Chapter 7

Linear Programming and Reductions

1 Linear Programming

Many of the problems we want to solve by algorithms are *optimization* tasks: Find 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 use edges of the graph and lead from s to t, the tree must touch all nodes, the subsequence must be increasing); and (b) amongst all solutions that satisfy these constraints, we want the best possible, with respect to some well-defined criterion.

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

1.1 An introductory example

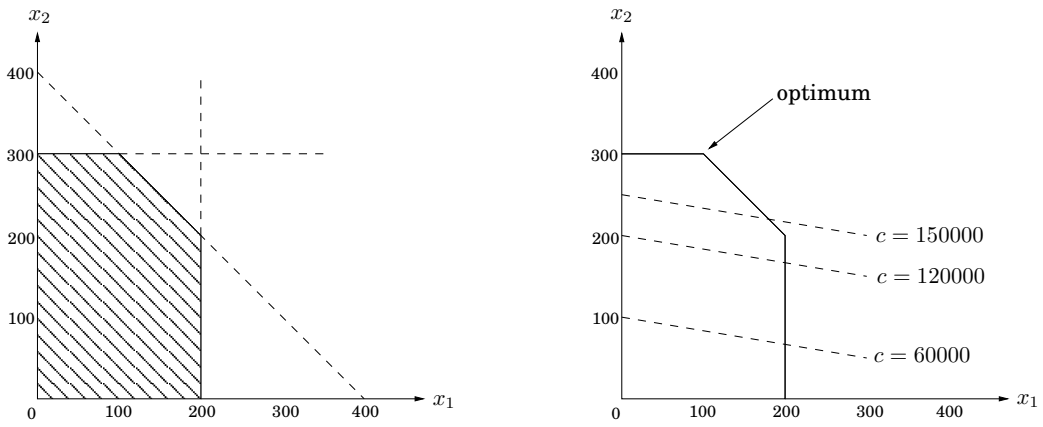
A small company has two products, and wants to figure out how much to produce of each of them so as to maximize its profits. Let's say it makes x_1 units of Product 1 per day, at a profit of $\$1$ each, and x_2 units of Product 2, at a more substantial profit of $\$6$ apiece; x_1 and x_2 are unknown 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, x_1 cannot be more than 200, and x_2 cannot be more than 300 – presumably because the demand for these products is limited. Also, the sum of the two can be at most 400, because of labor constraints (there is a fixed number of workers). What are the optimal levels of production?

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

$$\begin{aligned} \max\ 1x_1 + 6x_2 \\ x_1 &\le 200 \\ x_2 &\le 300 \\ x_1 + x_2 &\le 400 \\ x_1, x_2 &\ge 0 \end{aligned}$$

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.

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.



We wish to maximize the linear objective function $1x_1 + 6x_2$ over all points in this polygon. The equation $1x_1 + 6x_2 = c$ 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, 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 pathological 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,

$$x \le 1, \quad x \ge 2.$$

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

$$\begin{aligned} \max\ & x_1 + x_2 \\ & x_1, x_2 \geq 0 \end{aligned}$$

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 of 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.

More products

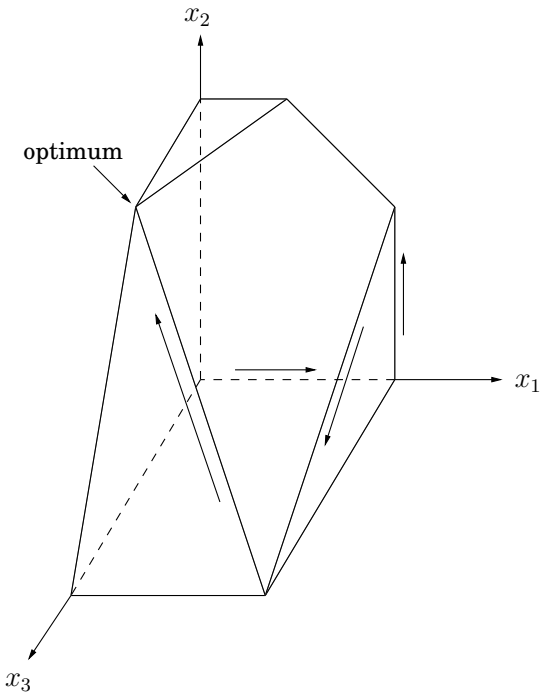
Our company decides to grow by adding a third, more lucrative, product, which will bring a profit of $\$13$ per unit. Let x_1, x_2, x_3 be the amounts of the three products. The old constraints on x_1 and x_2 persist, although the labor constraint now includes x_3 as well: the sum of all three variables 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.

$$\begin{aligned} \max\ \ 1x_1 + 6x_2 + 13x_3 & \\ x_1 & \leq 200 \\ x_2 & \leq 300 \\ x_1 + x_2 + x_3 & \leq 400 \\ x_2 + 3x_3 & \leq 600 \\ x_1, x_2, x_3 & \geq 0 \end{aligned}$$

The space of solutions is now three-dimensional. Each linear equation defines a 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).

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



A profit of c corresponds to the plane $1x_1 + 6x_2 + 13x_3 = c$. 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.

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

In this particular case, the optimum vertex is $(0, 300, 100)$, with total profit $\$3100$.

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 more rigorously.

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

1.2 Production planning

This time our company has only one product, whose demand is extremely seasonal. Our analyst has just given us demand estimates for all months of the next calendar year: $d_1, d_2, \ldots, d_{12}$. Unfortunately, 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

A magic trick called duality

Here is why you should believe $(0, 300, 100)$, with total profit 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 get it? This inequality says that no feasible solution (values x_1, x_2, x_3 satisfying the constraints) can have a profit greater than 3100. We have indeed found the optimum! The only question is, the multipliers (0,1,1,4) for the four inequalities were pulled out of a hat – is there a principled way to find them?
Later in this chapter we'll see how you can always come up with such multipliers by solving another LP! Except that (it gets 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 its *dual* — that solving the original LP solves the dual as well! But we are getting far ahead of our story.

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. We have currently no stored units on hand, 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.

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

All in all, there are 72 variables.

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 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.

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

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.

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

And overtime is limited,

$$o_i \leq 6w_i.$$

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

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

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

1.3 Optimum bandwidth allocation

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

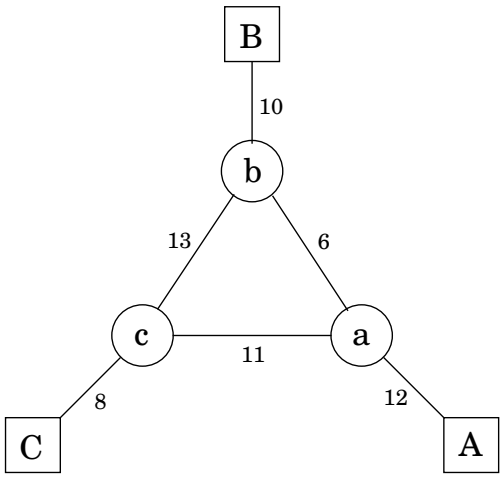
Connection 3: between A and C

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?

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.

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

Figure 1.3 A communication network; bandwidths are shown.



The solution, obtained instantaneously via simplex, 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 always 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*.

Another cautionary observation: Our LP had one variable for every possible path between the endpoints. In a larger network, there could be exponentially many paths. Formulating network problems as LPs often does not scale very well...

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 Variants of linear programming

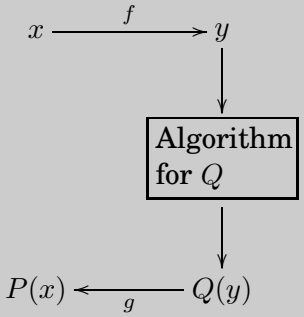
As evidenced in our examples, a general linear program can be either a maximization or a minimization problem and its constraints are allowed to be either equations or inequalities. We will now show that these various kinds of LP *can all be reduced to one another* via simple transformations. Here's how.

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:

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

Reductions

Some computational tasks Q are sufficiently general that any subroutine for them can also be used to solve other tasks P. We say P *reduces to* Q. For instance, P might be solvable by a single call to Q's subroutine, which means any instance x of P can be transformed into some instance y of Q such that $P(x)$ can be deduced from $Q(y)$:



If the pre- and post-processing procedures f and g are efficiently computable then this creates an efficient algorithm for P out of any efficient algorithm for Q!

We have already seen reductions in this book: we reduced signal processing to polynomial multiplication, and then we reduced polynomial multiplication to evaluation/interpolation, and finally reduced *that* problem to FFT. In the last chapter, we reduced the problem of finding the longest increasing subsequence of a sequence of numbers to the problem of finding the longest path in a dag (which in turn can be reduced to the shortest path problem, simply by setting all edge weights to -1).
Reductions enhance the power of algorithms: Once we have an algorithm for problem Q (which could be linear programming, for example) we can use it to solve other problems. The reason linear programming is important is because it has many *applications* — which is another way of saying that many problems reduce to it.

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.

2. To change an equality constraint into inequalities is trivial: $ax = b$ is equivalent to $ax \leq b, ax \geq b$.

3. 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.

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

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.

$$\begin{aligned} \min\ -1x_1 - 6x_2 & \\ x_1 + s_1 &= 200 \\ x_2 + s_2 &= 300 \\ x_1 + x_2 + s_3 &= 400 \\ x_1, x_2, s_1, s_2, s_3 &\geq 0 \end{aligned}$$

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 above.

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:

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

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

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

2 The simplex algorithm

At a high level, the simplex algorithm takes a set of 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'
```

This is simple to visualize in our two- and three-dimensional examples (Figures 1.1 and 1.2). The key to understanding it in generality is to study the geometry of n-dimensional space — the space we must work in when the linear program has n variables. Once we have a feel for basic geometric notions such as *vertex* and *hyperplane* in this context, the simplex algorithm becomes easy to understand and to implement.

2.1 Geometry of $\mathbb{R}^n$

Let's start with a generic linear program

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

where $\mathbf{x}$ is a vector of n variables. We'll think of these variables as labeling coordinate directions in an n-dimensional Euclidean space: a point in this space is an assignment of values to the variables, represented by an n-tuple of real numbers. A linear equation defines a *hyperplane* while a linear inequality defines a *half-space*, the set of all points which are either precisely on the hyperplane or lie on one particular side of it.

The feasible region of the linear program is described by a set of inequalities and is therefore the intersection of the corresponding half-spaces, a convex polyhedron.

What is a vertex?

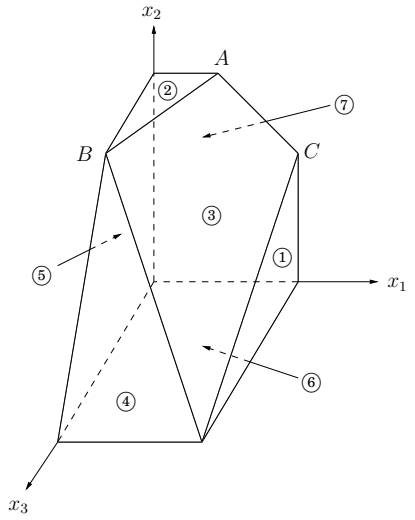
Figure 2.1 recalls an earlier example. Looking at it closely, we see that *each vertex is the unique point at which some subset of hyperplanes meet*. Vertex A, for instance, is the sole point at which constraints ②, ③ and ⑦ are satisfied with equality. On the other hand, the hyperplanes corresponding 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 equations 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 hyperplanes 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

Figure 2.1 A polyhedron defined by seven inequalities.



$$\begin{aligned} \max \quad & 1x_1 + 6x_2 + 13x_3 & \\ & x_1 \leq 200 & ① \\ & x_2 \leq 300 & ② \\ x_1 + x_2 + x_3 & \leq 400 & ③ \\ x_2 + 3x_3 & \leq 600 & ④ \\ & x_1 \geq 0 & ⑤ \\ & x_2 \geq 0 & ⑥ \\ & x_3 \geq 0 & ⑦ \end{aligned}$$

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 sets of n inequalities. There is one tricky issue here. Although any subset of the 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.

What is a neighbor?

Now that we've defined *vertex*, all we need to complete our specification of the simplex algorithm is a notion of *neighbor*. Here it is.

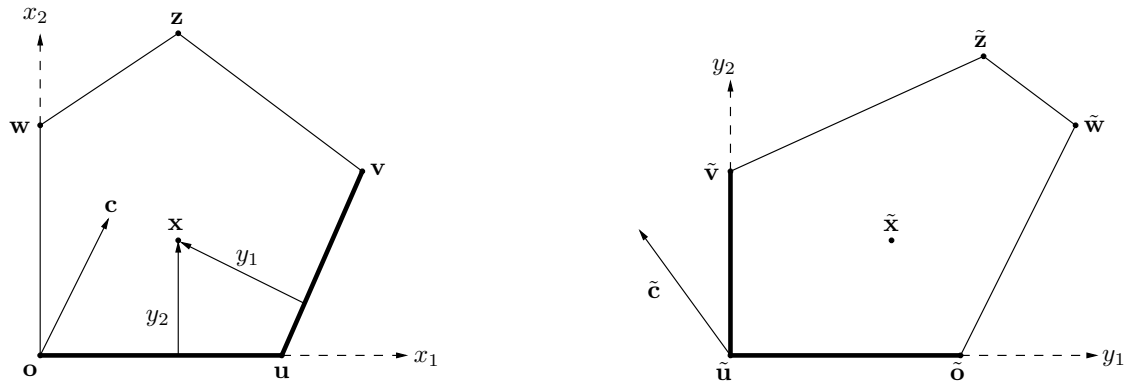
> Vertices u and v are *neighbors* if, of the two sets of n inequalities that define them, $n - 1$ are identical.

To understand this definition, we need to get a feel for the following question: from the viewpoint of a specific vertex – the current position of simplex – what does the rest of the polyhedron look like?

This is most easily answered if the vertex happens to be the origin $\mathbf{x} = 0$, the unique point of intersection of the n constraints $x_i = 0$. In its immediate vicinity, there are only these n hyperplanes (although others might be close by) because of our non-degeneracy assumption. So if one were to stand at the origin and look out into the polyhedron, one's view would be channeled between these n "walls". From this position, a natural way to describe other points in the polyhedron is in terms of distances to each of the walls, which are easy enough to determine: the distance from point $\mathbf{z} = (z_1, z_2, \ldots, z_n)$ to wall $x_i = 0$ is simply z_i.

What about the local view from some other vertex $\mathbf{u}$ of the polyhedron? By definition $\mathbf{u}$ is the intersection of n hyperplanes, the unique point at which some subset of n inequalities

Figure 2.2 Left: A 2-d feasible region whose optimal vertex is $\mathbf{z}$. The hyperplanes defining vertex $\mathbf{u}$ are shown in bold. Right: a transformation to the local view from $\mathbf{u}$.



from $\{\mathbf{A}\mathbf{x} \leq \mathbf{b}, \mathbf{x} \geq 0\}$ is satisfied with equality. These inequalities are the n walls that meet at $\mathbf{u}$. Let's generically write them as

$$\boldsymbol{\alpha}_i \cdot \mathbf{x} \leq \beta_i, \quad i = 1, \ldots, n.$$

(For instance, $x_2 \geq 0$ is the same as $-(0, 1, 0, 0, \ldots, 0) \cdot \mathbf{x} \leq 0$.) For any other point $\mathbf{x}$ in the feasible region, some of these inequalities will be loose, and we can introduce slack variables y_i to make them equalities: $\boldsymbol{\alpha}_i \cdot \mathbf{x} + y_i = \beta_i$. Geometrically, y_i *is the distance from the hyperplane* $\boldsymbol{\alpha}_i \cdot \mathbf{x} = \beta_i$, suitably scaled by $\|\boldsymbol{\alpha}_i\|$ (Figure 2.2). Moreover, $\mathbf{x}$ is completely specified by its distances from the n hyperplanes, in other words by the slack values $\mathbf{y} = (y_1, \ldots, y_n)$: more precisely, $\mathbf{x}$ and $\mathbf{y}$ are linear functions of each other (do you see why?). Thus the y_i's form a new system of coordinates for $\mathbb{R}^n$, the local view of the polyhedron as seen from vertex $\mathbf{u}$. In these new coordinates, $\mathbf{u}$ becomes the origin $\mathbf{y} = 0$, and any point in the polyhedron has $\mathbf{y} \geq 0$ since it satisfies the n constraints.

The linear relationship between coordinate systems makes it possible to rewrite the original objective function in terms of the $\mathbf{y}$ variables, as the entirely equivalent:

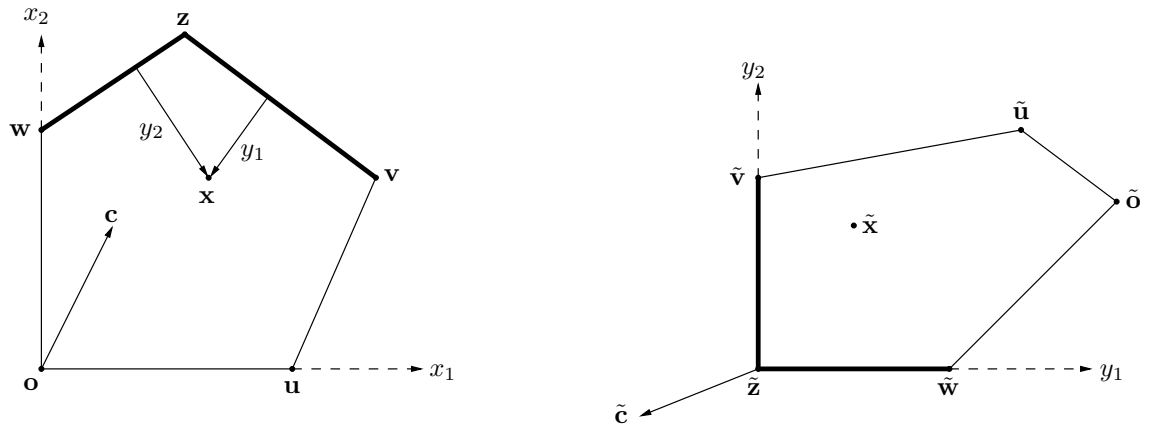
$$\max \; c_{\mathbf{u}} + \tilde{\mathbf{c}} \cdot \mathbf{y}$$

where $c_{\mathbf{u}}$ is the value of the objective function at $\mathbf{u}$.

If every coordinate of $\tilde{\mathbf{c}}$ is ≤ 0 then $c_{\mathbf{u}}$ is the maximum possible objective value (since any feasible point has $\mathbf{y} \geq 0$) and so $\mathbf{u}$ must be an optimum vertex. Geometrically this says that if the objective function $\tilde{\mathbf{c}}$ is pointing towards the negative orthant and the feasible region is in the positive orthant then the origin must be the optimum vertex (Figure 2.3).

On the other hand, if some coordinate $\tilde{c}_i > 0$, then the objective function can be improved by increasing y_i. In doing so, we release one tight constraint $\boldsymbol{\alpha}_i \cdot \mathbf{x} = \beta_i$ and effectively start sliding along an edge of the polyhedron. For instance, in Figure 2.2, we slide along $(\mathbf{u}, \mathbf{v})$. As we are moving on that edge, only $n - 1$ inequalities are satisfied with equality. We keep going until eventually some other inequality of the LP is violated. Just before this happens, that inequality is satisfied with an equality sign, and at that point we once again have n tight constraints, meaning that we have arrived at a new vertex of the polyhedron. Thus the starting

Figure 2.3 Continuation: this time, the transformation is to the local view of vertex z.



and ending points of this move are neighbors, and the edge between them is the intersection of the $n-1$ hyperplanes they have in common (or more precisely, the feasible portion of the intersection).

The simplex algorithm is now fully defined. It moves from vertex to neighboring vertex, stopping when the objective function is locally optimal, which as we've just seen implies global optimality. When 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 non-degeneracy assumption, this edge has nonzero length, and so we strictly improve the objective value. Thus the process must eventually halt.

2.2 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

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

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.

Figure 2.4 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 $(x_1, \ldots, x_n)$

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

$$\begin{aligned} x_1 \quad - \; 2x_3 \quad\quad &= 2 \\ x_2 + \; x_3 \quad\quad &= 3 \\ x_2 + 2x_3 - \; x_4 &= 2 \\ x_2 + 3x_3 + \; x_4 &= 5 \end{aligned}$$

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 2.4) — 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 $T(n) = T(n-1) + O(n^2) = O(n^3)$.

2.3 Loose ends

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

The starting vertex. In our 3-d example we had the obvious starting point $(0, 0, 0)$, which worked because the linear program had 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 LP* and solved by simplex!

To see how this is done, start with any linear program in standard form, 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 i^{th} equation by -1.

Then we create a new LP as follows:

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.

- 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 i^{th} 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 2.1 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 of 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.

2.4 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 which 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 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 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

$$\max\ c_{\mathbf{u}} + \tilde{\mathbf{c}} \cdot \mathbf{y}$$

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. In fact, there are examples of LPs in which simplex does indeed take an exponential number of iterations (we see such an example in the next section, more in the Exercises). 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.

3 Flows in networks

3.1 Shipping oil

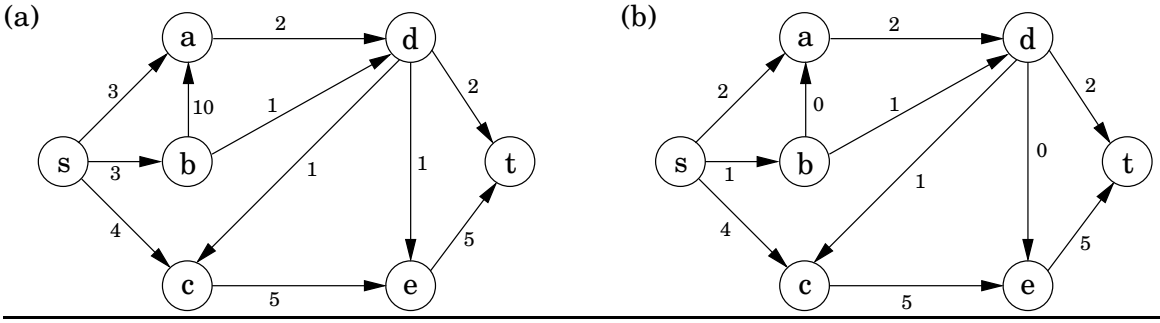
Figure 3.1(a) shows a directed graph representing a network of pipelines along which oil can be sent. The goal is to ship as much as oil as possible from the *source* s to the *sink* t. Each pipeline has a maximum *capacity* it can handle, and there are no opportunities for storing oil en route. Figure 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.

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



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, t, the amount of flow entering u equals the amount leaving u:

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

   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 $\{f_e : e \in E\}$ which 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 3.1 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 $f_{sc} + f_{ac} = f_{ce}$). 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.

3.3 A closer look at the algorithm

Now that we have reduced the max-flow problem to linear programming, we might want to examine how simplex handles the resulting LPs, as a source of inspiration for designing *direct* max-flow algorithms. In fact, a careful analysis (Exercise) shows that the behavior of simplex on flow LPs 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 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 gets around this problem by also allowing paths to *cancel existing flow*. In this particular case, it would subsequently choose the 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.

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.

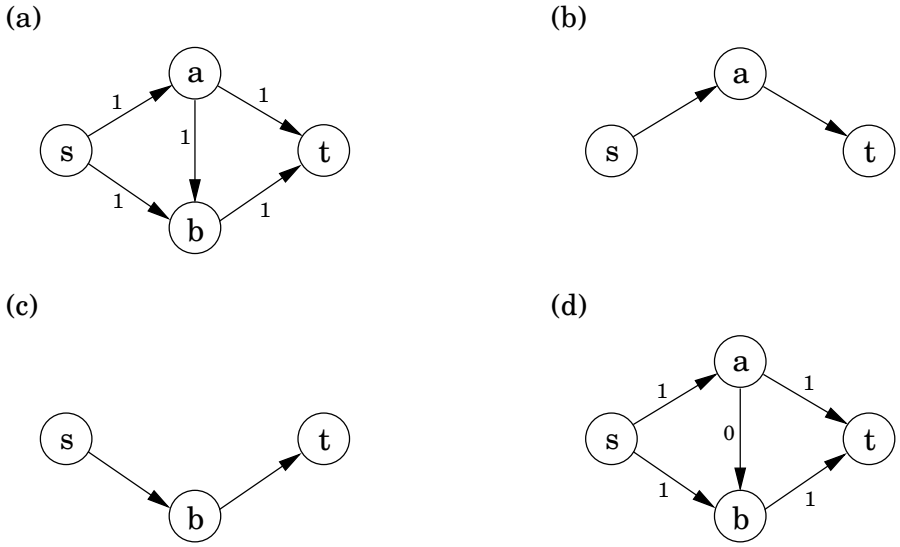
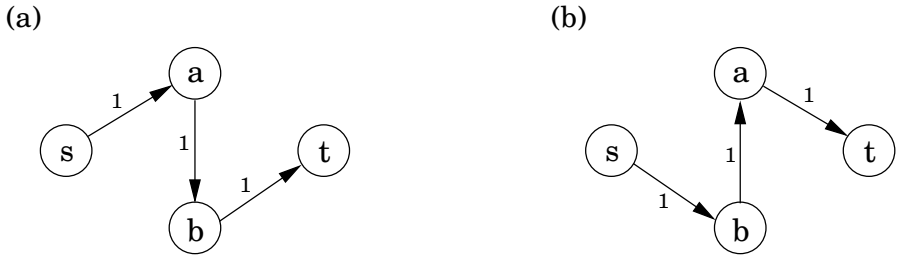


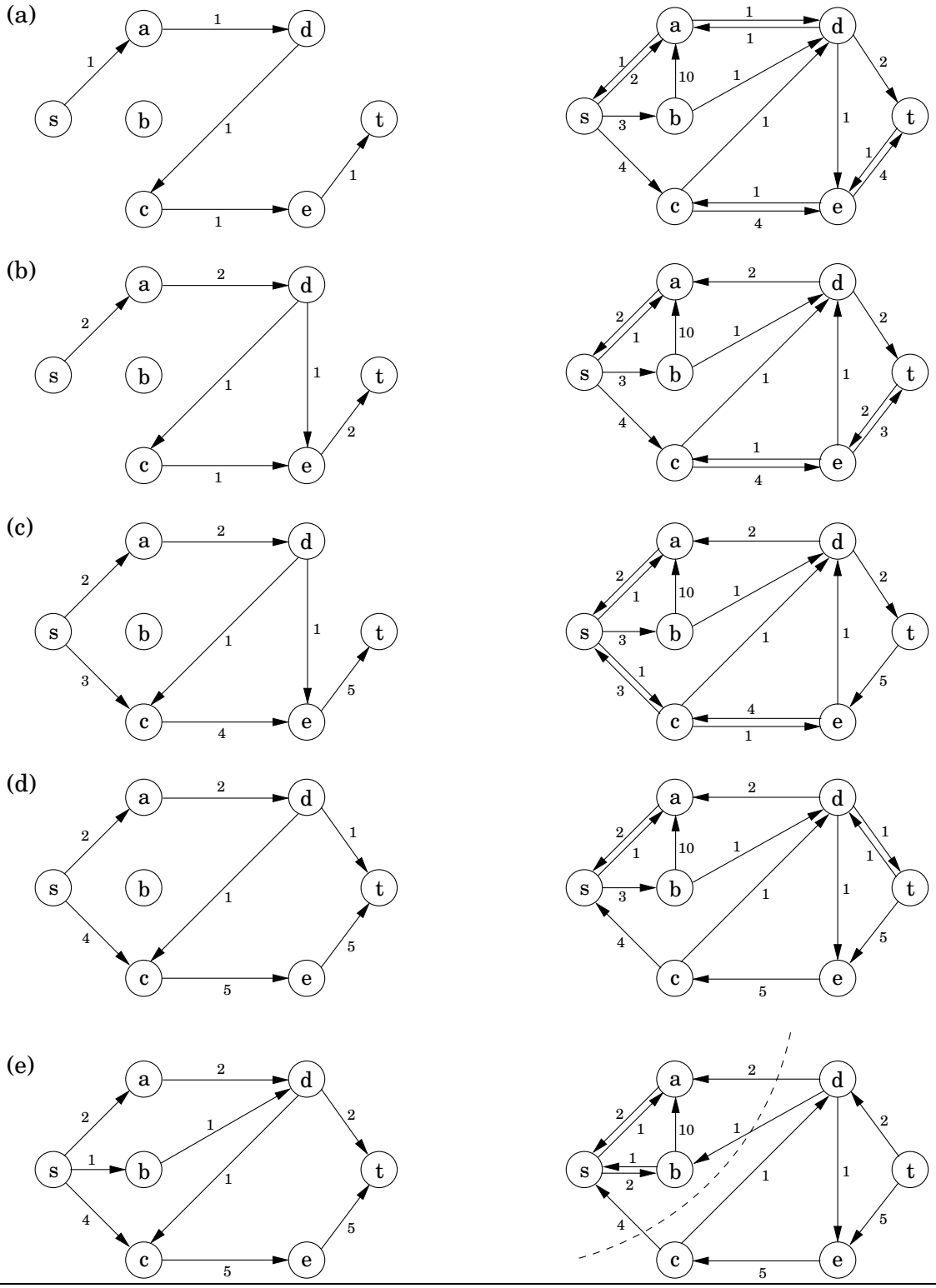
Figure 3.3 (a) We could have chosen this path first. (b) In which case, we would have to allow this as a second path.



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)).

By simulating the behavior of simplex, we get a direct algorithm for solving max-flow. It proceeds in iterations, each time finding a suitable $s - t$ path by using, say, linear-time breadth-first search; and it halts when there is no longer any such path along which flow can be increased. In fact, breadth-first search is a much better idea than depth-first search in this context because it ensures (Exercise) that the overall running time of the algorithm is polynomial! Figure 3.4 illustrates the algorithm on our oil example.

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 (the network of "flow-increasing opportunities" consisting of kinds (1) and (2) of edges described in the text) 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.



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

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

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.

4 Duality

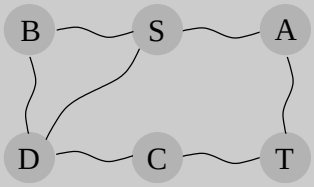
We have seen that in networks, flows are smaller than cuts, but the maximum flow and minimum cut exactly coincide and each is therefore a certificate of the other's optimality. This *duality* between the two problems is a general property of linear programs. Every linear maximization problem has a dual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

To understand duality, let us recall our LP with the three products:

$$\begin{array}{rrrrrrr} \max & 1x_1 & + & 6x_2 & + & 13x_3 & \\ & x_1 & & & & & \leq 200 \\ & & & x_2 & & & \leq 300 \\ & x_1 & + & x_2 & + & x_3 & \leq 400 \\ & & & x_2 & + & 3x_3 & \leq 600 \\ & & & & & x_1, x_2, x_3 & \geq 0 \end{array}$$

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 you notice that: Shortest path is a *minimization* problem, right? Then why are we *pulling* t away from s, an act whose purpose is, obviously, *maximization?* Answer: by pulling s away from t we solve *the dual* of the shortest path problem...

Its optimum solution is $(x_1, x_2, x_3) = (0, 1300, 100)$, with objective value 3100. When we first solved this LP we noticed something remarkable: we can multiply its inequalities with certain coefficients (0, 1, 1, and 4, respectively) and add the resulting inequalities together to obtain $1x_1 + 6x_2 + 13x_3 \leq 3100$, a certificate of optimality. The only question is, how can such multipliers be found? What is below the table in this magic trick?

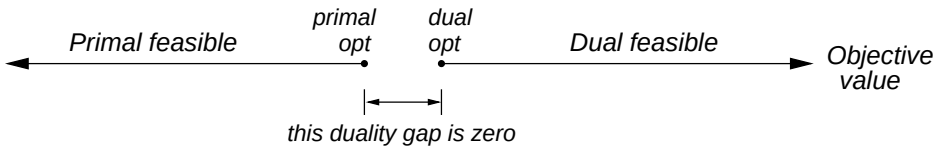
Let us try to describe what we expect of these four multipliers, call them y_1, y_2, y_3, y_4. For starters, they must of course be nonnegative, for otherwise they are unqualified to multiply inequalities. Let's focus on the coefficient of x_1 for concreteness. After the multiplication and addition steps, we get a final inequality in which x_1's coefficient is $y_1 + y_3$. To get a certificate of optimality, this value should be 1, the coefficient of x_1 in the objective. In fact, if you think about it, it would be fine if $y_1 + y_3$ were larger than 1 — the certificate would be all the more convincing. This gives us an inequality constraint on the y's: $y_1 + y_3 \geq 1$. Similarly, looking at the coefficients of x_2 and x_3 we get two more inequalities: $y_2 + y_3 + y_4 \geq 6$ and $y_3 + 3y_4 \geq 13$. For any multipliers y_1, y_2, y_3, y_4 satisfying these 3 inequalities, we get an upper bound on the optimum of the LP above:

$$1x_1 + 6x_2 + 13x_3 \ \leq \ 200y_1 + 300y_2 + 400y_3 + 600y_4.$$

But something is still missing: we can easily find y's that satisfy the 3 inequalities by simply taking them to be large enough, for example $(5, 3, 6, 10)$. These particular values of the y's do give a bound: they tell us that the optimum solution of the LP is at most $200 \cdot 5 + 300 \cdot 3 + 400 \cdot 6 + 600 \cdot 10 = 10300$. The trouble is that 10300 is far too large to be an interesting bound. We want upper bounds on our original maximization LP that are as tight — as low — as possible. In other words, *we want to minimize* $200y_1 + 300y_2 + 400y_3 + 600y_4$ *subject to these three inequalities* — a new linear program!

To summarize: finding multipliers that yield tight upper bounds on our original LP is

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



tantamount to solving a new LP:

$$\begin{array}{rrrrrrrrl} \min & 200y_1 & + & 300y_2 & + & 400y_3 & + & 600y_4 & \\ & y_1 & & & + & y_3 & & & \geq \ 1 \\ & & & y_2 & + & y_3 & + & y_4 & \geq \ 6 \\ & & & & & y_3 & + & 3y_4 & \geq \ 13 \\ & & & & & y_1, y_2, y_3, y_4 & & & \geq \ 0 \end{array}$$

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:

$$\text{Primal} : (x_1, x_2, x_3) = (0, 300, 100); \quad \text{Dual} : (y_1, y_2, y_3, y_4) = (0, 1, 1, 4).$$

They both have value 3100, and therefore they certify each other's optimality (Figure 4.1).

This wasn't just a lucky example, but a general phenomenon. The construction above (writing one constraint in the dual for every variable of the primal, requiring each sum to be above the objective coefficient of the corresponding variable, and optimizing the sum of these variables weighted by the right-hand sides), can be carried out for any LP. This transformation is summarized in Figure 4.2, and even more generically in Figure 4.3. The only new item in the latter figure is the following: If we have 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.

When the primal is the LP that expresses the max-flow problem, it is possible to assign interpretations to the dual variables which show the dual to be none other than the minimum cut problem (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,

Figure 4.2 Constructing the dual.

① Primal LP: $$\max \ \mathbf{c}^T\mathbf{x}$$ $$\mathbf{A}\mathbf{x} \le \mathbf{b}$$ $$\mathbf{x} \ge 0$$ where $\mathbf{A}$ has rows $\mathbf{a}_1, \ldots, \mathbf{a}_m$, and $\mathbf{x}$ has n coordinates.	② Introduce nonnegative dual variables $y_1, \ldots, y_m$, one per inequality of $\mathbf{A}\mathbf{x} \le \mathbf{b}$. $$y_i \times \{\mathbf{a}_i^T\mathbf{x} \le b_i\} \text{ gives } \ y_i\mathbf{a}_i^T\mathbf{x} \le y_i b_i$$ Adding them up: $$(y_1\mathbf{a}_1^T + \cdots + y_m\mathbf{a}_m^T)\mathbf{x} \le \mathbf{y}^T\mathbf{b}.$$
③ $\mathbf{y}^T\mathbf{b}$ is an upper bound on the primal if $$y_1\mathbf{a}_1^T + \cdots + y_m\mathbf{a}_m^T \ge \mathbf{c}^T.$$ Dual LP: $$\min \ \mathbf{y}^T\mathbf{b}$$ $$(y_1\mathbf{a}_1^T + \cdots + y_m\mathbf{a}_m^T) \ge \mathbf{c}^T$$ $$\mathbf{y} \ge 0$$	④ A more concise form of the dual: $$\min \ \mathbf{y}^T\mathbf{b}$$ $$\mathbf{y}^T\mathbf{A} \ge \mathbf{c}^T$$ $$\mathbf{y} \ge 0$$

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.

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

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 (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

[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.

Figure 4.3 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. $\mathbf{A}_j$ denotes the j^{th} column of the $(|M| + |M'|) \times (|N| + |N'|)$ matrix whose rows are the $\mathbf{a}_i$ (see Exercise ???).

Primal		*Dual*
$\max \mathbf{c}^T\mathbf{x}$		$\min \mathbf{y}^T\mathbf{b}$
$\mathbf{a}_i^T\mathbf{x} \leq b_i$	$i \in M$	$y_i \geq 0$
$\mathbf{a}_i^T\mathbf{x} = b_i$	$i \in M'$	y_i unrestricted
x_j unrestricted	$j \in N$	$\mathbf{y}^T\mathbf{A}_j = c_j$
$x_j \geq 0$	$j \in N'$	$\mathbf{y}^T\mathbf{A}_j \geq c_j$

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 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 — LP duality.

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.

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

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. 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 did with maximum flows and minimum cuts.

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

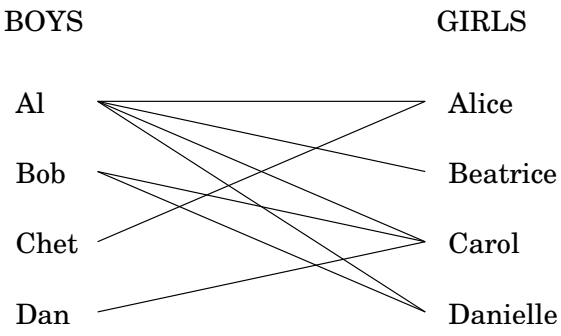
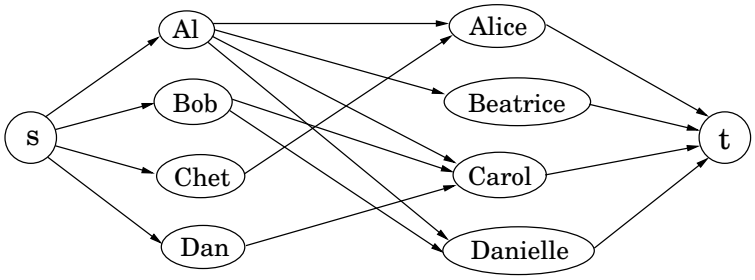


Figure 6.2 A matchmaking network. Each edge has a capacity of one.



6 Matchings

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 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, when we also demand that the variables be integers.

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*. We are asked the following question: 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 evaluation* 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 inequalities $x_g \geq x_h, x_g \geq x_{h'}, x_g \leq x_h + x_{h'}$. If it is the AND of h and h', use instead the inequalities $x_g \leq x_h, x_g \leq x_{h'}, x_g \geq x_h + x_{h'} - 1$ (notice the difference). If g is the NOT of h, just add $x_g = 1 - x_h$.

It is easy to see that 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 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!* Here is a justification of this statement: 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 superpositions of the computer's circuit, with the values of the gates in one layer used to compute the values for the next. Hence, the fact that circuit value problem reduces to LP 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*, LP'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.