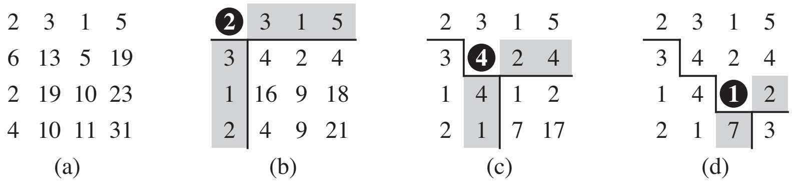

$$\underset{A}{\begin{pmatrix} 2 & 3 & 1 & 5 \\ 6 & 13 & 5 & 19 \\ 2 & 19 & 10 & 23 \\ 4 & 10 & 11 & 31 \end{pmatrix}} = \underset{L}{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 \\ 1 & 4 & 1 & 0 \\ 2 & 1 & 7 & 1 \end{pmatrix}} \underset{U}{\begin{pmatrix} 2 & 3 & 1 & 5 \\ 0 & 4 & 2 & 4 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 3 \end{pmatrix}}$$

(e)

Figure 28.1 The operation of LU-DECOMPOSITION. **(a)** The matrix A. **(b)** The element $a_{11} = 2$ in the black circle is the pivot, the shaded column is v/a_{11}, and the shaded row is w^{T}. The elements of U computed thus far are above the horizontal line, and the elements of L are to the left of the vertical line. The Schur complement matrix $A' - vw^{\mathrm{T}}/a_{11}$ occupies the lower right. **(c)** We now operate on the Schur complement matrix produced from part (b). The element $a_{22} = 4$ in the black circle is the pivot, and the shaded column and row are v/a_{22} and w^{T} (in the partitioning of the Schur complement), respectively. Lines divide the matrix into the elements of U computed so far (above), the elements of L computed so far (left), and the new Schur complement (lower right). **(d)** After the next step, the matrix A is factored. (The element 3 in the new Schur complement becomes part of U when the recursion terminates.) **(e)** The factorization $A = LU$.

trix A. (We don't need to divide by a_{kk} in line 12 because we already did so when we computed l_{ik} in line 8.) Because line 12 is triply nested, LU-DECOMPOSITION runs in time $\Theta(n^3)$.

Figure 28.1 illustrates the operation of LU-DECOMPOSITION. It shows a standard optimization of the procedure in which we store the significant elements of L and U in place in the matrix A. That is, we can set up a correspondence between each element a_{ij} and either l_{ij} (if $i > j$) or u_{ij} (if $i \le j$) and update the matrix A so that it holds both L and U when the procedure terminates. To obtain the pseudocode for this optimization from the above pseudocode, just replace each reference to l or u by a; you can easily verify that this transformation preserves correctness.

Computing an LUP decomposition

Generally, in solving a system of linear equations $Ax = b$, we must pivot on off-diagonal elements of A to avoid dividing by 0. Dividing by 0 would, of course, be disastrous. But we also want to avoid dividing by a small value—even if A is nonsingular—because numerical instabilities can result. We therefore try to pivot on a large value.

The mathematics behind LUP decomposition is similar to that of LU decomposition. Recall that we are given an $n \times n$ nonsingular matrix A, and we wish to find a permutation matrix P, a unit lower-triangular matrix L, and an upper-triangular matrix U such that $PA = LU$. Before we partition the matrix A, as we did for LU decomposition, we move a nonzero element, say a_{k1}, from somewhere in the first column to the $(1, 1)$ position of the matrix. For numerical stability, we choose a_{k1} as the element in the first column with the greatest absolute value. (The first column cannot contain only 0s, for then A would be singular, because its determinant would be 0, by Theorems D.4 and D.5.) In order to preserve the set of equations, we exchange row 1 with row k, which is equivalent to multiplying A by a permutation matrix Q on the left (Exercise D.1-4). Thus, we can write QA as

$$QA = \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ \nu & A' \end{pmatrix} ,$$

where $\nu = (a_{21}, a_{31}, \ldots, a_{n1})^{\mathrm{T}}$, except that a_{11} replaces a_{k1}; $w^{\mathrm{T}} = (a_{k2}, a_{k3}, \ldots, a_{kn})$; and A' is an $(n-1) \times (n-1)$ matrix. Since $a_{k1} \neq 0$, we can now perform much the same linear algebra as for LU decomposition, but now guaranteeing that we do not divide by 0:

$$\begin{aligned} QA &= \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ \nu & A' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ \nu/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & A' - \nu w^{\mathrm{T}}/a_{k1} \end{pmatrix} . \end{aligned}$$

As we saw for LU decomposition, if A is nonsingular, then the Schur complement $A' - \nu w^{\mathrm{T}}/a_{k1}$ is nonsingular, too. Therefore, we can recursively find an LUP decomposition for it, with unit lower-triangular matrix L', upper-triangular matrix U', and permutation matrix P', such that

$$P'(A' - \nu w^{\mathrm{T}}/a_{k1}) = L'U' .$$

Define

$$P = \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} Q ,$$

which is a permutation matrix, since it is the product of two permutation matrices (Exercise D.1-4). We now have

$$
\begin{aligned}
PA &= \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} QA \\
&= \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} \begin{pmatrix} 1 & 0 \\ \nu/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & A' - \nu w^{\mathrm{T}}/a_{k1} \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ P'\nu/a_{k1} & P' \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & A' - \nu w^{\mathrm{T}}/a_{k1} \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ P'\nu/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & P'(A' - \nu w^{\mathrm{T}}/a_{k1}) \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ P'\nu/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & L'U' \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ P'\nu/a_{k1} & L' \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & U' \end{pmatrix} \\
&= LU \, ,
\end{aligned}
$$

yielding the LUP decomposition. Because L' is unit lower-triangular, so is L, and because U' is upper-triangular, so is U.

Notice that in this derivation, unlike the one for LU decomposition, we must multiply both the column vector ν/a_{k1} and the Schur complement $A' - \nu w^{\mathrm{T}}/a_{k1}$ by the permutation matrix P'. Here is the pseudocode for LUP decomposition:

LUP-DECOMPOSITION(A)

```
n = A.rows
let π[1..n] be a new array
for i = 1 to n
    π[i] = i
for k = 1 to n
    p = 0
    for i = k to n
        if |a_ik| > p
            p = |a_ik|
            k' = i
    if p == 0
        error "singular matrix"
    exchange π[k] with π[k']
    for i = 1 to n
        exchange a_ki with a_k'i
    for i = k + 1 to n
        a_ik = a_ik / a_kk
        for j = k + 1 to n
            a_ij = a_ij - a_ik a_kj
```

Like LU-DECOMPOSITION, our LUP-DECOMPOSITION procedure replaces the recursion with an iteration loop. As an improvement over a direct implementation of the recursion, we dynamically maintain the permutation matrix P as an array π, where $\pi[i] = j$ means that the ith row of P contains a 1 in column j. We also implement the code to compute L and U "in place" in the matrix A. Thus, when the procedure terminates,

$$a_{ij} = \begin{cases} l_{ij} & \text{if } i > j , \\ u_{ij} & \text{if } i \le j . \end{cases}$$

Figure 28.2 illustrates how LUP-DECOMPOSITION factors a matrix. Lines 3–4 initialize the array π to represent the identity permutation. The outer **for** loop beginning in line 5 implements the recursion. Each time through the outer loop, lines 6–10 determine the element $a_{k'k}$ with largest absolute value of those in the current first column (column k) of the $(n - k + 1) \times (n - k + 1)$ matrix whose LUP decomposition we are finding. If all elements in the current first column are zero, lines 11–12 report that the matrix is singular. To pivot, we exchange $\pi[k']$ with $\pi[k]$ in line 13 and exchange the kth and k'th rows of A in lines 14–15, thereby making the pivot element a_{kk}. (The entire rows are swapped because in the derivation of the method above, not only is $A' - vw^{\mathrm{T}}/a_{k1}$ multiplied by P', but so is v/a_{k1}.) Finally, the Schur complement is computed by lines 16–19 in much the same way as it is computed by lines 7–12 of LU-DECOMPOSITION, except that here the operation is written to work in place.

Because of its triply nested loop structure, LUP-DECOMPOSITION has a running time of $\Theta(n^3)$, which is the same as that of LU-DECOMPOSITION. Thus, pivoting costs us at most a constant factor in time.

Exercises

28.1-1

Solve the equation

$$\begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ -6 & 5 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ 14 \\ -7 \end{pmatrix}$$

by using forward substitution.

28.1-2

Find an LU decomposition of the matrix

$$\begin{pmatrix} 4 & -5 & 6 \\ 8 & -6 & 7 \\ 12 & -7 & 12 \end{pmatrix} .$$

$$\begin{array}{c|cccc} 1 & 2 & 0 & 2 & 0.6 \\ 2 & 3 & 3 & 4 & -2 \\ 3 & 5 & 5 & 4 & 2 \\ 4 & -1 & -2 & 3.4 & -1 \end{array}$$

(a)

$$\begin{array}{c|cccc} 3 & 5 & 5 & 4 & 2 \\ 2 & 3 & 3 & 4 & -2 \\ 1 & 2 & 0 & 2 & 0.6 \\ 4 & -1 & -2 & 3.4 & -1 \end{array}$$

(b)

$$\begin{array}{c|cccc} 3 & 5 & 5 & 4 & 2 \\ 2 & 0.6 & 0 & 1.6 & -3.2 \\ 1 & 0.4 & -2 & 0.4 & -.2 \\ 4 & -0.2 & -1 & 4.2 & -0.6 \end{array}$$

(c)

$$\begin{array}{c|cccc} 3 & 5 & 5 & 4 & 2 \\ 2 & 0.6 & 0 & 1.6 & -3.2 \\ 1 & 0.4 & -2 & 0.4 & -0.2 \\ 4 & -0.2 & -1 & 4.2 & -0.6 \end{array}$$

(d)

$$\begin{array}{c|cccc} 3 & 5 & 5 & 4 & 2 \\ 1 & 0.4 & -2 & 0.4 & -0.2 \\ 2 & 0.6 & 0 & 1.6 & -3.2 \\ 4 & -0.2 & -1 & 4.2 & -0.6 \end{array}$$

(e)

$$\begin{array}{c|cccc} 3 & 5 & 5 & 4 & 2 \\ 1 & 0.4 & -2 & 0.4 & -0.2 \\ 2 & 0.6 & 0 & 1.6 & -3.2 \\ 4 & -0.2 & 0.5 & 4 & -0.5 \end{array}$$

(f)

$$\begin{array}{c|cccc} 3 & 5 & 5 & 4 & 2 \\ 1 & 0.4 & -2 & 0.4 & -0.2 \\ 2 & 0.6 & 0 & 1.6 & -3.2 \\ 4 & -0.2 & 0.5 & 4 & -0.5 \end{array}$$

(g)

$$\begin{array}{c|cccc} 3 & 5 & 5 & 4 & 2 \\ 1 & 0.4 & -2 & 0.4 & -0.2 \\ 4 & -0.2 & 0.5 & 4 & -0.5 \\ 2 & 0.6 & 0 & 1.6 & -3.2 \end{array}$$

(h)

$$\begin{array}{c|cccc} 3 & 5 & 5 & 4 & 2 \\ 1 & 0.4 & -2 & 0.4 & -0.2 \\ 4 & -0.2 & 0.5 & 4 & -0.5 \\ 2 & 0.6 & 0 & 0.4 & -3 \end{array}$$

(i)

$$\underbrace{\begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}}_{P} \underbrace{\begin{pmatrix} 2 & 0 & 2 & 0.6 \\ 3 & 3 & 4 & -2 \\ 5 & 5 & 4 & 2 \\ -1 & -2 & 3.4 & -1 \end{pmatrix}}_{A} = \underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0.4 & 1 & 0 & 0 \\ -0.2 & 0.5 & 1 & 0 \\ 0.6 & 0 & 0.4 & 1 \end{pmatrix}}_{L} \underbrace{\begin{pmatrix} 5 & 5 & 4 & 2 \\ 0 & -2 & 0.4 & -0.2 \\ 0 & 0 & 4 & -0.5 \\ 0 & 0 & 0 & -3 \end{pmatrix}}_{U}$$

(j)

Figure 28.2 The operation of LUP-DECOMPOSITION. **(a)** The input matrix A with the identity permutation of the rows on the left. The first step of the algorithm determines that the element 5 in the black circle in the third row is the pivot for the first column. **(b)** Rows 1 and 3 are swapped and the permutation is updated. The shaded column and row represent v and w^{T}. **(c)** The vector v is replaced by $v/5$, and the lower right of the matrix is updated with the Schur complement. Lines divide the matrix into three regions: elements of U (above), elements of L (left), and elements of the Schur complement (lower right). **(d)–(f)** The second step. **(g)–(i)** The third step. No further changes occur on the fourth (final) step. **(j)** The LUP decomposition $PA = LU$.

28.1-3

Solve the equation

$$\begin{pmatrix} 1 & 5 & 4 \\ 2 & 0 & 3 \\ 5 & 8 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 12 \\ 9 \\ 5 \end{pmatrix}$$

by using an LUP decomposition.

28.1-4

Describe the LUP decomposition of a diagonal matrix.

28.1-5

Describe the LUP decomposition of a permutation matrix A, and prove that it is unique.

28.1-6

Show that for all $n \geq 1$, there exists a singular $n \times n$ matrix that has an LU decomposition.

28.1-7

In LU-DECOMPOSITION, is it necessary to perform the outermost **for** loop iteration when $k = n$? How about in LUP-DECOMPOSITION?

28.2 Inverting matrices

Although in practice we do not generally use matrix inverses to solve systems of linear equations, preferring instead to use more numerically stable techniques such as LUP decomposition, sometimes we need to compute a matrix inverse. In this section, we show how to use LUP decomposition to compute a matrix inverse. We also prove that matrix multiplication and computing the inverse of a matrix are equivalently hard problems, in that (subject to technical conditions) we can use an algorithm for one to solve the other in the same asymptotic running time. Thus, we can use Strassen's algorithm (see Section 4.2) for matrix multiplication to invert a matrix. Indeed, Strassen's original paper was motivated by the problem of showing that a set of a linear equations could be solved more quickly than by the usual method.

Computing a matrix inverse from an LUP decomposition

Suppose that we have an LUP decomposition of a matrix A in the form of three matrices L, U, and P such that $PA = LU$. Using LUP-SOLVE, we can solve an equation of the form $Ax = b$ in time $\Theta(n^2)$. Since the LUP decomposition depends on A but not b, we can run LUP-SOLVE on a second set of equations of the form $Ax = b'$ in additional time $\Theta(n^2)$. In general, once we have the LUP decomposition of A, we can solve, in time $\Theta(kn^2)$, k versions of the equation $Ax = b$ that differ only in b.

We can think of the equation

$$AX = I_n , \tag{28.10}$$

which defines the matrix X, the inverse of A, as a set of n distinct equations of the form $Ax = b$. To be precise, let X_i denote the ith column of X, and recall that the unit vector e_i is the ith column of I_n. We can then solve equation (28.10) for X by using the LUP decomposition for A to solve each equation

$$AX_i = e_i$$

separately for X_i. Once we have the LUP decomposition, we can compute each of the n columns X_i in time $\Theta(n^2)$, and so we can compute X from the LUP decomposition of A in time $\Theta(n^3)$. Since we can determine the LUP decomposition of A in time $\Theta(n^3)$, we can compute the inverse A^{-1} of a matrix A in time $\Theta(n^3)$.

Matrix multiplication and matrix inversion

We now show that the theoretical speedups obtained for matrix multiplication translate to speedups for matrix inversion. In fact, we prove something stronger: matrix inversion is equivalent to matrix multiplication, in the following sense. If $M(n)$ denotes the time to multiply two $n \times n$ matrices, then we can invert a nonsingular $n \times n$ matrix in time $O(M(n))$. Moreover, if $I(n)$ denotes the time to invert a nonsingular $n \times n$ matrix, then we can multiply two $n \times n$ matrices in time $O(I(n))$. We prove these results as two separate theorems.

Theorem 28.1 (Multiplication is no harder than inversion)
If we can invert an $n \times n$ matrix in time $I(n)$, where $I(n) = \Omega(n^2)$ and $I(n)$ satisfies the regularity condition $I(3n) = O(I(n))$, then we can multiply two $n \times n$ matrices in time $O(I(n))$.

Proof Let A and B be $n \times n$ matrices whose matrix product C we wish to compute. We define the $3n \times 3n$ matrix D by

$$D = \begin{pmatrix} I_n & A & 0 \\ 0 & I_n & B \\ 0 & 0 & I_n \end{pmatrix} .$$

The inverse of D is

$$D^{-1} = \begin{pmatrix} I_n & -A & AB \\ 0 & I_n & -B \\ 0 & 0 & I_n \end{pmatrix} ,$$

and thus we can compute the product AB by taking the upper right $n \times n$ submatrix of D^{-1}.

We can construct matrix D in $\Theta(n^2)$ time, which is $O(I(n))$ because we assume that $I(n) = \Omega(n^2)$, and we can invert D in $O(I(3n)) = O(I(n))$ time, by the regularity condition on $I(n)$. We thus have $M(n) = O(I(n))$. ■

Note that $I(n)$ satisfies the regularity condition whenever $I(n) = \Theta(n^c \lg^d n)$ for any constants $c > 0$ and $d \geq 0$.

The proof that matrix inversion is no harder than matrix multiplication relies on some properties of symmetric positive-definite matrices that we will prove in Section 28.3.

Theorem 28.2 (Inversion is no harder than multiplication)
Suppose we can multiply two $n \times n$ real matrices in time $M(n)$, where $M(n) = \Omega(n^2)$ and $M(n)$ satisfies the two regularity conditions $M(n+k) = O(M(n))$ for any k in the range $0 \leq k \leq n$ and $M(n/2) \leq cM(n)$ for some constant $c < 1/2$. Then we can compute the inverse of any real nonsingular $n \times n$ matrix in time $O(M(n))$.

Proof We prove the theorem here for real matrices. Exercise 28.2-6 asks you to generalize the proof for matrices whose entries are complex numbers.

We can assume that n is an exact power of 2, since we have

$$\begin{pmatrix} A & 0 \\ 0 & I_k \end{pmatrix}^{-1} = \begin{pmatrix} A^{-1} & 0 \\ 0 & I_k \end{pmatrix}$$

for any $k > 0$. Thus, by choosing k such that $n + k$ is a power of 2, we enlarge the matrix to a size that is the next power of 2 and obtain the desired answer A^{-1} from the answer to the enlarged problem. The first regularity condition on $M(n)$ ensures that this enlargement does not cause the running time to increase by more than a constant factor.

For the moment, let us assume that the $n \times n$ matrix A is symmetric and positive-definite. We partition each of A and its inverse A^{-1} into four $n/2 \times n/2$ submatrices:

$$A = \begin{pmatrix} B & C^{\mathrm{T}} \\ C & D \end{pmatrix} \quad \text{and} \quad A^{-1} = \begin{pmatrix} R & T \\ U & V \end{pmatrix} . \tag{28.11}$$

Then, if we let

$$S = D - CB^{-1}C^{\mathrm{T}} \tag{28.12}$$

be the Schur complement of A with respect to B (we shall see more about this form of Schur complement in Section 28.3), we have

$$A^{-1} = \begin{pmatrix} R & T \\ U & V \end{pmatrix} = \begin{pmatrix} B^{-1} + B^{-1}C^{\mathrm{T}}S^{-1}CB^{-1} & -B^{-1}C^{\mathrm{T}}S^{-1} \\ -S^{-1}CB^{-1} & S^{-1} \end{pmatrix} , \tag{28.13}$$

since $AA^{-1} = I_n$, as you can verify by performing the matrix multiplication. Because A is symmetric and positive-definite, Lemmas 28.4 and 28.5 in Section 28.3 imply that B and S are both symmetric and positive-definite. By Lemma 28.3 in Section 28.3, therefore, the inverses B^{-1} and S^{-1} exist, and by Exercise D.2-6, B^{-1} and S^{-1} are symmetric, so that $(B^{-1})^{\mathrm{T}} = B^{-1}$ and $(S^{-1})^{\mathrm{T}} = S^{-1}$. Therefore, we can compute the submatrices R, T, U, and V of A^{-1} as follows, where all matrices mentioned are $n/2 \times n/2$:

1. Form the submatrices B, C, C^{T}, and D of A.
2. Recursively compute the inverse B^{-1} of B.
3. Compute the matrix product $W = CB^{-1}$, and then compute its transpose W^{T}, which equals $B^{-1}C^{\mathrm{T}}$ (by Exercise D.1-2 and $(B^{-1})^{\mathrm{T}} = B^{-1}$).
4. Compute the matrix product $X = WC^{\mathrm{T}}$, which equals $CB^{-1}C^{\mathrm{T}}$, and then compute the matrix $S = D - X = D - CB^{-1}C^{\mathrm{T}}$.
5. Recursively compute the inverse S^{-1} of S, and set V to S^{-1}.
6. Compute the matrix product $Y = S^{-1}W$, which equals $S^{-1}CB^{-1}$, and then compute its transpose Y^{T}, which equals $B^{-1}C^{\mathrm{T}}S^{-1}$ (by Exercise D.1-2, $(B^{-1})^{\mathrm{T}} = B^{-1}$, and $(S^{-1})^{\mathrm{T}} = S^{-1}$). Set T to $-Y^{\mathrm{T}}$ and U to $-Y$.
7. Compute the matrix product $Z = W^{\mathrm{T}}Y$, which equals $B^{-1}C^{\mathrm{T}}S^{-1}CB^{-1}$, and set R to $B^{-1} + Z$.

Thus, we can invert an $n \times n$ symmetric positive-definite matrix by inverting two $n/2 \times n/2$ matrices in steps 2 and 5; performing four multiplications of $n/2 \times n/2$ matrices in steps 3, 4, 6, and 7; plus an additional cost of $O(n^2)$ for extracting submatrices from A, inserting submatrices into A^{-1}, and performing a constant number of additions, subtractions, and transposes on $n/2 \times n/2$ matrices. We get the recurrence

$$\begin{aligned} I(n) &\leq 2I(n/2) + 4M(n/2) + O(n^2) \\ &= 2I(n/2) + \Theta(M(n)) \\ &= O(M(n)) . \end{aligned}$$

The second line holds because the second regularity condition in the statement of the theorem implies that $4M(n/2) < 2M(n)$ and because we assume that $M(n) = \Omega(n^2)$. The third line follows because the second regularity condition allows us to apply case 3 of the master theorem (Theorem 4.1).

It remains to prove that we can obtain the same asymptotic running time for matrix multiplication as for matrix inversion when A is invertible but not symmetric and positive-definite. The basic idea is that for any nonsingular matrix A, the matrix $A^{\mathrm{T}}A$ is symmetric (by Exercise D.1-2) and positive-definite (by Theorem D.6). The trick, then, is to reduce the problem of inverting A to the problem of inverting $A^{\mathrm{T}}A$.

The reduction is based on the observation that when A is an $n \times n$ nonsingular matrix, we have

$$A^{-1} = (A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}} ,$$

since $((A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}})A = (A^{\mathrm{T}}A)^{-1}(A^{\mathrm{T}}A) = I_n$ and a matrix inverse is unique. Therefore, we can compute A^{-1} by first multiplying A^{T} by A to obtain $A^{\mathrm{T}}A$, then inverting the symmetric positive-definite matrix $A^{\mathrm{T}}A$ using the above divide-and-conquer algorithm, and finally multiplying the result by A^{T}. Each of these three steps takes $O(M(n))$ time, and thus we can invert any nonsingular matrix with real entries in $O(M(n))$ time. ■

The proof of Theorem 28.2 suggests a means of solving the equation $Ax = b$ by using LU decomposition without pivoting, so long as A is nonsingular. We multiply both sides of the equation by A^{T}, yielding $(A^{\mathrm{T}}A)x = A^{\mathrm{T}}b$. This transformation doesn't affect the solution x, since A^{T} is invertible, and so we can factor the symmetric positive-definite matrix $A^{\mathrm{T}}A$ by computing an LU decomposition. We then use forward and back substitution to solve for x with the right-hand side $A^{\mathrm{T}}b$. Although this method is theoretically correct, in practice the procedure LUP-DECOMPOSITION works much better. LUP decomposition requires fewer arithmetic operations by a constant factor, and it has somewhat better numerical properties.

Exercises

28.2-1
Let $M(n)$ be the time to multiply two $n \times n$ matrices, and let $S(n)$ denote the time required to square an $n \times n$ matrix. Show that multiplying and squaring matrices have essentially the same difficulty: an $M(n)$-time matrix-multiplication algorithm implies an $O(M(n))$-time squaring algorithm, and an $S(n)$-time squaring algorithm implies an $O(S(n))$-time matrix-multiplication algorithm.

28.2-2

Let $M(n)$ be the time to multiply two $n \times n$ matrices, and let $L(n)$ be the time to compute the LUP decomposition of an $n \times n$ matrix. Show that multiplying matrices and computing LUP decompositions of matrices have essentially the same difficulty: an $M(n)$-time matrix-multiplication algorithm implies an $O(M(n))$-time LUP-decomposition algorithm, and an $L(n)$-time LUP-decomposition algorithm implies an $O(L(n))$-time matrix-multiplication algorithm.

28.2-3

Let $M(n)$ be the time to multiply two $n \times n$ matrices, and let $D(n)$ denote the time required to find the determinant of an $n \times n$ matrix. Show that multiplying matrices and computing the determinant have essentially the same difficulty: an $M(n)$-time matrix-multiplication algorithm implies an $O(M(n))$-time determinant algorithm, and a $D(n)$-time determinant algorithm implies an $O(D(n))$-time matrix-multiplication algorithm.

28.2-4

Let $M(n)$ be the time to multiply two $n \times n$ boolean matrices, and let $T(n)$ be the time to find the transitive closure of an $n \times n$ boolean matrix. (See Section 25.2.) Show that an $M(n)$-time boolean matrix-multiplication algorithm implies an $O(M(n) \lg n)$-time transitive-closure algorithm, and a $T(n)$-time transitive-closure algorithm implies an $O(T(n))$-time boolean matrix-multiplication algorithm.

28.2-5

Does the matrix-inversion algorithm based on Theorem 28.2 work when matrix elements are drawn from the field of integers modulo 2? Explain.

28.2-6 ★

Generalize the matrix-inversion algorithm of Theorem 28.2 to handle matrices of complex numbers, and prove that your generalization works correctly. (*Hint:* Instead of the transpose of A, use the ***conjugate transpose*** A^*, which you obtain from the transpose of A by replacing every entry with its complex conjugate. Instead of symmetric matrices, consider ***Hermitian*** matrices, which are matrices A such that $A = A^*$.)

28.3 Symmetric positive-definite matrices and least-squares approximation

Symmetric positive-definite matrices have many interesting and desirable properties. For example, they are nonsingular, and we can perform LU decomposition on them without having to worry about dividing by 0. In this section, we shall prove several other important properties of symmetric positive-definite matrices and show an interesting application to curve fitting by a least-squares approximation.

The first property we prove is perhaps the most basic.

Lemma 28.3
Any positive-definite matrix is nonsingular.

Proof Suppose that a matrix A is singular. Then by Corollary D.3, there exists a nonzero vector x such that $Ax = 0$. Hence, $x^{\mathrm{T}}Ax = 0$, and A cannot be positive-definite. ■

The proof that we can perform LU decomposition on a symmetric positive-definite matrix A without dividing by 0 is more involved. We begin by proving properties about certain submatrices of A. Define the kth ***leading submatrix*** of A to be the matrix A_k consisting of the intersection of the first k rows and first k columns of A.

Lemma 28.4
If A is a symmetric positive-definite matrix, then every leading submatrix of A is symmetric and positive-definite.

Proof That each leading submatrix A_k is symmetric is obvious. To prove that A_k is positive-definite, we assume that it is not and derive a contradiction. If A_k is not positive-definite, then there exists a k-vector $x_k \neq 0$ such that $x_k^{\mathrm{T}} A_k x_k \leq 0$. Let A be $n \times n$, and

$$A = \begin{pmatrix} A_k & B^{\mathrm{T}} \\ B & C \end{pmatrix} \tag{28.14}$$

for submatrices B (which is $(n-k) \times k$) and C (which is $(n-k) \times (n-k)$). Define the n-vector $x = (\, x_k^{\mathrm{T}} \quad 0 \,)^{\mathrm{T}}$, where $n - k$ 0s follow x_k. Then we have

$$\begin{aligned}
x^{\mathrm{T}}Ax &= (\, x_k^{\mathrm{T}} \quad 0 \,) \begin{pmatrix} A_k & B^{\mathrm{T}} \\ B & C \end{pmatrix} \begin{pmatrix} x_k \\ 0 \end{pmatrix} \\
&= (\, x_k^{\mathrm{T}} \quad 0 \,) \begin{pmatrix} A_k x_k \\ B x_k \end{pmatrix} \\
&= x_k^{\mathrm{T}} A_k x_k \\
&\leq 0 \,,
\end{aligned}$$

which contradicts A being positive-definite. ■

We now turn to some essential properties of the Schur complement. Let A be a symmetric positive-definite matrix, and let A_k be a leading $k \times k$ submatrix of A. Partition A once again according to equation (28.14). We generalize equation (28.9) to define the ***Schur complement*** S of A with respect to A_k as

$$S = C - BA_k^{-1}B^{\mathrm{T}} . \tag{28.15}$$

(By Lemma 28.4, A_k is symmetric and positive-definite; therefore, A_k^{-1} exists by Lemma 28.3, and S is well defined.) Note that our earlier definition (28.9) of the Schur complement is consistent with equation (28.15), by letting $k = 1$.

The next lemma shows that the Schur-complement matrices of symmetric positive-definite matrices are themselves symmetric and positive-definite. We used this result in Theorem 28.2, and we need its corollary to prove the correctness of LU decomposition for symmetric positive-definite matrices.

Lemma 28.5 (Schur complement lemma)
If A is a symmetric positive-definite matrix and A_k is a leading $k \times k$ submatrix of A, then the Schur complement S of A with respect to A_k is symmetric and positive-definite.

Proof Because A is symmetric, so is the submatrix C. By Exercise D.2-6, the product $BA_k^{-1}B^{\mathrm{T}}$ is symmetric, and by Exercise D.1-1, S is symmetric.

It remains to show that S is positive-definite. Consider the partition of A given in equation (28.14). For any nonzero vector x, we have $x^{\mathrm{T}}Ax > 0$ by the assumption that A is positive-definite. Let us break x into two subvectors y and z compatible with A_k and C, respectively. Because A_k^{-1} exists, we have

$$\begin{aligned}
x^{\mathrm{T}}Ax &= \begin{pmatrix} y^{\mathrm{T}} & z^{\mathrm{T}} \end{pmatrix} \begin{pmatrix} A_k & B^{\mathrm{T}} \\ B & C \end{pmatrix} \begin{pmatrix} y \\ z \end{pmatrix} \\
&= \begin{pmatrix} y^{\mathrm{T}} & z^{\mathrm{T}} \end{pmatrix} \begin{pmatrix} A_k y + B^{\mathrm{T}}z \\ By + Cz \end{pmatrix} \\
&= y^{\mathrm{T}}A_k y + y^{\mathrm{T}}B^{\mathrm{T}}z + z^{\mathrm{T}}By + z^{\mathrm{T}}Cz \\
&= (y + A_k^{-1}B^{\mathrm{T}}z)^{\mathrm{T}}A_k(y + A_k^{-1}B^{\mathrm{T}}z) + z^{\mathrm{T}}(C - BA_k^{-1}B^{\mathrm{T}})z , \qquad (28.16)
\end{aligned}$$

by matrix magic. (Verify by multiplying through.) This last equation amounts to "completing the square" of the quadratic form. (See Exercise 28.3-2.)

Since $x^{\mathrm{T}}Ax > 0$ holds for any nonzero x, let us pick any nonzero z and then choose $y = -A_k^{-1}B^{\mathrm{T}}z$, which causes the first term in equation (28.16) to vanish, leaving

$$z^{\mathrm{T}}(C - BA_k^{-1}B^{\mathrm{T}})z = z^{\mathrm{T}}Sz$$

as the value of the expression. For any $z \neq 0$, we therefore have $z^{\mathrm{T}}Sz = x^{\mathrm{T}}Ax > 0$, and thus S is positive-definite. ■

Corollary 28.6
LU decomposition of a symmetric positive-definite matrix never causes a division by 0.

Proof Let A be a symmetric positive-definite matrix. We shall prove something stronger than the statement of the corollary: every pivot is strictly positive. The first pivot is a_{11}. Let e_1 be the first unit vector, from which we obtain $a_{11} = e_1^{\mathrm{T}} A e_1 > 0$. Since the first step of LU decomposition produces the Schur complement of A with respect to $A_1 = (a_{11})$, Lemma 28.5 implies by induction that all pivots are positive. ■

Least-squares approximation

One important application of symmetric positive-definite matrices arises in fitting curves to given sets of data points. Suppose that we are given a set of m data points

$$(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m) ,$$

where we know that the y_i are subject to measurement errors. We would like to determine a function $F(x)$ such that the approximation errors

$$\eta_i = F(x_i) - y_i \tag{28.17}$$

are small for $i = 1, 2, \ldots, m$. The form of the function F depends on the problem at hand. Here, we assume that it has the form of a linearly weighted sum,

$$F(x) = \sum_{j=1}^{n} c_j f_j(x) ,$$

where the number of summands n and the specific ***basis functions*** f_j are chosen based on knowledge of the problem at hand. A common choice is $f_j(x) = x^{j-1}$, which means that

$$F(x) = c_1 + c_2 x + c_3 x^2 + \cdots + c_n x^{n-1}$$

is a polynomial of degree $n - 1$ in x. Thus, given m data points $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$, we wish to calculate n coefficients $c_1, c_2, \ldots, c_n$ that minimize the approximation errors $\eta_1, \eta_2, \ldots, \eta_m$.

By choosing $n = m$, we can calculate each y_i *exactly* in equation (28.17). Such a high-degree F "fits the noise" as well as the data, however, and generally gives poor results when used to predict y for previously unseen values of x. It is usually better to choose n significantly smaller than m and hope that by choosing the coefficients c_j well, we can obtain a function F that finds the significant patterns in the data points without paying undue attention to the noise. Some theoretical principles exist for choosing n, but they are beyond the scope of this text. In any case, once we choose a value of n that is less than m, we end up with an overdetermined set of equations whose solution we wish to approximate. We now show how to do so.

Let

$$A = \begin{pmatrix} f_1(x_1) & f_2(x_1) & \dots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \dots & f_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_m) & f_2(x_m) & \dots & f_n(x_m) \end{pmatrix}$$

denote the matrix of values of the basis functions at the given points; that is, $a_{ij} = f_j(x_i)$. Let $c = (c_k)$ denote the desired n-vector of coefficients. Then,

$$\begin{aligned} Ac &= \begin{pmatrix} f_1(x_1) & f_2(x_1) & \dots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \dots & f_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_m) & f_2(x_m) & \dots & f_n(x_m) \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix} \\ &= \begin{pmatrix} F(x_1) \\ F(x_2) \\ \vdots \\ F(x_m) \end{pmatrix} \end{aligned}$$

is the m-vector of "predicted values" for y. Thus,

$$\eta = Ac - y$$

is the m-vector of ***approximation errors***.

To minimize approximation errors, we choose to minimize the norm of the error vector η, which gives us a ***least-squares solution***, since

$$\|\eta\| = \left(\sum_{i=1}^{m} \eta_i^2 \right)^{1/2} .$$

Because

$$\|\eta\|^2 = \|Ac - y\|^2 = \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} c_j - y_i \right)^2 ,$$

we can minimize $\|\eta\|$ by differentiating $\|\eta\|^2$ with respect to each c_k and then setting the result to 0:

$$\frac{d\,\|\eta\|^2}{dc_k} = \sum_{i=1}^{m} 2\left(\sum_{j=1}^{n} a_{ij}c_j - y_i\right)a_{ik} = 0\,. \tag{28.18}$$

The n equations (28.18) for $k = 1, 2, \ldots, n$ are equivalent to the single matrix equation

$$(Ac - y)^{\mathrm{T}} A = 0$$

or, equivalently (using Exercise D.1-2), to

$$A^{\mathrm{T}}(Ac - y) = 0\,,$$

which implies

$$A^{\mathrm{T}}Ac = A^{\mathrm{T}}y\,. \tag{28.19}$$

In statistics, this is called the ***normal equation***. The matrix $A^{\mathrm{T}}A$ is symmetric by Exercise D.1-2, and if A has full column rank, then by Theorem D.6, $A^{\mathrm{T}}A$ is positive-definite as well. Hence, $(A^{\mathrm{T}}A)^{-1}$ exists, and the solution to equation (28.19) is

$$\begin{aligned} c &= \left((A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}}\right)y \\ &= A^{+}y\,, \end{aligned} \tag{28.20}$$

where the matrix $A^{+} = ((A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}})$ is the ***pseudoinverse*** of the matrix A. The pseudoinverse naturally generalizes the notion of a matrix inverse to the case in which A is not square. (Compare equation (28.20) as the approximate solution to $Ac = y$ with the solution $A^{-1}b$ as the exact solution to $Ax = b$.)

As an example of producing a least-squares fit, suppose that we have five data points

$$\begin{aligned} (x_1, y_1) &= (-1, 2)\,, \\ (x_2, y_2) &= (1, 1)\,, \\ (x_3, y_3) &= (2, 1)\,, \\ (x_4, y_4) &= (3, 0)\,, \\ (x_5, y_5) &= (5, 3)\,, \end{aligned}$$

shown as black dots in Figure 28.3. We wish to fit these points with a quadratic polynomial

$$F(x) = c_1 + c_2x + c_3x^2\,.$$

We start with the matrix of basis-function values

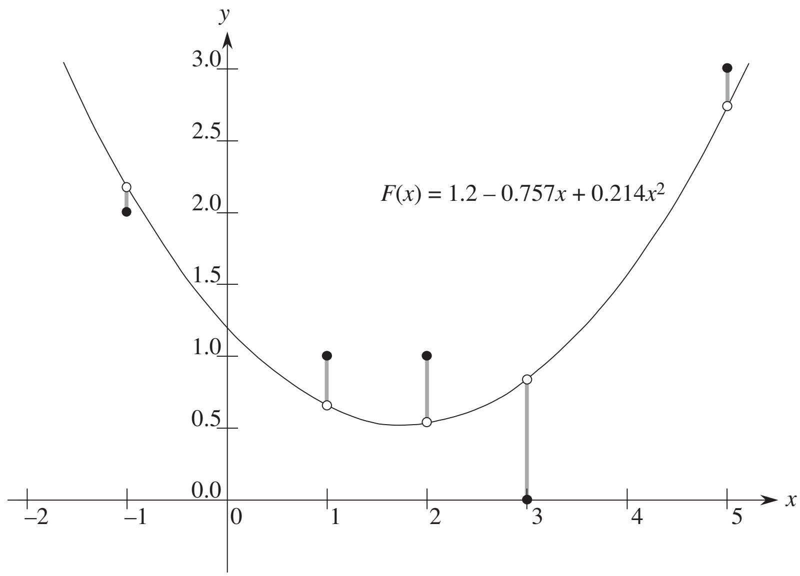

Figure 28.3 The least-squares fit of a quadratic polynomial to the set of five data points $\{(-1,2),(1,1),(2,1),(3,0),(5,3)\}$. The black dots are the data points, and the white dots are their estimated values predicted by the polynomial $F(x) = 1.2 - 0.757x + 0.214x^2$, the quadratic polynomial that minimizes the sum of the squared errors. Each shaded line shows the error for one data point.

$$A = \begin{pmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \\ 1 & x_4 & x_4^2 \\ 1 & x_5 & x_5^2 \end{pmatrix} = \begin{pmatrix} 1 & -1 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \\ 1 & 5 & 25 \end{pmatrix},$$

whose pseudoinverse is

$$A^+ = \begin{pmatrix} 0.500 & 0.300 & 0.200 & 0.100 & -0.100 \\ -0.388 & 0.093 & 0.190 & 0.193 & -0.088 \\ 0.060 & -0.036 & -0.048 & -0.036 & 0.060 \end{pmatrix}.$$

Multiplying y by A^+, we obtain the coefficient vector

$$c = \begin{pmatrix} 1.200 \\ -0.757 \\ 0.214 \end{pmatrix},$$

which corresponds to the quadratic polynomial

$$F(x) = 1.200 - 0.757x + 0.214x^2$$

as the closest-fitting quadratic to the given data, in a least-squares sense.

As a practical matter, we solve the normal equation (28.19) by multiplying y by A^{T} and then finding an LU decomposition of $A^{\mathrm{T}}A$. If A has full rank, the matrix $A^{\mathrm{T}}A$ is guaranteed to be nonsingular, because it is symmetric and positive-definite. (See Exercise D.1-2 and Theorem D.6.)

Exercises

28.3-1

Prove that every diagonal element of a symmetric positive-definite matrix is positive.

28.3-2

Let $A = \begin{pmatrix} a & b \\ b & c \end{pmatrix}$ be a 2×2 symmetric positive-definite matrix. Prove that its determinant $ac - b^2$ is positive by "completing the square" in a manner similar to that used in the proof of Lemma 28.5.

28.3-3

Prove that the maximum element in a symmetric positive-definite matrix lies on the diagonal.

28.3-4

Prove that the determinant of each leading submatrix of a symmetric positive-definite matrix is positive.

28.3-5

Let A_k denote the kth leading submatrix of a symmetric positive-definite matrix A. Prove that $\det(A_k)/\det(A_{k-1})$ is the kth pivot during LU decomposition, where, by convention, $\det(A_0) = 1$.

28.3-6

Find the function of the form

$$F(x) = c_1 + c_2 x \lg x + c_3 e^x$$

that is the best least-squares fit to the data points

$$(1, 1), (2, 1), (3, 3), (4, 8) .$$

28.3-7
Show that the pseudoinverse A^+ satisfies the following four equations:

$$\begin{aligned}
AA^+A &= A\,, \\
A^+AA^+ &= A^+\,, \\
(AA^+)^{\mathrm{T}} &= AA^+\,, \\
(A^+A)^{\mathrm{T}} &= A^+A\,.
\end{aligned}$$

Problems

28-1 Tridiagonal systems of linear equations
Consider the tridiagonal matrix

$$A = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{pmatrix}.$$

a. Find an LU decomposition of A.

b. Solve the equation $Ax = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \end{pmatrix}^{\mathrm{T}}$ by using forward and back substitution.

c. Find the inverse of A.

d. Show how, for any $n \times n$ symmetric positive-definite, tridiagonal matrix A and any n-vector b, to solve the equation $Ax = b$ in $O(n)$ time by performing an LU decomposition. Argue that any method based on forming A^{-1} is asymptotically more expensive in the worst case.

e. Show how, for any $n \times n$ nonsingular, tridiagonal matrix A and any n-vector b, to solve the equation $Ax = b$ in $O(n)$ time by performing an LUP decomposition.

28-2 Splines
A practical method for interpolating a set of points with a curve is to use ***cubic splines***. We are given a set $\{(x_i, y_i) : i = 0, 1, \ldots, n\}$ of $n + 1$ point-value pairs, where $x_0 < x_1 < \cdots < x_n$. We wish to fit a piecewise-cubic curve (spline) $f(x)$ to the points. That is, the curve $f(x)$ is made up of n cubic polynomials $f_i(x) = a_i + b_i x + c_i x^2 + d_i x^3$ for $i = 0, 1, \ldots, n - 1$, where if x falls in

the range $x_i \le x \le x_{i+1}$, then the value of the curve is given by $f(x) = f_i(x - x_i)$. The points x_i at which the cubic polynomials are "pasted" together are called ***knots***. For simplicity, we shall assume that $x_i = i$ for $i = 0, 1, \ldots, n$.

To ensure continuity of $f(x)$, we require that

$$\begin{aligned} f(x_i) &= f_i(0) &= y_i \;, \\ f(x_{i+1}) &= f_i(1) &= y_{i+1} \end{aligned}$$

for $i = 0, 1, \ldots, n-1$. To ensure that $f(x)$ is sufficiently smooth, we also insist that the first derivative be continuous at each knot:

$$f'(x_{i+1}) = f_i'(1) = f_{i+1}'(0)$$

for $i = 0, 1, \ldots, n-2$.

a. Suppose that for $i = 0, 1, \ldots, n$, we are given not only the point-value pairs $\{(x_i, y_i)\}$ but also the first derivatives $D_i = f'(x_i)$ at each knot. Express each coefficient a_i, b_i, c_i, and d_i in terms of the values y_i, y_{i+1}, D_i, and D_{i+1}. (Remember that $x_i = i$.) How quickly can we compute the $4n$ coefficients from the point-value pairs and first derivatives?

The question remains of how to choose the first derivatives of $f(x)$ at the knots. One method is to require the second derivatives to be continuous at the knots:

$$f''(x_{i+1}) = f_i''(1) = f_{i+1}''(0)$$

for $i = 0, 1, \ldots, n-2$. At the first and last knots, we assume that $f''(x_0) = f_0''(0) = 0$ and $f''(x_n) = f_{n-1}''(1) = 0$; these assumptions make $f(x)$ a ***natural*** cubic spline.

b. Use the continuity constraints on the second derivative to show that for $i = 1, 2, \ldots, n-1$,

$$D_{i-1} + 4D_i + D_{i+1} = 3(y_{i+1} - y_{i-1}) \;. \tag{28.21}$$

c. Show that

$$\begin{aligned} 2D_0 + D_1 &= 3(y_1 - y_0) \;, && (28.22) \\ D_{n-1} + 2D_n &= 3(y_n - y_{n-1}) \;. && (28.23) \end{aligned}$$

d. Rewrite equations (28.21)–(28.23) as a matrix equation involving the vector $D = \langle D_0, D_1, \ldots, D_n \rangle$ of unknowns. What attributes does the matrix in your equation have?

e. Argue that a natural cubic spline can interpolate a set of $n+1$ point-value pairs in $O(n)$ time (see Problem 28-1).

f. Show how to determine a natural cubic spline that interpolates a set of $n + 1$ points (x_i, y_i) satisfying $x_0 < x_1 < \cdots < x_n$, even when x_i is not necessarily equal to i. What matrix equation must your method solve, and how quickly does your algorithm run?

Chapter notes

Many excellent texts describe numerical and scientific computation in much greater detail than we have room for here. The following are especially readable: George and Liu [132], Golub and Van Loan [144], Press, Teukolsky, Vetterling, and Flannery [283, 284], and Strang [323, 324].

Golub and Van Loan [144] discuss numerical stability. They show why $\det(A)$ is not necessarily a good indicator of the stability of a matrix A, proposing instead to use $\|A\|_\infty \|A^{-1}\|_\infty$, where $\|A\|_\infty = \max_{1 \le i \le n} \sum_{j=1}^{n} |a_{ij}|$. They also address the question of how to compute this value without actually computing A^{-1}.

Gaussian elimination, upon which the LU and LUP decompositions are based, was the first systematic method for solving linear systems of equations. It was also one of the earliest numerical algorithms. Although it was known earlier, its discovery is commonly attributed to C. F. Gauss (1777–1855). In his famous paper [325], Strassen showed that an $n \times n$ matrix can be inverted in $O(n^{\lg 7})$ time. Winograd [358] originally proved that matrix multiplication is no harder than matrix inversion, and the converse is due to Aho, Hopcroft, and Ullman [5].

Another important matrix decomposition is the ***singular value decomposition***, or ***SVD***. The SVD factors an $m \times n$ matrix A into $A = Q_1 \Sigma Q_2^{\mathrm{T}}$, where Σ is an $m \times n$ matrix with nonzero values only on the diagonal, Q_1 is $m \times m$ with mutually orthonormal columns, and Q_2 is $n \times n$, also with mutually orthonormal columns. Two vectors are ***orthonormal*** if their inner product is 0 and each vector has a norm of 1. The books by Strang [323, 324] and Golub and Van Loan [144] contain good treatments of the SVD.

Strang [324] has an excellent presentation of symmetric positive-definite matrices and of linear algebra in general.

29 Linear Programming

Many problems take the form of maximizing or minimizing an objective, given limited resources and competing constraints. If we can specify the objective as a linear function of certain variables, and if we can specify the constraints on resources as equalities or inequalities on those variables, then we have a ***linear-programming problem***. Linear programs arise in a variety of practical applications. We begin by studying an application in electoral politics.

A political problem

Suppose that you are a politician trying to win an election. Your district has three different types of areas—urban, suburban, and rural. These areas have, respectively, 100,000, 200,000, and 50,000 registered voters. Although not all the registered voters actually go to the polls, you decide that to govern effectively, you would like at least half the registered voters in each of the three regions to vote for you. You are honorable and would never consider supporting policies in which you do not believe. You realize, however, that certain issues may be more effective in winning votes in certain places. Your primary issues are building more roads, gun control, farm subsidies, and a gasoline tax dedicated to improved public transit. According to your campaign staff's research, you can estimate how many votes you win or lose from each population segment by spending $1,000 on advertising on each issue. This information appears in the table of Figure 29.1. In this table, each entry indicates the number of thousands of either urban, suburban, or rural voters who would be won over by spending $1,000 on advertising in support of a particular issue. Negative entries denote votes that would be lost. Your task is to figure out the minimum amount of money that you need to spend in order to win 50,000 urban votes, 100,000 suburban votes, and 25,000 rural votes.

You could, by trial and error, devise a strategy that wins the required number of votes, but the strategy you come up with might not be the least expensive one. For example, you could devote $20,000 of advertising to building roads, $0 to gun control, $4,000 to farm subsidies, and $9,000 to a gasoline tax. In this case, you

policy	urban	suburban	rural
build roads	−2	5	3
gun control	8	2	−5
farm subsidies	0	0	10
gasoline tax	10	0	−2

Figure 29.1 The effects of policies on voters. Each entry describes the number of thousands of urban, suburban, or rural voters who could be won over by spending \$1,000 on advertising support of a policy on a particular issue. Negative entries denote votes that would be lost.

would win $20(-2)+0(8)+4(0)+9(10) = 50$ thousand urban votes, $20(5)+0(2)+4(0)+9(0) = 100$ thousand suburban votes, and $20(3)+0(-5)+4(10)+9(-2) = 82$ thousand rural votes. You would win the exact number of votes desired in the urban and suburban areas and more than enough votes in the rural area. (In fact, in the rural area, you would receive more votes than there are voters.) In order to garner these votes, you would have paid for $20 + 0 + 4 + 9 = 33$ thousand dollars of advertising.

Naturally, you may wonder whether this strategy is the best possible. That is, could you achieve your goals while spending less on advertising? Additional trial and error might help you to answer this question, but wouldn't you rather have a systematic method for answering such questions? In order to develop one, we shall formulate this question mathematically. We introduce 4 variables:

- x_1 is the number of thousands of dollars spent on advertising on building roads,
- x_2 is the number of thousands of dollars spent on advertising on gun control,
- x_3 is the number of thousands of dollars spent on advertising on farm subsidies, and
- x_4 is the number of thousands of dollars spent on advertising on a gasoline tax.

We can write the requirement that we win at least 50,000 urban votes as

$$-2x_1 + 8x_2 + 0x_3 + 10x_4 \geq 50 . \tag{29.1}$$

Similarly, we can write the requirements that we win at least 100,000 suburban votes and 25,000 rural votes as

$$5x_1 + 2x_2 + 0x_3 + 0x_4 \geq 100 \tag{29.2}$$

and

$$3x_1 - 5x_2 + 10x_3 - 2x_4 \geq 25 . \tag{29.3}$$

Any setting of the variables x_1, x_2, x_3, x_4 that satisfies inequalities (29.1)–(29.3) yields a strategy that wins a sufficient number of each type of vote. In order to keep costs as small as possible, you would like to minimize the amount spent on advertising. That is, you want to minimize the expression

$$x_1 + x_2 + x_3 + x_4 . \tag{29.4}$$

Although negative advertising often occurs in political campaigns, there is no such thing as negative-cost advertising. Consequently, we require that

$$x_1 \geq 0,\ x_2 \geq 0,\ x_3 \geq 0, \text{ and } x_4 \geq 0 . \tag{29.5}$$

Combining inequalities (29.1)–(29.3) and (29.5) with the objective of minimizing (29.4), we obtain what is known as a "linear program." We format this problem as

$$\begin{array}{lrcrcrcrcrl}
\text{minimize} & x_1 & + & x_2 & + & x_3 & + & x_4 & & & \quad (29.6) \\
\text{subject to} & & & & & & & & & & \\
 & -2x_1 & + & 8x_2 & + & 0x_3 & + & 10x_4 & \geq & 50 & \quad (29.7) \\
 & 5x_1 & + & 2x_2 & + & 0x_3 & + & 0x_4 & \geq & 100 & \quad (29.8) \\
 & 3x_1 & - & 5x_2 & + & 10x_3 & - & 2x_4 & \geq & 25 & \quad (29.9) \\
 & & & x_1, x_2, x_3, x_4 & & & & & \geq & 0 . & \quad (29.10)
\end{array}$$

The solution of this linear program yields your optimal strategy.

General linear programs

In the general linear-programming problem, we wish to optimize a linear function subject to a set of linear inequalities. Given a set of real numbers $a_1, a_2, \ldots, a_n$ and a set of variables $x_1, x_2, \ldots, x_n$, we define a ***linear function*** f on those variables by

$$f(x_1, x_2, \ldots, x_n) = a_1 x_1 + a_2 x_2 + \cdots + a_n x_n = \sum_{j=1}^{n} a_j x_j .$$

If b is a real number and f is a linear function, then the equation

$$f(x_1, x_2, \ldots, x_n) = b$$

is a ***linear equality*** and the inequalities

$$f(x_1, x_2, \ldots, x_n) \leq b$$

and

$$f(x_1, x_2, \ldots, x_n) \geq b$$

are ***linear inequalities***. We use the general term ***linear constraints*** to denote either linear equalities or linear inequalities. In linear programming, we do not allow strict inequalities. Formally, a ***linear-programming problem*** is the problem of either minimizing or maximizing a linear function subject to a finite set of linear constraints. If we are to minimize, then we call the linear program a ***minimization linear program***, and if we are to maximize, then we call the linear program a ***maximization linear program***.

The remainder of this chapter covers how to formulate and solve linear programs. Although several polynomial-time algorithms for linear programming have been developed, we will not study them in this chapter. Instead, we shall study the simplex algorithm, which is the oldest linear-programming algorithm. The simplex algorithm does not run in polynomial time in the worst case, but it is fairly efficient and widely used in practice.

An overview of linear programming

In order to describe properties of and algorithms for linear programs, we find it convenient to express them in canonical forms. We shall use two forms, ***standard*** and ***slack***, in this chapter. We will define them precisely in Section 29.1. Informally, a linear program in standard form is the maximization of a linear function subject to linear *inequalities*, whereas a linear program in slack form is the maximization of a linear function subject to linear *equalities*. We shall typically use standard form for expressing linear programs, but we find it more convenient to use slack form when we describe the details of the simplex algorithm. For now, we restrict our attention to maximizing a linear function on n variables subject to a set of m linear inequalities.

Let us first consider the following linear program with two variables:

$$\text{maximize} \quad x_1 + x_2 \tag{29.11}$$

subject to

$$4x_1 - x_2 \leq 8 \tag{29.12}$$

$$2x_1 + x_2 \leq 10 \tag{29.13}$$

$$5x_1 - 2x_2 \geq -2 \tag{29.14}$$

$$x_1, x_2 \geq 0 \; . \tag{29.15}$$

We call any setting of the variables x_1 and x_2 that satisfies all the constraints (29.12)–(29.15) a ***feasible solution*** to the linear program. If we graph the constraints in the (x_1, x_2)-Cartesian coordinate system, as in Figure 29.2(a), we see

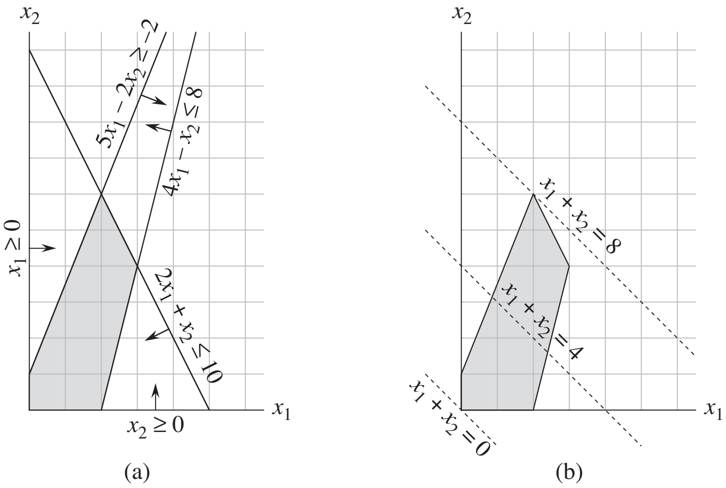

Figure 29.2 **(a)** The linear program given in (29.12)–(29.15). Each constraint is represented by a line and a direction. The intersection of the constraints, which is the feasible region, is shaded. **(b)** The dotted lines show, respectively, the points for which the objective value is 0, 4, and 8. The optimal solution to the linear program is $x_1 = 2$ and $x_2 = 6$ with objective value 8.

that the set of feasible solutions (shaded in the figure) forms a convex region[1] in the two-dimensional space. We call this convex region the ***feasible region*** and the function we wish to maximize the ***objective function***. Conceptually, we could evaluate the objective function $x_1 + x_2$ at each point in the feasible region; we call the value of the objective function at a particular point the ***objective value***. We could then identify a point that has the maximum objective value as an optimal solution. For this example (and for most linear programs), the feasible region contains an infinite number of points, and so we need to determine an efficient way to find a point that achieves the maximum objective value without explicitly evaluating the objective function at every point in the feasible region.

In two dimensions, we can optimize via a graphical procedure. The set of points for which $x_1+x_2 = z$, for any z, is a line with a slope of -1. If we plot $x_1+x_2 = 0$, we obtain the line with slope -1 through the origin, as in Figure 29.2(b). The intersection of this line and the feasible region is the set of feasible solutions that have an objective value of 0. In this case, that intersection of the line with the feasible region is the single point $(0, 0)$. More generally, for any z, the intersection

[1] An intuitive definition of a convex region is that it fulfills the requirement that for any two points in the region, all points on a line segment between them are also in the region.

of the line $x_1 + x_2 = z$ and the feasible region is the set of feasible solutions that have objective value z. Figure 29.2(b) shows the lines $x_1 + x_2 = 0$, $x_1 + x_2 = 4$, and $x_1 + x_2 = 8$. Because the feasible region in Figure 29.2 is bounded, there must be some maximum value z for which the intersection of the line $x_1 + x_2 = z$ and the feasible region is nonempty. Any point at which this occurs is an optimal solution to the linear program, which in this case is the point $x_1 = 2$ and $x_2 = 6$ with objective value 8.

It is no accident that an optimal solution to the linear program occurs at a vertex of the feasible region. The maximum value of z for which the line $x_1 + x_2 = z$ intersects the feasible region must be on the boundary of the feasible region, and thus the intersection of this line with the boundary of the feasible region is either a single vertex or a line segment. If the intersection is a single vertex, then there is just one optimal solution, and it is that vertex. If the intersection is a line segment, every point on that line segment must have the same objective value; in particular, both endpoints of the line segment are optimal solutions. Since each endpoint of a line segment is a vertex, there is an optimal solution at a vertex in this case as well.

Although we cannot easily graph linear programs with more than two variables, the same intuition holds. If we have three variables, then each constraint corresponds to a half-space in three-dimensional space. The intersection of these half-spaces forms the feasible region. The set of points for which the objective function obtains a given value z is now a plane (assuming no degenerate conditions). If all coefficients of the objective function are nonnegative, and if the origin is a feasible solution to the linear program, then as we move this plane away from the origin, in a direction normal to the objective function, we find points of increasing objective value. (If the origin is not feasible or if some coefficients in the objective function are negative, the intuitive picture becomes slightly more complicated.) As in two dimensions, because the feasible region is convex, the set of points that achieve the optimal objective value must include a vertex of the feasible region. Similarly, if we have n variables, each constraint defines a half-space in n-dimensional space. We call the feasible region formed by the intersection of these half-spaces a ***simplex***. The objective function is now a hyperplane and, because of convexity, an optimal solution still occurs at a vertex of the simplex.

The ***simplex algorithm*** takes as input a linear program and returns an optimal solution. It starts at some vertex of the simplex and performs a sequence of iterations. In each iteration, it moves along an edge of the simplex from a current vertex to a neighboring vertex whose objective value is no smaller than that of the current vertex (and usually is larger.) The simplex algorithm terminates when it reaches a local maximum, which is a vertex from which all neighboring vertices have a smaller objective value. Because the feasible region is convex and the objective function is linear, this local optimum is actually a global optimum. In Section 29.4, we shall use a concept called “duality” to show that the solution returned by the simplex algorithm is indeed optimal.

Although the geometric view gives a good intuitive view of the operations of the simplex algorithm, we shall not refer to it explicitly when developing the details of the simplex algorithm in Section 29.3. Instead, we take an algebraic view. We first write the given linear program in slack form, which is a set of linear equalities. These linear equalities express some of the variables, called “basic variables,” in terms of other variables, called “nonbasic variables.” We move from one vertex to another by making a basic variable become nonbasic and making a nonbasic variable become basic. We call this operation a “pivot” and, viewed algebraically, it is nothing more than rewriting the linear program in an equivalent slack form.

The two-variable example described above was particularly simple. We shall need to address several more details in this chapter. These issues include identifying linear programs that have no solutions, linear programs that have no finite optimal solution, and linear programs for which the origin is not a feasible solution.

Applications of linear programming

Linear programming has a large number of applications. Any textbook on operations research is filled with examples of linear programming, and linear programming has become a standard tool taught to students in most business schools. The election scenario is one typical example. Two more examples of linear programming are the following:

- An airline wishes to schedule its flight crews. The Federal Aviation Administration imposes many constraints, such as limiting the number of consecutive hours that each crew member can work and insisting that a particular crew work only on one model of aircraft during each month. The airline wants to schedule crews on all of its flights using as few crew members as possible.
- An oil company wants to decide where to drill for oil. Siting a drill at a particular location has an associated cost and, based on geological surveys, an expected payoff of some number of barrels of oil. The company has a limited budget for locating new drills and wants to maximize the amount of oil it expects to find, given this budget.

With linear programs, we also model and solve graph and combinatorial problems, such as those appearing in this textbook. We have already seen a special case of linear programming used to solve systems of difference constraints in Section 24.4. In Section 29.2, we shall study how to formulate several graph and network-flow problems as linear programs. In Section 35.4, we shall use linear programming as a tool to find an approximate solution to another graph problem.

Algorithms for linear programming

This chapter studies the simplex algorithm. This algorithm, when implemented carefully, often solves general linear programs quickly in practice. With some carefully contrived inputs, however, the simplex algorithm can require exponential time. The first polynomial-time algorithm for linear programming was the ***ellipsoid algorithm***, which runs slowly in practice. A second class of polynomial-time algorithms are known as ***interior-point methods***. In contrast to the simplex algorithm, which moves along the exterior of the feasible region and maintains a feasible solution that is a vertex of the simplex at each iteration, these algorithms move through the interior of the feasible region. The intermediate solutions, while feasible, are not necessarily vertices of the simplex, but the final solution is a vertex. For large inputs, interior-point algorithms can run as fast as, and sometimes faster than, the simplex algorithm. The chapter notes point you to more information about these algorithms.

If we add to a linear program the additional requirement that all variables take on integer values, we have an ***integer linear program***. Exercise 34.5-3 asks you to show that just finding a feasible solution to this problem is NP-hard; since no polynomial-time algorithms are known for any NP-hard problems, there is no known polynomial-time algorithm for integer linear programming. In contrast, we can solve a general linear-programming problem in polynomial time.

In this chapter, if we have a linear program with variables $x = (x_1, x_2, \ldots, x_n)$ and wish to refer to a particular setting of the variables, we shall use the notation $\bar{x} = (\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n)$.

29.1 Standard and slack forms

This section describes two formats, standard form and slack form, that are useful when we specify and work with linear programs. In standard form, all the constraints are inequalities, whereas in slack form, all constraints are equalities (except for those that require the variables to be nonnegative).

Standard form

In ***standard form***, we are given n real numbers $c_1, c_2, \ldots, c_n$; m real numbers $b_1, b_2, \ldots, b_m$; and mn real numbers a_{ij} for $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n$. We wish to find n real numbers $x_1, x_2, \ldots, x_n$ that

$$\text{maximize} \quad \sum_{j=1}^{n} c_j x_j \tag{29.16}$$

subject to

$$\sum_{j=1}^{n} a_{ij} x_j \le b_i \quad \text{for } i = 1, 2, \ldots, m \tag{29.17}$$

$$x_j \ge 0 \quad \text{for } j = 1, 2, \ldots, n . \tag{29.18}$$

Generalizing the terminology we introduced for the two-variable linear program, we call expression (29.16) the ***objective function*** and the $n + m$ inequalities in lines (29.17) and (29.18) the ***constraints***. The n constraints in line (29.18) are the ***nonnegativity constraints***. An arbitrary linear program need not have nonnegativity constraints, but standard form requires them. Sometimes we find it convenient to express a linear program in a more compact form. If we create an $m \times n$ matrix $A = (a_{ij})$, an m-vector $b = (b_i)$, an n-vector $c = (c_j)$, and an n-vector $x = (x_j)$, then we can rewrite the linear program defined in (29.16)–(29.18) as

$$\text{maximize} \quad c^{\mathrm{T}} x \tag{29.19}$$

subject to

$$Ax \le b \tag{29.20}$$

$$x \ge 0 . \tag{29.21}$$

In line (29.19), $c^{\mathrm{T}}x$ is the inner product of two vectors. In inequality (29.20), Ax is a matrix-vector product, and in inequality (29.21), $x \ge 0$ means that each entry of the vector x must be nonnegative. We see that we can specify a linear program in standard form by a tuple (A, b, c), and we shall adopt the convention that A, b, and c always have the dimensions given above.

We now introduce terminology to describe solutions to linear programs. We used some of this terminology in the earlier example of a two-variable linear program. We call a setting of the variables $\bar{x}$ that satisfies all the constraints a ***feasible solution***, whereas a setting of the variables $\bar{x}$ that fails to satisfy at least one constraint is an ***infeasible solution***. We say that a solution $\bar{x}$ has ***objective value*** $c^{\mathrm{T}}\bar{x}$. A feasible solution $\bar{x}$ whose objective value is maximum over all feasible solutions is an ***optimal solution***, and we call its objective value $c^{\mathrm{T}}\bar{x}$ the ***optimal objective value***. If a linear program has no feasible solutions, we say that the linear program is ***infeasible***; otherwise it is ***feasible***. If a linear program has some feasible solutions but does not have a finite optimal objective value, we say that the linear program is ***unbounded***. Exercise 29.1-9 asks you to show that a linear program can have a finite optimal objective value even if the feasible region is not bounded.

Converting linear programs into standard form

It is always possible to convert a linear program, given as minimizing or maximizing a linear function subject to linear constraints, into standard form. A linear program might not be in standard form for any of four possible reasons:

1. The objective function might be a minimization rather than a maximization.
2. There might be variables without nonnegativity constraints.
3. There might be ***equality constraints***, which have an equal sign rather than a less-than-or-equal-to sign.
4. There might be ***inequality constraints***, but instead of having a less-than-or-equal-to sign, they have a greater-than-or-equal-to sign.

When converting one linear program L into another linear program L', we would like the property that an optimal solution to L' yields an optimal solution to L. To capture this idea, we say that two maximization linear programs L and L' are ***equivalent*** if for each feasible solution $\bar{x}$ to L with objective value z, there is a corresponding feasible solution $\bar{x}'$ to L' with objective value z, and for each feasible solution $\bar{x}'$ to L' with objective value z, there is a corresponding feasible solution $\bar{x}$ to L with objective value z. (This definition does not imply a one-to-one correspondence between feasible solutions.) A minimization linear program L and a maximization linear program L' are equivalent if for each feasible solution $\bar{x}$ to L with objective value z, there is a corresponding feasible solution $\bar{x}'$ to L' with objective value $-z$, and for each feasible solution $\bar{x}'$ to L' with objective value z, there is a corresponding feasible solution $\bar{x}$ to L with objective value $-z$.

We now show how to remove, one by one, each of the possible problems in the list above. After removing each one, we shall argue that the new linear program is equivalent to the old one.

To convert a minimization linear program L into an equivalent maximization linear program L', we simply negate the coefficients in the objective function. Since L and L' have identical sets of feasible solutions and, for any feasible solution, the objective value in L is the negative of the objective value in L', these two linear programs are equivalent. For example, if we have the linear program

$$
\begin{array}{lrcrcl}
\text{minimize} & -2x_1 & + & 3x_2 & & \\
\text{subject to} & & & & & \\
& x_1 & + & x_2 & = & 7 \\
& x_1 & - & 2x_2 & \leq & 4 \\
& x_1 & & & \geq & 0 \; ,
\end{array}
$$

and we negate the coefficients of the objective function, we obtain

$$
\begin{array}{llcrcl}
\text{maximize} & 2x_1 & - & 3x_2 & & \\
\text{subject to} & & & & & \\
& x_1 & + & x_2 & = & 7 \\
& x_1 & - & 2x_2 & \le & 4 \\
& x_1 & & & \ge & 0 .
\end{array}
$$

Next, we show how to convert a linear program in which some of the variables do not have nonnegativity constraints into one in which each variable has a nonnegativity constraint. Suppose that some variable x_j does not have a nonnegativity constraint. Then, we replace each occurrence of x_j by $x'_j - x''_j$, and add the nonnegativity constraints $x'_j \ge 0$ and $x''_j \ge 0$. Thus, if the objective function has a term $c_j x_j$, we replace it by $c_j x'_j - c_j x''_j$, and if constraint i has a term $a_{ij} x_j$, we replace it by $a_{ij} x'_j - a_{ij} x''_j$. Any feasible solution $\hat{x}$ to the new linear program corresponds to a feasible solution $\bar{x}$ to the original linear program with $\bar{x}_j = \hat{x}'_j - \hat{x}''_j$ and with the same objective value. Also, any feasible solution $\bar{x}$ to the original linear program corresponds to a feasible solution $\hat{x}$ to the new linear program with $\hat{x}'_j = \bar{x}_j$ and $\hat{x}''_j = 0$ if $\bar{x}_j \ge 0$, or with $\hat{x}''_j = \bar{x}_j$ and $\hat{x}'_j = 0$ if $\bar{x}_j < 0$. The two linear programs have the same objective value regardless of the sign of $\bar{x}_j$. Thus, the two linear programs are equivalent. We apply this conversion scheme to each variable that does not have a nonnegativity constraint to yield an equivalent linear program in which all variables have nonnegativity constraints.

Continuing the example, we want to ensure that each variable has a corresponding nonnegativity constraint. Variable x_1 has such a constraint, but variable x_2 does not. Therefore, we replace x_2 by two variables x'_2 and x''_2, and we modify the linear program to obtain

$$
\begin{array}{llcrcrcl}
\text{maximize} & 2x_1 & - & 3x'_2 & + & 3x''_2 & & \\
\text{subject to} & & & & & & & \\
& x_1 & + & x'_2 & - & x''_2 & = & 7 \\
& x_1 & - & 2x'_2 & + & 2x''_2 & \le & 4 \\
& \multicolumn{5}{l}{\quad x_1, x'_2, x''_2} & \ge & 0 .
\end{array}
\tag{29.22}
$$

Next, we convert equality constraints into inequality constraints. Suppose that a linear program has an equality constraint $f(x_1, x_2, \ldots, x_n) = b$. Since $x = y$ if and only if both $x \ge y$ and $x \le y$, we can replace this equality constraint by the pair of inequality constraints $f(x_1, x_2, \ldots, x_n) \le b$ and $f(x_1, x_2, \ldots, x_n) \ge b$. Repeating this conversion for each equality constraint yields a linear program in which all constraints are inequalities.

Finally, we can convert the greater-than-or-equal-to constraints to less-than-or-equal-to constraints by multiplying these constraints through by -1. That is, any inequality of the form

$$\sum_{j=1}^{n} a_{ij} x_j \geq b_i$$

is equivalent to

$$\sum_{j=1}^{n} -a_{ij} x_j \leq -b_i \ .$$

Thus, by replacing each coefficient a_{ij} by $-a_{ij}$ and each value b_i by $-b_i$, we obtain an equivalent less-than-or-equal-to constraint.

Finishing our example, we replace the equality in constraint (29.22) by two inequalities, obtaining

$$\begin{array}{lrcrcrcr}
\text{maximize} & 2x_1 & - & 3x_2' & + & 3x_2'' & & \\
\text{subject to} & & & & & & & \\
& x_1 & + & x_2' & - & x_2'' & \leq & 7 \\
& x_1 & + & x_2' & - & x_2'' & \geq & 7 \\
& x_1 & - & 2x_2' & + & 2x_2'' & \leq & 4 \\
& & & x_1, x_2', x_2'' & & & \geq & 0 \ .
\end{array} \tag{29.23}$$

Finally, we negate constraint (29.23). For consistency in variable names, we rename x_2' to x_2 and x_2'' to x_3, obtaining the standard form

$$\text{maximize} \quad 2x_1 - 3x_2 + 3x_3 \tag{29.24}$$

subject to

$$x_1 + x_2 - x_3 \leq 7 \tag{29.25}$$

$$-x_1 - x_2 + x_3 \leq -7 \tag{29.26}$$

$$x_1 - 2x_2 + 2x_3 \leq 4 \tag{29.27}$$

$$x_1, x_2, x_3 \geq 0 \ . \tag{29.28}$$

Converting linear programs into slack form

To efficiently solve a linear program with the simplex algorithm, we prefer to express it in a form in which some of the constraints are equality constraints. More precisely, we shall convert it into a form in which the nonnegativity constraints are the only inequality constraints, and the remaining constraints are equalities. Let

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i \tag{29.29}$$

be an inequality constraint. We introduce a new variable s and rewrite inequality (29.29) as the two constraints

$$s = b_i - \sum_{j=1}^{n} a_{ij} x_j \ , \tag{29.30}$$

$$s \geq 0 \ . \tag{29.31}$$

We call s a ***slack variable*** because it measures the ***slack***, or difference, between the left-hand and right-hand sides of equation (29.29). (We shall soon see why we find it convenient to write the constraint with only the slack variable on the left-hand side.) Because inequality (29.29) is true if and only if both equation (29.30) and inequality (29.31) are true, we can convert each inequality constraint of a linear program in this way to obtain an equivalent linear program in which the only inequality constraints are the nonnegativity constraints. When converting from standard to slack form, we shall use x_{n+i} (instead of s) to denote the slack variable associated with the ith inequality. The ith constraint is therefore

$$x_{n+i} = b_i - \sum_{j=1}^{n} a_{ij} x_j \ , \tag{29.32}$$

along with the nonnegativity constraint $x_{n+i} \geq 0$.

By converting each constraint of a linear program in standard form, we obtain a linear program in a different form. For example, for the linear program described in (29.24)–(29.28), we introduce slack variables x_4, x_5, and x_6, obtaining

$$\text{maximize} \quad 2x_1 - 3x_2 + 3x_3 \tag{29.33}$$

subject to

$$x_4 = 7 - x_1 - x_2 + x_3 \tag{29.34}$$

$$x_5 = -7 + x_1 + x_2 - x_3 \tag{29.35}$$

$$x_6 = 4 - x_1 + 2x_2 - 2x_3 \tag{29.36}$$

$$x_1, x_2, x_3, x_4, x_5, x_6 \geq 0 \ . \tag{29.37}$$

In this linear program, all the constraints except for the nonnegativity constraints are equalities, and each variable is subject to a nonnegativity constraint. We write each equality constraint with one of the variables on the left-hand side of the equality and all others on the right-hand side. Furthermore, each equation has the same set of variables on the right-hand side, and these variables are also the only ones that appear in the objective function. We call the variables on the left-hand side of the equalities ***basic variables*** and those on the right-hand side ***nonbasic variables***.

For linear programs that satisfy these conditions, we shall sometimes omit the words "maximize" and "subject to," as well as the explicit nonnegativity constraints. We shall also use the variable z to denote the value of the objective function. We call the resulting format ***slack form***. If we write the linear program given in (29.33)–(29.37) in slack form, we obtain

$$
\begin{aligned}
z &= & & 2x_1 - 3x_2 + 3x_3 && (29.38)\\
x_4 &= 7 & - & x_1 - x_2 + x_3 && (29.39)\\
x_5 &= -7 & + & x_1 + x_2 - x_3 && (29.40)\\
x_6 &= 4 & - & x_1 + 2x_2 - 2x_3 \; . && (29.41)
\end{aligned}
$$

As with standard form, we find it convenient to have a more concise notation for describing a slack form. As we shall see in Section 29.3, the sets of basic and nonbasic variables will change as the simplex algorithm runs. We use N to denote the set of indices of the nonbasic variables and B to denote the set of indices of the basic variables. We always have that $|N| = n$, $|B| = m$, and $N \cup B = \{1, 2, \ldots, n+m\}$. The equations are indexed by the entries of B, and the variables on the right-hand sides are indexed by the entries of N. As in standard form, we use b_i, c_j, and a_{ij} to denote constant terms and coefficients. We also use ν to denote an optional constant term in the objective function. (We shall see a little later that including the constant term in the objective function makes it easy to determine the value of the objective function.) Thus we can concisely define a slack form by a tuple (N, B, A, b, c, ν), denoting the slack form

$$z = \nu + \sum_{j \in N} c_j x_j \qquad (29.42)$$

$$x_i = b_i - \sum_{j \in N} a_{ij} x_j \quad \text{for } i \in B \; , \qquad (29.43)$$

in which all variables x are constrained to be nonnegative. Because we subtract the sum $\sum_{j \in N} a_{ij} x_j$ in (29.43), the values a_{ij} are actually the negatives of the coefficients as they "appear" in the slack form.

For example, in the slack form

$$
\begin{aligned}
z &= 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3}\\
x_1 &= 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3}\\
x_2 &= 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3}\\
x_4 &= 18 - \frac{x_3}{2} + \frac{x_5}{2} \; ,
\end{aligned}
$$

we have $B = \{1, 2, 4\}$, $N = \{3, 5, 6\}$,

$$A = \begin{pmatrix} a_{13} & a_{15} & a_{16} \\ a_{23} & a_{25} & a_{26} \\ a_{43} & a_{45} & a_{46} \end{pmatrix} = \begin{pmatrix} -1/6 & -1/6 & 1/3 \\ 8/3 & 2/3 & -1/3 \\ 1/2 & -1/2 & 0 \end{pmatrix},$$

$$b = \begin{pmatrix} b_1 \\ b_2 \\ b_4 \end{pmatrix} = \begin{pmatrix} 8 \\ 4 \\ 18 \end{pmatrix},$$

$c = \begin{pmatrix} c_3 & c_5 & c_6 \end{pmatrix}^{\mathrm{T}} = \begin{pmatrix} -1/6 & -1/6 & -2/3 \end{pmatrix}^{\mathrm{T}}$, and $v = 28$. Note that the indices into A, b, and c are not necessarily sets of contiguous integers; they depend on the index sets B and N. As an example of the entries of A being the negatives of the coefficients as they appear in the slack form, observe that the equation for x_1 includes the term $x_3/6$, yet the coefficient a_{13} is actually $-1/6$ rather than $+1/6$.

Exercises

29.1-1

If we express the linear program in (29.24)–(29.28) in the compact notation of (29.19)–(29.21), what are n, m, A, b, and c?

29.1-2

Give three feasible solutions to the linear program in (29.24)–(29.28). What is the objective value of each one?

29.1-3

For the slack form in (29.38)–(29.41), what are N, B, A, b, c, and v?

29.1-4

Convert the following linear program into standard form:

$$\begin{array}{lrcrcrcr}
\text{minimize} & 2x_1 & + & 7x_2 & + & x_3 & & \\
\text{subject to} & & & & & & & \\
& x_1 & & & - & x_3 & = & 7 \\
& 3x_1 & + & x_2 & & & \geq & 24 \\
& & & x_2 & & & \geq & 0 \\
& & & & & x_3 & \leq & 0 \ .
\end{array}$$

29.1-5

Convert the following linear program into slack form:

$$
\begin{array}{lrcrcrcl}
\text{maximize} & 2x_1 & & & - & 6x_3 & & \\
\text{subject to} & & & & & & & \\
& x_1 & + & x_2 & - & x_3 & \leq & 7 \\
& 3x_1 & - & x_2 & & & \geq & 8 \\
& -x_1 & + & 2x_2 & + & 2x_3 & \geq & 0 \\
& & x_1, x_2, x_3 & & & & \geq & 0 .
\end{array}
$$

What are the basic and nonbasic variables?

29.1-6

Show that the following linear program is infeasible:

$$
\begin{array}{lrcrcr}
\text{maximize} & 3x_1 & - & 2x_2 & & \\
\text{subject to} & & & & & \\
& x_1 & + & x_2 & \leq & 2 \\
& -2x_1 & - & 2x_2 & \leq & -10 \\
& & x_1, x_2 & & \geq & 0 .
\end{array}
$$

29.1-7

Show that the following linear program is unbounded:

$$
\begin{array}{lrcrcr}
\text{maximize} & x_1 & - & x_2 & & \\
\text{subject to} & & & & & \\
& -2x_1 & + & x_2 & \leq & -1 \\
& -x_1 & - & 2x_2 & \leq & -2 \\
& & x_1, x_2 & & \geq & 0 .
\end{array}
$$

29.1-8

Suppose that we have a general linear program with n variables and m constraints, and suppose that we convert it into standard form. Give an upper bound on the number of variables and constraints in the resulting linear program.

29.1-9

Give an example of a linear program for which the feasible region is not bounded, but the optimal objective value is finite.

29.2 Formulating problems as linear programs

Although we shall focus on the simplex algorithm in this chapter, it is also important to be able to recognize when we can formulate a problem as a linear program. Once we cast a problem as a polynomial-sized linear program, we can solve it in polynomial time by the ellipsoid algorithm or interior-point methods. Several linear-programming software packages can solve problems efficiently, so that once the problem is in the form of a linear program, such a package can solve it.

We shall look at several concrete examples of linear-programming problems. We start with two problems that we have already studied: the single-source shortest-paths problem (see Chapter 24) and the maximum-flow problem (see Chapter 26). We then describe the minimum-cost-flow problem. Although the minimum-cost-flow problem has a polynomial-time algorithm that is not based on linear programming, we won't describe the algorithm. Finally, we describe the multicommodity-flow problem, for which the only known polynomial-time algorithm is based on linear programming.

When we solved graph problems in Part VI, we used attribute notation, such as $v.d$ and $(u,v).f$. Linear programs typically use subscripted variables rather than objects with attached attributes, however. Therefore, when we express variables in linear programs, we shall indicate vertices and edges through subscripts. For example, we denote the shortest-path weight for vertex v not by $v.d$ but by d_v. Similarly, we denote the flow from vertex u to vertex v not by $(u,v).f$ but by f_{uv}. For quantities that are given as inputs to problems, such as edge weights or capacities, we shall continue to use notations such as $w(u,v)$ and $c(u.v)$.

Shortest paths

We can formulate the single-source shortest-paths problem as a linear program. In this section, we shall focus on how to formulate the single-pair shortest-path problem, leaving the extension to the more general single-source shortest-paths problem as Exercise 29.2-3.

In the single-pair shortest-path problem, we are given a weighted, directed graph $G = (V, E)$, with weight function $w : E \to \mathbb{R}$ mapping edges to real-valued weights, a source vertex s, and destination vertex t. We wish to compute the value d_t, which is the weight of a shortest path from s to t. To express this problem as a linear program, we need to determine a set of variables and constraints that define when we have a shortest path from s to t. Fortunately, the Bellman-Ford algorithm does exactly this. When the Bellman-Ford algorithm terminates, it has computed, for each vertex v, a value d_v (using subscript notation here rather than attribute notation) such that for each edge $(u,v) \in E$, we have $d_v \leq d_u + w(u,v)$.

The source vertex initially receives a value $d_s = 0$, which never changes. Thus we obtain the following linear program to compute the shortest-path weight from s to t:

$$\text{maximize} \quad d_t \tag{29.44}$$

subject to

$$d_v \leq d_u + w(u,v) \quad \text{for each edge } (u,v) \in E , \tag{29.45}$$

$$d_s = 0 . \tag{29.46}$$

You might be surprised that this linear program maximizes an objective function when it is supposed to compute shortest paths. We do not want to minimize the objective function, since then setting $\bar{d}_v = 0$ for all $v \in V$ would yield an optimal solution to the linear program without solving the shortest-paths problem. We maximize because an optimal solution to the shortest-paths problem sets each $\bar{d}_v$ to $\min_{u:(u,v)\in E} \left\{\bar{d}_u + w(u,v)\right\}$, so that $\bar{d}_v$ is the largest value that is less than or equal to all of the values in the set $\left\{\bar{d}_u + w(u,v)\right\}$. We want to maximize d_v for all vertices v on a shortest path from s to t subject to these constraints on all vertices v, and maximizing d_t achieves this goal.

This linear program has $|V|$ variables d_v, one for each vertex $v \in V$. It also has $|E| + 1$ constraints: one for each edge, plus the additional constraint that the source vertex's shortest-path weight always has the value 0.

Maximum flow

Next, we express the maximum-flow problem as a linear program. Recall that we are given a directed graph $G = (V, E)$ in which each edge $(u, v) \in E$ has a nonnegative capacity $c(u, v) \geq 0$, and two distinguished vertices: a source s and a sink t. As defined in Section 26.1, a flow is a nonnegative real-valued function $f : V \times V \to \mathbb{R}$ that satisfies the capacity constraint and flow conservation. A maximum flow is a flow that satisfies these constraints and maximizes the flow value, which is the total flow coming out of the source minus the total flow into the source. A flow, therefore, satisfies linear constraints, and the value of a flow is a linear function. Recalling also that we assume that $c(u, v) = 0$ if $(u, v) \notin E$ and that there are no antiparallel edges, we can express the maximum-flow problem as a linear program:

$$\text{maximize} \quad \sum_{v\in V} f_{sv} - \sum_{v\in V} f_{vs} \tag{29.47}$$

subject to

$$f_{uv} \leq c(u,v) \quad \text{for each } u, v \in V , \tag{29.48}$$

$$\sum_{v\in V} f_{vu} = \sum_{v\in V} f_{uv} \quad \text{for each } u \in V - \{s,t\} , \tag{29.49}$$

$$f_{uv} \geq 0 \quad \text{for each } u, v \in V . \tag{29.50}$$

This linear program has $|V|^2$ variables, corresponding to the flow between each pair of vertices, and it has $2|V|^2 + |V| - 2$ constraints.

It is usually more efficient to solve a smaller-sized linear program. The linear program in (29.47)–(29.50) has, for ease of notation, a flow and capacity of 0 for each pair of vertices u, v with $(u, v) \notin E$. It would be more efficient to rewrite the linear program so that it has $O(V + E)$ constraints. Exercise 29.2-5 asks you to do so.

Minimum-cost flow

In this section, we have used linear programming to solve problems for which we already knew efficient algorithms. In fact, an efficient algorithm designed specifically for a problem, such as Dijkstra's algorithm for the single-source shortest-paths problem, or the push-relabel method for maximum flow, will often be more efficient than linear programming, both in theory and in practice.

The real power of linear programming comes from the ability to solve new problems. Recall the problem faced by the politician in the beginning of this chapter. The problem of obtaining a sufficient number of votes, while not spending too much money, is not solved by any of the algorithms that we have studied in this book, yet we can solve it by linear programming. Books abound with such real-world problems that linear programming can solve. Linear programming is also particularly useful for solving variants of problems for which we may not already know of an efficient algorithm.

Consider, for example, the following generalization of the maximum-flow problem. Suppose that, in addition to a capacity $c(u, v)$ for each edge (u, v), we are given a real-valued cost $a(u, v)$. As in the maximum-flow problem, we assume that $c(u, v) = 0$ if $(u, v) \notin E$, and that there are no antiparallel edges. If we send f_{uv} units of flow over edge (u, v), we incur a cost of $a(u, v) f_{uv}$. We are also given a flow demand d. We wish to send d units of flow from s to t while minimizing the total cost $\sum_{(u,v)\in E} a(u, v) f_{uv}$ incurred by the flow. This problem is known as the ***minimum-cost-flow problem***.

Figure 29.3(a) shows an example of the minimum-cost-flow problem. We wish to send 4 units of flow from s to t while incurring the minimum total cost. Any particular legal flow, that is, a function f satisfying constraints (29.48)–(29.49), incurs a total cost of $\sum_{(u,v)\in E} a(u, v) f_{uv}$. We wish to find the particular 4-unit flow that minimizes this cost. Figure 29.3(b) shows an optimal solution, with total cost $\sum_{(u,v)\in E} a(u, v) f_{uv} = (2 \cdot 2) + (5 \cdot 2) + (3 \cdot 1) + (7 \cdot 1) + (1 \cdot 3) = 27$.

There are polynomial-time algorithms specifically designed for the minimum-cost-flow problem, but they are beyond the scope of this book. We can, however, express the minimum-cost-flow problem as a linear program. The linear program looks similar to the one for the maximum-flow problem with the additional con-

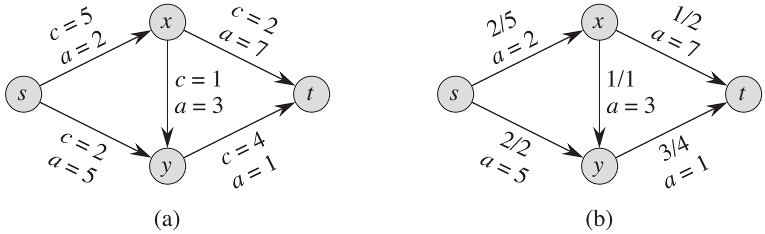

Figure 29.3 **(a)** An example of a minimum-cost-flow problem. We denote the capacities by c and the costs by a. Vertex s is the source and vertex t is the sink, and we wish to send 4 units of flow from s to t. **(b)** A solution to the minimum-cost flow problem in which 4 units of flow are sent from s to t. For each edge, the flow and capacity are written as flow/capacity.

straint that the value of the flow be exactly d units, and with the new objective function of minimizing the cost:

$$\text{minimize} \quad \sum_{(u,v)\in E} a(u,v) f_{uv} \tag{29.51}$$

subject to

$$
\begin{aligned}
f_{uv} &\le c(u,v) && \text{for each } u, v \in V \; , \\
\sum_{v\in V} f_{vu} - \sum_{v\in V} f_{uv} &= 0 && \text{for each } u \in V - \{s,t\} \; , \\
\sum_{v\in V} f_{sv} - \sum_{v\in V} f_{vs} &= d \; , \\
f_{uv} &\ge 0 && \text{for each } u, v \in V \; .
\end{aligned} \tag{29.52}
$$

Multicommodity flow

As a final example, we consider another flow problem. Suppose that the Lucky Puck company from Section 26.1 decides to diversify its product line and ship not only hockey pucks, but also hockey sticks and hockey helmets. Each piece of equipment is manufactured in its own factory, has its own warehouse, and must be shipped, each day, from factory to warehouse. The sticks are manufactured in Vancouver and must be shipped to Saskatoon, and the helmets are manufactured in Edmonton and must be shipped to Regina. The capacity of the shipping network does not change, however, and the different items, or ***commodities***, must share the same network.

This example is an instance of a ***multicommodity-flow problem***. In this problem, we are again given a directed graph $G = (V, E)$ in which each edge $(u,v) \in E$ has a nonnegative capacity $c(u,v) \ge 0$. As in the maximum-flow problem, we implicitly assume that $c(u,v) = 0$ for $(u,v) \notin E$, and that the graph has no antiparallel edges. In addition, we are given k different commodities, $K_1, K_2, \ldots, K_k$, where we specify commodity i by the triple $K_i = (s_i, t_i, d_i)$. Here, vertex s_i is the source of commodity i, vertex t_i is the sink of commodity i, and d_i is the demand for commodity i, which is the desired flow value for the commodity from s_i to t_i. We define a flow for commodity i, denoted by f_i, (so that f_{iuv} is the flow of commodity i from vertex u to vertex v) to be a real-valued function that satisfies the flow-conservation and capacity constraints. We now define f_{uv}, the ***aggregate flow***, to be the sum of the various commodity flows, so that $f_{uv} = \sum_{i=1}^{k} f_{iuv}$. The aggregate flow on edge (u, v) must be no more than the capacity of edge (u, v). We are not trying to minimize any objective function in this problem; we need only determine whether such a flow exists. Thus, we write a linear program with a "null" objective function:

$$
\begin{array}{lrcll}
\text{minimize} & & & 0 & \\
\text{subject to} & & & & \\
& \displaystyle\sum_{i=1}^{k} f_{iuv} & \leq & c(u,v) & \text{for each } u, v \in V \, , \\
& \displaystyle\sum_{v \in V} f_{iuv} - \sum_{v \in V} f_{ivu} & = & 0 & \text{for each } i = 1, 2, \ldots, k \text{ and} \\
& & & & \text{for each } u \in V - \{s_i, t_i\} \, , \\
& \displaystyle\sum_{v \in V} f_{i,s_i,v} - \sum_{v \in V} f_{i,v,s_i} & = & d_i & \text{for each } i = 1, 2, \ldots, k \, , \\
& f_{iuv} & \geq & 0 & \text{for each } u, v \in V \text{ and} \\
& & & & \text{for each } i = 1, 2, \ldots, k \, .
\end{array}
$$

The only known polynomial-time algorithm for this problem expresses it as a linear program and then solves it with a polynomial-time linear-programming algorithm.

Exercises

29.2-1
Put the single-pair shortest-path linear program from (29.44)–(29.46) into standard form.

29.2-2
Write out explicitly the linear program corresponding to finding the shortest path from node s to node y in Figure 24.2(a).

29.2-3
In the single-source shortest-paths problem, we want to find the shortest-path weights from a source vertex s to all vertices $v \in V$. Given a graph G, write a

linear program for which the solution has the property that d_v is the shortest-path weight from s to v for each vertex $v \in V$.

29.2-4
Write out explicitly the linear program corresponding to finding the maximum flow in Figure 26.1(a).

29.2-5
Rewrite the linear program for maximum flow (29.47)–(29.50) so that it uses only $O(V + E)$ constraints.

29.2-6
Write a linear program that, given a bipartite graph $G = (V, E)$, solves the maximum-bipartite-matching problem.

29.2-7
In the ***minimum-cost multicommodity-flow problem***, we are given directed graph $G = (V, E)$ in which each edge $(u, v) \in E$ has a nonnegative capacity $c(u, v) \geq 0$ and a cost $a(u, v)$. As in the multicommodity-flow problem, we are given k different commodities, $K_1, K_2, \ldots, K_k$, where we specify commodity i by the triple $K_i = (s_i, t_i, d_i)$. We define the flow f_i for commodity i and the aggregate flow f_{uv} on edge (u, v) as in the multicommodity-flow problem. A feasible flow is one in which the aggregate flow on each edge (u, v) is no more than the capacity of edge (u, v). The cost of a flow is $\sum_{u,v \in V} a(u, v) f_{uv}$, and the goal is to find the feasible flow of minimum cost. Express this problem as a linear program.

29.3 The simplex algorithm

The simplex algorithm is the classical method for solving linear programs. In contrast to most of the other algorithms in this book, its running time is not polynomial in the worst case. It does yield insight into linear programs, however, and is often remarkably fast in practice.

In addition to having a geometric interpretation, described earlier in this chapter, the simplex algorithm bears some similarity to Gaussian elimination, discussed in Section 28.1. Gaussian elimination begins with a system of linear equalities whose solution is unknown. In each iteration, we rewrite this system in an equivalent form that has some additional structure. After some number of iterations, we have rewritten the system so that the solution is simple to obtain. The simplex algorithm proceeds in a similar manner, and we can view it as Gaussian elimination for inequalities.

We now describe the main idea behind an iteration of the simplex algorithm. Associated with each iteration will be a "basic solution" that we can easily obtain from the slack form of the linear program: set each nonbasic variable to 0 and compute the values of the basic variables from the equality constraints. An iteration converts one slack form into an equivalent slack form. The objective value of the associated basic feasible solution will be no less than that at the previous iteration, and usually greater. To achieve this increase in the objective value, we choose a nonbasic variable such that if we were to increase that variable's value from 0, then the objective value would increase, too. The amount by which we can increase the variable is limited by the other constraints. In particular, we raise it until some basic variable becomes 0. We then rewrite the slack form, exchanging the roles of that basic variable and the chosen nonbasic variable. Although we have used a particular setting of the variables to guide the algorithm, and we shall use it in our proofs, the algorithm does not explicitly maintain this solution. It simply rewrites the linear program until an optimal solution becomes "obvious."

An example of the simplex algorithm

We begin with an extended example. Consider the following linear program in standard form:

$$\text{maximize} \quad 3x_1 + x_2 + 2x_3 \tag{29.53}$$

subject to

$$x_1 + x_2 + 3x_3 \le 30 \tag{29.54}$$

$$2x_1 + 2x_2 + 5x_3 \le 24 \tag{29.55}$$

$$4x_1 + x_2 + 2x_3 \le 36 \tag{29.56}$$

$$x_1, x_2, x_3 \ge 0 \; . \tag{29.57}$$

In order to use the simplex algorithm, we must convert the linear program into slack form; we saw how to do so in Section 29.1. In addition to being an algebraic manipulation, slack is a useful algorithmic concept. Recalling from Section 29.1 that each variable has a corresponding nonnegativity constraint, we say that an equality constraint is ***tight*** for a particular setting of its nonbasic variables if they cause the constraint's basic variable to become 0. Similarly, a setting of the nonbasic variables that would make a basic variable become negative ***violates*** that constraint. Thus, the slack variables explicitly maintain how far each constraint is from being tight, and so they help to determine how much we can increase values of nonbasic variables without violating any constraints.

Associating the slack variables x_4, x_5, and x_6 with inequalities (29.54)–(29.56), respectively, and putting the linear program into slack form, we obtain

$$\begin{aligned} z &= & & 3x_1 &+& x_2 &+& 2x_3 \\ x_4 &= 30 &-& x_1 &-& x_2 &-& 3x_3 \\ x_5 &= 24 &-& 2x_1 &-& 2x_2 &-& 5x_3 \\ x_6 &= 36 &-& 4x_1 &-& x_2 &-& 2x_3 \ . \end{aligned} \tag{29.58–29.61}$$

The system of constraints (29.59)–(29.61) has 3 equations and 6 variables. Any setting of the variables x_1, x_2, and x_3 defines values for x_4, x_5, and x_6; therefore, we have an infinite number of solutions to this system of equations. A solution is feasible if all of $x_1, x_2, \ldots, x_6$ are nonnegative, and there can be an infinite number of feasible solutions as well. The infinite number of possible solutions to a system such as this one will be useful in later proofs. We focus on the ***basic solution***: set all the (nonbasic) variables on the right-hand side to 0 and then compute the values of the (basic) variables on the left-hand side. In this example, the basic solution is $(\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_6) = (0, 0, 0, 30, 24, 36)$ and it has objective value $z = (3 \cdot 0) + (1 \cdot 0) + (2 \cdot 0) = 0$. Observe that this basic solution sets $\bar{x}_i = b_i$ for each $i \in B$. An iteration of the simplex algorithm rewrites the set of equations and the objective function so as to put a different set of variables on the right-hand side. Thus, a different basic solution is associated with the rewritten problem. We emphasize that the rewrite does not in any way change the underlying linear-programming problem; the problem at one iteration has the identical set of feasible solutions as the problem at the previous iteration. The problem does, however, have a different basic solution than that of the previous iteration.

If a basic solution is also feasible, we call it a ***basic feasible solution***. As we run the simplex algorithm, the basic solution is almost always a basic feasible solution. We shall see in Section 29.5, however, that for the first few iterations of the simplex algorithm, the basic solution might not be feasible.

Our goal, in each iteration, is to reformulate the linear program so that the basic solution has a greater objective value. We select a nonbasic variable x_e whose coefficient in the objective function is positive, and we increase the value of x_e as much as possible without violating any of the constraints. The variable x_e becomes basic, and some other variable x_l becomes nonbasic. The values of other basic variables and of the objective function may also change.

To continue the example, let's think about increasing the value of x_1. As we increase x_1, the values of x_4, x_5, and x_6 all decrease. Because we have a nonnegativity constraint for each variable, we cannot allow any of them to become negative. If x_1 increases above 30, then x_4 becomes negative, and x_5 and x_6 become negative when x_1 increases above 12 and 9, respectively. The third constraint (29.61) is the tightest constraint, and it limits how much we can increase x_1. Therefore, we switch the roles of x_1 and x_6. We solve equation (29.61) for x_1 and obtain

$$x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4} \ . \tag{29.62}$$

To rewrite the other equations with x_6 on the right-hand side, we substitute for x_1 using equation (29.62). Doing so for equation (29.59), we obtain

$$\begin{aligned}
x_4 &= 30 - x_1 - x_2 - 3x_3 \\
&= 30 - \left(9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4}\right) - x_2 - 3x_3 \\
&= 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4} . \qquad (29.63)
\end{aligned}$$

Similarly, we combine equation (29.62) with constraint (29.60) and with objective function (29.58) to rewrite our linear program in the following form:

$$\begin{aligned}
z &= 27 + \frac{x_2}{4} + \frac{x_3}{2} - \frac{3x_6}{4} \qquad (29.64) \\
x_1 &= 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4} \qquad (29.65) \\
x_4 &= 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4} \qquad (29.66) \\
x_5 &= 6 - \frac{3x_2}{2} - 4x_3 + \frac{x_6}{2} . \qquad (29.67)
\end{aligned}$$

We call this operation a ***pivot***. As demonstrated above, a pivot chooses a nonbasic variable x_e, called the ***entering variable***, and a basic variable x_l, called the ***leaving variable***, and exchanges their roles.

The linear program described in equations (29.64)–(29.67) is equivalent to the linear program described in equations (29.58)–(29.61). We perform two operations in the simplex algorithm: rewrite equations so that variables move between the left-hand side and the right-hand side, and substitute one equation into another. The first operation trivially creates an equivalent problem, and the second, by elementary linear algebra, also creates an equivalent problem. (See Exercise 29.3-3.)

To demonstrate this equivalence, observe that our original basic solution $(0, 0, 0, 30, 24, 36)$ satisfies the new equations (29.65)–(29.67) and has objective value $27 + (1/4) \cdot 0 + (1/2) \cdot 0 - (3/4) \cdot 36 = 0$. The basic solution associated with the new linear program sets the nonbasic values to 0 and is $(9, 0, 0, 21, 6, 0)$, with objective value $z = 27$. Simple arithmetic verifies that this solution also satisfies equations (29.59)–(29.61) and, when plugged into objective function (29.58), has objective value $(3 \cdot 9) + (1 \cdot 0) + (2 \cdot 0) = 27$.

Continuing the example, we wish to find a new variable whose value we wish to increase. We do not want to increase x_6, since as its value increases, the objective value decreases. We can attempt to increase either x_2 or x_3; let us choose x_3. How far can we increase x_3 without violating any of the constraints? Constraint (29.65) limits it to 18, constraint (29.66) limits it to $42/5$, and constraint (29.67) limits it to $3/2$. The third constraint is again the tightest one, and therefore we rewrite the third constraint so that x_3 is on the left-hand side and x_5 is on the right-hand side. We then substitute this new equation, $x_3 = 3/2 - 3x_2/8 - x_5/4 + x_6/8$, into equations (29.64)–(29.66) and obtain the new, but equivalent, system

$$z = \frac{111}{4} + \frac{x_2}{16} - \frac{x_5}{8} - \frac{11x_6}{16} \tag{29.68}$$

$$x_1 = \frac{33}{4} - \frac{x_2}{16} + \frac{x_5}{8} - \frac{5x_6}{16} \tag{29.69}$$

$$x_3 = \frac{3}{2} - \frac{3x_2}{8} - \frac{x_5}{4} + \frac{x_6}{8} \tag{29.70}$$

$$x_4 = \frac{69}{4} + \frac{3x_2}{16} + \frac{5x_5}{8} - \frac{x_6}{16} . \tag{29.71}$$

This system has the associated basic solution $(33/4, 0, 3/2, 69/4, 0, 0)$, with objective value $111/4$. Now the only way to increase the objective value is to increase x_2. The three constraints give upper bounds of 132, 4, and ∞, respectively. (We get an upper bound of ∞ from constraint (29.71) because, as we increase x_2, the value of the basic variable x_4 increases also. This constraint, therefore, places no restriction on how much we can increase x_2.) We increase x_2 to 4, and it becomes nonbasic. Then we solve equation (29.70) for x_2 and substitute in the other equations to obtain

$$z = 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \tag{29.72}$$

$$x_1 = 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \tag{29.73}$$

$$x_2 = 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \tag{29.74}$$

$$x_4 = 18 - \frac{x_3}{2} + \frac{x_5}{2} . \tag{29.75}$$

At this point, all coefficients in the objective function are negative. As we shall see later in this chapter, this situation occurs only when we have rewritten the linear program so that the basic solution is an optimal solution. Thus, for this problem, the solution $(8, 4, 0, 18, 0, 0)$, with objective value 28, is optimal. We can now return to our original linear program given in (29.53)–(29.57). The only variables in the original linear program are x_1, x_2, and x_3, and so our solution is $x_1 = 8$, $x_2 = 4$, and $x_3 = 0$, with objective value $(3 \cdot 8) + (1 \cdot 4) + (2 \cdot 0) = 28$. Note that the values of the slack variables in the final solution measure how much slack remains in each inequality. Slack variable x_4 is 18, and in inequality (29.54), the left-hand side, with value $8 + 4 + 0 = 12$, is 18 less than the right-hand side of 30. Slack variables x_5 and x_6 are 0 and indeed, in inequalities (29.55) and (29.56), the left-hand and right-hand sides are equal. Observe also that even though the coefficients in the original slack form are integral, the coefficients in the other linear programs are not necessarily integral, and the intermediate solutions are not necessarily integral. Furthermore, the final solution to a linear program need not be integral; it is purely coincidental that this example has an integral solution.

Pivoting

We now formalize the procedure for pivoting. The procedure PIVOT takes as input a slack form, given by the tuple (N, B, A, b, c, ν), the index l of the leaving variable x_l, and the index e of the entering variable x_e. It returns the tuple $(\widehat{N}, \widehat{B}, \widehat{A}, \widehat{b}, \widehat{c}, \widehat{\nu})$ describing the new slack form. (Recall again that the entries of the $m \times n$ matrices A and $\widehat{A}$ are actually the negatives of the coefficients that appear in the slack form.)

PIVOT$(N, B, A, b, c, \nu, l, e)$

1 // Compute the coefficients of the equation for new basic variable x_e.
2 let $\widehat{A}$ be a new $m \times n$ matrix
3 $\widehat{b}_e = b_l / a_{le}$
4 **for** each $j \in N - \{e\}$
5     $\widehat{a}_{ej} = a_{lj} / a_{le}$
6 $\widehat{a}_{el} = 1 / a_{le}$
7 // Compute the coefficients of the remaining constraints.
8 **for** each $i \in B - \{l\}$
9     $\widehat{b}_i = b_i - a_{ie} \widehat{b}_e$
10   **for** each $j \in N - \{e\}$
11         $\widehat{a}_{ij} = a_{ij} - a_{ie} \widehat{a}_{ej}$
12     $\widehat{a}_{il} = -a_{ie} \widehat{a}_{el}$
13 // Compute the objective function.
14 $\widehat{\nu} = \nu + c_e \widehat{b}_e$
15 **for** each $j \in N - \{e\}$
16     $\widehat{c}_j = c_j - c_e \widehat{a}_{ej}$
17 $\widehat{c}_l = -c_e \widehat{a}_{el}$
18 // Compute new sets of basic and nonbasic variables.
19 $\widehat{N} = N - \{e\} \cup \{l\}$
20 $\widehat{B} = B - \{l\} \cup \{e\}$
21 **return** $(\widehat{N}, \widehat{B}, \widehat{A}, \widehat{b}, \widehat{c}, \widehat{\nu})$

PIVOT works as follows. Lines 3–6 compute the coefficients in the new equation for x_e by rewriting the equation that has x_l on the left-hand side to instead have x_e on the left-hand side. Lines 8–12 update the remaining equations by substituting the right-hand side of this new equation for each occurrence of x_e. Lines 14–17 do the same substitution for the objective function, and lines 19 and 20 update the sets of nonbasic and basic variables. Line 21 returns the new slack form. As given, if $a_{le} = 0$, PIVOT would cause an error by dividing by 0, but as we shall see in the proofs of Lemmas 29.2 and 29.12, we call PIVOT only when $a_{le} \neq 0$.

We now summarize the effect that PIVOT has on the values of the variables in the basic solution.

Lemma 29.1

Consider a call to PIVOT$(N, B, A, b, c, \nu, l, e)$ in which $a_{le} \neq 0$. Let the values returned from the call be $(\widehat{N}, \widehat{B}, \widehat{A}, \widehat{b}, \widehat{c}, \widehat{\nu})$, and let $\bar{x}$ denote the basic solution after the call. Then

1. $\bar{x}_j = 0$ for each $j \in \widehat{N}$.
2. $\bar{x}_e = b_l / a_{le}$.
3. $\bar{x}_i = b_i - a_{ie}\widehat{b}_e$ for each $i \in \widehat{B} - \{e\}$.

Proof The first statement is true because the basic solution always sets all nonbasic variables to 0. When we set each nonbasic variable to 0 in a constraint

$$x_i = \widehat{b}_i - \sum_{j \in \widehat{N}} \widehat{a}_{ij} x_j ,$$

we have that $\bar{x}_i = \widehat{b}_i$ for each $i \in \widehat{B}$. Since $e \in \widehat{B}$, line 3 of PIVOT gives

$$\bar{x}_e = \widehat{b}_e = b_l / a_{le} ,$$

which proves the second statement. Similarly, using line 9 for each $i \in \widehat{B} - \{e\}$, we have

$$\bar{x}_i = \widehat{b}_i = b_i - a_{ie}\widehat{b}_e ,$$

which proves the third statement. ■

The formal simplex algorithm

We are now ready to formalize the simplex algorithm, which we demonstrated by example. That example was a particularly nice one, and we could have had several other issues to address:

- How do we determine whether a linear program is feasible?
- What do we do if the linear program is feasible, but the initial basic solution is not feasible?
- How do we determine whether a linear program is unbounded?
- How do we choose the entering and leaving variables?

In Section 29.5, we shall show how to determine whether a problem is feasible, and if so, how to find a slack form in which the initial basic solution is feasible. Therefore, let us assume that we have a procedure INITIALIZE-SIMPLEX(A, b, c) that takes as input a linear program in standard form, that is, an $m \times n$ matrix $A = (a_{ij})$, an m-vector $b = (b_i)$, and an n-vector $c = (c_j)$. If the problem is infeasible, the procedure returns a message that the program is infeasible and then terminates. Otherwise, the procedure returns a slack form for which the initial basic solution is feasible.

The procedure SIMPLEX takes as input a linear program in standard form, as just described. It returns an n-vector $\bar{x} = (\bar{x}_j)$ that is an optimal solution to the linear program described in (29.19)–(29.21).

```
SIMPLEX(A, b, c)
 1  (N, B, A, b, c, ν) = INITIALIZE-SIMPLEX(A, b, c)
 2  let Δ be a new vector of length n
 3  while some index j ∈ N has c_j > 0
 4      choose an index e ∈ N for which c_e > 0
 5      for each index i ∈ B
 6          if a_ie > 0
 7              Δ_i = b_i / a_ie
 8          else Δ_i = ∞
 9      choose an index l ∈ B that minimizes Δ_i
10      if Δ_l == ∞
11          return "unbounded"
12      else (N, B, A, b, c, ν) = PIVOT(N, B, A, b, c, ν, l, e)
13  for i = 1 to n
14      if i ∈ B
15          x̄_i = b_i
16      else x̄_i = 0
17  return (x̄_1, x̄_2, ..., x̄_n)
```

The SIMPLEX procedure works as follows. In line 1, it calls the procedure INITIALIZE-SIMPLEX(A, b, c), described above, which either determines that the linear program is infeasible or returns a slack form for which the basic solution is feasible. The **while** loop of lines 3–12 forms the main part of the algorithm. If all coefficients in the objective function are negative, then the **while** loop terminates. Otherwise, line 4 selects a variable x_e, whose coefficient in the objective function is positive, as the entering variable. Although we may choose any such variable as the entering variable, we assume that we use some prespecified deterministic rule. Next, lines 5–9 check each constraint and pick the one that most severely limits the amount by which we can increase x_e without violating any of the nonnegativity constraints; the basic variable associated with this constraint is x_l. Again, we are free to choose one of several variables as the leaving variable, but we assume that we use some prespecified deterministic rule. If none of the constraints limits the amount by which the entering variable can increase, the algorithm returns "unbounded" in line 11. Otherwise, line 12 exchanges the roles of the entering and leaving variables by calling PIVOT(N, B, A, b, c, v, l, e), as described above. Lines 13–16 compute a solution $\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n$ for the original linear-programming variables by setting all the nonbasic variables to 0 and each basic variable $\bar{x}_i$ to b_i, and line 17 returns these values.

To show that SIMPLEX is correct, we first show that if SIMPLEX has an initial feasible solution and eventually terminates, then it either returns a feasible solution or determines that the linear program is unbounded. Then, we show that SIMPLEX terminates. Finally, in Section 29.4 (Theorem 29.10) we show that the solution returned is optimal.

Lemma 29.2

Given a linear program (A, b, c), suppose that the call to INITIALIZE-SIMPLEX in line 1 of SIMPLEX returns a slack form for which the basic solution is feasible. Then if SIMPLEX returns a solution in line 17, that solution is a feasible solution to the linear program. If SIMPLEX returns "unbounded" in line 11, the linear program is unbounded.

Proof We use the following three-part loop invariant:

At the start of each iteration of the **while** loop of lines 3–12,

1. the slack form is equivalent to the slack form returned by the call of INITIALIZE-SIMPLEX,
2. for each $i \in B$, we have $b_i \geq 0$, and
3. the basic solution associated with the slack form is feasible.

Initialization: The equivalence of the slack forms is trivial for the first iteration. We assume, in the statement of the lemma, that the call to INITIALIZE-SIMPLEX in line 1 of SIMPLEX returns a slack form for which the basic solution is feasible. Thus, the third part of the invariant is true. Because the basic solution is feasible, each basic variable x_i is nonnegative. Furthermore, since the basic solution sets each basic variable x_i to b_i, we have that $b_i \geq 0$ for all $i \in B$. Thus, the second part of the invariant holds.

Maintenance: We shall show that each iteration of the **while** loop maintains the loop invariant, assuming that the **return** statement in line 11 does not execute. We shall handle the case in which line 11 executes when we discuss termination.

An iteration of the **while** loop exchanges the role of a basic and a nonbasic variable by calling the PIVOT procedure. By Exercise 29.3-3, the slack form is equivalent to the one from the previous iteration which, by the loop invariant, is equivalent to the initial slack form.

We now demonstrate the second part of the loop invariant. We assume that at the start of each iteration of the **while** loop, $b_i \geq 0$ for each $i \in B$, and we shall show that these inequalities remain true after the call to PIVOT in line 12. Since the only changes to the variables b_i and the set B of basic variables occur in this assignment, it suffices to show that line 12 maintains this part of the invariant. We let b_i, a_{ij}, and B refer to values before the call of PIVOT, and $\hat{b}_i$ refer to values returned from PIVOT.

First, we observe that $\hat{b}_e \geq 0$ because $b_l \geq 0$ by the loop invariant, $a_{le} > 0$ by lines 6 and 9 of SIMPLEX, and $\hat{b}_e = b_l / a_{le}$ by line 3 of PIVOT.

For the remaining indices $i \in B - \{l\}$, we have that

$$\begin{aligned} \hat{b}_i &= b_i - a_{ie}\hat{b}_e && \text{(by line 9 of PIVOT)} \\ &= b_i - a_{ie}(b_l/a_{le}) && \text{(by line 3 of PIVOT)} \; . \end{aligned} \tag{29.76}$$

We have two cases to consider, depending on whether $a_{ie} > 0$ or $a_{ie} \leq 0$. If $a_{ie} > 0$, then since we chose l such that

$$b_l/a_{le} \leq b_i/a_{ie} \quad \text{for all } i \in B \; , \tag{29.77}$$

we have

$$\begin{aligned} \hat{b}_i &= b_i - a_{ie}(b_l/a_{le}) && \text{(by equation (29.76))} \\ &\geq b_i - a_{ie}(b_i/a_{ie}) && \text{(by inequality (29.77))} \\ &= b_i - b_i \\ &= 0 \; , \end{aligned}$$

and thus $\hat{b}_i \geq 0$. If $a_{ie} \leq 0$, then because a_{le}, b_i, and b_l are all nonnegative, equation (29.76) implies that $\hat{b}_i$ must be nonnegative, too.

We now argue that the basic solution is feasible, i.e., that all variables have nonnegative values. The nonbasic variables are set to 0 and thus are nonnegative. Each basic variable x_i is defined by the equation

$$x_i = b_i - \sum_{j \in N} a_{ij} x_j \; .$$

The basic solution sets $\bar{x}_i = b_i$. Using the second part of the loop invariant, we conclude that each basic variable $\bar{x}_i$ is nonnegative.

Termination: The **while** loop can terminate in one of two ways. If it terminates because of the condition in line 3, then the current basic solution is feasible and line 17 returns this solution. The other way it terminates is by returning "unbounded" in line 11. In this case, for each iteration of the **for** loop in lines 5–8, when line 6 is executed, we find that $a_{ie} \leq 0$. Consider the solution $\bar{x}$ defined as

$$\bar{x}_i = \begin{cases} \infty & \text{if } i = e , \\ 0 & \text{if } i \in N - \{e\} , \\ b_i - \sum_{j \in N} a_{ij}\bar{x}_j & \text{if } i \in B . \end{cases}$$

We now show that this solution is feasible, i.e., that all variables are nonnegative. The nonbasic variables other than $\bar{x}_e$ are 0, and $\bar{x}_e = \infty > 0$; thus all nonbasic variables are nonnegative. For each basic variable $\bar{x}_i$, we have

$$\begin{aligned} \bar{x}_i &= b_i - \sum_{j \in N} a_{ij}\bar{x}_j \\ &= b_i - a_{ie}\bar{x}_e . \end{aligned}$$

The loop invariant implies that $b_i \geq 0$, and we have $a_{ie} \leq 0$ and $\bar{x}_e = \infty > 0$. Thus, $\bar{x}_i \geq 0$.

Now we show that the objective value for the solution $\bar{x}$ is unbounded. From equation (29.42), the objective value is

$$\begin{aligned} z &= \nu + \sum_{j \in N} c_j\bar{x}_j \\ &= \nu + c_e\bar{x}_e . \end{aligned}$$

Since $c_e > 0$ (by line 4 of SIMPLEX) and $\bar{x}_e = \infty$, the objective value is ∞, and thus the linear program is unbounded. ■

It remains to show that SIMPLEX terminates, and when it does terminate, the solution it returns is optimal. Section 29.4 will address optimality. We now discuss termination.

Termination

In the example given in the beginning of this section, each iteration of the simplex algorithm increased the objective value associated with the basic solution. As Exercise 29.3-2 asks you to show, no iteration of SIMPLEX can decrease the objective value associated with the basic solution. Unfortunately, it is possible that an iteration leaves the objective value unchanged. This phenomenon is called ***degeneracy***, and we shall now study it in greater detail.

The assignment in line 14 of PIVOT, $\widehat{\nu} = \nu + c_e \widehat{b}_e$, changes the objective value. Since SIMPLEX calls PIVOT only when $c_e > 0$, the only way for the objective value to remain unchanged (i.e., $\widehat{\nu} = \nu$) is for $\widehat{b}_e$ to be 0. This value is assigned as $\widehat{b}_e = b_l / a_{le}$ in line 3 of PIVOT. Since we always call PIVOT with $a_{le} \neq 0$, we see that for $\widehat{b}_e$ to equal 0, and hence the objective value to be unchanged, we must have $b_l = 0$.

Indeed, this situation can occur. Consider the linear program

$$
\begin{array}{rcrcrcrcr}
z & = & & & x_1 & + & x_2 & + & x_3 \\
x_4 & = & 8 & - & x_1 & - & x_2 & & \\
x_5 & = & & & & & x_2 & - & x_3 \; .
\end{array}
$$

Suppose that we choose x_1 as the entering variable and x_4 as the leaving variable. After pivoting, we obtain

$$
\begin{array}{rcrcrcrcr}
z & = & 8 & & & + & x_3 & - & x_4 \\
x_1 & = & 8 & - & x_2 & & & - & x_4 \\
x_5 & = & & & x_2 & - & x_3 & . &
\end{array}
$$

At this point, our only choice is to pivot with x_3 entering and x_5 leaving. Since $b_5 = 0$, the objective value of 8 remains unchanged after pivoting:

$$
\begin{array}{rcrcrcrcr}
z & = & 8 & + & x_2 & - & x_4 & - & x_5 \\
x_1 & = & 8 & - & x_2 & - & x_4 & & \\
x_3 & = & & & x_2 & & & - & x_5 \; .
\end{array}
$$

The objective value has not changed, but our slack form has. Fortunately, if we pivot again, with x_2 entering and x_1 leaving, the objective value increases (to 16), and the simplex algorithm can continue.

Degeneracy can prevent the simplex algorithm from terminating, because it can lead to a phenomenon known as ***cycling***: the slack forms at two different iterations of SIMPLEX are identical. Because of degeneracy, SIMPLEX could choose a sequence of pivot operations that leave the objective value unchanged but repeat a slack form within the sequence. Since SIMPLEX is a deterministic algorithm, if it cycles, then it will cycle through the same series of slack forms forever, never terminating.

Cycling is the only reason that SIMPLEX might not terminate. To show this fact, we must first develop some additional machinery.

At each iteration, SIMPLEX maintains A, b, c, and ν in addition to the sets N and B. Although we need to explicitly maintain A, b, c, and ν in order to implement the simplex algorithm efficiently, we can get by without maintaining them. In other words, the sets of basic and nonbasic variables suffice to uniquely determine the slack form. Before proving this fact, we prove a useful algebraic lemma.

Lemma 29.3

Let I be a set of indices. For each $j \in I$, let α_j and β_j be real numbers, and let x_j be a real-valued variable. Let γ be any real number. Suppose that for any settings of the x_j, we have

$$\sum_{j \in I} \alpha_j x_j = \gamma + \sum_{j \in I} \beta_j x_j \; . \tag{29.78}$$

Then $\alpha_j = \beta_j$ for each $j \in I$, and $\gamma = 0$.

Proof Since equation (29.78) holds for any values of the x_j, we can use particular values to draw conclusions about α, β, and γ. If we let $x_j = 0$ for each $j \in I$, we conclude that $\gamma = 0$. Now pick an arbitrary index $j \in I$, and set $x_j = 1$ and $x_k = 0$ for all $k \neq j$. Then we must have $\alpha_j = \beta_j$. Since we picked j as any index in I, we conclude that $\alpha_j = \beta_j$ for each $j \in I$. ■

A particular linear program has many different slack forms; recall that each slack form has the same set of feasible and optimal solutions as the original linear program. We now show that the slack form of a linear program is uniquely determined by the set of basic variables. That is, given the set of basic variables, a unique slack form (unique set of coefficients and right-hand sides) is associated with those basic variables.

Lemma 29.4

Let (A, b, c) be a linear program in standard form. Given a set B of basic variables, the associated slack form is uniquely determined.

Proof Assume for the purpose of contradiction that there are two different slack forms with the same set B of basic variables. The slack forms must also have identical sets $N = \{1, 2, \ldots, n + m\} - B$ of nonbasic variables. We write the first slack form as

$$z = v + \sum_{j \in N} c_j x_j \tag{29.79}$$

$$x_i = b_i - \sum_{j \in N} a_{ij} x_j \quad \text{for } i \in B \; , \tag{29.80}$$

and the second as

$$z = v' + \sum_{j \in N} c'_j x_j \tag{29.81}$$

$$x_i = b'_i - \sum_{j \in N} a'_{ij} x_j \quad \text{for } i \in B \; . \tag{29.82}$$