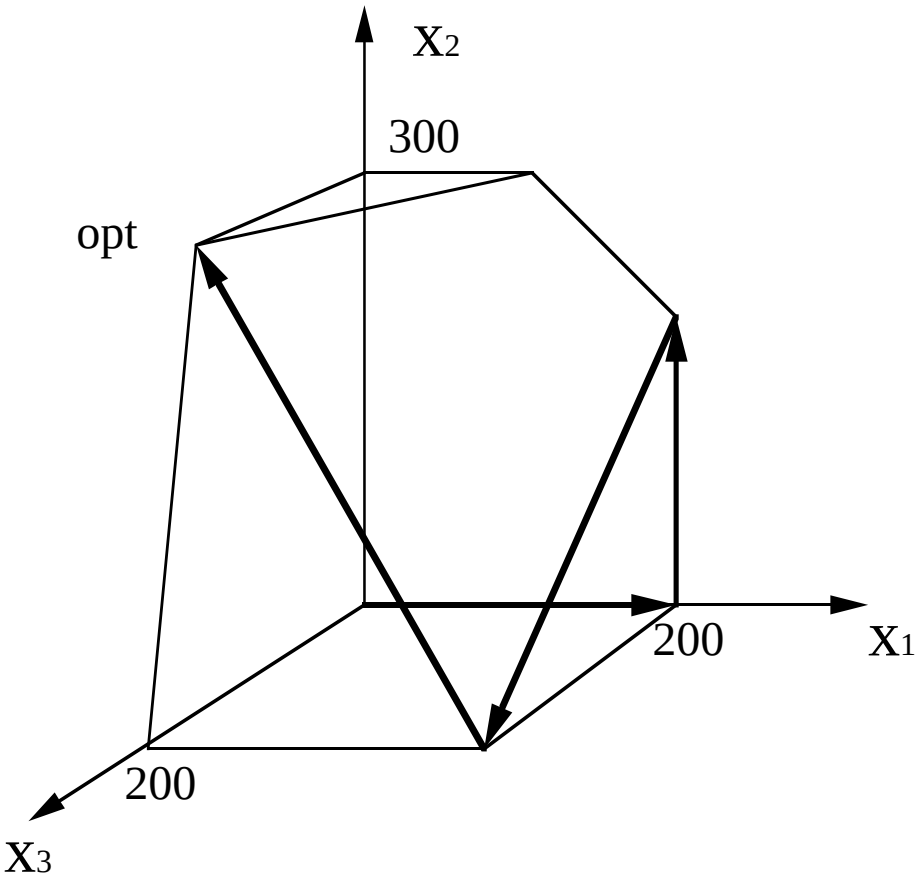

Figure 16.1: The feasible region

For this problem, an optimal solution exists, and moreover we shall show that it is easy to find.

Linear programs

Linear programs, in general, have the following form: there is an *objective function* that one seeks to optimize, along with *constraints* on the variables. The objective function and the constraints are all *linear* in the variables; that is, all equations have no powers of the variables, nor are the variables multiplied together. As we shall see, many problems can be represented by linear programs, and for many problems it is an extremely convenient representation. So once we explain how to solve linear programs, the question then becomes how to reduce other problems to linear programming (LP).

There are polynomial time algorithms for solving linear programs. In practice, however, such problems are solved by the *simplex method* devised by George Dantzig in 1947. The simplex method starts from a vertex (in this

case the vertex $(0,0,0)$) and repeatedly looks for a vertex that is adjacent, and has better objective value. That is, it is a kind of *hill-climbing* in the vertices of the polytope. When a vertex is found that has no better neighbor, simplex stops and declares this vertex to be the optimum. For example, in the figure one of the possible paths followed by simplex is shown. No known variant of the simplex algorithm has been proven to take polynomial time, and most of the variations used in practice have been shown to take exponential time on some examples. Fortunately, in practice, bad cases rarely arise, and the simplex algorithm runs extremely quickly. There are now implementations of simplex that solve routinely linear programs with *many thousands* of variables and constraints.

Of course, given a linear program, it is possible either that (a) the optimum solution may be infinity, or (b) that there may be no feasible solution at all. If this is the case, the simplex algorithm will discover it.

Reductions between versions of simplex

A general linear programming problem may involve constraints that are equalities or inequalities in either direction. Its variables may be nonnegative, or could be unrestricted in sign. And we may be either minimizing or maximizing a linear function. It turns out that we can easily translate any such version to any other. One such translation that is particularly useful is from the general form to the one required by simplex: *minimization, nonnegative variables, and equality constraints.*

To turn an inequality $\sum a_i x_i \leq b$ into an equality constraint, we introduce a new variable s (the *slack variable* for this inequality), and rewrite this inequality as $\sum a_i x_i + s = b, s \geq 0$. Similarly, any inequality $\sum a_i x_i \geq b$ is rewritten as $\sum a_i x_i - s = b, s \geq 0$; s is now called a *surplus* variable.

We handle an unrestricted variable x as follows: we introduce two nonnegative variables, x^+ and x^-, and replace x by $x^+ - x^-$ everywhere. The idea is that we let $x = x^+ - x^-$, where we may restrict both x^+ and x^- to be nonnegative. This way, x can take on any value, but there are only nonnegative variables.

Finally, to turn a maximization problem into a minimization one, we just multiply the objective function by -1.

A production scheduling example

We have the demand estimates for our product for all months of 1997, $d_i : i = 1, \ldots, 12$, and they are very uneven, ranging from 440 to 920. We currently have 30 employees, each of which produce 20 units of the product each month at a salary of 2,000; we have no stock of the product. How can we handle such fluctuations in demand? Three ways:

- overtime —but this is expensive since it costs 80% more than regular production, and has limitations, as workers can only work 30% overtime.
- hire and fire workers —but hiring costs 320, and firing costs 400.
- store the surplus production —but this costs 8 per item per month

This rather involved problem can be formulated and solved as a linear program. As in all such reductions, the crucial first step is defining the variables:

- Let w_0 be the number of workers we have the ith month —we have $w_0 = 30$.
- Let x_i be the production for month i.
- o_i is the number of items produced by overtime in month i.
- h_i and f_i are the number of workers hired/fired in the beginning of month i.
- s_i is the amount of product stored after the end of month i.

We now must write the constraints:

- $x_i = 20w_i + o_i$ —the amount produced is the one produced by regular production, plus overtime.
- $w_i = w_{i-1} + h_i - f_i, w_i \geq 0$ —the changing number of workers.
- $s_i = s_{i-1} + x_i - d_i \geq 0$ —the amount stored in the end of this month is what we started with, plus the production, minus the demand.
- $o_i \leq 6w_i$ —only 30% overtime.

Finally, what is the objective function? It is

$$\min 2000 \sum w_i + 400 \sum f_i + 320 \sum h_i + 8 \sum s_i + 180 \sum o_i,$$

where the summations are from $i = 1$ to 12.

A Communication Network Problem

We have a network whose lines have the bandwidth shown in Figure 16.2. We wish to establish three calls: one between A and B (call 1), one between B and C (call 2), and one between A and C (call 3). We must give each call at least 2 units of bandwidth, but possibly more. The link from A to B pays 3 per unit of bandwidth, from B to C pays 2, and from A to C pays 4. Notice that each call can be routed in two ways (the long and the short path), or by a combination (for example, two units of bandwidth via the short route, and three via the long route). Suppose we are a shady network administrator, and our goals is to maximize the network's income (rather than minimize the overall cost). How do we route these calls to maximize the network's income?

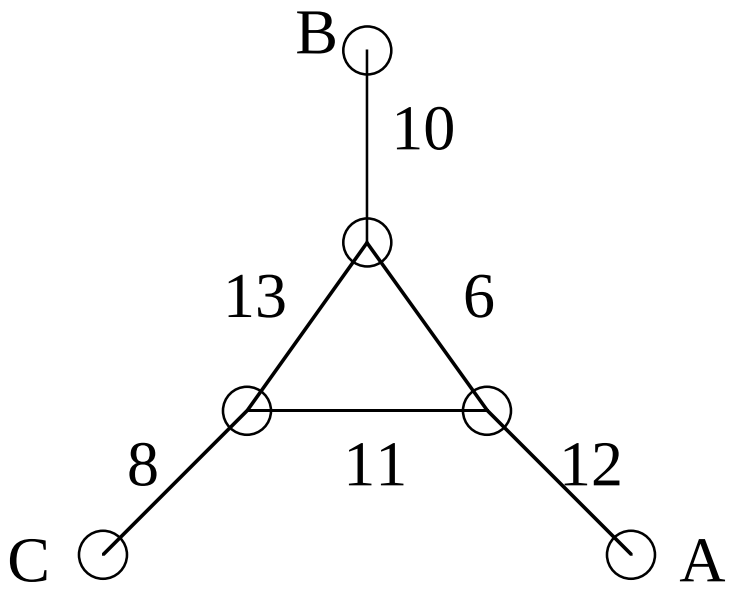

Figure 16.2: A communication network

This is also a linear program. We have variables for each call and each path (long or short); for example x_1 is the short path for call 1, and x_2' the long path for call 2. We demand that (1) no edge bandwidth is exceeded, and (2) each call gets a bandwidth of 2.

$$\begin{aligned} \max\ 3x_1 + 3x_1' + 2x_2 + 2x_2' + 4x_3 + 4x_3' & \\ x_1 + x_1' + x_2 + x_2' &\leq 10 \\ x_1 + x_1' + x_3 + x_3' &\leq 12 \\ x_2 + x_2' + x_3 + x_3' &\leq 8 \\ x_1 + x_2' + x_3' &\leq 6 \end{aligned}$$

$$\begin{aligned} x'_1 + x_2 + x'_3 &\leq 13 \\ x'_1 + x'_2 + x_3 &\leq 11 \\ x_1 + x'_1 &\geq 2 \\ x_2 + x'_2 &\geq 2 \\ x_3 + x'_3 &\geq 2 \\ x_1, x'_1 \ldots, x'_3 &\geq 0 \end{aligned}$$

The solution, obtained via simplex in a few milliseconds, is the following: $x_1 = 0, x'_1 = 7, x_2 = x'_2 = 1.5, x_3 = .5, x'_3 = 4.5$.

Question: Suppose that we removed the constraints stating that each call should receive at least two units. Would the optimum change?

Approximate Separation

An interesting last application: Suppose that we have two sets of points in the plane, the *black points* $(x_i, y_i) : i = 1, \ldots, m$ and the *white points* $(x_i, y_i) : i = m+1, \ldots, m+n$. We wish to separate them by a straight line $ax + by = c$, so that for all black points $ax + by \leq c$, and for all white points $ax + by \geq c$. In general, this would be impossible. Still, we may want to separate them by a line that minimizes the sum of the "displacement errors" (distance from the boundary) over all misclassified points. Here is the LP that achieves this:

$$\begin{array}{rc} \min e_1 & +e_2 + \ldots + e_m + e_{m+1} + \ldots + e_{m+n} \\ e_1 \geq & ax_1 + by_1 - c \\ e_2 \geq & ax_2 + by_2 - c \\ \vdots & \\ e_m \geq & ax_m + by_m - c \\ e_{m+1} \geq & c - ax_{m+1} - by_{m+1} \\ \vdots & \\ e_{m+n} \geq & c - ax_{m+n} - by_{m+n} \\ & e_i \geq 0 \end{array}$$

Network Flows

Suppose that we are given the network in top of Figure 16.3, 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.

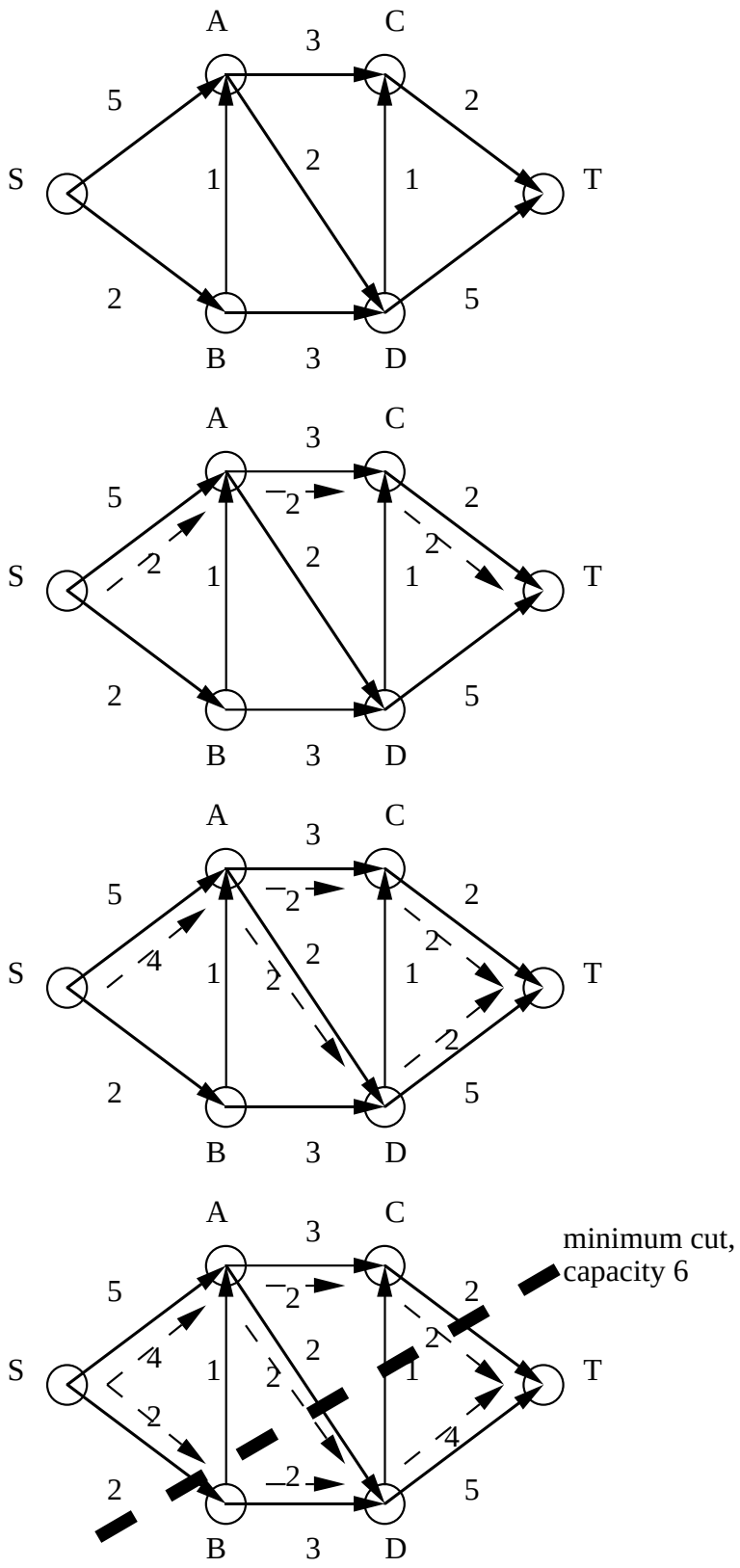

Figure 16.3: Max flow

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 16.3).

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 Figure 16.3.

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 16.3.

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 16.4. 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).

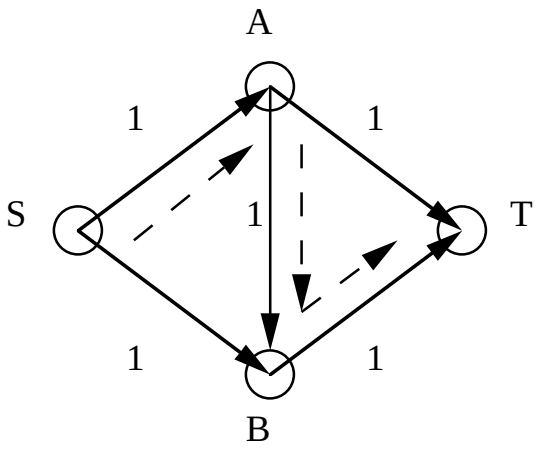

Figure 16.4: Flows may have to be canceled

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 16.3.

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 $O(E f^*)$, 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 the algorithm just described.