Linear Programming and Reductions

1 Linear Programming

A lot of the problems we want to solve are *optimization* tasks: the *shortest* path, the *cheapest* spanning tree, the *longest* increasing subsequence, and so on. In these problems, we seek a solution which (a) satisfies certain constraints (for instance, the path must lead from s to t, the tree must touch all nodes, the subsequence must be increasing); and (b) is the best possible, with respect to some well-defined criterion, amongst those that do.

Linear programming (or *LP)* describes a very general class of optimization problems in which both the constraints and the optimization criterion are *linear*. In an LP problem we are given a set of variables, and we want to assign values to them so as to (a) satisfy a set of linear equations and/or inequalities involving these variables; and (b) maximize a given linear objective function.

1.1 An introductory example

A company has two products, and wants to figure out how much to produce of each of them so as to maximize profits. Let's say it makes x_1 units of Product 1, at a profit of $\$100$ each, and x_2 units of Product 2, at a more substantial profit of $\$600$ apiece. If this were the end of the story, the most favorable option would be to focus entirely upon Product 2, but there are also some constraints on x_1 and x_2 that must be accommodated (besides the obvious one, $x_1, x_2 \geq 0$). 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 two can be at most 400, because of labor constraints (a fixed number of workers). What are the optimal levels of production?

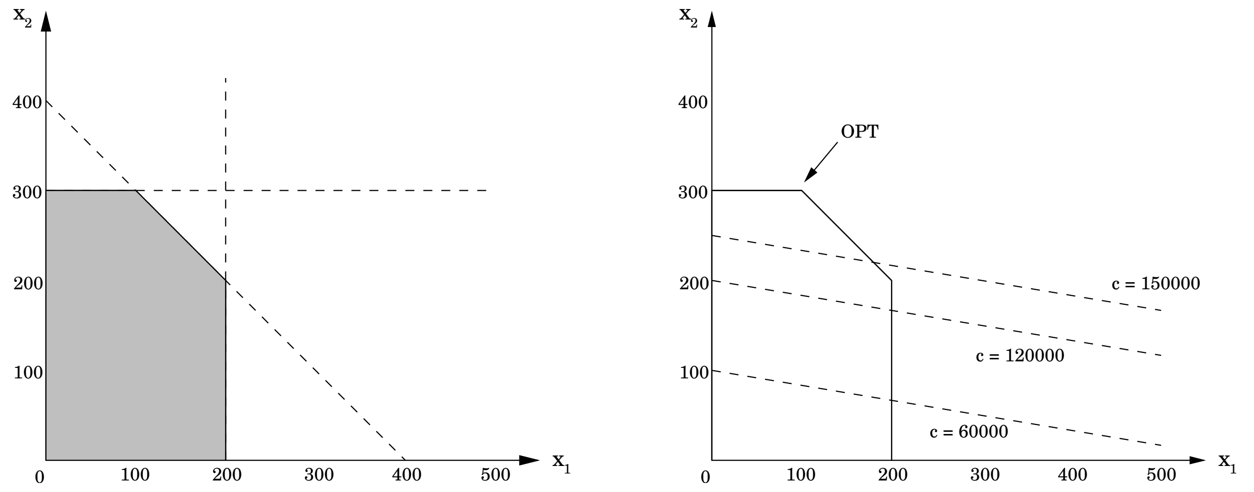
We represent the situation by a *linear program*, as follows.

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

A linear equation in x_1 and x_2 defines a line in the 2-d plane, and a linear inequality designates a *half-space*, the region on one side of the line. Thus the set of all *feasible solutions* of this linear program, that is, the points (x_1, x_2) which satisfy all constraints, is the intersection of five half-spaces. It is a polygon, shown in Figure 1.1.

We wish to maximize the linear objective function $100x_1 + 600x_2$ over all points in this polygon. The equation $100x_1 + 600x_2 = c$ defines a line of slope $-1/6$, and is shown in Figure 1.1 for selected values of c. As c increases, this "profit line" moves parallel to itself, up and to the right. Since the goal is to maximize c, we must move the line as far up and to the right as possible, while still touching the feasible region. The optimum solution will be the very last feasible point that the profit line sees, and must therefore be a vertex of the polygon, as shown in the figure. If the slope of the profit line were slightly different, -1 instead of $-1/6$, then its last contact with the polygon would be an entire edge rather than a single vertex. In this case,

Figure 1.1 *Left:* The feasible region for a linear program. *Right:* Contour lines of the objective function: $100x_1 + 600x_2 = c$ for different values of the profit c.



the optimum solution would not be unique, but there would certainly be an optimum vertex.

It is a general rule of linear programs that the optimum is achieved at a vertex of the feasible region, except in two special cases.

1. The program is *infeasible*, that is, the constraints are so tight that it is impossible to satisfy all of them. For instance,
$$x \leq 1, \quad x \geq 2.$$

2. The constraints are so loose that the feasible region is *unbounded*, and it is possible to achieve arbitrarily high objective values. For instance,

$$\max\ x_1 + x_2$$
$$x_1, x_2 \geq 0$$

Solving linear programs

Linear programs can be solved by the *simplex method*, devised by George Dantzig in 1947. We shall explain it in more detail later, but briefly, this algorithm starts at a vertex, in our case perhaps $(0, 0)$, and repeatedly looks for an adjacent vertex (connected by an edge of the feasible region) of better objective value. In this way it does *hill-climbing* on the vertices of the polygon, walking from neighbor to neighbor so as to steadily increase profit along the way. Here's a possible trajectory.

$$(0, 0) \;\rightarrow\; (200, 0) \;\rightarrow\; (200, 200) \;\rightarrow\; (100, 300).$$

Upon reaching a vertex that has no better neighbor, simplex declares it to be optimal and halts. Why does this local test imply global optimality? By simple geometry – think of the profit line passing through this vertex. Since all the vertex's neighbors lie below the line, the rest of the feasible polygon must lie also below this line.

The simplex algorithm is based on elementary principles, but an actual implementation involves a large number of tricky details, such as numerical precision. Luckily, these issues are very well understood and there are many professional, industrial-strength linear programming packages available. In a typical application, the main task is therefore to correctly express the problem as a linear program. The package then takes care of the rest.

Adding more products

Now let's extend our example somewhat. The company in question decides to grow by adding a third, more lucrative, product, which will bring a profit of $\$1400$ per unit. Let x_1, x_2, x_3 be the amounts of the three products. The old constraints on x_1 and x_2 persist, although the labor constraint now includes x_3 as well: the sum of all three variables can be at most 400. What's more, it turns out that Products 2 and 3 require the same piece of machinery, except that Product 3 uses it three times as much, which imposes another constraint $x_2 + 3x_3 \leq 600$. What are the best possible levels of production?

Here is the updated linear program.

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

The space of solutions is now three-dimensional. Each linear equation defines a plane in 3-d, and each inequality a half-space on one side of the plane. The feasible region is an intersection of seven half-spaces, a polyhedron (Figure 1.2).

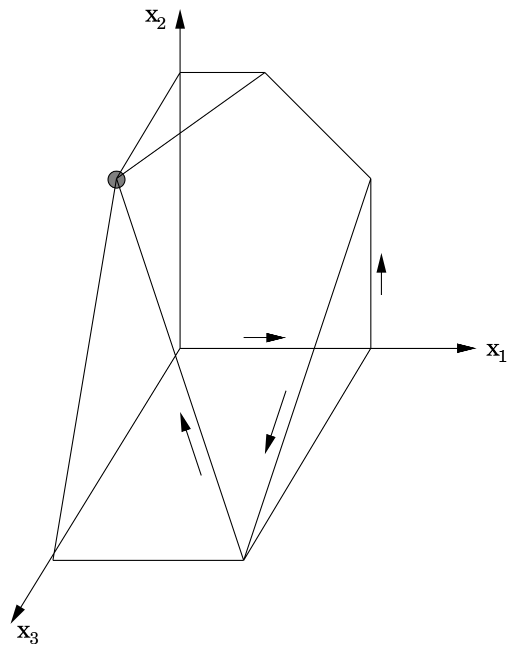
A profit of c corresponds to the plane $100x_1 + 600x_2 + 400x_3 = c$. As c increases, this profit-plane moves parallel to itself, further and further into the positive orthant until it no longer touches the feasible region. The point of final contact is the optimal vertex.

How would the simplex algorithm behave on this modified problem? As before, it would move from vertex to vertex, along edges of the polyhedron, increasing profit steadily. A possible trajectory is shown in Figure 1.2. Finally, upon reaching a vertex with no better neighbor, it would stop and declare this to be the optimal point. Once again by basic geometry, if all vertex's neighbors lie on one side of the profit-plane, then so must the entire polyhedron.

What if we add a fourth product, or hundreds of more products? Then the problem becomes high-dimensional, and hard to visualize. Simplex continues to work just fine in this general setting, but we can no longer rely upon simple geometric intuitions and therefore need to describe and justify the algorithm in more algebraic terms.

We will get to this later; meanwhile, let's look at a higher-dimensional application.

Figure 1.2 The feasible polyhedron for a three-variable linear program.



1.2 Application: production planning

Our analyst has just given us demand-estimates for our product for all months of the next calendar year: $d_1, d_2, \ldots, d_{12}$. Unfortunately, these demands are very uneven, ranging from 440 to 920.

Here's a quick snapshot of the company. We currently have 30 employees, each of whom produces 20 units of the product each month and gets a monthly salary of $2,000. We have no initial surplus of the product.

How can we handle the fluctuations in demand? There are three ways:

1. *Overtime*, but this is expensive since overtime pay is 80% more than regular pay. Also, workers can put in at most 30% overtime.

2. *Hiring and firing*, but these cost $320 and $400, respectively, per worker.

3. *Storing surplus production*, but this costs $8 per item per month. And we must end the year without any items stored.

This rather involved problem can be formulated and solved as a linear program!

A crucial first step is defining the variables.

$$\begin{aligned}
w_i &= \text{number of workers during } i^{\text{th}} \text{ month; } w_0 = 30. \\
x_i &= \text{number of items produced during the } i^{\text{th}} \text{ month.} \\
o_i &= \text{number of items produced by overtime in month } i. \\
h_i, f_i &= \text{number of workers hired/fired at the beginning of month } i. \\
s_i &= \text{amount stored at the end of month } i\text{; } s_0 = 0.
\end{aligned}$$

We must now write the constraints. The total amount produced per month consists of regular production plus overtime,

$$x_i = 20w_i + o_i,$$

(one constraint for each $i = 1, \ldots, 12$). The number of workers can potentially change at the start of each month, and can of course never be negative.

$$w_i = w_{i-1} + h_i - f_i, \quad w_i \geq 0, \quad f_i \geq 0, \quad h_i \geq 0.$$

The amount stored at the end of each month is what we started with, plus the amount we produced, minus the demand for the month.

$$s_i = s_{i-1} + x_i - d_i, \quad s_i \geq 0.$$

And overtime is limited,

$$0 \leq o_i \leq 6w_i.$$

Finally, what is the objective function? It is to minimize the total cost,

$$\min \quad 2000\sum_i w_i \, + \, 320\sum_i h_i \, + \, 500\sum_i f_i \, + \, 8\sum_i s_i \, + \, 180\sum_i o_i,$$

a linear function of the variables. Solving this linear program by simplex will give us the optimum business strategy for our company.

1.3 Application: a communication network problem

We are managing a network whose lines have the bandwidths shown in Figure 1.3, and we need to establish three connections:

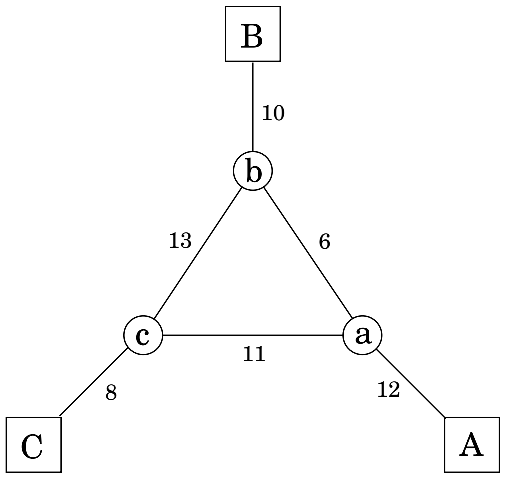
Connection 1: between users A and B

Connection 2: between B and C

Connection 3: between A and C

Each connection requires at least 2 units of bandwidth, but can be assigned more. Connection 1 pays \$3 per unit of bandwidth, and connections 2,3 pay \$2 and \$4 respectively. Notice that each connection can be routed in two ways, a long path and a short path, or by a combination: for example, two units of bandwidth via the short route, one via the long route. How do we route these calls to maximize our network's revenue?

Figure 1.3 A communication network; bandwidths are shown.



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

$$\begin{array}{rl}
\max \;\; 3x_1 + 3x_1' + 2x_2 + 2x_2' + 4x_3 + 4x_3' & \\
x_1 + x_1' + x_2 + x_2' \leq 10 & \text{[edge } (b,B)\text{]} \\
x_1 + x_1' + x_3 + x_3' \leq 12 & \text{[edge } (a,A)\text{]} \\
x_2 + x_2' + x_3 + x_3' \leq 8 & \text{[edge } (c,C)\text{]} \\
x_1 + x_2' + x_3' \leq 6 & \text{[edge } (a,b)\text{]} \\
x_1' + x_2 + x_3' \leq 13 & \text{[edge } (b,c)\text{]} \\
x_1' + x_2' + x_3 \leq 11 & \text{[edge } (a,c)\text{]} \\
x_1 + x_1' \geq 2 & [A - B] \\
x_2 + x_2' \geq 2 & [B - C] \\
x_3 + x_3' \geq 2 & [A - C] \\
x_1, x_1', x_2, x_2', x_3, x_3' \geq 0 &
\end{array}$$

The solution, obtained via simplex in a few microseconds, is

$$x_1 = 0,\; x_1' = 7,\; x_2 = x_2' = 1.5,\; x_3 = 0.5,\; x_3' = 4.5.$$

Interestingly, this optimal solution is not integral. Fractional values don't pose a problem in this particular example, but they would have been troublesome in production scheduling: how does one hire 4.5 workers, for instance? There is frequently a tension in LP between the ease of obtaining fractional solutions and the desirability of integer ones. As we shall see, finding the optimum integer solution of an LP is a very hard problem, called *integer linear programming.*

Here's a parting question for you to consider. Suppose we removed the constraint that each call should receive at least two units of bandwidth. Would the optimum change?

1.4 Linear programs in standard form

As evidenced in our examples, a general linear program can be either a maximization or a minimization problem and its constraints are allowed to be either equations or inequalities. We will now show that any such program can be reduced to a much more constrained *standard form*, in which the variables are all forced to be nonnegative, the constraints are all equations, and the objective function is to be minimized!

This transformation can be effected through a few simple steps.

1. To turn an inequality constraint like $\sum_{i=1}^{n} a_i x_i \leq b$ into an equation, introduce a new variable s and use:

$$\begin{aligned} \sum_{i=1}^{n} a_i x_i + s &= b \\ s &\geq 0. \end{aligned}$$

   s is called the *slack variable* for this inequality. Then a vector $(x_1, \ldots, x_n)$ satisfies the original inequality constraint if and only if there is some $s \geq 0$ for which it satisfies the new equality constraint.

2. To deal with a variable x that is unrestricted in sign, introduce two nonnegative variables, $x^+, x^- \geq 0$, and replace x, wherever it occurs in the constraints or the objective function, by $x^+ - x^-$. This way, x can take on any real value by appropriately adjusting the new variables. More precisely, any feasible solution to the original LP involving x can be mapped to a feasible solution of the new LP involving x^+, x^-, and vice versa.

3. Finally, to turn a maximization problem into a minimization, just multiply the coefficients of the objective function by -1.

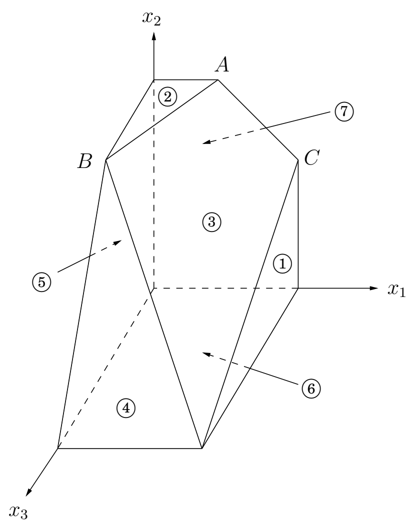
For example, our first linear program gets rewritten thus.

$$\begin{aligned} \min\ -100x_1 - 600x_2 \\ x_1 + s_1 = 200 \\ x_2 + s_2 = 300 \\ x_1 + x_2 + s_3 = 400 \\ x_1, x_2, s_1, s_2, s_3 \geq 0 \end{aligned}$$

When talking about linear programs in generality, it helps to collect all the variables into a vector x and to combine the individual linear equations into a single matrix-vector equation. Standard form is then

$$\begin{gathered} \max\ c^T x \\ Ax = b \\ x \geq 0. \end{gathered}$$

Figure 2.1 A polyhedron defined by seven inequalities.



$$\begin{aligned} \max \quad & 100x_1 + 600x_2 + 1400x_3 & \\ & x_1 \le 200 & ① \\ & x_2 \le 300 & ② \\ & x_1 + x_2 + x_3 \le 400 & ③ \\ & x_2 + 3x_3 \le 600 & ④ \\ & x_1 \ge 0 & ⑤ \\ & x_2 \ge 0 & ⑥ \\ & x_3 \ge 0 & ⑦ \end{aligned}$$

Some other formats are also common, such as

$$\max\ c^T x$$
$$Ax \le b$$
$$x \ge 0,$$

sometimes called *canonical form*. Do you see why all LPs can be written in this way?

2 The simplex algorithm

2.1 A rough outline

Suppose we are given an LP in canonical form,

$$\text{maximize } c^T x \text{ subject to } Ax \le b \text{ and } x \ge 0,$$

where x consists of n variables and A is an $m \times n$ matrix. The space of solutions is then n-dimensional, and the total number of inequalities is $m + n$. Each inequality defines a half-space in $\mathbf{R}^n$, and their intersection is the feasible region, a polyhedron.

Simplex finds the optimum feasible point by the following high-level strategy:

```
let $v$ be any vertex of the polyhedron
while there is a neighbor $v'$ of $v$ with better objective value:
    set $v = v'$
```

In our 2-d and 3-d examples, it was obvious what *vertex* and *neighbor* meant, so we didn't bother defining them formally. In higher-dimensional spaces, we need to be more careful since we can no longer visualize the feasible region.

Figure 2.1 recalls an earlier example. Looking at it closely, we see that *each vertex is the unique point at which some subset of hyperplanes meet*. Vertex A, for instance, is the sole point at which constraints ②, ③ and ⑦ are satisfied with equality. On the other hand, the hyperplanes corresponding to inequalities ④ and ⑥ do not define a vertex, because their intersection is not just a single point but an entire line.

Let's make this definition precise.

> Pick a subset of the inequalities. If there is a unique point which satisfies them with equality, and this point happens to be feasible, then it is a *vertex*.

How many inequalities are needed to uniquely identify a point? Since the space of solutions is n-dimensional, and each hyperplane constraint brings the dimension down by at most one, at least n inequalities are needed. Or equivalently, from an algebraic viewpoint, since we have n variables we need at least n linear equations if we want a unique solution. On the other hand, more than n equations are redundant: at least one of them must depend linearly on the others, and can be removed.

In other words, we have an association between vertices and subsets of inequalities. Let's number the inequalities: the first n are $x \geq 0$, and these are followed by $Ax \leq b$.

$$\boxed{\text{vertex } u} \longleftrightarrow \boxed{I_u \subset [n+m], \text{ a subset of } n \text{ inequalities}}$$

There is one tricky issue here. Although any subset of inequalities can correspond to at most one vertex, it is possible that a vertex might be generated by various different subsets. In the figure, vertex B is generated by $\{②, ③, ④\}$, but also by $\{②, ④, ⑤\}$. Such vertices are called *degenerate*, and require special consideration. Let's assume for the time being that they don't exist, and we'll return to them later.

Now that we've defined vertices, what about neighbors? Look at A and C: their corresponding hyperplanes $I_A = \{②, ③, ⑦\}$ and $I_C = \{①, ③, ⑦\}$ differ in just one constraint.

> Vertices u, v are ***neighbors*** if $|I_u \cap I_v| = n - 1$.

The edge between them is the intersection of these $n - 1$ hyperplanes (or more precisely, the feasible portion of the intersection).

2.2 Why does simplex work?

We've seen how simplex works on an LP in canonical form:

$$\text{maximize } c^T x \text{ subject to } Ax \leq b \text{ and } x \geq 0.$$

To prove its correctness, it helps to switch to standard form. This involves introducing m slack variables, one for each inequality in $Ax \leq b$. For simplicity, let's call these new variables $x_{n+1}, \ldots, x_{n+m}$. Notice how this numbering gives us a useful correspondence between the canonical and standard forms of our LP:

Canonical	*Standard*
Satisfy constraints $I \subset [n+m]$ with equality	Set variables $\{x_i : i \in I\}$ to zero

By definition, at any vertex $\tilde{x}$,

- the constraints $I_{\tilde{x}}$ are exactly satisfied, and these uniquely determine $\tilde{x}$.

- the remaining inequalities are not satisfied exactly (recall that we have disallowed degenerate vertices).

Paraphrasing, $\{\tilde{x}_i : i \in I_{\tilde{x}}\} = 0$ while $\{\tilde{x}_i : i \notin I_{\tilde{x}}\} > 0$. Think of the n variables $\{x_i : i \in I_{\tilde{x}}\}$ as being the *free variables*; the remaining *dependent variables* are a linear function of them by the uniqueness condition (check!). This means that the objective function $c^T x$ can also be expressed solely in terms of the free variables:

$$c^T x \;\equiv\; \tilde{c} + \sum_{i \in I_{\tilde{x}}} c_i x_i, \qquad (\dagger)$$

where $\tilde{c}, \{c_i\}$ are constants. Notice that $\tilde{c}$ is exactly the objective value at vertex $\tilde{x}$.

Suppose that some c_i is positive. Increasing the free variable $\tilde{x}_i$ (currently zero) will improve the objective value. However, as a side-effect, it will produce changes in the dependent variables, so we can only increase it while these other variables remain nonnegative. That is, we increase $\tilde{x}_i$ until the point where one of the dependent variables becomes zero. This brings us to a neighbor of $\tilde{x}$. [If we can keep increasing $\tilde{x}_i$ indefinitely, then the LP is unbounded.]

On the other hand, suppose none of the c_i are positive. Then the very form of the objective function $(\dagger)$ implies that $c^T x \leq \tilde{c}$, and so $\tilde{x}$ is the optimum feasible point.

2.3 Gaussian elimination

In fleshing out the outline of the simplex algorithm, there is one issue that arises immediately: under our algebraic definition, merely writing down the coordinates of a vertex involves solving a system of linear equations. How is this done?

We are given a system of n linear equations in n unknowns, say $n = 4$ and

$$\begin{array}{ccccccccc}
x_1 & & & - & 2x_3 & & & = & 2 \\
 & & x_2 & + & x_3 & & & = & 3 \\
x_1 & + & x_2 & & & - & x_4 & = & 4 \\
 & & x_2 & + & 3x_3 & + & x_4 & = & 5
\end{array}$$

The high school method for solving such systems is to repeatedly apply the following property.

> If we add a multiple of one equation to another equation, the overall system of equations remains equivalent.

For example, adding -1 times the first equation to the third one, we get the equivalent system

$$\begin{array}{ccccccccc}
x_1 & & & - & 2x_3 & & & = & 2 \\
 & & x_2 & + & x_3 & & & = & 3 \\
 & & x_2 & + & 2x_3 & - & x_4 & = & 2 \\
 & & x_2 & + & 3x_3 & + & x_4 & = & 5
\end{array}$$

This transformation is clever in the following sense: it *eliminates* the variable x_1 from the third equation, leaving just one equation with x_1. In other words, ignoring the first equation, we have a system of *three* equations in *three* unknowns: we decreased n by one! We can solve this smaller system to get x_2, x_3, x_4, and then plug these into the first equation to get x_1.

This suggests an algorithm (Figure 2.2) — once more due to Gauss. It does $O(n^2)$ work to reduce the problem size from n to $n-1$, and therefore has a running time of $T(n) = T(n-1) + O(n^2) = O(n^3)$.

Figure 2.2 Gaussian elimination: solving n linearly independent equations in n variables.

```
procedure gauss(E, X)
Input:   A system $E = \{e_1, \ldots, e_n\}$ of equations in $n$ unknowns $X = \{x_1, \ldots, x_n\}$:
         $e_1 : a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1;\ \cdots;\ e_n : a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n$.
Output:  A solution of the system, if one exists

if all coefficients $a_{i1}$ are zero:
   halt with message ''either infeasible or not linearly independent''
choose a nonzero coefficient $a_{p1}$, and swap equations $e_1, e_p$
for $i = 2$ to $n$:
   $e_i = e_i - (a_{i1}/a_{11}) \cdot e_1$
$(x_2, \ldots, x_n) =$ gauss $(E - \{e_1\}, X - \{x_1\})$
$x_1 = (b_1 - \sum_{j>1} a_{1j}x_j)/a_{11}$
return $(x_1, \ldots, x_n)$
```

2.4 Loose ends

There are several important issues in the simplex algorithm that we haven't yet mentioned.

The starting vertex. In our example we had the obvious starting point $(0, 0)$, but in general we can't hope to be so fortunate. It turns out that finding a starting vertex can be in turn be reduced to LP! To do this, alter the original linear program slightly:

- Create new artificial variables $z_1, \ldots, z_m \geq 0$.
- Add z_i to the left-hand side of the ith equation.
- Let the objective (to be minimized) be $\sum_{i=1}^{m} z_i$.

For this new LP, it's easy to come up with a starting vertex, namely the one with $z_i = b_i$ for all i and all other variables zero. Therefore we can solve it by simplex. If the optimum objective is positive, this means that the original linear program is infeasible. If it is zero, we have our starting vertex!

Unboundedness. In some cases an LP is unbounded, in that its objective function can be made arbitrarily small. This possibility is easily handled by simplex, by performing one small test while exploring each neighbor of a vertex.

Time complexity. As presented, simplex runs in time $O(Lm^4n)$, where L is the length of the longest objective-decreasing path of vertices. There are specialized examples where L is exponential in m and n! This is highly unusual, but even the m^4n factor is forbidding.

This m^4n can be improved to mn, making simplex a practical algorithm. One improvement is that, once we have solved a system of equations, we can solve another system that differs in one column in time $O(mn)$ instead of the usual $O(m^3)$. An extension of the same method makes it possible to quickly choose a neighboring vertex of better objective, rather than trying out all $O(mn)$ possibilities.

How much precision is needed?

An alarming thought: could vertices of the feasible region require so much precision to specify that just writing them down takes exponential time? As we learned in Chapter 1, this kind of question arises whenever numbers aren't constrained to a fixed size.
The input to simplex (in standard form, say) is a system of m linear equations, $Ax = b$, and we can assume that the entries of A, b are scaled to integers. Let γ be the maximum of their absolute values: how big can the vertices of the LP get, in terms of γ?
By definition, each vertex z is the solution to a set of equations $\tilde{A}z = b$ in which $\tilde{A}$ consists of m columns of A. *Cramer's rule* expresses this solution in closed form using determinants:

$$z_j = \det(\tilde{A}_j)/\det(\tilde{A}).$$

(Here $\tilde{A}_j$ is $\tilde{A}$ with the j^{th} column replaced by b.) Since A, b are integral, these two determinants must also be integral, so z_j is a rational number. One way to represent it is simply as a tuple of two integers, its numerator and denominator. How large can they be?
Recall that the determinant of an $m \times m$ matrix R can be broken down into

$$\det(R) = r_{11}\det(R_{11}) - r_{12}\det(R_{12}) + r_{13}\det(R_{13}) - \cdots \pm r_{1m}\det(R_{1m}),$$

where R_{ij} is the $(m-1) \times (m-1)$ submatrix that remains when row i and column j are removed from R. Thus $\det(R)$ is bounded by $U(m) \leq m\gamma U(m-1)$, which works out to $U(m) \leq m!\,\gamma^m$.
Returning to the expression for z_j, we see that its size is at most $2\log(m!\,\gamma^m)$, only about m times the size of γ and therefore polynomial.

Numerical issues are of great importance in optimization and scientific computing. Even if the input is a small and well-behaved quantity, the output or the intermediate computations may get perilously close to zero or to infinity, and need to be handled with care. Fortunately, these issues have been worked out thoroughly and incorporated into excellent software packages which shield them from our concern.

3 Flows in networks

3.1 Shipping oil

Figure 3.1(a) shows a directed graph representing a network of pipelines along which oil can be sent. The goal is to ship as much as oil as possible from the *source* s to the *sink* t. Each pipeline has a maximum *capacity* it can handle, and there are no opportunities for storing oil en route. Figure 3.1(b) shows a possible *flow* from s to t, which ships 7 units in all. Is this the best that can be done?

3.2 Maximizing flow

The networks we are dealing with consist of: a directed graph $G = (V, E)$; two special nodes $s, t \in V$, which are, respectively, a source and sink of G; and *capacities* $c_e > 0$ on the edges.

We would like to send as much oil as possible from s to t without exceeding the capacity of any of the edges. A particular shipping scheme is called a *flow*, and is formally a function $f : E \to \mathbf{R}$ with the following two properties.

1. It doesn't violate edge capacities: $0 \le f_e \le c_e$ for all $e \in E$.
2. For all nodes u except s, t, the amount of flow entering u equals the amount leaving u:

$$\sum_{(w,u)\in E} f_{wu} \quad = \quad \sum_{(u,z)\in E} f_{uz}.$$

   In other words, flow is *conserved*.

The *size* of a flow is the total quantity sent from s to t, and by the conservation principle, is equal to the quantity leaving s:

$$\text{size}(f) = \sum_{(s,u)\in E} f_{su}.$$

The upshot is that we want to assign values to $\{f_e : e \in E\}$ which will satisfy a set of linear constraints and maximize a linear objective function. In other words, the maximum flow problem reduces to linear programming! Simplex will correctly solve the problem, and will confirm that in our example, a flow of 7 is in fact optimal.

3.3 A closer look at the algorithm

A careful analysis of the behavior of simplex on flow LPs (which we will not attempt here) reveals that its pattern of incremental improvements is extremely simple.

> Start with zero flow.
>
> Repeat: choose an appropriate path from s to t, and increase flow along this path as much as possible.

Figure 3.2 shows a small example in which simplex halts after two iterations. The final flow has size two, which is easily seen to be optimal.

There is just one complication. What if we had initially chosen a different path, the one in Figure 3.3(a)? This gives only one unit of flow and yet seems to block all other paths. Simplex

Figure 3.1 (a) A network with edge capacities. (b) A *flow* in the network.

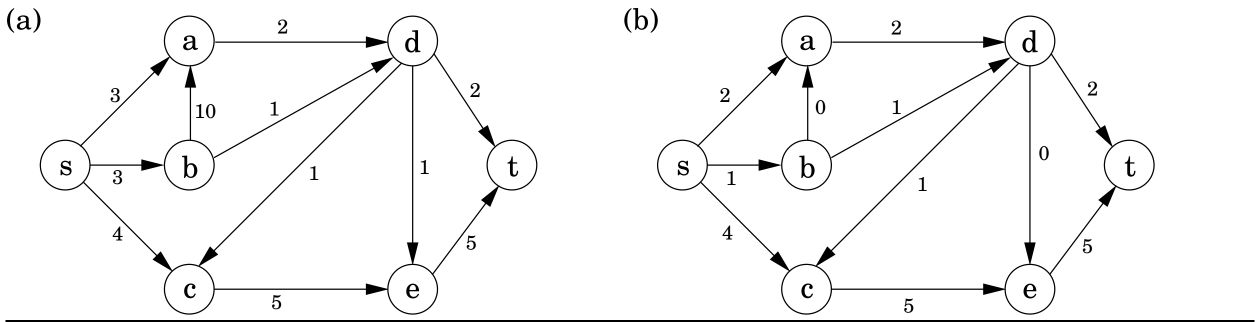


Figure 3.2 An illustration of the max-flow algorithm. (a) A toy network. (b) The first path chosen. (c) The second path chosen. (d) The final flow.

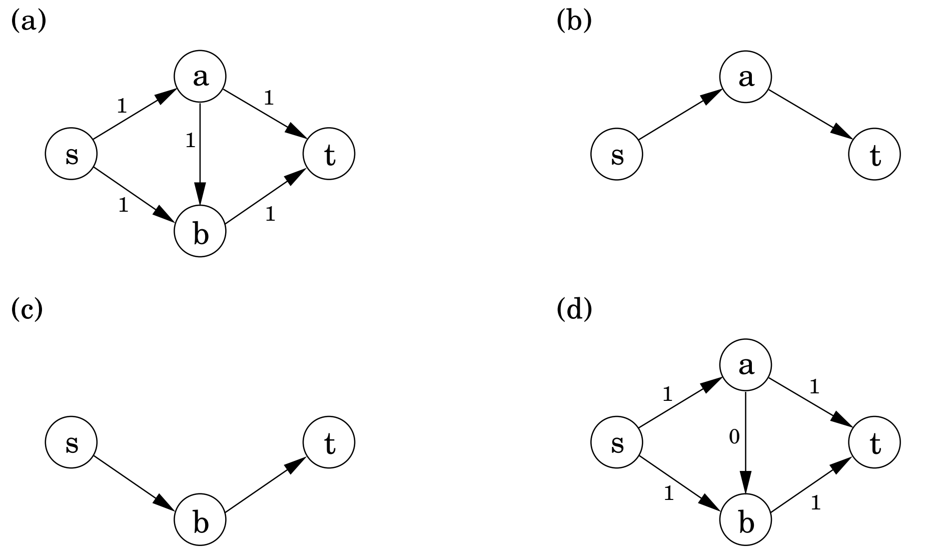
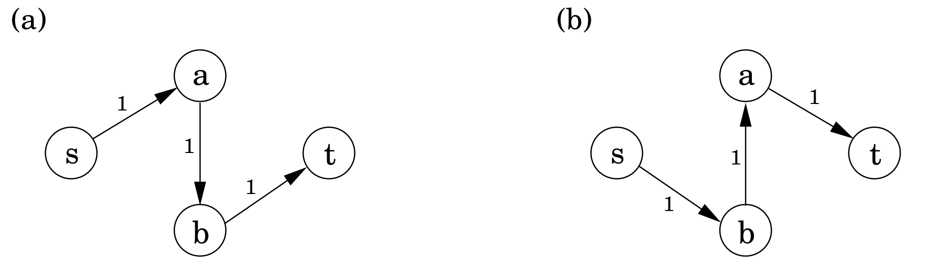


Figure 3.3 (a) We could have chosen this path first. (b) In which case, we would have to allow this as a second path.



gets around this problem by also allowing paths to *cancel existing flow*. In this particular case, it would subsequently choose path 3.3(b). Edge (b, a) of this path isn't in the original network, and has the effect of canceling flow previously assigned to edge (a, b).

To summarize, simplex looks for an $s - t$ path whose edges (u, v) can be of two types:

1. (u, v) is in the original network, and is not yet at full capacity;

2. the reverse edge (v, u) is in the original network, and there is some flow along it.

If the current flow is f, then in the first case edge (u, v) can handle upto $c_{uv} - f_{uv}$ additional units of flow, and in the second case, upto f_{vu} additional units (cancel all or part of the existing flow on (v, u)).

Finding a suitable $s-t$ path takes linear time with, say, depth-first search. The overall time complexity of simplex therefore hinges upon the number of iterations, and in the Exercises, we will see how to ensure that this is always polynomial. Figure 3.4 illustrates the algorithm on our oil example.

3.4 A certificate of optimality

Now for a truly remarkable fact: not only does simplex correctly compute a maximum flow, but it also generates a short proof of the optimality of this flow!

Let's see an example of what this means. Partition the nodes of the oil network (Figure 3.1) into groups $L = \{s, a, b\}$ and $R = \{c, d, e, t\}$. Any oil transmitted must pass from L to R. Therefore, no flow can possibly exceed the total capacity of the edges from L to R, which is 7. But this means that the flow we found earlier, of size 7, must be optimal!

More generally, a *cut* partitions the vertices into two disjoint groups L and R such that s is in L and t is in R. Its capacity is the total capacity of the edges from L to R, and as argued above, is an upper bound on *any* flow:

> Pick any flow f and any cut (L, R). Then size(f) $\leq$ capacity(L, R).

Some cuts are large and give loose upper bounds – cut $(\{s, b, c\}, \{a, d, e, t\})$ has a capacity of 19. But there is also a cut of capacity 7, which is effectively a *certificate of optimality* of the maximum flow. This isn't just a lucky property of our oil network; such a cut *always* exists.

> *Max-flow min-cut theorem.* The size of the maximum flow in a network equals the capacity of the smallest cut.

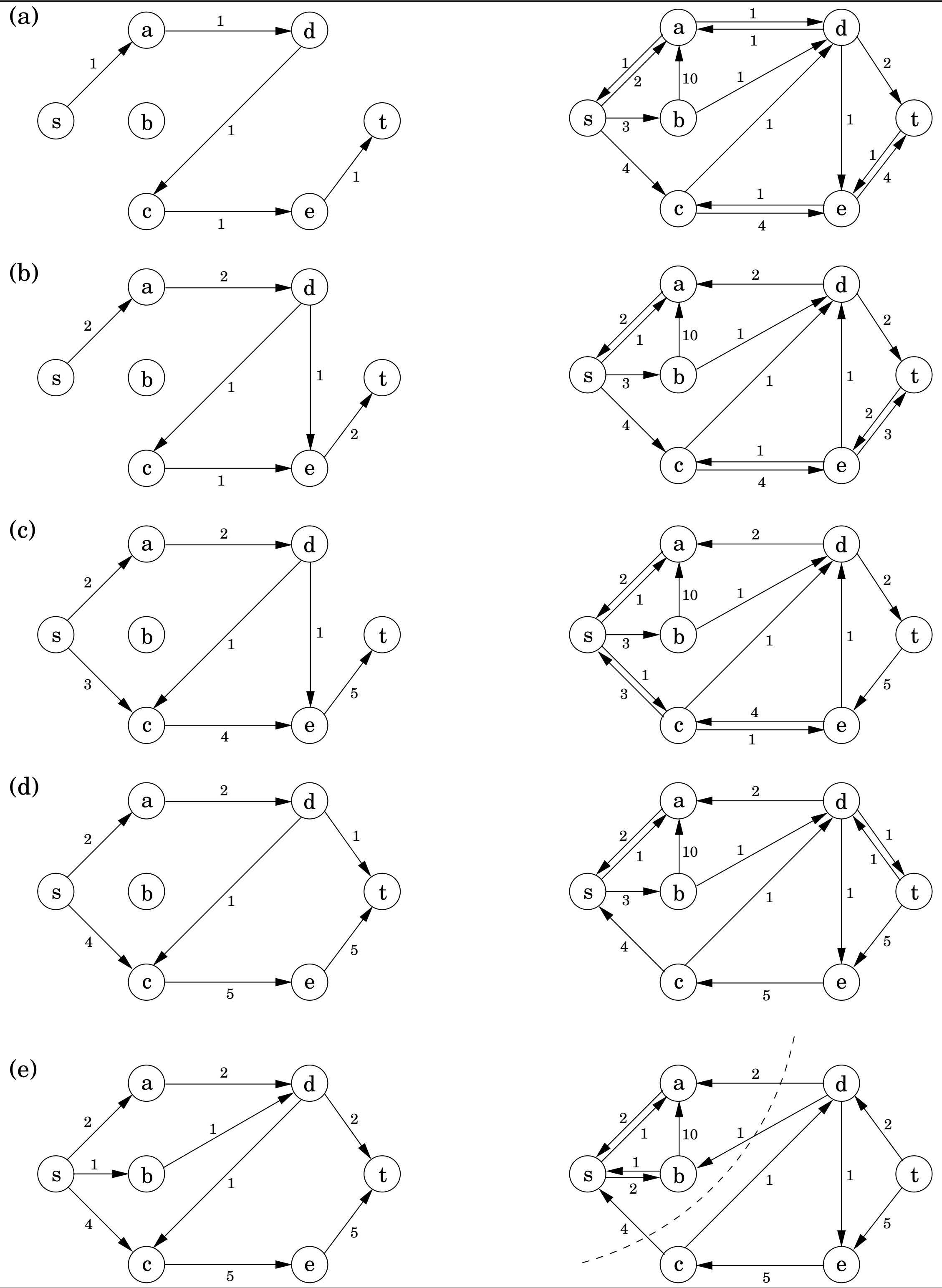
Moreover the simplex algorithm automatically finds this cut as a by-product!

Let's see why this is true. Suppose f is the final flow when simplex terminates. We know that node t is no longer reachable from s using the two types of edges we listed above. Let L be the nodes that are reachable from s in this way, and let $R = V - L$ be the rest of the nodes. Then (L, R) is a cut in the graph, and we claim

$$\text{size}(f) \;=\; \text{capacity}(L, R).$$

To see this, observe that because there is no viable $s - t$ path, any edge going from L to R must be at full capacity (in the current flow f), and any edge from R to L must have zero flow. Therefore the flow across (L, R) is exactly the capacity of the cut.

Figure 3.4 The max-flow algorithm applied to the network of Figure 3.1. At each iteration, the current flow is shown on the left and the residual graph is shown on the right. Paths chosen: (a) s, a, d, c, e, t; (b) s, a, d, e, t; (c) s, c, e, t; (d) s, c, e, d, t; (e) s, b, d, t.



4 Duality

We have seen that in networks, flows are smaller than cuts, but the maximum flow and minimum cut exactly coincide and each is therefore a certificate of the other's optimality. This *duality* between the two problems is a general property of linear programs. Every linear maximization problem has a dual minimization problem, and they relate to each other in the same way as flows and cuts.

To understand duality, let's start with a simple LP:

$$\begin{array}{rrcrcll}
\max & x_1 & + & 2x_2 & & & \\
& x_1 & & & \leq & 10 & \cdots \text{①} \\
& & & x_2 & \leq & 5 & \cdots \text{②} \\
& -3x_1 & + & 4x_2 & \leq & 4 & \cdots \text{③} \\
& x_1 & + & x_2 & = & 8 & \cdots \text{④} \\
& & & x_1, x_2 & \geq & 0 &
\end{array}$$

After playing with it a bit, we find a solution that looks pretty good, $(x_1, x_2) = (4, 4)$. This is definitely feasible, but how can we tell if its objective value, $x_1 + 2x_2 = 12$, is *optimal*? Well, the first two inequalities tell us that we certainly can't hope for a value of more than 20:

$$\begin{array}{ccc l}
x_1 & \leq & 10 & \cdots \text{①} \\
2x_2 & \leq & 10 & \cdots 2 \times \text{②} \\
\hline
x_1 + 2x_2 & \leq & 20 &
\end{array}$$

Looking a little more closely, we can get an even better upper bound of 13.

$$\begin{array}{ccc l}
x_2 & \leq & 5 & \cdots \text{②} \\
x_1 + x_2 & = & 8 & \cdots \text{④} \\
\hline
x_1 + 2x_2 & \leq & 13 &
\end{array}$$

We're getting very close.... Can we bring this upper bound down to 12, and thereby prove the optimality of our candidate solution, by looking at some other linear combination of the constraints? What is the *tightest* upper bound obtainable in this way?

Interestingly enough, this problem can itself be cast as a linear program. Let y_1, y_2, y_3 be the multipliers of the three inequalities, and z that of the equation. The resulting linear combination, $y_1 \times \text{①} + y_2 \times \text{②} + y_3 \times \text{③} + z \times \text{④}$, is

$$y_1 x_1 + y_2 x_2 + y_3(-3x_1 + 4x_2) + z(x_1 + x_2) \quad \leq \quad 10y_1 + 5y_2 + 4y_3 + 8z.$$

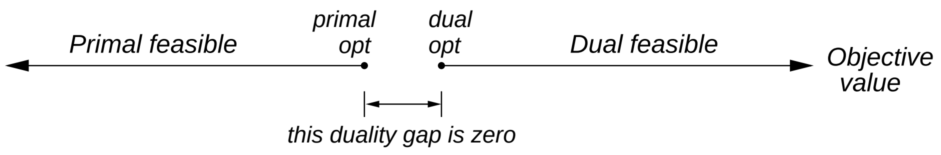
Here, the y's need to be nonnegative (other the sign of the inequality gets flipped), but z is unrestricted. In order for the left-hand side to be an upper bound on the objective function $x_1 + 2x_2$, we need to make sure that the coefficient of x_1 is at least 1,

$$y_1 - 3y_3 + z \geq 1,$$

and that the coefficient of x_2 is at least 2,

$$y_2 + 4y_3 + z \geq 2.$$

Figure 4.1 By design, dual feasible values $\geq$ primal feasible values. The duality theorem tells us that moreover their optima coincide.



The dual optimization task is then to find the linear combination which gives the tightest (that is, smallest) upper bound. It is an LP!

$$\begin{array}{lrrrrrrrrr}
\min & 10y_1 & + & 5y_2 & + & 4y_3 & + & 8z & & \\
& y_1 & & & - & 3y_3 & + & z & \geq & 1 \\
& & & y_2 & + & 4y_3 & + & z & \geq & 2 \\
& & & & & & & y_1, y_2, y_3 & \geq & 0
\end{array}$$

By design, any feasible value of this *dual* LP is an upper bound on the original *primal* LP. So if we somehow find a pair of primal and dual feasible values which are equal, then they must both be optimal. Here is just such a pair:

$$\text{Primal}: (x_1, x_2) = (4, 4); \quad \text{Dual}: (y_1, y_2, y_3, z) = (0, 0, 1/7, 10/7).$$

They both have value 12, and therefore certify each other's optimality (Figure 4.1).

This wasn't just a lucky example, but a general phenomenon. Although we've only talked about the dual of one particular LP, the same steps can be followed mechanically to find the dual of any linear program. This generic transformation is summarized in Figure 4.2. In the case of maximum flow LPs, it is possible to assign interpretations to the dual variables which show the dual to be none other than the minimum cut problem (see Exercise ???). The relation between flows and cuts is therefore just a specific instance of the following *duality theorem*.

> If a linear program has a bounded optimum, then so does its dual, and the two optimum values coincide.

We won't get into the proof of this, but it falls out of the simplex algorithm, in much the same way as the max-flow min-cut theorem fell out of the analysis of the max-flow algorithm.

5 Games

We can represent various conflict sitations in life by *matrix games*. For example, the schoolyard *rock-paper-scissors* game is specified by the *payoff matrix* below. There are two players, called Row and Column, and they each pick a move from $\{r, p, s\}$. Then they look up the matrix entry corresponding to their moves, and Column pays Row this amount. It is Row's gain and Column's loss.

$$G = \begin{array}{|c|rrr|}
\hline
 & r & p & s \\
\hline
r & 0 & -1 & 1 \\
p & 1 & 0 & -1 \\
s & -1 & 1 & 0 \\
\hline
\end{array}$$

Figure 4.2 Constructing the dual. The generic primal LP shown here has both inequalities and equations, over a variety of either nonnegative or unrestricted variables. Its dual has a nonnegative variable for each of the $|M|$ inequalities and an unrestricted variable for each of the $|M'|$ equations. A_j denotes the j^{th} column of the $(|M|+|M'|) \times (|N|+|N'|)$ matrix whose rows are the a_i (see Exercise ???).

Primal		***Dual***
$\max c^T x$		$\min y^T b$
$a_i^T x \leq b_i$	$i \in M$	$y_i \geq 0$
$a_i^T x = b_i$	$i \in M'$	y_i unrestricted
x_j unrestricted	$j \in N$	$y^T A_j = c_j$
$x_j \geq 0$	$j \in N'$	$y^T A_j \geq c_j$

Now suppose the two of them play this game repeatedly. If Row always makes the same move, Column will quickly catch on and will always play the countermove, winning every time. Therefore Row should mix things up: we can model this by allowing Row to have a *mixed strategy*, in which on each turn she plays r with probability x_1, p with probability x_2, and s with probability x_3. This strategy is specified by the vector (x_1, x_2, x_3), positive numbers which add up to one. Similarly, Column's mixed strategy is some (y_1, y_2, y_3).[1]

On any given round of the game, there is an $x_i y_j$ chance that Row and Column will play the i^{th} and j^{th} moves, respectively. Therefore the *expected* (average) payoff is

$$\sum_{i,j} x_i y_j G_{ij}.$$

For instance, if Row plays the "completely random" strategy $(1/3, 1/3, 1/3)$, the payoff is

$$\frac{1}{3} \sum_j y_j G_{ij} = \frac{1}{3} \left((y_1 + y_2 + y_3) \cdot 0 + (y_3 + y_1 + y_2) \cdot 1 + (y_2 + y_3 + y_1) \cdot -1 \right) = 0,$$

no matter what Column does. And vice versa. Moreover, since either player can force a payoff of zero, neither one can hope for any better.

Let's think about this in a slightly different way, by considering two scenarios:

1: First Row announces her strategy, then Column picks his.

2: First Column announces his strategy, then Row chooses hers.

We've seen that because of the symmetry in rock-paper-scissors, the average payoff is the same (zero) in either case if both parties play optimally. In general games, however, we'd expect the first option to favor Column, since he knows Row's strategy and can fully exploit it while choosing his own. Likewise, we'd expect the second option to favor Column. Amazingly, this is not the case: if both play optimally, then it doesn't hurt a player to announce his or her strategy in advance! What's more, this remarkable property is a consequence of LP duality.

[1]Also of interest are scenarios in which players alter their strategies from round to round, but these can get very complicated and are a vast subject unto themselves.

A simple example will make this clear. Imagine a *Presidential Election* scenario in which there are two candidates for office, and the moves they can make correspond to campaign issues on which they can focus (the initials stand for "economy," "society," "morality," and "tax-cut"). The payoff entries are millions of votes lost by Column.

$$G = \begin{array}{|c|rr|} \hline & m & t \\ \hline e & 3 & -1 \\ s & -2 & 1 \\ \hline \end{array}$$

Suppose Row announces that she will play the mixed strategy $(0.5, 0.5)$. What should Column do? Move m will incur an expected loss of 0.5, while t will incur an expected loss of 0. The best response of Column is therefore the *pure* strategy $(0, 1)$. More generally, if Row's strategy x is fixed, there is always a pure strategy that is optimal for Column, and it is found by comparing the options $\sum_i G_{ij} x_i$ for different j, and picking the smallest (since the entries denote Column's loss). Therefore, if Row is forced to announce x before Column plays, she should choose *defensively*, that is, maximize her payoff under Column's best response,

$$\max_{\{x_i\}} \min_j \sum_i G_{ij} x_i.$$

This choice of x_i's gives Row the best possible *guarantee* about her expected payoff. In the election example, finding these x_i's corresponds to maximizing $\min\{3x_1 - 2x_2, -x_1 + x_2\}$, which translates into the following LP:

$$\begin{array}{rcrcrcl} & & \max & & z & & \\ -3x_1 & + & 2x_2 & + & z & \leq & 0 \\ x_1 & - & x_2 & + & z & \leq & 0 \\ x_1 & + & x_2 & & & = & 1 \\ & & & & x_1, x_2 & \geq & 0 \end{array}$$

Symmetrically, if Column had to announce his strategy first, he would be best off choosing the mixed strategy y that minimized his loss under Row's best response, in other words,

$$\min_{\{y_j\}} \max_i \sum_j G_{ij} y_j.$$

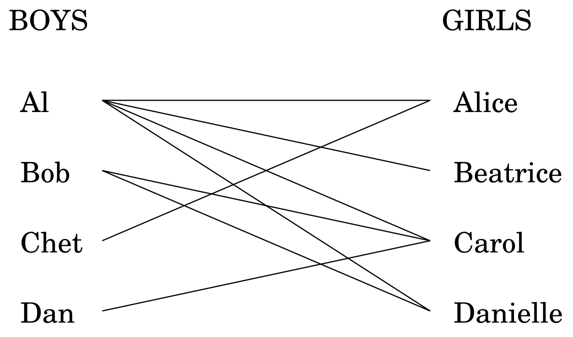
In our example, Column would minimize $\max\{3y_1 - y_2, -2y_1 + y_2\}$, which in LP form is:

$$\begin{array}{rcrcrcl} & & \min & & w & & \\ -3y_1 & + & y_2 & + & w & \geq & 0 \\ 2y_1 & - & y_2 & + & w & \geq & 0 \\ y_1 & + & y_2 & & & = & 1 \\ & & & & y_1, y_2 & \geq & 0 \end{array}$$

The crucial observation now is that *these two LP's are dual to each other* (see Figure 4.1), and hence have the same optimum, call it V.

Let us summarize: By solving an LP, Row can guarantee an expected gain of at least V, and by solving the dual LP, Column can guarantee an expected loss of at most the same value.

Figure 6.1 An edge between two people means they like each other. Is it possible to pair everyone up happily?



It follows that this is the uniquely defined optimal play; *a priori* it wasn't even certain that such a play exists. V is called *the value* of the game. In this case, it is $1/7$, and is realized when Row plays her optimum mixed strategy $(3/7, 4/7)$, and Column plays his optimum mixed strategy $(2/7, 5/7)$.

The existence of mixed strategies that are optimal for both players and achieve the same value is a fundamental result in Game Theory called *the min-max theorem*. It can be written in equations as follows:

$$\max_x \min_y \sum_{i,j} x_i y_j G_{ij} \quad = \quad \min_y \max_x \sum_{i,j} x_i y_j G_{ij}.$$

This is surprising, because the left-hand side, in which Row has to announce her strategy first, should presumably be better for Column than the right-hand side, in which he has to go first. Duality equalizes the two, as it does with maximum flows and minimum cuts.

6 Reductions

Sometimes the cleverest way to solve a new problem is by realizing that with a little pre- and post-processing it can be rewritten in the form of something more familiar.

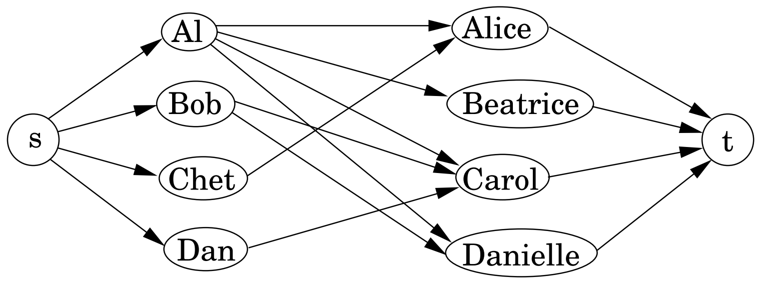
Specifically, we say that *Problem A reduces to Problem B* if any algorithm for B can be used to solve A as well. For instance, we showed earlier in this chapter that the maximum flow problem reduces to linear programming. We have also encountered reductions in numerous other contexts. While studying dynamic programming, we saw that computing the longest increasing subsequence reduces to finding longest paths in dags, and that in turn finding the longest path in a dag reduces to finding the shortest path (by making all edge weights -1).

In this last case, we can *compose* the two reductions: if Problem A reduces to B, and B to C, then effectively A reduces to C. We now turn to another such example.

6.1 Matching

Figure 6.1 shows a bipartite graph with four nodes on the left representing boys and four nodes on the right representing girls. There is an edge between a boy and girl if they like each

Figure 6.2 A matchmaking network. Each edge has a capacity of one.



other (for instance, Al likes all the girls). Is it possible to choose couples so that everyone has exactly one partner, and it is someone they like? In graph-theoretic jargon, is there a *perfect matching*?

This matchmaking game can be reduced to the maximum flow problem, and thereby to linear programming! Create a new source node, s, with outgoing edges to all the boys; a new sink node, t, with incoming edges from all the girls; and direct all the edges in the original bipartite graph from boy to girl (Figure 6.2). Finally, give every edge a weight of one. Then there is a perfect matching if and only if this network has a flow whose size equals the number of couples. Can you find such a flow in the example?

Actually, the situation is slightly more complicated than stated above: what is easy to see is that the optimum *integer-valued* flow corresponds to the optimum matching. We would be at a bit of a loss interpreting a flow that ships 0.7 units along the edge Al–Carol, for instance! Fortunately, the maximum flow problem has the following property: if all edge capacities are integers, then the optimal flow (or at least one of them, if it is not unique) is integral. We can see this directly from our algorithm, which would only ever generate integer flows in this case.

Hence integrality comes for free in the maximum flow problem. Unfortunately, this is the exception rather than the rule: as we will see in the next chapter, it is a very difficult problem to find the optimum solution (or for that matter, *any* solution) of a general linear program, under the additional constraint that the variables be integers.

6.2 Circuit evaluation

The importance of linear programming stems from the astounding variety of problems which reduce to it, and which thereby bear witness to its expressive power. In a sense, this next one is the *ultimate* application.

Suppose that we are given a *Boolean circuit*, that is, a dag of gates, each of which is one of the following: (1) an *input gate* of indegree zero, with value `true` or `false` ; (2) an *OR gate* of indegree two; (3) an *AND gate* of indegree two; or (4) a *NOT gate* with indegree one. In addition, one of the gates is designated as the *output*. When the laws of Boolean logic are applied to the gates in topological order, does the output gate evaluate to `true` ? This is known as the *circuit value* problem.

There is a very simple and automatic way of translating this problem into an LP. Create a variable x_g for each gate g, with constraints $0 \leq x_g \leq 1$. If g is a `true` input gate, add the equation $x_g = 1$; if it is `false`, add $x_g = 0$. If g is an OR gate, with incoming edges from, say, gates h and h', then include the inequality $x_g \leq x_h + x_{h'}$. If it is the AND of h and h', use instead the pair of inequalities $x_g \leq x_h$ and $x_g \leq x_{h'}$ (notice the difference). If g is the NOT of h, put in $x_g = 1 - x_h$. Finally, we want to maximize x_o, where o is the output gate. It is easy to see that the optimum value of x_o will be 1 if the circuit value is `true`, and 0 if it is `false`.

This is a straightforward reduction to linear programming, from a problem that may not seem very interesting at first. However, the circuit value problem is in a sense the most general problem solvable in polynomial time! Here is a justification of this statement: any algorithm runs on a computer, and the computer is ultimately a Boolean combinational circuit implemented on a chip. If the algorithm runs in polynomial time, it can be rendered as a Boolean circuit consisting of polynomially many superpositions of the computer's circuit. Hence, the fact that circuit value problem reduces to LP means that *all polynomially solvable problems do!*

In our next topic, *Complexity and NP-completeness*, we shall see that a class containing many hard problems reduces, much the same way, to *integer programming*.