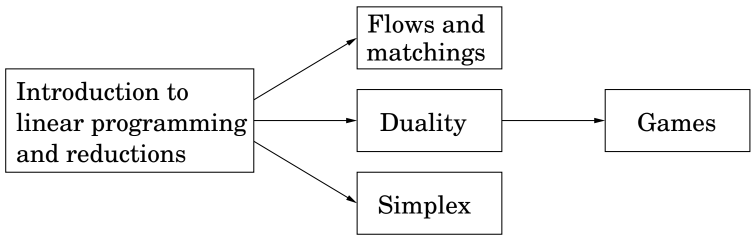
Chapter 7

Linear programming and reductions

Many of the problems for which we want algorithms are *optimization* tasks: the *shortest* path, the *cheapest* spanning tree, the *longest* increasing subsequence, and so on. In such cases, we seek a solution that (1) satisfies certain constraints (for instance, the path must use edges of the graph and lead from s to t, the tree must touch all nodes, the subsequence must be increasing); and (2) is the best possible, with respect to some well-defined criterion, among all solutions that satisfy these constraints.

Linear programming describes a broad class of optimization tasks in which both the constraints and the optimization criterion are *linear functions*. It turns out an enormous number of problems can be expressed in this way.

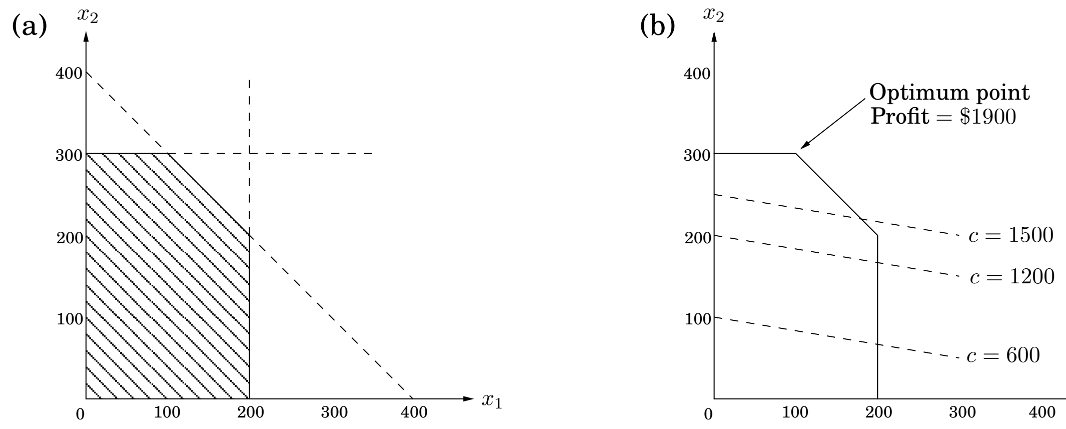
Given the vastness of its topic, this chapter is divided into several parts, which can be read separately subject to the following dependencies.



7.1 An introduction to linear programming

In a linear programming problem we are given a set of variables, and we want to assign real values to them so as to (1) satisfy a set of linear equations and/or linear inequalities involving these variables and (2) maximize or minimize a given linear objective function.

Figure 7.1 (a) The feasible region for a linear program. (b) Contour lines of the objective function: $x_1 + 6x_2 = c$ for different values of the profit c.



7.1.1 Example: profit maximization

A boutique chocolatier has two products: its flagship assortment of triangular chocolates, called *Pyramide*, and the more decadent and deluxe *Pyramide Nuit*. How much of each should it produce to maximize profits? Let's say it makes x_1 boxes of Pyramide per day, at a profit of \$1 each, and x_2 boxes of Nuit, at a more substantial profit of \$6 apiece; x_1 and x_2 are unknown values that we wish to determine. But this is not all; 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, the daily demand for these exclusive chocolates is limited to at most 200 boxes of Pyramide and 300 boxes of Nuit. Also, the current workforce can produce a total of at most 400 boxes of chocolate per day. What are the optimal levels of production?

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

$$\begin{array}{lr}
\text{Objective function} & \max x_1 + 6x_2 \\
\text{Constraints} & x_1 \leq 200 \\
 & x_2 \leq 300 \\
 & x_1 + x_2 \leq 400 \\
 & x_1, x_2 \geq 0
\end{array}$$

A linear equation in x_1 and x_2 defines a line in the two-dimensional (2D) 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 convex polygon, shown in Figure 7.1.

We want to find the point in this polygon at which the objective function—the profit—is maximized. The points with a profit of c dollars lie on the line $x_1 + 6x_2 = c$, which has a slope of $-1/6$ and is shown in Figure 7.1 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 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 different, then its last contact with the polygon could be an entire edge rather than a single vertex. In this case, the optimum solution would not be unique, but there would certainly be an optimum vertex.

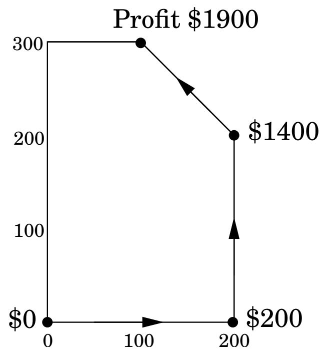
It is a general rule of linear programs that the optimum is achieved at a vertex of the feasible region. The only exceptions are cases in which there is no optimum; this can happen in two ways:

1. The linear 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,
$$\max\ x_1 + x_2$$
$$x_1, x_2 \geq 0$$

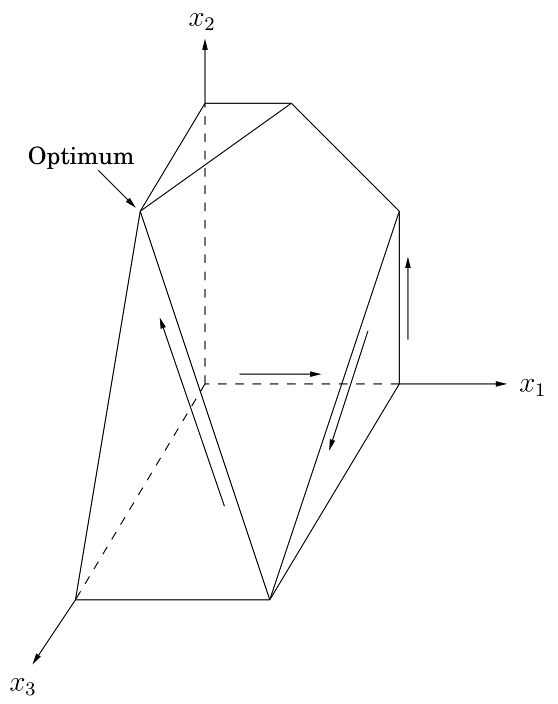
Solving linear programs

Linear programs (LPs) can be solved by the *simplex method*, devised by George Dantzig in 1947. We shall explain it in more detail in Section 7.6, 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.



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 also lie below this line.

Figure 7.2 The feasible polyhedron for a three-variable linear program.



More products

Encouraged by consumer demand, the chocolatier decides to introduce a third and even more exclusive line of chocolates, called *Pyramide Luxe*. One box of these will bring in a profit of $\$13$. Let x_1, x_2, x_3 denote the number of boxes of each chocolate produced daily, with x_3 referring to Luxe. The old constraints on x_1 and x_2 persist, although the labor restriction now extends to x_3 as well: the sum of all three variables can be at most 400. What's more, it turns out that Nuit and Luxe require the same packaging machinery, except that Luxe 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.

$$\begin{aligned} \max \quad x_1 + 6x_2 + 13x_3 & \\ x_1 &\leq 200 \\ x_2 &\leq 300 \\ x_1 + x_2 + x_3 &\leq 400 \\ x_2 + 3x_3 &\leq 600 \\ x_1, x_2, x_3 &\geq 0 \end{aligned}$$

The space of solutions is now three-dimensional. Each linear equation defines a 3D plane, 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 7.2). Looking at the figure, can you decipher which inequality corresponds to each face of the polyhedron?

A profit of c corresponds to the plane $x_1 + 6x_2 + 13x_3 = c$. 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: $(0, 300, 100)$, with total profit $\$3100$.

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 7.2, corresponding to the following sequence of vertices and profits:

$$\begin{array}{c} (0,0,0) \\ \$0 \end{array} \longrightarrow \begin{array}{c} (200,0,0) \\ \$200 \end{array} \longrightarrow \begin{array}{c} (200,200,0) \\ \$1400 \end{array} \longrightarrow \begin{array}{c} (200,0,200) \\ \$2800 \end{array} \longrightarrow \begin{array}{c} (0,300,100) \\ \$3100 \end{array}$$

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 the vertex's neighbors lie on one side of the profit-plane, then so must the entire polyhedron.

A magic trick called duality

Here is why you should believe that $(0, 300, 100)$, with a total profit of $\$3100$, is the optimum: Look back at the linear program. Add the second inequality to the third, and add to them the fourth multiplied by 4. The result is the inequality $x_1 + 6x_2 + 13x_3 \leq 3100$.

Do you see? This inequality says that no feasible solution (values x_1, x_2, x_3 satisfying the constraints) can possibly have a profit greater than 3100. So we must indeed have found the optimum! The only question is, where did we get these mysterious multipliers $(0, 1, 1, 4)$ for the four inequalities?

In Section 7.4 we'll see that it is always possible to come up with such multipliers by solving another LP! Except that (it gets even better) we do not even need to solve this other LP, because it is in fact so intimately connected to the original one—it is called the *dual*—that solving the original LP solves the dual as well! But we are getting far ahead of our story.

What if we add a fourth line of chocolates, or hundreds more of them? Then the problem becomes high-dimensional, and hard to visualize. Simplex continues to work in this general setting, although we can no longer rely upon simple geometric intuitions for its description and justification. We will study the full-fledged simplex algorithm in Section 7.6.

In the meantime, we can rest assured in the knowledge that there are many professional, industrial-strength packages that implement simplex and take care of all the tricky details like numeric precision. 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.

With this in mind, let's look at a high-dimensional application.

7.1.2 Example: production planning

This time, our company makes handwoven carpets, a product for which the demand is extremely seasonal. Our analyst has just obtained demand estimates for all months of the next calendar year: $d_1, d_2, \ldots, d_{12}$. As feared, they are very uneven, ranging from 440 to 920.

Here's a quick snapshot of the company. We currently have 30 employees, each of whom makes 20 carpets per month and gets a monthly salary of $\$2{,}000$. We have no initial surplus of carpets.

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 carpet per month. We currently have no stored carpets on hand, and we must end the year without any carpets stored.

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

A crucial first step is defining the variables.

$$\begin{aligned} w_i &= \text{number of workers during } i\text{th month; } w_0 = 30. \\ x_i &= \text{number of carpets made during } i\text{th month.} \\ o_i &= \text{number of carpets made by overtime in month } i. \\ h_i, f_i &= \text{number of workers hired and fired, respectively, at beginning of month } i. \\ s_i &= \text{number of carpets stored at end of month } i\text{; } s_0 = 0. \end{aligned}$$

All in all, there are 72 variables (74 if you count w_0 and s_0).

We now write the constraints. First, all variables must be nonnegative:

$$w_i, x_i, o_i, h_i, f_i, s_i \geq 0, \quad i = 1, \ldots, 12.$$

The total number of carpets made 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:

$$w_i = w_{i-1} + h_i - f_i.$$

The number of carpets stored at the end of each month is what we started with, plus the number we made, minus the demand for the month:

$$s_i = s_{i-1} + x_i - d_i.$$

And overtime is limited:

$$o_i \leq 6w_i.$$

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

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

a linear function of the variables. Solving this linear program by simplex should take less than a second and will give us the optimum business strategy for our company.

Well, almost. The optimum solution might turn out to be *fractional*; for instance, it might involve hiring 10.6 workers in the month of March. This number would have to be rounded to either 10 or 11 in order to make sense, and the overall cost would then increase correspondingly. In the present example, most of the variables take on fairly large (double-digit) values, and thus rounding is unlikely to affect things too much. There are other LPs, however, in which rounding decisions have to be made very carefully in order to end up with an integer solution of reasonable quality.

In general, there is a tension in linear programming between the ease of obtaining fractional solutions and the desirability of integer ones. As we shall see in Chapter 8, finding the optimum integer solution of an LP is an important but very hard problem, called *integer linear programming.*

7.1.3 Example: optimum bandwidth allocation

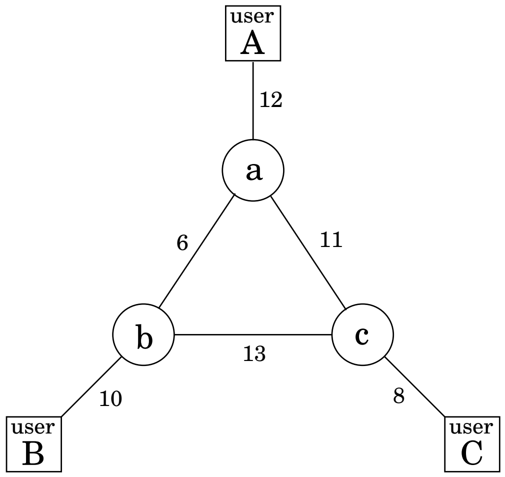
Next we turn to a miniaturized version of the kind of problem a network service provider might face.

Suppose we are managing a network whose lines have the bandwidths shown in Figure 7.3, and we need to establish three connections: between users A and B, between B and C, and between A and C. Each connection requires at least two units of bandwidth, but can be assigned more. Connection A–B pays \$3 per unit of bandwidth, and connections B–C and A–C pay \$2 and \$4, respectively.

Each connection can be routed in two ways, a long path and a short path, or by a combination: for instance, two units of bandwidth via the short route, one via the long route. How do we route these connections to maximize our network's revenue?

This is a linear program. We have variables for each connection and each path (long or short); for example, x_{AB} is the short-path bandwidth allocated to the connection between A and B, and x'_{AB} the long-path bandwidth for this same connection. We demand that no edge's bandwidth is exceeded and that each connection gets a bandwidth of at least 2 units.

Figure 7.3 A communications network between three users A, B, and C. Bandwidths are shown.



$$
\begin{array}{rll}
\max & 3x_{AB} + 3x'_{AB} + 2x_{BC} + 2x'_{BC} + 4x_{AC} + 4x'_{AC} & \\
& x_{AB} + x'_{AB} + x_{BC} + x'_{BC} \leq 10 & \text{[edge } (b, B)\text{]} \\
& x_{AB} + x'_{AB} + x_{AC} + x'_{AC} \leq 12 & \text{[edge } (a, A)\text{]} \\
& x_{BC} + x'_{BC} + x_{AC} + x'_{AC} \leq 8 & \text{[edge } (c, C)\text{]} \\
& x_{AB} + x'_{BC} + x'_{AC} \leq 6 & \text{[edge } (a, b)\text{]} \\
& x'_{AB} + x_{BC} + x'_{AC} \leq 13 & \text{[edge } (b, c)\text{]} \\
& x'_{AB} + x'_{BC} + x_{AC} \leq 11 & \text{[edge } (a, c)\text{]} \\
& x_{AB} + x'_{AB} \geq 2 & \\
& x_{BC} + x'_{BC} \geq 2 & \\
& x_{AC} + x'_{AC} \geq 2 & \\
& x_{AB}, x'_{AB}, x_{BC}, x'_{BC}, x_{AC}, x'_{AC} \geq 0 &
\end{array}
$$

Even a tiny example like this one is hard to solve on one's own (try it!), and yet the optimal solution is obtained instantaneously via simplex:

$$x_{AB} = 0,\ x'_{AB} = 7,\ x_{BC} = x'_{BC} = 1.5,\ x_{AC} = 0.5,\ x'_{AC} = 4.5.$$

This solution is not integral, but in the present application we don't need it to be, and thus no rounding is required. Looking back at the original network, we see that every edge except a–c is used at full capacity.

One cautionary observation: our LP has one variable for every possible path between the users. In a larger network, there could easily be exponentially many such paths, and therefore this particular way of translating the network problem into an LP will not scale well. We will see a cleverer and more scalable formulation in Section 7.2.

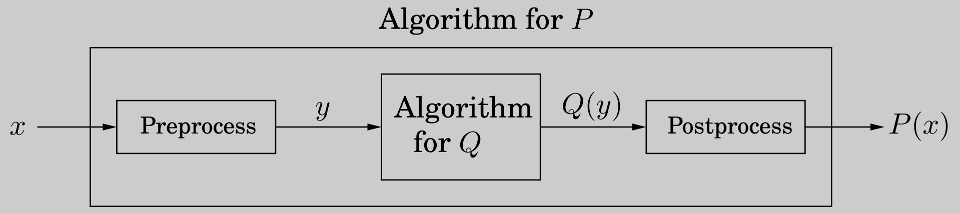
Here's a parting question for you to consider. Suppose we removed the constraint that each connection should receive at least two units of bandwidth. Would the optimum change?

Reductions

Sometimes a computational task is sufficiently general that any subroutine for it can also be used to solve a variety of other tasks, which at first glance might seem unrelated. For instance, we saw in Chapter 6 how an algorithm for finding the longest path in a dag can, surprisingly, also be used for finding longest increasing subsequences. We describe this phenomenon by saying that the longest increasing subsequence problem *reduces to* the longest path problem in a dag. In turn, the longest path in a dag reduces to the shortest path in a dag; here's how a subroutine for the latter can be used to solve the former:

```
function LONGEST PATH($G$)
  negate all edge weights of $G$
  return SHORTEST PATH($G$)
```

Let's step back and take a slightly more formal view of reductions. If any subroutine for task Q can also be used to solve P, we say P *reduces to* Q. Often, P is solvable by a single call to Q's subroutine, which means any instance x of P can be transformed into an instance y of Q such that $P(x)$ can be deduced from $Q(y)$:



(Do you see that the reduction from $P =$ LONGEST PATH to $Q =$ SHORTEST PATH follows this schema?) If the pre- and postprocessing procedures are efficiently computable then this creates an efficient algorithm for P out of *any* efficient algorithm for Q!

Reductions enhance the power of algorithms: Once we have an algorithm for problem Q (which could be shortest path, for example) we can use it to solve other problems. In fact, most of the computational tasks we study in this book are considered core computer science problems precisely because they arise in so many different applications, which is another way of saying that many problems reduce to them. This is especially true of linear programming.

7.1.4 Variants of linear programming

As evidenced in our examples, a general linear program has many degrees of freedom.

1. It can be either a maximization or a minimization problem.

2. Its constraints can be equations and/or inequalities.

3. The variables are often restricted to be nonnegative, but they can also be unrestricted in sign.

We will now show that these various LP options *can all be reduced to one another* via simple transformations. Here's how.

1. To turn a maximization problem into a minimization (or vice versa), just multiply the coefficients of the objective function by -1.

2a. 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

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

This s is called the *slack variable* for the inequality. As justification, observe that 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.

2b. To change an equality constraint into inequalities is easy: rewrite $ax = b$ as the equivalent pair of constraints $ax \leq b$ and $ax \geq b$.

3. Finally, to deal with a variable x that is unrestricted in sign, do the following:

   - Introduce two nonnegative variables, $x^+, x^- \geq 0$.
   - 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.

By applying these transformations we can reduce any LP (maximization or minimization, with both inequalities and equations, and with both nonnegative and unrestricted variables) into an LP of a much more constrained kind that we call the *standard form*, in which the variables are all nonnegative, the constraints are all equations, and the objective function is to be minimized.

For example, our first linear program gets rewritten thus:

$$\begin{aligned} \max\ & x_1 + 6x_2 \\ & x_1 \leq 200 \\ & x_2 \leq 300 \\ & x_1 + x_2 \leq 400 \\ & x_1, x_2 \geq 0 \end{aligned} \quad \Longrightarrow \quad \begin{aligned} \min\ & -x_1 - 6x_2 \\ & x_1 + s_1 = 200 \\ & x_2 + s_2 = 300 \\ & x_1 + x_2 + s_3 = 400 \\ & x_1, x_2, s_1, s_2, s_3 \geq 0 \end{aligned}$$

The original was also in a useful form: maximize an objective subject to certain inequalities. Any LP can likewise be recast in this way, using the reductions given earlier.

Matrix-vector notation

A linear function like $x_1 + 6x_2$ can be written as the dot product of two vectors

$$\mathbf{c} = \begin{pmatrix} 1 \\ 6 \end{pmatrix} \text{ and } \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix},$$

denoted $\mathbf{c} \cdot \mathbf{x}$ or $\mathbf{c}^T\mathbf{x}$. Similarly, linear constraints can be compiled into matrix-vector form:

$$\begin{array}{rcl} x_1 & \leq & 200 \\ x_2 & \leq & 300 \\ x_1 + x_2 & \leq & 400 \end{array} \quad \Longrightarrow \quad \underbrace{\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix}}_{\mathbf{A}} \underbrace{\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}}_{\mathbf{x}} \underbrace{\leq}_{\leq} \underbrace{\begin{pmatrix} 200 \\ 300 \\ 400 \end{pmatrix}}_{\mathbf{b}}.$$

Here each row of matrix $\mathbf{A}$ corresponds to one constraint: its dot product with $\mathbf{x}$ is at most the value in the corresponding row of $\mathbf{b}$. In other words, if the rows of $\mathbf{A}$ are the vectors $\mathbf{a}_1, \ldots, \mathbf{a}_m$, then the statement $\mathbf{Ax} \leq \mathbf{b}$ is equivalent to

$$\mathbf{a}_i \cdot \mathbf{x} \leq b_i \text{ for all } i = 1, \ldots, m.$$

With these notational conveniences, a generic LP can be expressed simply as

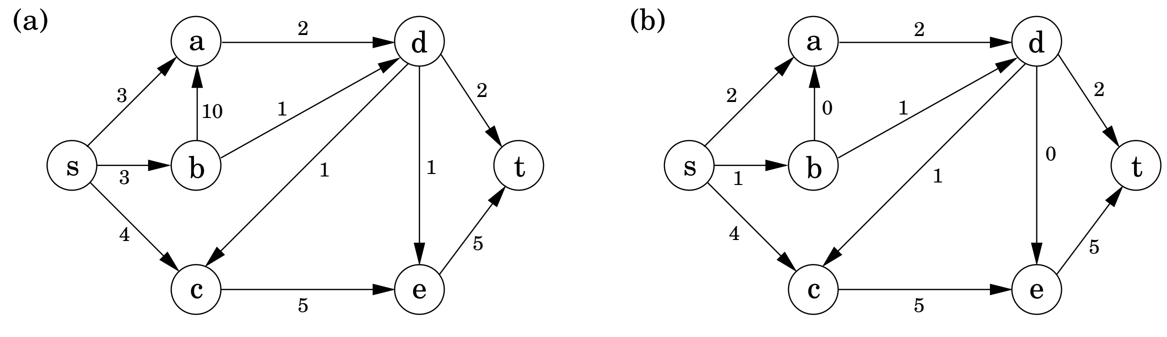
$$\begin{aligned} \max\ & \mathbf{c}^T\mathbf{x} \\ & \mathbf{Ax} \leq \mathbf{b} \\ & \mathbf{x} \geq 0. \end{aligned}$$

7.2 Flows in networks

7.2.1 Shipping oil

Figure 7.4(a) shows a directed graph representing a network of pipelines along which oil can be sent. The goal is to ship as much 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

Figure 7.4 (a) A network with edge capacities. (b) A *flow* in the network.



en route. Figure 7.4(b) shows a possible *flow* from s to t, which ships 7 units in all. Is this the best that can be done?

7.2.2 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 capacities of any of the edges. A particular shipping scheme is called a *flow* and consists of a variable f_e for each edge e of the network, satisfying the following two properties:

1. It doesn't violate edge capacities: $0 \leq f_e \leq c_e$ for all $e \in E$.

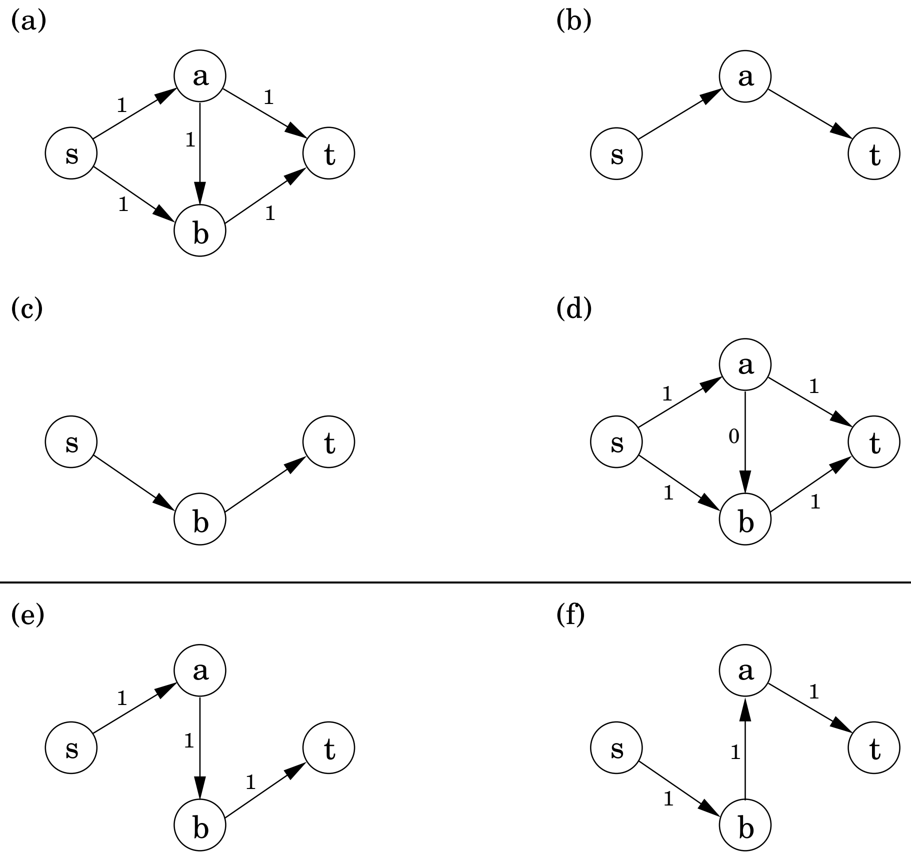
2. For all nodes u except s and t, the amount of flow entering u equals the amount leaving u:
$$\sum_{(w,u)\in E} f_{wu} \quad = \quad \sum_{(u,z)\in E} f_{uz}.$$
In other words, flow is *conserved*.

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

In short, our goal is to assign values to $\{f_e : e \in E\}$ that will satisfy a set of linear constraints and maximize a linear objective function. But this is a linear program! *The maximum-flow problem reduces to linear programming.*

For example, for the network of Figure 7.4 the LP has 11 variables, one per edge. It seeks to maximize $f_{sa} + f_{sb} + f_{sc}$ subject to a total of 27 constraints: 11 for nonnegativity (such as $f_{sa} \geq 0$), 11 for capacity (such as $f_{sa} \leq 3$), and 5 for flow conservation (one for each node of the graph other than s and t, such as $f_{sc} + f_{dc} = f_{ce}$). Simplex would take no time at all to correctly solve the problem and to confirm that, in our example, a flow of 7 is in fact optimal.

Figure 7.5 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. (e) We could have chosen this path first. (f) In which case, we would have to allow this second path.



7.2.3 A closer look at the algorithm

All we know so far of the simplex algorithm is the vague geometric intuition that it keeps making local moves on the surface of a convex feasible region, successively improving the objective function until it finally reaches the optimal solution. Once we have studied it in more detail (Section 7.6), we will be in a position to understand exactly how it handles flow LPs, which is useful as a source of inspiration for designing *direct* max-flow algorithms.

It turns out that in fact the behavior of simplex has an elementary interpretation:

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

Figure 7.5(a)–(d) shows a small example in which simplex halts after two iterations. The final flow has size 2, 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 7.5(e)? This gives only one unit of flow and yet seems to block all other paths. Simplex gets around this problem by also allowing paths to *cancel existing flow*. In this particular case, it would subsequently choose the path of Figure 7.5(f). 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, in each iteration 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 up to $c_{uv} - f_{uv}$ additional units of flow, and in the second case, upto f_{vu} additional units (canceling all or part of the existing flow on (v, u)). These flow-increasing opportunities can be captured in a *residual network* $G^f = (V, E^f)$, which has exactly the two types of edges listed, with residual capacities c^f:

$$\begin{cases} c_{uv} - f_{uv} & \text{if } (u, v) \in E \text{ and } f_{uv} < c_{uv} \\ f_{vu} & \text{if } (v, u) \in E \text{ and } f_{vu} > 0 \end{cases}$$

Thus we can equivalently think of simplex as choosing an $s - t$ path in the residual network.

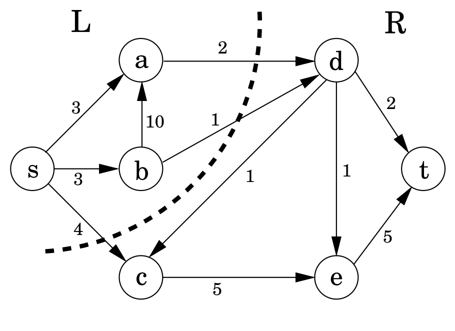
By simulating the behavior of simplex, we get a direct algorithm for solving max-flow. It proceeds in iterations, each time explicitly constructing G^f, finding a suitable $s - t$ path in G^f by using, say, a linear-time breadth-first search, and halting if there is no longer any such path along which flow can be increased.

Figure 7.6 illustrates the algorithm on our oil example.

7.2.4 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 7.4) into two 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, an (s,t)-*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 previously, is an upper bound on *any* flow:

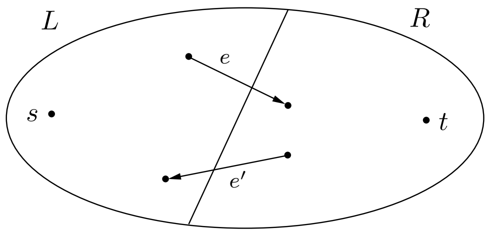
Pick any flow f and any (s,t)-cut (L,R). Then size$(f) \leq$ capacity(L,R).

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* (s,t)*-cut.*

Moreover, our algorithm automatically finds this cut as a by-product!

Let's see why this is true. Suppose f is the final flow when the algorithm terminates. We know that node t is no longer reachable from s in the residual network G^f. Let L be the nodes that *are* reachable from s in G^f, and let $R = V - L$ be the rest of the nodes. Then (L,R) is a cut in the graph G:



We claim that

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

To see this, observe that by the way L is defined, 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. (So, in the figure, $f_e = c_e$ and $f_{e'} = 0$.) Therefore the net flow across (L,R) is exactly the capacity of the cut.

7.2.5 Efficiency

Each iteration of our maximum-flow algorithm is efficient, requiring $O(|E|)$ time if a depth-first or breadth-first search is used to find an $s-t$ path. But how many iterations are there?

Suppose all edges in the original network have *integer* capacities $\leq C$. Then an inductive argument shows that on each iteration of the algorithm, the flow is always an integer and increases by an integer amount. Therefore, since the maximum flow is at most $C|E|$ (why?), it follows that the number of iterations is at most this much. But this is hardly a reassuring bound: what if C is in the millions?

We examine this issue further in Exercise 7.31. It turns out that it is indeed possible to construct bad examples in which the number of iterations is proportional to C, *if* $s-t$ paths are not carefully chosen. However, if paths are chosen in a sensible manner—in particular, by using a breadth-first search, which finds the path with the fewest edges—then the number of iterations is at most $O(|V| \cdot |E|)$, no matter what the capacities are. This latter bound gives an overall running time of $O(|V| \cdot |E|^2)$ for maximum flow.

Figure 7.6 The max-flow algorithm applied to the network of Figure 7.4. At each iteration, the current flow is shown on the left and the residual network on the right. The paths chosen are shown in bold.

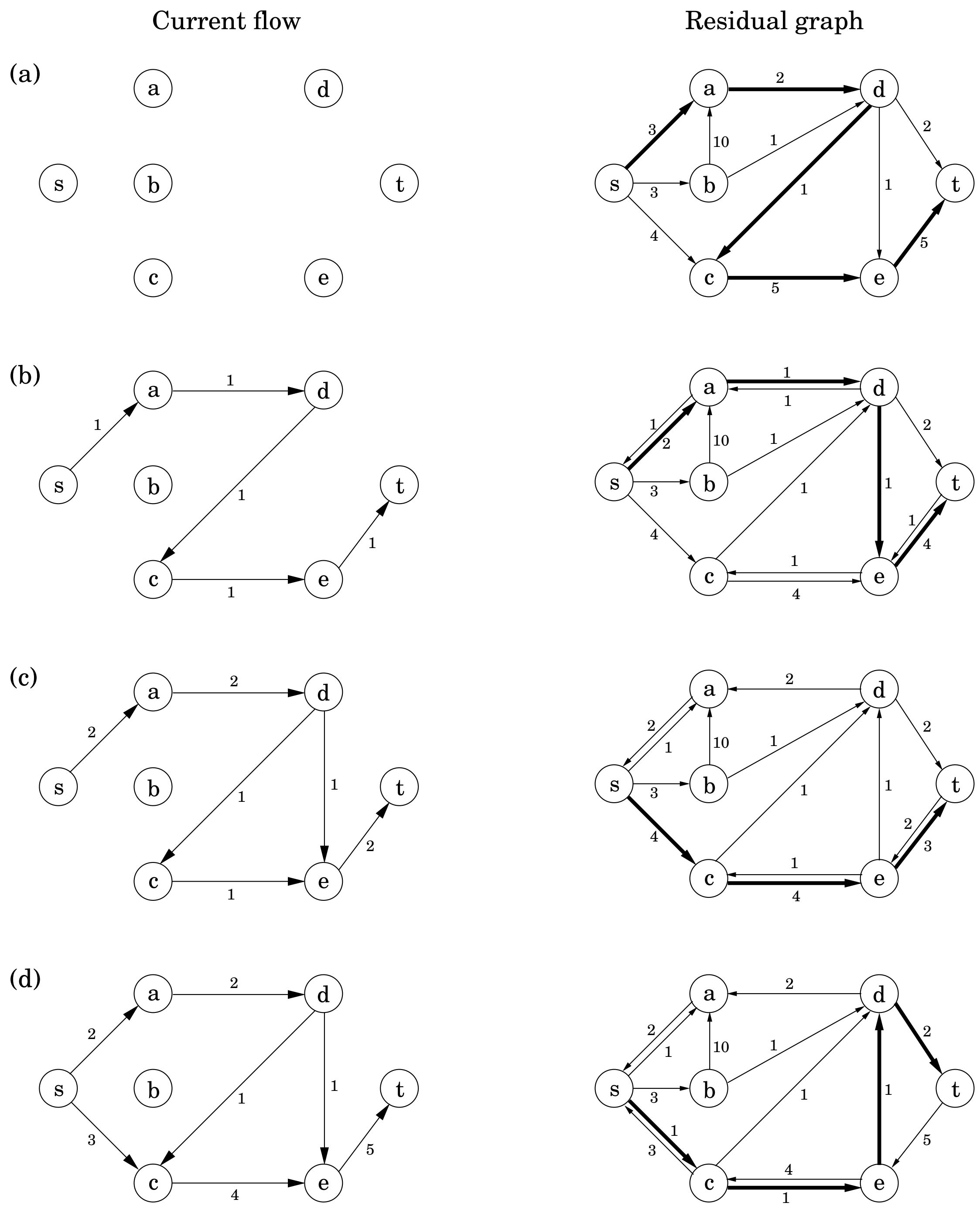


Figure 7.6 *Continued*

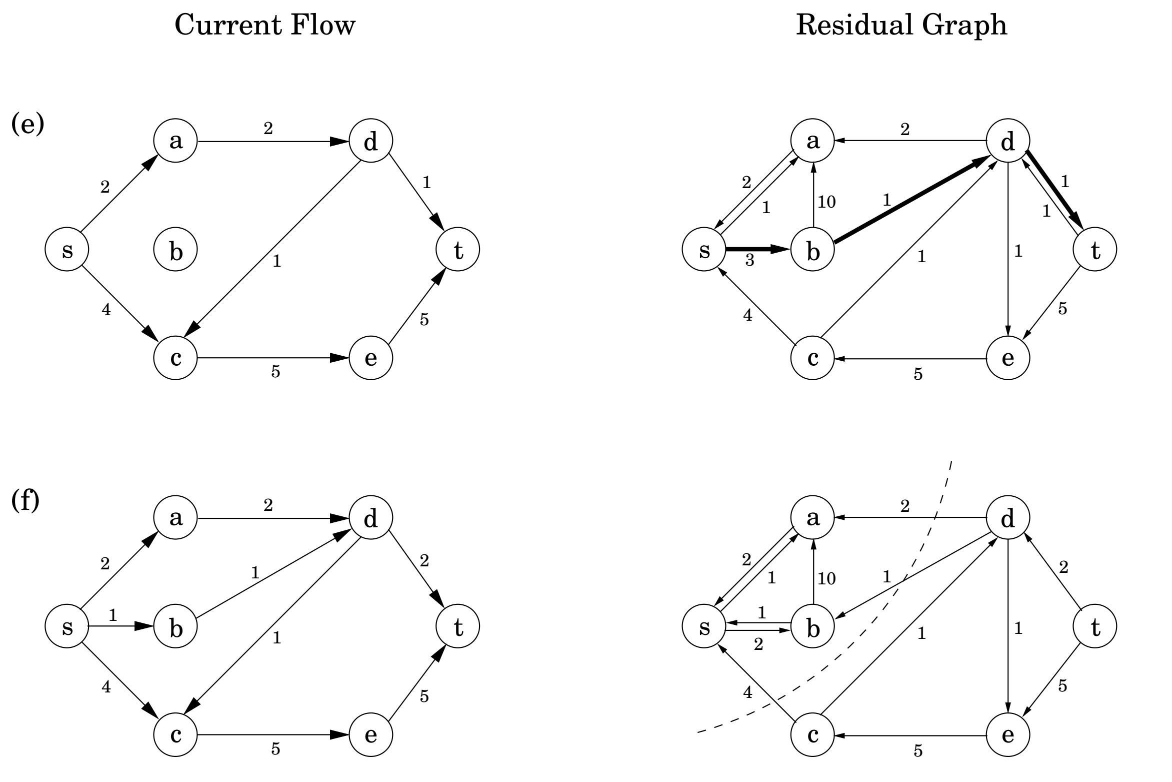
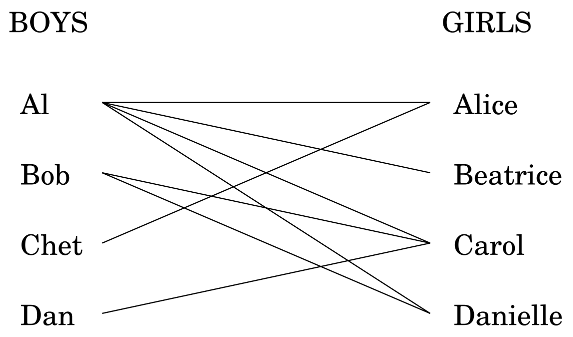


Figure 7.7 An edge between two people means they like each other. Is it possible to pair everyone up happily?



7.3 Bipartite matching

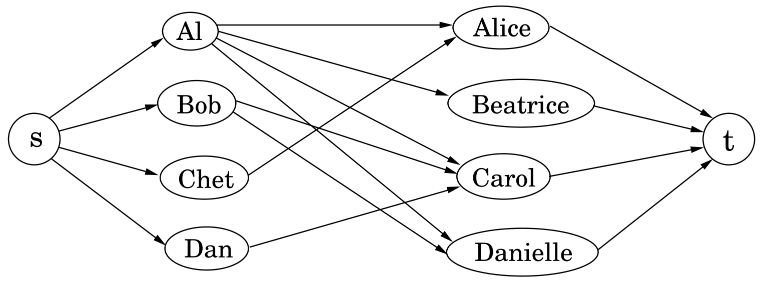
Figure 7.7 shows a graph with four nodes on the left representing boys and four nodes on the right representing girls.[1] There is an edge between a boy and girl if they like each other (for instance, Al likes all the girls). Is it possible to choose couples so that everyone has exactly one partner, 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 7.8). Finally, give every edge a capacity of 1. 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 just stated: 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!

[1]This kind of graph, in which the nodes can be partitioned into two groups such that all edges are *between* the groups, is called *bipartite*.

Figure 7.8 A matchmaking network. Each edge has a capacity of one.



Fortunately, the maximum-flow problem has the following property: *if all edge capacities are integers, then the optimal flow found by our algorithm is integral*. We can see this directly from the algorithm, which in such cases would increment the flow by an integer amount on each iteration.

Hence integrality comes for free in the maximum-flow problem. Unfortunately, this is the exception rather than the rule: as we will see in Chapter 8, it is a very difficult problem to find the optimum solution (or for that matter, *any* solution) of a general linear program, if we also demand that the variables be integers.

7.4 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. Remarkable as this phenomenon is, we now generalize it from maximum flow to *any* problem that can be solved by linear programming! It turns out that every linear maximization problem has a *dual* minimization problem, and they relate to each other in much the same way as flows and cuts.

To understand what duality is about, recall our introductory LP with the two types of chocolate:

$$\begin{aligned} \max\ x_1 + 6x_2 & \\ x_1 &\leq 200 \\ x_2 &\leq 300 \\ x_1 + x_2 &\leq 400 \\ x_1, x_2 &\geq 0 \end{aligned}$$

Simplex declares the optimum solution to be $(x_1, x_2) = (100, 300)$, with objective value 1900. Can this answer be checked somehow? Let's see: suppose we take the first inequality and add it to six times the second inequality. We get

$$x_1 + 6x_2 \ \leq \ 2000.$$

This is interesting, because it tells us that it is impossible to achieve a profit of more than 2000. Can we add together some other combination of the LP constraints and bring this upper bound even closer to 1900? After a little experimentation, we find that multiplying the three inequalities by 0, 5, and 1, respectively, and adding them up yields

$$x_1 + 6x_2 \ \leq \ 1900.$$

So 1900 must indeed be the best possible value! The multipliers $(0, 5, 1)$ magically constitute a *certificate of optimality*! It is remarkable that such a certificate exists for this LP—and even if we knew there were one, how would we systematically go about finding it?

Let's investigate the issue by describing what we expect of these three multipliers, call them y_1, y_2, y_3.

Multiplier	Inequality
y_1	$x_1 \leq 200$
y_2	$x_2 \leq 300$
y_3	$x_1 + x_2 \leq 400$

To start with, these y_i's must be nonnegative, for otherwise they are unqualified to multiply inequalities (multiplying an inequality by a negative number would flip the $\leq$ to $\geq$). After the multiplication and addition steps, we get the bound:

$$(y_1 + y_3)x_1 + (y_2 + y_3)x_2 \;\leq\; 200y_1 + 300y_2 + 400y_3.$$

We want the left-hand side to look like our objective function $x_1 + 6x_2$ so that the right-hand side is an upper bound on the optimum solution. For this we need $y_1 + y_3$ to be 1 and $y_2 + y_3$ to be 6. Come to think of it, it would be fine if $y_1 + y_3$ were larger than 1—the resulting certificate would be all the more convincing. Thus, we get an upper bound

$$x_1 + 6x_2 \;\leq\; 200y_1 + 300y_2 + 400y_3 \quad \text{if} \quad \left\{ \begin{array}{c} y_1, y_2, y_3 \geq 0 \\ y_1 + y_3 \geq 1 \\ y_2 + y_3 \geq 6 \end{array} \right\}.$$

We can easily find y's that satisfy the inequalities on the right by simply making them large enough, for example $(y_1, y_2, y_3) = (5, 3, 6)$. But these particular multipliers would tell us that the optimum solution of the LP is at most $200 \cdot 5 + 300 \cdot 3 + 400 \cdot 6 = 4300$, a bound that is far too loose to be of interest. What we want is a bound that is as tight as possible, so we should minimize $200y_1 + 300y_2 + 400y_3$ subject to the preceding inequalities. *And this is a new linear program*!

Therefore, finding the set of multipliers that gives the best upper bound on our original LP is tantamount to solving a new LP:

$$\begin{aligned} \min\ 200y_1 + 300y_2 + 400y_3 \\ y_1 + y_3 \geq 1 \\ y_2 + y_3 \geq 6 \\ y_1, y_2, y_3 \geq 0 \end{aligned}$$

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 that are equal, then they must both be optimal. Here is just such a pair:

$$\text{Primal} : (x_1, x_2) = (100, 300); \quad \text{Dual} : (y_1, y_2, y_3) = (0, 5, 1).$$

They both have value 1900, and therefore they certify each other's optimality (Figure 7.9).

Amazingly, this is not just a lucky example, but a general phenomenon. To start with, the preceding construction—creating a multiplier for each primal constraint; writing a constraint

Figure 7.9 By design, dual feasible values $\geq$ primal feasible values. The duality theorem tells us that moreover their optima coincide.

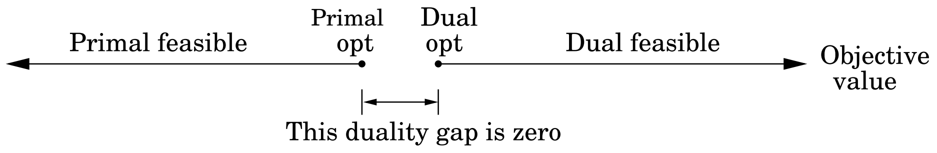


Figure 7.10 A generic primal LP in matrix-vector form, and its dual.

Primal LP:

$$\begin{aligned} \max\ & \mathbf{c}^T\mathbf{x} \\ & \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq 0 \end{aligned}$$

Dual LP:

$$\begin{aligned} \min\ & \mathbf{y}^T\mathbf{b} \\ & \mathbf{y}^T\mathbf{A} \geq \mathbf{c}^T \\ & \mathbf{y} \geq 0 \end{aligned}$$

in the dual for every variable of the primal, in which the sum is required to be above the objective coefficient of the corresponding primal variable; and optimizing the sum of the multipliers weighted by the primal right-hand sides—can be carried out for any LP, as shown in Figure 7.10, and in even greater generality in Figure 7.11. The second figure has one noteworthy addition: if the primal has an equality constraint, then the corresponding multiplier (or *dual variable*) need not be nonnegative, because the validity of equations is preserved when multiplied by negative numbers. So, the multipliers of equations are unrestricted variables. Notice also the simple symmetry between the two LPs, in that the matrix $A = (a_{ij})$ defines one primal constraint with each of its *rows*, and one dual constraint with each of its *columns*.

By construction, any feasible solution of the dual is an upper bound on any feasible solution of the primal. But moreover, their optima coincide!

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

When the primal is the LP that expresses the max-flow problem, it is possible to assign interpretations to the dual variables that show the dual to be none other than the minimum-cut problem (Exercise 7.25). The relation between flows and cuts is therefore just a specific instance of the duality theorem. And in fact, the proof of this theorem 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.

Figure 7.11 In the most general case of linear programming, we have a set I of inequalities and a set E of equalities (a total of $m = |I| + |E|$ constraints) over n variables, of which a subset N are constrained to be nonnegative. The dual has $m = |I| + |E|$ variables, of which only those corresponding to I have nonnegativity constraints.

Primal LP:

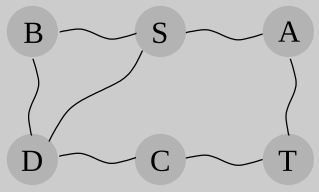
$$\begin{aligned} \max\ & c_1x_1 + \cdots + c_nx_n \\ a_{i1}x_1 + \cdots + a_{in}x_n &\leq b_i \quad \text{for } i \in I \\ a_{i1}x_1 + \cdots + a_{in}x_n &= b_i \quad \text{for } i \in E \\ x_j &\geq 0 \quad \text{for } j \in N \end{aligned}$$

Dual LP:

$$\begin{aligned} \min\ & b_1y_1 + \cdots + b_my_m \\ a_{1j}y_1 + \cdots + a_{mj}y_m &\geq c_j \quad \text{for } j \in N \\ a_{1j}y_1 + \cdots + a_{mj}y_m &= c_j \quad \text{for } j \notin N \\ y_i &\geq 0 \quad \text{for } i \in I \end{aligned}$$

Visualizing duality

One can solve the shortest-path problem by the following "analog" device: Given a weighted undirected graph, build a *physical model* of it in which each edge is a string of length equal to the edge's weight, and each node is a knot at which the appropriate endpoints of strings are tied together. Then to find the shortest path from s to t, just *pull* s away from t until the gadget is taut. It is intuitively clear that this finds the shortest path from s to t.



There is nothing remarkable or surprising about all this until we notice the following: the shortest-path problem is a *minimization* problem, right? Then why are we *pulling* s away from t, an act whose purpose is, obviously, *maximization?* Answer: By pulling s away from t we solve *the dual* of the shortest-path problem! This dual has a very simple form (Exercise 7.28), with one variable x_u for each node u:

$$\begin{aligned} \max\ & x_S - x_T \\ |x_u - x_v| \leq w_{uv} & \quad \text{for all edges } \{u, v\} \end{aligned}$$

In words, the dual problem is to stretch s and t as far apart as possible, subject to the constraint that the endpoints of any edge $\{u, v\}$ are separated by a distance of at most w_{uv}.

7.5 Zero-sum games

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

$$G \;=\; \begin{array}{|cc|rrr|} \hline & & \multicolumn{3}{c|}{\text{Column}} \\ & & r & p & s \\ \hline & r & 0 & -1 & 1 \\ \text{Row} & p & 1 & 0 & -1 \\ & s & -1 & 1 & 0 \\ \hline \end{array}$$

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 $\mathbf{x} = (x_1, x_2, x_3)$, positive numbers that add up to 1. Similarly, Column's mixed strategy is some $\mathbf{y} = (y_1, y_2, y_3)$.[2]

On any given round of the game, there is an $x_i y_j$ chance that Row and Column will play the ith and jth moves, respectively. Therefore the *expected* (average) payoff is

$$\sum_{i,j} G_{ij} \cdot \text{Prob[Row plays } i\text{, Column plays } j] \;=\; \sum_{i,j} G_{ij} x_i y_j.$$

Row wants to *maximize* this, while Column wants to *minimize* it. What payoffs can they hope to achieve in rock-paper-scissors? Well, suppose for instance that Row plays the "completely random" strategy $\mathbf{x} = (1/3, 1/3, 1/3)$. If Column plays r, then the average payoff (reading the first column of the game matrix) will be

$$\frac{1}{3} \cdot 0 + \frac{1}{3} \cdot 1 + \frac{1}{3} \cdot -1 \;=\; 0.$$

This is also true if Column plays p, or s. And since the payoff of any mixed strategy (y_1, y_2, y_3) is just a weighted average of the individual payoffs for playing r, p, and s, it must also be zero. This can be seen directly from the preceding formula,

$$\sum_{i,j} G_{ij} x_i y_j \;=\; \sum_{i,j} G_{ij} \cdot \frac{1}{3} y_j \;=\; \sum_j y_j \left(\sum_i \frac{1}{3} G_{ij} \right) \;=\; \sum_j y_j \cdot 0 \;=\; 0,$$

where the second-to-last equality is the observation that every column of G adds up to zero. Thus by playing the "completely random" strategy, Row forces an expected payoff of zero, *no matter what Column does*. This means that Column cannot hope for a negative (expected)

[2]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.

payoff (remember that he wants the payoff to be as small as possible). But symmetrically, if Column plays the completely random strategy, he also forces an expected payoff of zero, and thus Row cannot hope for a positive (expected) payoff. In short, the best each player can do is to play completely randomly, with an expected payoff of zero. We have mathematically confirmed what you knew all along about rock-paper-scissors!

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

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

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

We've seen that the average payoff is the same (zero) in either case if both parties play optimally. But this might well be due to the high level of symmetry in rock-paper-scissors. In general games, 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 Row. 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—and in fact equivalent to—linear programming duality.

Let's investigate this with a nonsymmetric game. Imagine a *presidential election* scenario in which there are two candidates for office, and the moves they 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 $\mathbf{x} = (1/2, 1/2)$. What should Column do? Move m will incur an expected loss of $1/2$, while t will incur an expected loss of 0. The best response of Column is therefore the *pure* strategy $\mathbf{y} = (0, 1)$.

More generally, once Row's strategy $\mathbf{x} = (x_1, x_2)$ is fixed, there is always a *pure* strategy that is optimal for Column: either move m, with payoff $3x_1 - 2x_2$, or t, with payoff $-x_1 + x_2$, whichever is smaller. After all, any mixed strategy $\mathbf{y}$ is a weighted average of these two pure strategies and thus cannot beat the better of the two.

Therefore, if Row is forced to announce $\mathbf{x}$ before Column plays, she knows that his best response will achieve an expected payoff of $\min\{3x_1 - 2x_2, -x_1 + x_2\}$. She should choose $\mathbf{x}$ *defensively* to maximize her payoff against this best response:

$$\text{Pick } (x_1, x_2) \text{ that maximizes} \qquad \underbrace{\min\{3x_1 - 2x_2, -x_1 + x_2\}}_{\text{payoff from Column's best response to } \mathbf{x}}$$

This choice of x_i's gives Row the best possible *guarantee* about her expected payoff. And we will now see that it can be found by an LP! The main trick is to notice that for *fixed* x_1 and x_2 the following are equivalent:

$$z = \min\{3x_1 - 2x_2, -x_1 + x_2\} \qquad\qquad \begin{array}{c} \max\ z \\ z \le 3x_1 - 2x_2 \\ z \le -x_1 + x_2 \end{array}$$

And Row needs to choose x_1 and x_2 to maximize this z.

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

Symmetrically, if Column has to announce his strategy first, his best bet is to choose the mixed strategy $\mathbf{y}$ that minimizes his loss under Row's best response, in other words,

$$\text{Pick } (y_1, y_2) \text{ that minimizes} \quad \underbrace{\max\{3y_1 - y_2, -2y_1 + y_2\}}_{\text{outcome of Row's best response to } \mathbf{y}}$$

In LP form, this is

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

The crucial observation now is that *these two LPs are dual to each other* (see Figure 7.11)! Hence, they have the same optimum, call it V.

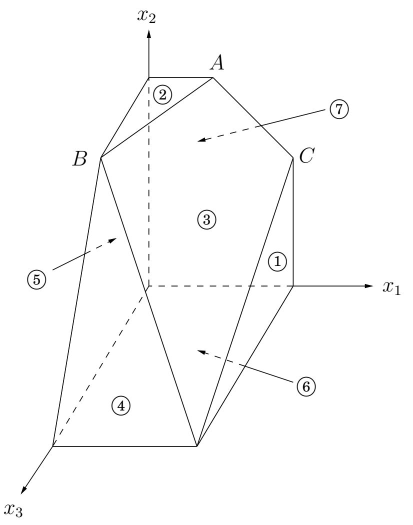
Let us summarize. By solving an LP, Row (the maximizer) can determine a strategy for herself that guarantees an expected outcome of at least V no matter what Column does. And by solving the dual LP, Column (the minimizer) can guarantee an expected outcome of at most V, no matter what Row does. It follows that this is the uniquely defined optimal play: a priori it wasn't even certain that such a play existed. V is known as the *value* of the game. In our example, 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 $(2/7, 5/7)$.

This example is easily generalized to arbitrary games and shows the existence of mixed strategies that are optimal for both players and achieve the same value—a fundamental result of game theory called the *min-max theorem*. It can be written in equation form as follows:

$$\max_{\mathbf{x}} \min_{\mathbf{y}} \sum_{i,j} G_{ij} x_i y_j \quad = \quad \min_{\mathbf{y}} \max_{\mathbf{x}} \sum_{i,j} G_{ij} x_i y_j.$$

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 did with maximum flows and minimum cuts.

Figure 7.12 A polyhedron defined by seven inequalities.



$$
\begin{array}{rr}
\max \quad x_1 + 6x_2 + 13x_3 & \\
x_1 \le 200 & ① \\
x_2 \le 300 & ② \\
x_1 + x_2 + x_3 \le 400 & ③ \\
x_2 + 3x_3 \le 600 & ④ \\
x_1 \ge 0 & ⑤ \\
x_2 \ge 0 & ⑥ \\
x_3 \ge 0 & ⑦
\end{array}
$$

7.6 The simplex algorithm

The extraordinary power and expressiveness of linear programs would be little consolation if we did not have a way to solve them efficiently. This is the role of the simplex algorithm.

At a high level, the simplex algorithm takes a set of linear inequalities and a linear objective function and finds the optimal feasible point by the following strategy:

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

In our 2D and 3D examples (Figure 7.1 and Figure 7.2), this was simple to visualize and made intuitive sense. But what if there are n variables, $x_1, \ldots, x_n$?

Any setting of the x_i's can be represented by an n-tuple of real numbers and plotted in n-dimensional space. A linear equation involving the x_i's defines a *hyperplane* in this same space $\mathbb{R}^n$, and the corresponding linear inequality defines a *half-space*, all points that are either precisely on the hyperplane or lie on one particular side of it. Finally, the feasible region of the linear program is specified by a set of inequalities and is therefore the intersection of the corresponding half-spaces, a convex polyhedron.

But what do the concepts of *vertex* and *neighbor* mean in this general context?

7.6.1 Vertices and neighbors in n-dimensional space

Figure 7.12 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 that satisfies them with equality, and this point happens to be feasible, then it is a *vertex*.

How many equations are needed to uniquely identify a point? When there are n variables, we need at least n linear equations if we want a unique solution. On the other hand, having more than n equations is redundant: at least one of them can be rewritten as a linear combination of the others and can therefore be disregarded. In short,

> Each vertex is specified by a set of n inequalities.[3]

A notion of *neighbor* now follows naturally.

> Two vertices are *neighbors* if they have $n - 1$ defining inequalities in common.

In Figure 7.12, for instance, vertices A and C share the two defining inequalities $\{③, ⑦\}$ and are thus neighbors.

7.6.2 The algorithm

On each iteration, simplex has two tasks:

1. Check whether the current vertex is optimal (and if so, halt).

2. Determine where to move next.

As we will see, both tasks are easy if the vertex happens to be at the origin. And if the vertex is elsewhere, we will transform the coordinate system to move it to the origin!

First let's see why the origin is so convenient. Suppose we have some generic LP

$$\begin{gathered}\max\ \mathbf{c}^T\mathbf{x}\\ \mathbf{A}\mathbf{x} \le \mathbf{b}\\ \mathbf{x} \ge 0\end{gathered}$$

where $\mathbf{x}$ is the vector of variables, $\mathbf{x} = (x_1, \ldots, x_n)$. Suppose the origin is feasible. Then it is certainly a vertex, since it is the unique point at which the n inequalities $\{x_1 \ge 0, \ldots, x_n \ge 0\}$ are *tight*. Now let's solve our two tasks. Task 1:

> The origin is optimal if and only if all $c_i \le 0$.

[3]There is one tricky issue here. It is possible that the same vertex might be generated by different subsets of inequalities. In Figure 7.12, 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.

If all $c_i \leq 0$, then considering the constraints $\mathbf{x} \geq 0$, we can't hope for a better objective value. Conversely, if some $c_i > 0$, then the origin is not optimal, since we can increase the objective function by raising x_i.

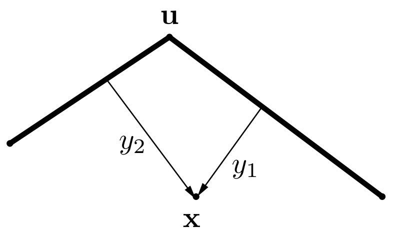
Thus, for task 2, we can move by increasing some x_i for which $c_i > 0$. How much can we increase it? *Until we hit some other constraint*. That is, we release the tight constraint $x_i \geq 0$ and increase x_i until some other inequality, previously loose, now becomes tight. At that point, we again have exactly n tight inequalities, so we are at a new vertex.

For instance, suppose we're dealing with the following linear program.

$$\begin{array}{rcll} \multicolumn{3}{c}{\max\ 2x_1 + 5x_2} & \\ 2x_1 - x_2 & \leq & 4 & \text{①} \\ x_1 + 2x_2 & \leq & 9 & \text{②} \\ -x_1 + x_2 & \leq & 3 & \text{③} \\ x_1 & \geq & 0 & \text{④} \\ x_2 & \geq & 0 & \text{⑤} \end{array}$$

Simplex can be started at the origin, which is specified by constraints ④ and ⑤. To move, we release the tight constraint $x_2 \geq 0$. As x_2 is gradually increased, the first constraint it runs into is $-x_1 + x_2 \leq 3$, and thus it has to stop at $x_2 = 3$, at which point this new inequality is tight. The new vertex is thus given by ③ and ④.

So we know what to do if we are at the origin. But what if our current vertex $\mathbf{u}$ is elsewhere? The trick is to transform $\mathbf{u}$ into the origin, by shifting the coordinate system from the usual $(x_1, \ldots, x_n)$ to the "local view" from $\mathbf{u}$. These local coordinates consist of (appropriately scaled) distances $y_1, \ldots, y_n$ to the n hyperplanes (inequalities) that define and enclose $\mathbf{u}$:



Specifically, if one of these enclosing inequalities is $\mathbf{a}_i \cdot \mathbf{x} \leq b_i$, then the distance from a point $\mathbf{x}$ to that particular "wall" is

$$y_i = b_i - \mathbf{a}_i \cdot \mathbf{x}.$$

The n equations of this type, one per wall, define the y_i's as linear functions of the x_i's, and this relationship can be inverted to express the x_i's as a linear function of the y_i's. Thus we can rewrite the entire LP in terms of the y's. This doesn't fundamentally change it (for instance, the optimal value stays the same), but expresses it in a different coordinate frame. The revised "local" LP has the following three properties:

1. It includes the inequalities $\mathbf{y} \geq 0$, which are simply the transformed versions of the inequalities defining $\mathbf{u}$.

2. $\mathbf{u}$ itself is the origin in $\mathbf{y}$-space.

3. The cost function becomes $\max c_{\mathbf{u}} + \tilde{\mathbf{c}}^T \mathbf{y}$, where $c_{\mathbf{u}}$ is the value of the objective function at $\mathbf{u}$ and $\tilde{\mathbf{c}}$ is a transformed cost vector.

In short, we are back to the situation we know how to handle! Figure 7.13 shows this algorithm in action, continuing with our earlier example.

The simplex algorithm is now fully defined. It moves from vertex to neighboring vertex, stopping when the objective function is locally optimal, that is, when the coordinates of the local cost vector are all zero or negative. As we've just seen, a vertex with this property must also be globally optimal. On the other hand, if the current vertex is not locally optimal, then its local coordinate system includes some dimension along which the objective function can be improved, so we move along this direction—along this edge of the polyhedron—until we reach a neighboring vertex. By the nondegeneracy assumption (see footnote 3 in Section 7.6.1), this edge has nonzero length, and so we strictly improve the objective value. Thus the process must eventually halt.

Figure 7.13 Simplex in action.

Initial LP:

$$\max\ 2x_1 + 5x_2$$

$$\begin{array}{rcll} 2x_1 - x_2 & \le & 4 & \text{①} \\ x_1 + 2x_2 & \le & 9 & \text{②} \\ -x_1 + x_2 & \le & 3 & \text{③} \\ x_1 & \ge & 0 & \text{④} \\ x_2 & \ge & 0 & \text{⑤} \end{array}$$

Current vertex: {④, ⑤} (origin).
Objective value: 0.

Move: increase x_2.
⑤ is released, ③ becomes tight. Stop at $x_2 = 3$.

New vertex {④, ③} has local coordinates (y_1, y_2):

$$y_1 = x_1, \quad y_2 = 3 + x_1 - x_2$$

Rewritten LP:

$$\max\ 15 + 7y_1 - 5y_2$$

$$\begin{array}{rcll} y_1 + y_2 & \le & 7 & \text{①} \\ 3y_1 - 2y_2 & \le & 3 & \text{②} \\ y_2 & \ge & 0 & \text{③} \\ y_1 & \ge & 0 & \text{④} \\ -y_1 + y_2 & \le & 3 & \text{⑤} \end{array}$$

Current vertex: {④, ③}.
Objective value: 15.

Move: increase y_1.
④ is released, ② becomes tight. Stop at $y_1 = 1$.

New vertex {②, ③} has local coordinates (z_1, z_2):

$$z_1 = 3 - 3y_1 + 2y_2, \quad z_2 = y_2$$

Rewritten LP:

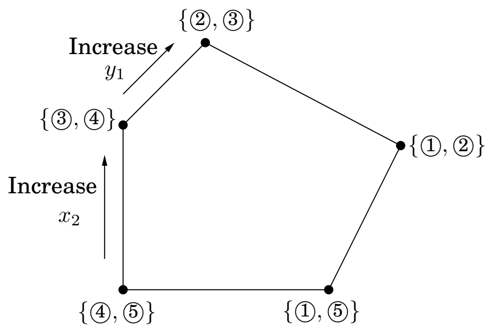
$$\max\ 22 - \tfrac{7}{3}z_1 - \tfrac{1}{3}z_2$$

$$\begin{array}{rcll} -\tfrac{1}{3}z_1 + \tfrac{5}{3}z_2 & \le & 6 & \text{①} \\ z_1 & \ge & 0 & \text{②} \\ z_2 & \ge & 0 & \text{③} \\ \tfrac{1}{3}z_1 - \tfrac{2}{3}z_2 & \le & 1 & \text{④} \\ \tfrac{1}{3}z_1 + \tfrac{1}{3}z_2 & \le & 4 & \text{⑤} \end{array}$$

Current vertex: {②, ③}.
Objective value: 22.

Optimal: all $c_i < 0$.

Solve ②, ③ (in original LP) to get optimal solution $(x_1, x_2) = (1, 4)$.



7.6.3 Loose ends

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

The starting vertex. How do we find a vertex at which to start simplex? In our 2D and 3D examples we always started at the origin, which worked because the linear programs happened to have inequalities with positive right-hand sides. In a general LP we won't always be so fortunate. However, it turns out that finding a starting vertex *can be reduced to an LP* and solved by simplex!

To see how this is done, start with any linear program in standard form (recall Section 7.1.4), since we know LPs can always be rewritten this way.

$$\min\ \mathbf{c}^T\mathbf{x}\ \text{ such that }\ \mathbf{A}\mathbf{x} = \mathbf{b} \text{ and } \mathbf{x} \geq 0.$$

We first make sure that the right-hand sides of the equations are all nonnegative: if $b_i < 0$, just multiply both sides of the ith equation by -1.

Then we create a new LP as follows:

- Create m new *artificial variables* $z_1, \ldots, z_m \geq 0$, where m is the number of equations.
- Add z_i to the left-hand side of the ith equation.
- Let the objective, to be *minimized*, be $z_1 + z_2 + \cdots + z_m$.

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, to obtain the optimum solution.

There are two cases. If the optimum value of $z_1 + \cdots + z_m$ is zero, then all z_i's obtained by simplex are zero, and hence from the optimum vertex of the new LP we get a starting feasible vertex of the original LP, just by ignoring the z_i's. We can at last start simplex!

But what if the optimum objective turns out to be positive? Let us think. We tried to minimize the sum of the z_i's, but simplex decided that it cannot be zero. But this means that the original linear program is infeasible: it *needs* some nonzero z_i's to become feasible. This is how simplex discovers and reports that an LP is infeasible.

Degeneracy. In the polyhedron of Figure 7.12 vertex B is *degenerate*. Geometrically, this means that it is the intersection of more than $n = 3$ faces of the polyhedron (in this case, ②, ③, ④, ⑤). Algebraically, it means that if we choose any one of four sets of three inequalities ({②, ③, ④}, {②, ③, ⑤}, {②, ④, ⑤}, and {③, ④, ⑤}) and solve the corresponding system of three linear equations in three unknowns, we'll get the same solution in all four cases: $(0, 300, 100)$. This is a serious problem: simplex may return a suboptimal degenerate vertex simply because all its neighbors are identical to it and thus have no better objective. And if we modify simplex so that it detects degeneracy and continues to hop from vertex to vertex despite lack of any improvement in the cost, it may end up looping forever.

One way to fix this is by a *perturbation*: change each b_i by a tiny random amount to $b_i \pm \epsilon_i$. This doesn't change the essence of the LP since the ϵ_i's are tiny, but it has the effect of differentiating between the solutions of the linear systems. To see why geometrically, imagine that the four planes ②, ③, ④, ⑤ were jolted a little. Wouldn't vertex B split into two vertices, very close to one another?

Unboundedness. In some cases an LP is unbounded, in that its objective function can be made arbitrarily large (or small, if it's a minimization problem). If this is the case, simplex will discover it: in exploring the neighborhood of a vertex, it will notice that taking out an inequality and adding another leads to an underdetermined system of equations that has an infinity of solutions. And in fact (this is an easy test) the space of solutions contains a whole line across which the objective can become larger and larger, all the way to ∞. In this case simplex halts and complains.

7.6.4 The running time of simplex

What is the running time of simplex, for a generic linear program

$$\max\ \mathbf{c}^T\mathbf{x} \text{ such that } \mathbf{A}\mathbf{x} \leq 0 \text{ and } \mathbf{x} \geq 0,$$

where there are n variables and $\mathbf{A}$ contains m inequality constraints? Since it is an iterative algorithm that proceeds from vertex to vertex, let's start by computing the time taken for a single iteration. Suppose the current vertex is $\mathbf{u}$. By definition, it is the unique point at which n inequality constraints are satisfied with equality. Each of its neighbors shares $n-1$ of these inequalities, so $\mathbf{u}$ can have at most $n \cdot m$ neighbors: choose which inequality to drop and which new one to add.

A naive way to perform an iteration would be to check each potential neighbor to see whether it really is a vertex of the polyhedron and to determine its cost. Finding the cost is quick, just a dot product, but checking whether it is a true vertex involves solving a system of n equations in n unknowns (that is, satisfying the n chosen inequalities exactly) and checking whether the result is feasible. By Gaussian elimination (see the following box) this takes $O(n^3)$ time, giving an unappetizing running time of $O(mn^4)$ per iteration.

Fortunately, there is a much better way, and this mn^4 factor can be improved to mn, making simplex a practical algorithm. Recall our earlier discussion (Section 7.6.2) about the *local view* from vertex $\mathbf{u}$. It turns out that the per-iteration overhead of rewriting the LP in terms of the current local coordinates is just $O((m+n)n)$; this exploits the fact that the local view changes only slightly between iterations, in just one of its defining inequalities.

Next, to select the best neighbor, we recall that the (local view of) the objective function is of the form "$\max\ c_{\mathbf{u}} + \tilde{\mathbf{c}} \cdot \mathbf{y}$" where $c_{\mathbf{u}}$ is the value of the objective function at $\mathbf{u}$. This immediately identifies a promising direction to move: we pick any $\tilde{c}_i > 0$ (if there is none, then the current vertex is optimal and simplex halts). Since the rest of the LP has now been rewritten in terms of the $\mathbf{y}$-coordinates, it is easy to determine how much y_i can be increased before some other inequality is violated. (And if we can increase y_i indefinitely, we know the LP is unbounded.)

It follows that the running time per iteration of simplex is just $O(mn)$. But how many iterations could there be? Naturally, there can't be more than $\binom{m+n}{n}$, which is an upper bound on the number of vertices. But this upper bound is exponential in n. And in fact, there are examples of LPs for which simplex does indeed take an exponential number of iterations. In other words, *simplex is an exponential-time algorithm.* However, such exponential examples do not occur in practice, and it is this fact that makes simplex so valuable and so widely used.

Gaussian elimination

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

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

The high school method for solving such systems is to repeatedly apply the following rule: *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

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

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 1! 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—once more due to Gauss.

```
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''
if $n = 1$:  return $b_1/a_{11}$

choose the coefficient $a_{p1}$ of largest magnitude, 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 $(x_1, \ldots, x_n)$
```

(When choosing the equation to swap into first place, we pick the one with largest $|a_{p1}|$ for reasons of *numerical accuracy*; after all, we will be dividing by a_{p1}.)

Gaussian elimination uses $O(n^2)$ *arithmetic operations* to reduce the problem size from n to $n-1$, and thus uses $O(n^3)$ operations overall. To show that this is also a good estimate of the total *running time*, we need to argue that the numbers involved remain polynomially bounded—for instance, that the solution $(x_1, \ldots, x_n)$ does not require too much more precision to write down than the original coefficients a_{ij} and b_i. Do you see why this is true?

Linear programming in polynomial time

Simplex is not a polynomial time algorithm. Certain rare kinds of linear programs cause it to go from one corner of the feasible region to a better corner and then to a still better one, and so on for an exponential number of steps. For a long time, linear programming was considered a paradox, a problem that can be solved in practice, but not in theory!

Then, in 1979, a young Soviet mathematician called Leonid Khachiyan came up with the *ellipsoid algorithm*, one that is very different from simplex, extremely simple in its conception (but sophisticated in its proof) and yet one that solves any linear program in polynomial time. Instead of chasing the solution from one corner of the polyhedron to the next, Khachiyan's algorithm confines it to smaller and smaller ellipsoids (skewed high-dimensional balls). When this algorithm was announced, it became a kind of "mathematical Sputnik," a splashy achievement that had the U.S. establishment worried, in the height of the Cold War, about the possible scientific superiority of the Soviet Union. The ellipsoid algorithm turned out to be an important theoretical advance, but did not compete well with simplex in practice. The paradox of linear programming deepened: A problem with two algorithms, one that is efficient in theory, and one that is efficient in practice!

A few years later Narendra Karmarkar, a graduate student at UC Berkeley, came up with a completely different idea, which led to another provably polynomial algorithm for linear programming. Karmarkar's algorithm is known as *the interior point method*, because it does just that: it dashes to the optimum corner not by hopping from corner to corner on the surface of the polyhedron like simplex does, but by cutting a clever path in the interior of the polyhedron. And it does perform well in practice.

But perhaps the greatest advance in linear programming algorithms was not Khachiyan's theoretical breakthrough or Karmarkar's novel approach, but an unexpected consequence of the latter: the fierce competition between the two approaches, simplex and interior point, resulted in the development of very fast code for linear programming.

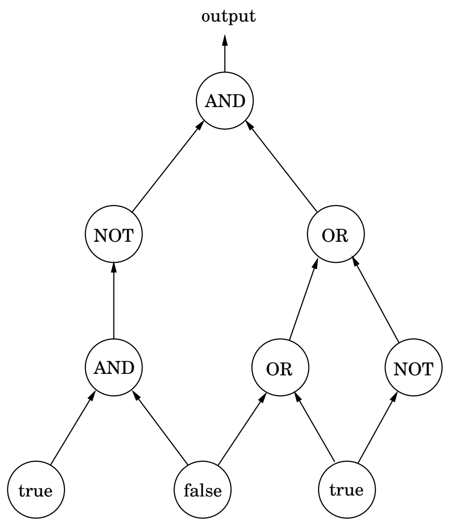
7.7 Postscript: circuit evaluation

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

We are given a *Boolean circuit*, that is, a dag of gates of the following types.

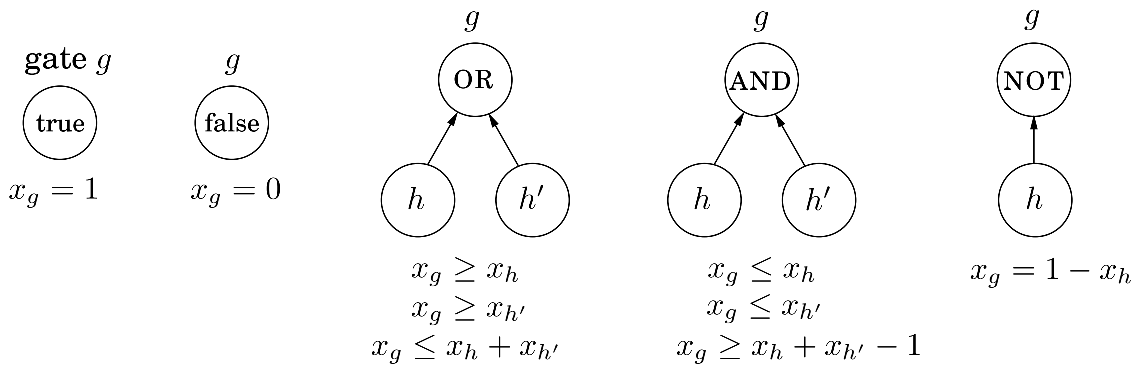
- *Input gates* have indegree zero, with value `true` or `false`.
- AND gates and OR gates have indegree 2.
- NOT gates have indegree 1.

In addition, one of the gates is designated as the *output*. Here's an example.



The CIRCUIT VALUE problem is the following: when the laws of Boolean logic are applied to the gates in topological order, does the output evaluate to `true`?

There is a simple, automatic way of translating this problem into a linear program. Create a variable x_g for each gate g, with constraints $0 \leq x_g \leq 1$. Add additional constraints for each type of gate:



These constraints force all the gates to take on exactly the right values—0 for `false`, and 1 for `true`. We don't need to maximize or minimize anything, and we can read the answer off from the variable x_o corresponding to the output gate.

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!* After all, any algorithm will eventually run 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 copies of the computer's circuit, one per unit of time, with the values of the gates in one layer used to compute the values for the next. Hence, the fact that CIRCUIT VALUE reduces to linear programming means that *all problems that can be solved in polynomial time do!*

In our next topic, **NP**-*completeness*, we shall see that many *hard* problems reduce, much the same way, to *integer programming*, linear programming's difficult twin.

Another parting thought: by what other means can the circuit evaluation problem be solved? Let's think—a circuit is a dag. And what algorithmic technique is most appropriate for solving problems on dags? That's right: dynamic programming! Together with linear programming, the world's two most general algorithmic techniques.