519,852
* 3ER

Introduction to Linear Optimization

Dimitris Bertsimas

John N. Tsitsiklis

Massachusetts Institute of Technology

Athena Scientific, Belmont, Massachusetts

Contents

linear optimization.max

Preface

The purpose of this book is to provide a unified, insightful, and modern treatment of linear optimization, that is, linear programming, network flow problems, and discrete linear optimization. We discuss both classical topics, as well as the state of the art. We give special attention to theory, but also cover applications and present case studies. Our main objective is to help the reader become a sophisticated practitioner of (linear) optimization, or a researcher. More specifically, we wish to develop the ability to formulate fairly complex optimization problems, provide an appreciation of the main classes of problems that are practically solvable, describe the available solution methods, and build an understanding of the qualitative properties of the solutions they provide.

Our general philosophy is that insight matters most. For the subject matter of this book, this necessarily requires a geometric view. On the other hand, problems are solved by algorithms, and these can only be described algebraically. Hence, our focus is on the beautiful interplay between algebra and geometry. We build understanding using figures and geometric arguments, and then translate ideas into algebraic formulas and algorithms. Given enough time, we expect that the reader will develop the ability to pass from one domain to the other without much effort.

Another of our objectives is to be comprehensive, but economical. We have made an effort to cover and highlight all of the principal ideas in this field. However, we have not tried to be encyclopedic, or to discuss every possible detail relevant to a particular algorithm. Our premise is that once mature understanding of the basic principles is in place, further details can be acquired by the reader with little additional effort.

Our last objective is to bring the reader up to date with respect to the state of the art. This is especially true in our treatment of interior point methods, large scale optimization, and the presentation of case studies that stretch the limits of currently available algorithms and computers.

The success of any optimization methodology hinges on its ability to deal with large and important problems. In that sense, the last chapter, on the art of linear optimization, is a critical part of this book. It will, we hope, convince the reader that progress on challenging problems requires both problem specific insight, as well as a deeper understanding of the underlying theory

In any book dealing with linear programming, there are some important choices to be made regarding the treatment of the simplex method. Traditionally, the simplex method is developed in terms of the full simplex tableau, which tends to become the central topic. We have found that the full simplex tableau is a useful device for working out numerical examples. But other than that, we have tried not to overemphasize its importance.

Let us also mention another departure from many other textbooks. Introductory treatments often focus on standard form problems, which is sufficient for the purposes of the simplex method. On the other hand, this approach often leaves the reader wondering whether certain properties are generally true, and can hinder the deeper understanding of the subject. We depart from this tradition: we consider the general form of linear programming problems and define key concepts (e.g., extreme points) within this context. (Of course, when it comes to algorithms, we often have to specialize to the standard form.) In the same spirit, we separate the structural understanding of linear programming from the particulars of the simplex method. For example, we include a derivation of duality theory that does not rely on the simplex method.

Finally, this book contains a treatment of several important topics that are not commonly covered. These include a discussion of the column geometry and of the insights it provides into the efficiency of the simplex method, the connection between duality and the pricing of financial assets, a unified view of delayed column generation and cutting plane methods, stochastic programming and Benders decomposition, the auction algorithm for the assignment problem, certain theoretical implications of the ellipsoid algorithm, a thorough treatment of interior point methods, and a whole chapter on the practice of linear optimization. There are also several noteworthy topics that are covered in the exercises, such as Leontief systems, strict complementarity, options pricing, von Neumann's algorithm, submodular function minimization, and bounds for a number of integer programming problems.

Here is a chapter by chapter description of the book.

Chapter 1: Introduces the linear programming problem, together with a number of examples, and provides some background material on linear algebra.

Chapter 2: Deals with the basic geometric properties of polyhedra, focusing on the definition and the existence of extreme points, and emphasizing the interplay between the geometric and the algebraic viewpoints.

Chapter 3: Contains more or less the classical material associated with the simplex method. It starts with a discussion of the column geometry. It starts with a high-level and geometrically motivated derivation of the simplex method. It then introduces the revised simplex method, and concludes with the simplex tableau. The usual topics of Phase I and anticycling are

also covered.

Chapter 4: It is a comprehensive treatment of linear programming duality. The duality theorem is first obtained as a corollary of the simplex method. A more abstract derivation is also provided, based on the separating hyperplane theorem, which is developed from first principles. It ends with a deeper look into the geometry of polyhedra.

Chapter 5: Discusses sensitivity analysis, that is, the dependence of solutions and the optimal cost on the problem data, including parametric programming. It also develops a characterization of dual optimal solutions as subgradients of a suitably defined optimal cost function.

Chapter 6: Presents the complementary ideas of delayed column generation and cutting planes. These methods are first developed at a high level, and are then made concrete by discussing the cutting stock problem, Dantzig-Wolfe decomposition, stochastic programming, and Benders decomposition.

Chapter 7: Provides a comprehensive review of the principal results and methods for the different variants of the network flow problem. It contains representatives from all major types of algorithms: primal descent (the simplex method), dual ascent (the primal-dual method), and approximate dual ascent (the auction algorithm). The focus is on the major algorithmic ideas, rather than on the refinements that can lead to better complexity estimates.

Chapter 8: Includes a discussion of complexity, a development of the ellipsoid method, and a proof of the polynomiality of linear programming. It also discusses the equivalence of separation and optimization, and provides examples where the ellipsoid algorithm can be used to derive polynomial time results for problems involving an exponential number of constraints.

Chapter 9: Contains an overview of all major classes of interior point methods, including affine scaling, potential reduction, and path following (both primal and primal-dual) methods. It includes a discussion of the underlying geometric ideas and computational issues, as well as convergence proofs and complexity analysis.

Chapter 10: Introduces integer programming formulations of discrete optimization problems. It provides a number of examples, as well as some intuition as to what constitutes a "strong" formulation.

Chapter 11: Covers the major classes of integer programming algorithms, including exact methods (branch and bound, cutting planes, dynamic programming), approximation algorithms, and heuristic methods (local search and simulated annealing). It also introduces a duality theory for integer programming.

Chapter 12: Deals with the art in linear optimization, i.e., the process

of modeling, exploiting problem structure, and fine tuning of optimization algorithms. We discuss the relative performance of interior point methods and different variants of the simplex method, in a realistic large scale setting. We also give some indication of the size of problems that can be currently solved.

An important theme that runs through several chapters is the modeling, complexity, and algorithms for problems with an exponential number of constraints. We discuss modeling in Section 10.3, complexity in Section 8.5, algorithmic approaches in Chapter 6 and 8.5, and we conclude with a case study in Section 12.5.

There is a fair number of exercises that are given at the end of each chapter. Most of them are intended to deepen the understanding of the subject, or to explore extensions of the theory in the text, as opposed to routine drills. However, several numerical exercises are also included. Starred exercises are supposed to be fairly hard. A solutions manual for qualified instructors can be obtained from the authors.

We have made a special effort to keep the text as modular as possible, allowing the reader to omit certain topics without loss of continuity. For example, much of the material in Chapters 5 and 6 is rarely used in the rest of the book. Furthermore, in Chapter 7 (on network flow problems), a reader who has gone through the problem formulation (Sections 7.1-7.2) can immediately move to any later section in that chapter. Also, the interior point algorithms of Chapter 9 are not used later, with the exception of some of the applications in Chapter 12. Even within the core chapters (Chapters 1-4), there are many sections that can be skipped during a first reading. Some sections have been marked with a star indicating that they contain somewhat more advanced material that is not usually covered in an introductory course.

The book was developed while we took turns teaching a first-year graduate course at M.I.T., for students in engineering and operations research. The only prerequisite is a working knowledge of linear algebra. In fact, it is only a small subset of linear algebra that is needed (e.g., the concepts of subspaces, linear independence, and the rank of a matrix). However, these elementary tools are sometimes used in subtle ways, and some mathematical maturity on the part of the reader can lead to a better appreciation of the subject.

The book can be used to teach several different types of courses. The first two suggestions below are one-semester variants that we have tried at M.I.T., but there are also other meaningful alternatives, depending on the students' background and the course's objectives.

(a) Cover most of Chapters 1-7, and if time permits, cover a small number of topics from Chapters 9-12.

(b) An alternative could be the same as above, except that interior point

algorithms (Chapter 9) are fully covered, replacing network flow problems (Chapter 7).

(c) A broad overview course can be constructed by concentrating on the easier material in most of the chapters. The core of such a course could consist of Chapter 1, Sections 2.1-2.4, 3.1-3.5, 4.1-4.3, 5.1, 7.1-7.3, 9.1, 10.1, some of the easier material in Chapter 11, and an application from Chapter 12.

(d) Finally, the book is also suitable for a half-course on integer programming, based on parts of Chapters 1 and 8, as well as Chapters 10-12.

There is a truly large literature on linear optimization, and we make no attempt to provide a comprehensive bibliography. To a great extent, the sources that we cite are either original references of historical interest, or recent texts where additional information can be found. For those topics, however, that touch upon current research, we also provide pointers to recent journal articles.

We would like to express our thanks to a number of individuals. We are grateful to our colleagues Dimitri Bertsekas and Rob Freund, for many discussions on the subjects in this book, as well as for reading parts of the manuscript. Several of our students, colleagues, and friends have contributed by reading parts of the manuscript, providing critical comments, and working on the exercises Jim Christodouleas, Thalia Chryssikou, Austin Frakt, David Gamarnik Leon Hsu, Spyros Kontogiorgis, Peter Marbach, Gina Mourtzinou, Yannis Paschalidis, Georgia Perakis, Lakis Polymenakos, Jay Sethuraman, Sarah Stock, Paul Tseng, and Ben Van Roy. But mostly, we are grateful to our families for their patience, love, and support in the course of this long project.

Dimitris Bertsimas
John N. Tsitsiklis
Cambridge, January 1997

Chapter 1

Introduction

Contents

linear optimization.max

In this chapter, we introduce *linear programming*, the problem of minimizing a linear cost function subject to linear equality and inequality constraints. We consider a few equivalent forms and then present a number of examples to illustrate the applicability of linear programming to a wide variety of contexts. We also solve a few simple examples and obtain some basic geometric intuition on the nature of the problem. The chapter ends with a review of linear algebra and of the conventions used in describing the computational requirements (operation count) of algorithms.

1.1 Variants of the linear programming problem

In this section, we pose the linear programming problem, discuss a few special forms that it takes, and establish some standard notation that we will be using. Rather than starting abstractly, we first state a concrete example, which is meant to facilitate understanding of the formal definition that will follow. The example we give is devoid of any interpretation. Later on, in Section 1.2, we will have ample opportunity to develop examples that arise in practical settings.

Example 1.1 The following is a linear programming problem:

$$
\begin{array}{lrcl}
\text{minimize} & 2x_1 - x_2 + 4x_3 & & \\
\text{subject to} & x_1 + x_2 + x_3 - x_4 & \leq & 2 \\
& 3x_2 - x_3 & = & 5 \\
& x_3 - x_4 & \geq & 3 \\
& x_1 & \geq & 0 \\
& x_3 & \leq & 0.
\end{array}
$$

Here x_1, x_2, x_3, and x_4 are variables whose values are to be chosen to minimize the linear cost function $2x_1 - x_2 + 4x_3$, subject to a set of linear equality and inequality constraints. Some of these constraints, such as $x_1 \geq 0$ and $x_3 \leq 0$, amount to simple restrictions on the sign of certain variables. The remaining constraints are of the form $a'x \leq b$, $a'x = b$, or $a'x \geq b$, where $a = (a_1, a_2, a_3, a_4)$ is a given vector[1], $x = (x_1, x_2, x_3, x_4)$ is the vector of decision variables, $a'x$ is their inner product $\sum_{i=1}^4 a_i x_i$, and b is a given scalar. For example, in the first constraint, we have $a = (1, 1, 1, 0)$ and $b = 2$.

We now generalize. In a *general* linear programming problem, we are given a cost vector $c = (c_1, \ldots, c_n)$ and we seek to minimize a linear cost function $c'x = \sum_{i=1}^n c_i x_i$ over all n-dimensional vectors $x = (x_1, \ldots, x_n)$,

---

[1] As discussed further in Section 1.5, all vectors are assumed to be column vectors, and are treated as such in matrix-vector products. Row vectors are indicated as transposes of (column) vectors. However, whenever we refer to a vector x inside the text, we use the more economical notation $x = (x_1, \ldots, x_n)$, even though x is a column vector. The reader who is unfamiliar with our notation may wish to consult Section 1.5 before continuing

subject to a set of linear equality and inequality constraints. In particular, let M_1, M_2, M_3 be some finite index sets, and suppose that for every i in any one of these sets, we are given an n-dimensional vector a_i and a scalar b_i, that will be used to form the ith constraint. Let also N_1 and N_2 be subsets of $\{1, \ldots, n\}$ that indicate which variables x_j are constrained to be nonnegative or nonpositive, respectively. We then consider the problem

$$
\begin{array}{lrcll}
\text{minimize} & c'x & & & \\
\text{subject to} & a_i'x & \geq & b_i, & i \in M_1, \\
& a_i'x & \leq & b_i, & i \in M_2, \\
& a_i'x & = & b_i, & i \in M_3, \\
& x_j & \geq & 0, & j \in N_1, \\
& x_j & \leq & 0, & j \in N_2.
\end{array}
\tag{1.1}
$$

The variables $x_1, \ldots, x_n$ are called *decision variables*, and a vector x satisfying all of the constraints is called a *feasible solution* or *feasible vector*. The set of all feasible solutions is called the *feasible set* or *feasible region*. If j is in neither N_1 nor N_2, there are no restrictions on the sign of x_j; in which case we say that x_j is a *free* or *unrestricted* variable. The function $c'x$ is called the *objective function* or *cost function*. A feasible solution x^* that minimizes the objective function (that is, $c'x^* \leq c'x$, for all feasible x) is called an *optimal feasible solution* or, simply, an *optimal solution*. The value of $c'x^*$ is then called the *optimal cost*. On the other hand, if for every real number K we can find a feasible solution x whose cost is less than K, we say that the optimal cost is $-\infty$ or that the cost is *unbounded below*. (Sometimes, we will abuse terminology and say that the problem is *unbounded*.) We finally note that there is no need to study maximization problems separately, because maximizing $c'x$ is equivalent to minimizing the linear cost function $-c'x$.

An equality constraint $a_i'x = b_i$ is equivalent to the two constraints $a_i'x \leq b_i$ and $a_i'x \geq b_i$. In addition, any constraint of the form $a_i'x \leq b_i$ can be rewritten as $(-a_i)'x \geq -b_i$. Finally, constraints of the form $x_j \geq 0$ or $x_j \leq 0$ are special cases of constraints of the form $a_i'x \geq b_i$, where a_i is a unit vector and $b_i = 0$. We conclude that the feasible set in a general linear programming problem can be expressed exclusively in terms of inequality constraints of the form $a_i'x \geq b_i$. Suppose that there is a total of m such constraints, indexed by $i = 1, \ldots, m$, let $b = (b_1, \ldots, b_m)$, and let A be the $m \times n$ matrix whose rows are the row vectors $a_1', \ldots, a_m'$, that is,

$$
A = \begin{bmatrix} - & a_1' & - \\ & \vdots & \\ - & a_m' & - \end{bmatrix}.
$$

Then, the constraints $a_i'x \geq b_i$, $i = 1, \ldots, m$, can be expressed compactly in the form $Ax \geq b$, and the linear programming problem can be written

as

$$\text{minimize} \quad c'x$$
$$\text{subject to} \quad Ax \geq b. \tag{1.2}$$

Inequalities such as $Ax \geq b$ will always be interpreted componentwise; that is, for every i, the ith component of the vector Ax, which is $a_i'x$, is greater than or equal to the ith component b_i of the vector b.

Example 1.2 The linear programming problem in Example 1.1 can be rewritten as

$$\text{minimize} \quad 2x_1 - x_2 + 4x_3$$
$$\text{subject to} \quad -x_1 - x_2 \qquad - x_4 \geq -2$$
$$3x_2 - x_3 \qquad \geq 5$$
$$-3x_2 + x_3 \qquad \geq -5$$
$$x_3 + x_4 \geq 3$$
$$x_1 \qquad \geq 0,$$
$$- x_3 \qquad \geq 0,$$

which is of the same form as the problem (1.2), with $c = (2, -1, 4, 0)$,

$$A = \begin{bmatrix} -1 & -1 & 0 & -1 \\ 0 & 3 & -1 & 0 \\ 0 & -3 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix},$$

and $b = (-2, 5, -5, 3, 0, 0)$.

Standard form problems

A linear programming problem of the form

$$\text{minimize} \quad c'x$$
$$\text{subject to} \quad Ax = b \tag{1.3}$$
$$x \geq 0,$$

is said to be in *standard form*. We provide an interpretation of problems in standard form. Suppose that x has dimension n and let $A_1, \ldots, A_n$ be the columns of A. Then, the constraint $Ax = b$ can be written in the form

$$\sum_{i=1}^{n} A_i x_i = b.$$

Intuitively, there are n available resource vectors $A_1, \ldots, A_n$, and a target vector b. We wish to "synthesize" the target vector b by using a nonnegative amount x_i of each resource vector A_i, while minimizing the cost $\sum_{i=1}^{n} c_i x_i$, where c_i is the unit cost of the ith resource. The following is a more concrete example.

Example 1.3 (The diet problem) Suppose that there are n different foods and m different nutrients, and that we are given the following table with the nutritional content of a unit of each food:

	food 1	$\cdots$	food n
nutrient 1	a_{11}	$\cdots$	a_{1n}
$\cdots$	$\cdots$		$\cdots$
nutrient m	a_{m1}	$\cdots$	a_{mn}

Let A be the $m \times n$ matrix with entries a_{ij}. Note that the jth column A_j of this matrix represents the nutritional content of the jth food. *Let b be a* vector with the requirements of an ideal diet or, equivalently, a specification of the nutritional contents of an "ideal food." We then interpret the standard form problem *as the problem of mixing nonnegative quantities x_i of the available foods, to synthesize the ideal food at minimal cost.* In a variant of this problem, the vector b specifies the *minimal requirements of an adequate diet*; in that case, the constraints $Ax = b$ are replaced by $Ax \geq b$, and the problem is not in standard form.

Reduction to standard form

As argued earlier, any linear programming problem, including the standard form problem (1.3), is a special case of the general form (1.1). We now argue that the converse is also true and that a general linear programming problem can be transformed into an equivalent problem in standard form. Here, when we say that the two problems are equivalent, *we mean that given a feasible solution to one problem, we can construct a feasible solution to the other, with the same cost.* In particular, the two problems have the same optimal cost and given an optimal solution to one problem, we can construct an optimal solution to the other. The problem transformation we have in mind involves two steps:

(a) *Elimination of free variables:* Given an unrestricted variable x_j in a problem in general form, we replace it by $x_j^+ - x_j^-$, where x_j^+ and x_j^- are new variables on which we impose the sign constraints $x_j^+ \geq 0$ and $x_j^- \geq 0$. The underlying idea is that any real number can be written as the difference of two nonnegative numbers.

(b) *Elimination of inequality constraints:* Given an inequality constraint of the form

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

we introduce a new variable s_i and the standard form constraints

$$\sum_{j=1}^{n} a_{ij} x_j + s_i = b_i,$$
$$s_i \ge 0.$$

Such a variable s_i is called a *slack* variable. Similarly, an inequality constraint $\sum_{j=1}^{n} a_{ij} x_j \ge b_i$ can be put in standard form by introducing a *surplus* variable s, and the constraints $\sum_{j=1}^{n} a_{ij} x_j - s_i = b_i$, $s_i \ge 0$.

We conclude that a general problem can be brought into standard form and, therefore, we only need to develop methods that are capable of solving standard form problems.

Example 1.4 The problem

minimize $2x_1 + 4x_2$

subject to $x_1 + x_2 \ge 3$

         $3x_1 + 2x_2 = 14$

         $x_1 \ge 0,$

is equivalent to the standard form problem

minimize $2x_1 + 4x_2^+ - 4x_2^-$

subject to $x_1 + x_2^+ - x_2^- - x_3 = 3$

         $3x_1 + 2x_2^+ - 2x_2^- = 14$

         $x_1, x_2^+, x_2^-, x_3 \ge 0.$

For example, given the feasible solution $(x_1, x_2) = (6, -2)$ to the original problem, we obtain the feasible solution $(x_1, x_2^+, x_2^-, x_3) = (6, 0, 2, 1)$ to the standard form problem, which has the same cost. Conversely, given the feasible solution $(x_1, x_2^+, x_2^-, x_3) = (8, 1, 6, 0)$ to the standard form problem, we obtain the feasible solution $(x_1, x_2) = (8, -5)$ to the original problem with the same cost.

In the sequel, we will often use the general form $Ax \ge b$ to develop the theory of linear programming. However, when it comes to algorithms, and especially the simplex and interior point methods, we will be focusing on the standard form $Ax = b$, $x \ge 0$, which is computationally more convenient.

1.2 Examples of linear programming problems

In this section, we discuss a number of examples of linear programming problems. One of our purposes is to indicate the vast range of situations to which linear programming can be applied. Another purpose is to develop some familiarity with the art of constructing mathematical formulations of loosely defined optimization problems.

A production problem

A firm produces n different goods using m different raw materials. Let b_i, $i = 1,\ldots,m$, be the available amount of the ith raw material. The jth good, $j = 1,\ldots,n$, requires a_{ij} units of the ith material and results in a revenue of c_j per unit produced. The firm faces the problem of deciding how much of each good to produce in order to maximize its total revenue.

In this example, the choice of the decision variables is simple. Let x_j, $j = 1,\ldots,n$, be the amount of the jth good. Then, the problem facing the firm can be formulated as follows:

maximize $c_1 x_1 + \cdots + c_n x_n$

subject to $a_{i1} x_1 + \cdots + a_{in} x_n \le b_i,$ $i = 1,\ldots,m,$

         $x_j \ge 0,$ $j = 1,\ldots,n.$

Production planning by a computer manufacturer

The example that we consider here is a problem that Digital Equipment Corporation (DEC) had faced in the fourth quarter of 1988. It illustrates the complexities and uncertainties of real world applications, as well as the usefulness of mathematical modeling for making important strategic decisions.

In the second quarter of 1988, DEC introduced a new family of (single CPU) computer systems and workstations: GP-1, GP-2, and GP-3, which are general purpose computer systems with different memory, disk storage, and expansion capabilities, as well as WS-1 and WS-2, which are workstations. In Table 1.1, we list the models, the list prices, the average disk usage per system, and the memory usage. For example, GP-1 uses four 256K memory boards, and 3 out of every 10 units are produced with a disk drive.

System	Price	# disk drives	# 256K boards
GP-1	$60,000	0.3	4
GP-2	$40,000	.7	2
GP-3	$30,000	0	2
WS-1	$30,000	1.4	2
WS-2	$15,000	0	1

Table 1.1: Features of the five different DEC systems.

Shipments of this new family of products started in the third quarter and ramped slowly during the fourth quarter. The following difficulties were anticipated for the next quarter:

(a) The in-house supplier of CPUs could provide at most 7,000 units, due to debugging problems.

(b) The supply of disk drives was uncertain and was estimated by the manufacturer to be in the range of 3,000 to 7,000 units.

(c) The supply of 256K memory boards was also limited in the range of 3,000 to 16,000 units.

On the demand side, the marketing department established that the maximum demand for the first quarter of 1989 would be 1,800 for GP-1 systems, 3,800 systems for the whole GP family, and 3,200 systems for the WS family. Included in these projections were 500 orders for GP-2, 500 orders for WS-1, and 400 orders for WS-2 that had already been received and had to be fulfilled in the next quarter.

In the previous quarters, in order to address the disk drive shortage, DEC had produced GP-1, GP-3, and WS-2 with no disk drive (although 3 out of 10 customers for GP-1 systems wanted a disk drive); and GP-2, WS-1 with one disk drive. We refer to this way of configuring the systems as the constrained mode of production.

In addition, DEC could address the shortage of 256K memory boards by using two alternative boards, instead of four 256K memory boards, in the GP-1 system. DEC could provide 4,000 alternative boards for the next quarter.

It was clear to the manufacturing staff that the problem had become complex, as revenue, profitability, and customer satisfaction were at risk. The following decisions needed to be made:

(a) The production plan for the first quarter of 1989.

(b) Concerning disk drive usage, should DEC continue to manufacture products in the constrained mode, or should it plan to satisfy customer preferences?

(c) Concerning memory boards, should DEC use alternative memory boards for its GP-1 systems?

(d) A final decision that had to be made was related to tradeoffs between shortages of disk drives and of 256K memory boards. The manufacturing staff would like to concentrate their efforts on either decreasing the shortage of disks or decreasing the shortage of 256K memory boards. Hence, they would like to know which alternative would have a larger effect on revenue.

In order to model the problem that DEC faced, we introduce variables x_1, x_2, x_3, x_4, x_5, that represent the number (in thousands) of GP-1, GP-2, GP-3, WS-1, and WS-2 systems, respectively, to be produced in the next quarter. Strictly speaking, since $1000x_i$ stands for number of units, it must be an integer. This can be accomplished by truncating each x_i after the third decimal point; given the size of the demand and the size of the

variables x_i, this has a negligible effect and the integrality constraint on $1000x_i$ can be ignored.

DEC had to make two distinct decisions: whether to use the constrained mode of production regarding disk drive usage, and whether to use alternative memory boards for the GP-1 system. As a result, there are four different combinations of possible choices.

We first develop a model for the case where alternative memory boards are not used and the constrained mode of production of disk drives is selected. The problem can be formulated as follows:

maximize $60x_1 + 40x_2 + 30x_3 + 30x_4 + 15x_5$ (total revenue)

subject to the following constraints

$$
\begin{aligned}
x_1 + x_2 + x_3 + x_4 + x_5 &\le 7 & &\text{(CPU availability)}\\
4x_1 + 2x_2 + 2x_3 + 2x_4 + x_5 &\le 8 & &\text{(256K availability)}\\
x_2 + x_4 &\le 3 & &\text{(disk drive availability)}\\
x_1 &\le 1.8 & &\text{(max demand for GP-1)}\\
x_3 &\le 0.3 & &\text{(max demand for GP-3)}\\
x_1 + x_2 + x_3 &\le 3.8 & &\text{(max demand for GP)}\\
x_4 + x_5 &\le 3.2 & &\text{(max demand for WS)}\\
x_2 &\ge 0.5 & &\text{(min demand for GP-2)}\\
x_4 &\ge 0.5 & &\text{(min demand for WS-1)}\\
x_5 &\ge 0.4 & &\text{(min demand for WS-2)}\\
x_1, x_2, x_3, x_4, x_5 &\ge 0. & &
\end{aligned}
$$

Notice that the objective function is in millions of dollars. In some respects, this is a pessimistic formulation, because the 256K memory and disk drive availability were set to 8 and 3, respectively, which is the lowest value in the range that was estimated. It is actually of interest to determine the solution to this problem as the 256K memory availability ranges from 8 to 16, and the disk drive availability ranges from 3 to 7, because this provides valuable information on the sensitivity of the optimal solution on availability. In another respect, the formulation is optimistic because, for example, it assumes that the revenue from GP-1 systems is $60x_1$ for any $x_1 \le 1.8$, even though a demand for 1,800 GP-1 systems is not guaranteed.

In order to accommodate the other three choices that DEC had, some of the problem constraints have to be modified, as follows. If we use the unconstrained mode of production for disk drives, the constraint $x_2 + x_4 \le 3$ is replaced by

$$0.3x_1 + 1.7x_2 + 1.4x_4 \le 3.$$

Furthermore, if we wish to use alternative memory boards in GP-1 systems, we replace the constraint $4x_1 + x_2 + 2x_3 + 2x_4 + x_5 \le 8$ by the two

constraints

$$2x_1 \le 4,$$
$$2x_2 + 2x_3 + 2x_4 + x_5 \le 8.$$

The four combinations of choices lead to four different linear programming problems, each of which needs to be solved for a variety of parameter values because, as discussed earlier, the right-hand side of some of the constraints is only known to lie within a certain range. Methods for solving linear programming problems, when certain parameters are allowed to vary, will be studied in Chapter 5, where this case study is revisited.

Multiperiod planning of electric power capacity

A state wants to plan its electricity capacity for the next T years. The state has a forecast of d_t megawatts, presumed accurate, of the demand for electricity during year $t = 1, \ldots, T$. The existing capacity, which is in oil-fired plants, that will not be retired and will be available during year t, is e_t. There are two alternatives for expanding electric capacity: coal-fired or nuclear power plants. There is a capital cost of c_t per megawatt of coal-fired capacity that becomes operational at the beginning of year t. The corresponding capital cost for nuclear power plants is n_t. For various political and safety reasons, it has been decided that no more than 20% of the total capacity should ever be nuclear. Coal plants last for 20 years, while nuclear plants last for 15 years. A least cost capacity expansion plan is desired.

The first step in formulating this problem as a linear programming problem is to define the decision variables. Let x_t and y_t be the amount of coal (respectively, nuclear) capacity brought on line at the beginning of year t. Let w_t and z_t be the total coal (respectively, nuclear) capacity available in year t. The cost of a capacity expansion plan is therefore,

$$\sum_{t=1}^{T} (c_t x_t + n_t y_t).$$

Since coal-fired plants last for 20 years, we have

$$w_t = \sum_{s=\max\{1, t-19\}}^{t} x_s, \quad t = 1, \ldots, T.$$

Similarly, for nuclear power plants,

$$z_t = \sum_{s=\max\{1, t-14\}}^{t} y_s, \quad t = 1, \ldots, T.$$

Since the available capacity must meet the forecasted demand, we require

$$w_t + z_t + e_t \ge d_t, \quad t = 1, \ldots, T.$$

Finally, since no more than 20% of the total capacity should ever be nuclear, we have

$$\frac{z_t}{w_t + z_t + e_t} \le 0.2,$$

which can be written as

$$0.8z_t - 0.2w_t \le 0.2e_t.$$

Summarizing, the capacity expansion problem is as follows:

$$\text{minimize} \quad \sum_{t=1}^{T} (c_t x_t + n_t y_t)$$

$$\text{subject to} \quad w_t - \sum_{s=\max\{1,t-19\}}^{t} x_s = 0, \qquad t = 1, \ldots, T,$$

$$z_t - \sum_{t=\max\{1,t-14\}}^{t} y_s = 0, \qquad t = 1, \ldots, T,$$

$$w_t + z_t \ge d_t - e_t, \qquad t = 1, \ldots, T,$$

$$0.8z_t - 0.2w_t \le 0.2e_t, \qquad t = 1, \ldots, T,$$

$$x_t, y_t, w_t, z_t \ge 0, \qquad t = 1, \ldots, T.$$

We note that this formulation is not entirely realistic, because it disregards certain economies of scale that may favor larger plants. However, it can provide a ballpark estimate of the true cost.

A scheduling problem

In the previous examples, the choice of the decision variables was fairly straightforward. We now discuss an example where this choice is less obvious.

A hospital wants to make a weekly night shift (12pm-8am) schedule for its nurses. The demand for nurses for the night shift on day j is an integer d_j, $j = 1, \ldots, 7$. Every nurse works 5 days in a row on the night shift. The problem is to find the minimal number of nurses the hospital needs to hire.

One could try using a decision variable y_j equal to the number of nurses that work on day j. With this definition, however, we would not be able to capture the constraint that every nurse works 5 days in a row. For this reason, we choose the decision variables differently, and define x_j as

the number of nurses starting their week on day j. (For example, a nurse whose week starts on day 5 will work days 5, 6, 7, 1, 2.) We then have the following problem formulation:

minimize $\quad x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7$

subject to
$$
\begin{aligned}
x_1 &\;\;+\; x_4 + x_5 + x_6 + x_7 \geq d_1 \\
x_1 + x_2 &\;\;+\; x_5 + x_6 + x_7 \geq d_2 \\
x_1 + x_2 + x_3 &\;\;+\; x_6 + x_7 \geq d_3 \\
x_1 + x_2 + x_3 + x_4 &\;\;+\; x_7 \geq d_4 \\
x_1 + x_2 + x_3 + x_4 + x_5 &\;\;\geq d_5 \\
x_2 + x_3 + x_4 + x_5 + x_6 &\;\;\geq d_6 \\
x_3 + x_4 + x_5 + x_6 + x_7 &\;\;\geq d_7 \\
x_j \geq 0, \quad x_j \text{ integer}
\end{aligned}
$$

This would be a linear programming problem, except for the constraint that each x_j must be an integer, and we actually have a linear *integer programming* problem. One way of dealing with this issue is to ignore ("relax") the integrality constraints and obtain the so-called *linear programming relaxation* of the original problem. Because the linear programming problem has fewer constraints, and therefore more options, the optimal cost will be less than or equal to the optimal cost of the original problem. If the optimal solution to the linear programming relaxation happens to be integer, then it is also an optimal solution to the original problem. If it is not integer, we can round each x_j upwards, thus obtaining a feasible, but not necessarily optimal, solution to the original problem. It turns out that for this particular problem, an optimal solution can be found without too much effort. However, this is the exception rather than the rule: finding optimal solutions to general integer programming problems is typically difficult; some methods will be discussed in Chapter 11.

Choosing paths in a communication network

Consider a communication network consisting of n nodes. Nodes are connected by communication links. A link allowing one-way transmission from node i to node j is described by an ordered pair (i,j). Let $\mathcal{A}$ be the set of all links. We assume that each link $(i,j) \in \mathcal{A}$ can carry up to u_{ij} bits per second. There is a positive charge c_{ij} per bit transmitted along that link. Each node k generates data, at the rate of $b^{k\ell}$ bits per second, that have to be transmitted to node ℓ, either through a direct link (k,ℓ) or by tracing a sequence of links. The problem is to choose paths along which all data reach their intended destinations, while minimizing the total cost. We allow the data with the same origin and destination to be split and be transmitted along different paths.

In order to formulate this problem as a linear programming problem, we introduce variables $x_{ij}^{k\ell}$ indicating the amount of data with origin k and

destination ℓ that traverse link (i,j). Let

$$
b_i^{k\ell} = \begin{cases} b^{k\ell} & \text{if } i = k, \\ -b^{k\ell} & \text{if } i = \ell, \\ 0 & \text{otherwise.} \end{cases}
$$

Thus, $b_i^{k\ell}$ is the net inflow at node i, from outside the network, of data with origin k and destination ℓ. We then have the following formulation:

minimize $\quad \displaystyle\sum_{(i,j)\in\mathcal{A}} \sum_{k=1}^{n} \sum_{\ell=1}^{n} c_{ij} x_{ij}^{k\ell}$

subject to $\quad \displaystyle\sum_{\{j|(i,j)\in\mathcal{A}\}} x_{ij}^{k\ell} - \sum_{\{j|(j,i)\in\mathcal{A}\}} x_{ji}^{k\ell} = b_i^{k\ell}, \qquad i,k,\ell = 1,\ldots,n,$

$$\sum_{k=1}^{n} \sum_{\ell=1}^{n} x_{ij}^{k\ell} \leq u_{ij}, \qquad (i,j)\in\mathcal{A},$$

$$x_{ij}^{k\ell} \geq 0, \qquad (i,j)\in\mathcal{A}, \quad k,\ell = 1,\ldots,n.$$

The first constraint is a flow conservation constraint at node i for data with origin k and destination ℓ. The expression

$$\sum_{\{j|(i,j)\in\mathcal{A}\}} x_{ij}^{k\ell}$$

represents the amount of data with origin and destination k and ℓ, respectively, that leave node i along some link. The expression

$$\sum_{\{j|(j,i)\in\mathcal{A}\}} x_{ji}^{k\ell}$$

represents the amount of data with the same origin and destination that enter node i through some link. Finally, $b_i^{k\ell}$ is the net amount of such data that enter node i from outside the network. The second constraint expresses the requirement that the total traffic through a link (i,j) cannot exceed the link's capacity.

This problem is known as the *multicommodity flow* problem, with the traffic corresponding to each origin-destination pair viewed as a different commodity. A mathematically similar problem arises when we consider a transportation company that wishes to transport several commodities from their origins to their destinations through a network. There is a version of this problem, known as the minimum cost *network flow* problem, in which we do not distinguish between different commodities. Instead, we are given the amount b_i of external supply or demand at each node i, and the objective is to transport material from the supply nodes to the demand nodes, at minimum cost. The network flow problem, which is the subject of Chapter 7, contains as special cases some important problems such as the shortest path problem, the maximum flow problem, and the assignment problem.

Pattern classification

We are given m examples of objects and for each one, say the ith one, a description of its features in terms of an n-dimensional vector a_i. Objects belong to one of two classes, and for each example we are told the class that it belongs to.

More concretely, suppose that each object is an image of an apple or an orange (these are our two classes). In this context, we can use a three-dimensional feature vector a_i to summarize the contents of the ith image. The three components of a_i (the features) could be the ellipticity of the object, the length of its stem, and its color, as measured in some scale. We are interested in designing a *classifier* which, given a new object (other than the originally available examples), will figure out whether it is an image of an apple or of an orange.

A *linear classifier* is defined in terms of an n-dimensional vector x and a scalar x_{n+1}, and operates as follows. Given a new object with feature vector a, the classifier declares it to be an object of the first class if

$$a'x \geq x_{n+1},$$

and of the second class if

$$a'x < x_{n+1}.$$

In words, a linear classifier makes decisions on the basis of a linear combination of the different features. Our objective is to use the available examples in order to design a "good" linear classifier.

There are many ways of approaching this problem, but a reasonable starting point could be the requirement that the classifier must give the correct answer for each one of the available examples. Let S be the set of examples of the first class. We are then looking for some x and x_{n+1} that satisfy the constraints

$$a_i'x \geq x_{n+1}, \qquad i \in S,$$
$$a_i'x < x_{n+1}, \qquad i \notin S.$$

Note that the second set of constraints involves a strict inequality and is not quite of the form arising in linear programming. This issue can be bypassed by observing that if some choice of x and x_{n+1} satisfies all of the above constraints, then there exists some other choice (obtained by multiplying x and x_{n+1} by a suitably large positive scalar) that satisfies

$$a_i'x \geq x_{n+1}, \qquad i \in S,$$
$$a_i'x \leq x_{n+1} - 1, \qquad i \notin S.$$

We conclude that the search for a linear classifier consistent with all available examples is a problem of finding a feasible solution to a linear programming problem.

1.3 Piecewise linear convex objective functions

All of the examples in the preceding section involved a *linear* objective function. However, there is an important class of optimization problems with a nonlinear objective function that can be cast as linear programming problems; these are examined next.

We first need some definitions:

Definition 1.1

(a) *A function $f: \Re^n \mapsto \Re$ is called convex if for every $x, y \in \Re^n$, and every $\lambda \in [0,1]$, we have*

$$f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1-\lambda)f(y).$$

(b) *A function $f: \Re^n \mapsto \Re$ is called concave if for every $x, y \in \Re^n$, and every $\lambda \in [0,1]$, we have*

$$f(\lambda x + (1 - \lambda)y) \geq \lambda f(x) + (1-\lambda)f(y).$$

Note that if x and y are vectors in $\Re^n$ and if λ ranges in $[0,1]$, then points of the form $\lambda x + (1 - \lambda)y$ belong to the line segment joining x and y. The definition of a convex function refers to the values of f, as its argument traces this segment. If f were linear, the inequality in part (a) of the definition would hold with equality. The inequality therefore means that when we restrict attention to such a segment, the graph of the function lies no higher than the graph of a corresponding linear function; see Figure 1.1(a).

It is easily seen that a function f is convex if and only if the function $-f$ is concave. Note that a function of the form $f(x) = a_0 + \sum_{i=1}^n a_i x_i$, where $a_0, \ldots, a_n$ are scalars, called an *affine* function, is both convex and concave. (It turns out that affine functions are the only functions that are both convex and concave.) Convex (as well as concave) functions play a central role in optimization.

We say that a vector x is a *local minimum* of f if $f(x) \leq f(y)$ for all y in the vicinity of x. We also say that x is a *global minimum* if $f(x) \leq f(y)$ for all y. A convex function cannot have local minima that fail to be global minima (see Figure 1.1), and this property is of great help in designing efficient optimization algorithms.

Let $c_1, \ldots, c_m$ be vectors in $\Re^n$, let $d_1, \ldots, d_m$ be scalars, and consider the function $f: \Re^n \mapsto \Re$ defined by

$$f(x) = \max_{i=1,\ldots,m} (c_i'x + d_i).$$
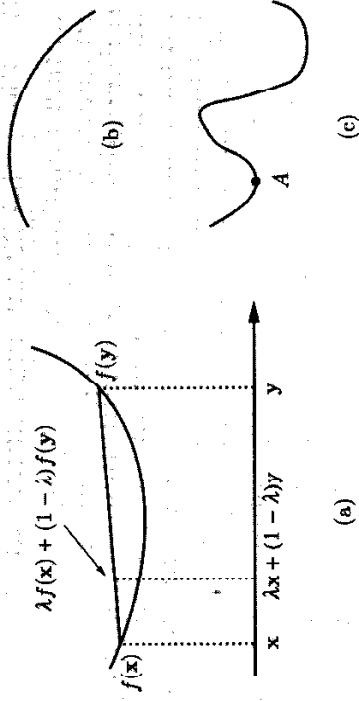
Figure 1.1: (a) Illustration of the definition of a convex function. (b) A concave function. (c) A function that is neither convex nor concave; note that A is a local, but not global, minimum.

[see Figure 1.2(a)]. Such a function is convex, as a consequence of the following result.

Theorem 1.1 *Let $f_1,\ldots,f_m : \Re^n \mapsto \Re$ be convex functions. Then, the function f defined by $f(\mathbf{x}) = \max_{i=1,\ldots,m} f_i(\mathbf{x})$ is also convex.*

Proof. Let $\mathbf{x}, \mathbf{y} \in \Re^n$ and let $\lambda \in [0,1]$. We have

$$f(\lambda \mathbf{x} + (1-\lambda)\mathbf{y}) = \max_{i=1,\ldots,m} f_i(\lambda \mathbf{x} + (1-\lambda)\mathbf{y})$$
$$\leq \max_{i=1,\ldots,m} \big(\lambda f_i(\mathbf{x}) + (1-\lambda)f_i(\mathbf{y})\big)$$
$$\leq \max_{i=1,\ldots,m} \lambda f_i(\mathbf{x}) + \max_{i=1,\ldots,m}(1 - \lambda)f_i(\mathbf{y})$$
$$= \lambda f(\mathbf{x}) + (1-\lambda)f(\mathbf{y}).$$

$\square$

A function of the form $\max_{i=1,\ldots,m}(\mathbf{c}_i'\mathbf{x}+d_i)$ is called a *piecewise linear convex* function. A simple example is the absolute value function defined by $f(x) = |x| = \max\{x, -x\}$. As illustrated in Figure 1.2(b), a piecewise linear convex function can be used to approximate a general convex function.

We now consider a generalization of linear programming, where the objective function is piecewise linear and convex rather than linear:

$$\text{minimize} \quad \max_{i=1,\ldots,m}(\mathbf{c}_i'\mathbf{x}+d_i)$$
$$\text{subject to} \quad \mathbf{Ax} \geq \mathbf{b}.$$
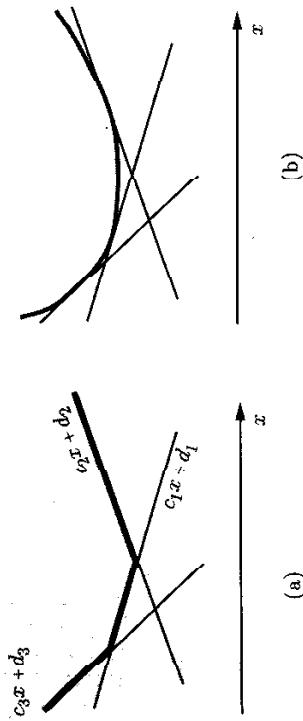
Figure 1.2: (a) A piecewise linear convex function of a single variable. (b) An approximation of a convex function by a piecewise linear convex function.

Note that $\max_{i=1,\ldots,m}(\mathbf{c}_i'\mathbf{x} + d_i)$ is equal to the smallest number z that satisfies $z \geq \mathbf{c}_i'\mathbf{x} + d_i$ for all i. For this reason, the optimization problem under consideration is equivalent to the linear programming problem

$$\text{minimize} \quad z$$
$$\text{subject to} \quad z \geq \mathbf{c}_i'\mathbf{x} + d_i, \quad i = 1,\ldots,m,$$
$$\mathbf{Ax} \geq \mathbf{b},$$

where the decision variables are z and $\mathbf{x}$.

To summarize, linear programming can be used to solve problems with piecewise linear convex cost functions, and the latter class of functions can be used as an approximation of more general convex cost functions. On the other hand, such a piecewise linear approximation is not always a good idea because it can turn a smooth function into a nonsmooth one ,piecewise linear functions have discontinuous derivatives).

We finally note that if we are given a constraint of the form $f(\mathbf{x}) \leq h$, where f is the piecewise linear convex function $f(\mathbf{x}) = \max_{i=1,\ldots,m}(\mathbf{f}_i'\mathbf{x}+g_i)$, such a constraint can be rewritten as

$$\mathbf{f}_i'\mathbf{x} + g_i \leq h, \quad i = 1,\ldots,m,$$

and linear programming is again applicable.

Problems involving absolute values

Consider a problem of the form

$$\text{minimize} \quad \sum_{i=1}^{n} c_i|x_i|$$
$$\text{subject to} \quad \mathbf{Ax} \geq \mathbf{b},$$

where $\mathbf{x} = (x_1, \ldots, x_n)$, and where the cost coefficients c_i are assumed to be nonnegative. The cost criterion, being the sum of the piecewise linear convex functions $c_i|x_i|$ is easily shown to be piecewise linear and convex (Exercise 1.2). However, expressing this cost criterion in the form $\max_j(\mathbf{c}'_j\mathbf{x} + d_j)$ is a bit involved, and a more direct route is preferable. We observe that $|x_i|$ is the smallest number z_i that satisfies $x_i \le z_i$ and $-x_i \le z_i$, and we obtain the linear programming formulation

$$\text{minimize} \quad \sum_{i=1}^{n} c_i z_i$$

$$\text{subject to} \quad \mathbf{Ax} \ge \mathbf{b}$$
$$x_i \le z_i, \qquad i = 1, \ldots, n,$$
$$-x_i \le z_i, \qquad i = 1, \ldots, n.$$

An alternative method for dealing with absolute values is to introduce new variables x_i^+, x_i^-, constrained to be nonnegative, and let $x_i = x_i^+ - x_i^-$. (Our intention is to have $x_i = x_i^+$ or $x_i = -x_i^-$, depending on whether x_i is positive or negative.) We then replace every occurrence of $|x_i|$ with $x_i^+ + x_i^-$ and obtain the alternative formulation

$$\text{minimize} \quad \sum_{i=1}^{n} c_i(x_i^+ + x_i^-)$$

$$\text{subject to} \quad \mathbf{Ax}^+ - \mathbf{Ax}^- \ge \mathbf{b}$$
$$\mathbf{x}^+, \mathbf{x}^- \ge 0,$$

where $\mathbf{x}^+ = (x_1^+, \ldots, x_n^+)$ and $\mathbf{x}^- = (x_1^-, \ldots, x_n^-)$.

The relations $x_i = x_i^+ - x_i^-$, $x_i^+ \ge 0$, $x_i^- \ge 0$, are not enough to guarantee that $|x_i| = x_i^+ + x_i^-$, and the validity of this reformulation may not be entirely obvious. Let us assume for simplicity that $c_i > 0$ for all i. At an optimal solution to the reformulated problem, and for each i, we must have either $x_i^+ = 0$ or $x_i^- = 0$, because otherwise we could reduce both x_i^+ and x_i^- by the same amount and preserve feasibility, while reducing the cost, in contradiction of optimality. Having guaranteed that either $x_i^+ = 0$ or $x_i^- = 0$, the desired relation $|x_i| = x_i^+ + x_i^-$ now follows.

The formal correctness of the two reformulations that have been presented here, and in a somewhat more general setting, is the subject of Exercise 1.5. We also note that the nonnegativity assumption on the cost coefficients c_i is crucial because, otherwise, the cost criterion is nonconvex.

Example 1.5 Consider the problem

$$\text{minimize} \quad 2|x_1| + x_2$$
$$\text{subject to} \quad x_1 + x_2 \ge 4.$$

Our first reformulation yields

$$\text{minimize} \quad 2z_1 + x_2$$
$$\text{subject to} \quad x_1 + x_2 \ge 4$$
$$x_1 \le z_1$$
$$-x_1 \le z_1,$$

while the second yields

$$\text{minimize} \quad 2x_1^+ + 2x_1^- + x_2$$
$$\text{subject to} \quad x_1^+ - x_1^- + x_2 \ge 4$$
$$x_1^+ \ge 0$$
$$x_1^- \ge 0.$$

We now continue with some applications involving piecewise linear convex objective functions.

Data fitting

We are given m data points of the form $(\mathbf{a}_i, b_i)$, $i = 1, \ldots, m$, where $\mathbf{a}_i \in \Re^n$ and $b_i \in \Re$, and wish to build a model that predicts the value of the variable b from knowledge of the vector $\mathbf{a}$. In such a situation, one often uses a linear model of the form $b = \mathbf{a}'\mathbf{x}$, where $\mathbf{x}$ is a parameter vector to be determined. Given a particular parameter vector $\mathbf{x}$, the *residual*, or prediction error, at the ith data point is defined as $|b_i - \mathbf{a}'_i\mathbf{x}|$. Given a choice between alternative models, one should choose a model that "explains" the available data as best as possible, i.e., a model that results in small residuals. One possibility is to minimize the largest residual. This is the problem of minimizing

$$\max_i |b_i - \mathbf{a}'_i\mathbf{x}|,$$

with respect to $\mathbf{x}$, subject to no constraints. Note that we are dealing here with a piecewise linear convex cost criterion. The following is an equivalent linear programming formulation:

$$\text{minimize} \quad z$$
$$\text{subject to} \quad b_i - \mathbf{a}'_i\mathbf{x} \le z, \qquad i = 1, \ldots, m,$$
$$-b_i + \mathbf{a}'_i\mathbf{x} \le z, \qquad i = 1, \ldots, m,$$

the decision variables being z and $\mathbf{x}$.

In an alternative formulation, we could adopt the cost criterion

$$\sum_{i=1}^{m} |b_i - \mathbf{a}'_i\mathbf{x}|.$$

Since $|b_i - a_i'\mathbf{x}|$ is the smallest number z_i that satisfies $b_i - a_i'\mathbf{x} \le z_i$ and $-b_i + a_i'\mathbf{x} \le z_i$, we obtain the formulation

minimize $\quad z_1 + \cdots + z_m$

subject to $\quad b_i - a_i'\mathbf{x} \le z_i, \qquad i = 1, \ldots, m,$

$\qquad\qquad -b_i + a_i'\mathbf{x} \le z_i, \qquad i = 1, \ldots, m.$

In practice, one may wish to use the quadratic cost criterion $\sum_{i=1}^m (b_i - a_i'\mathbf{x})^2$, in order to obtain a "least squares fit." This is a problem which is easier than linear programming; it can be solved using calculus methods, but its discussion is outside the scope of this book.

Optimal control of linear systems

Consider a dynamical system that evolves according to a model of the form

$$\mathbf{x}(t+1) = \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t),$$
$$y(t) = \mathbf{c}'\mathbf{x}(t).$$

Here $\mathbf{x}(t)$ is the state of the system at time t, $y(t)$ is the system output, assumed scalar, and $\mathbf{u}(t)$ is a control vector that we are free to choose subject to linear constraints of the form $\mathbf{D}\mathbf{u}(t) \le \mathbf{d}$ [these might include saturation constraints, i.e., hard bounds on the magnitude of each component of $\mathbf{u}(t)$]. To mention some possible applications, this could be a model of an airplane, an engine, an electrical circuit, a mechanical system, a manufacturing system, or even a model of economic growth. We are also given the initial state $\mathbf{x}(0)$. In one possible problem, we are to choose the values of the control variables $\mathbf{u}(0), \ldots, \mathbf{u}(T-1)$ to drive the state $\mathbf{x}(T)$ to a target state, assumed for simplicity to be zero. In addition to zeroing the state, it is often desirable to keep the magnitude of the output small at all intermediate times, and we may wish to minimize

$$\max_{t=1,\ldots,T-1} |y(t)|.$$

We then obtain the following linear programming problem:

minimize $\quad z$

subject to $\quad -z \le y(t) \le z,$ $\qquad\qquad t = 1, \ldots, T-1,$

$\qquad\qquad \mathbf{x}(t+1) = \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t), \qquad t = 0, \ldots, T-1,$

$\qquad\qquad y(t) = \mathbf{c}'\mathbf{x}(t), \qquad\qquad t = 1, \ldots, T-1,$

$\qquad\qquad \mathbf{D}\mathbf{u}(t) \le \mathbf{d}, \qquad\qquad t = 0, \ldots, T-1,$

$\qquad\qquad \mathbf{x}(T) = \mathbf{0},$

$\qquad\qquad \mathbf{x}(0) = \text{given.}$

Additional linear constraints on the state vectors $\mathbf{x}(t)$, or a more general piecewise linear convex cost function of the state and the control, can also be incorporated.

Rocket control

Consider a rocket that travels along a straight path. Let x_t, v_t, and a_t be the position, velocity, and acceleration, respectively, of the rocket at time t. By discretizing time and by taking the time increment to be unity, we obtain an approximate discrete-time model of the form

$$x_{t+1} = x_t + v_t,$$
$$v_{t+1} = v_t + a_t.$$

We assume that the acceleration a_t is under our control, as it is determined by the rocket thrust. In a rough model, the magnitude $|a_t|$ of the acceleration can be assumed to be proportional to the rate of fuel consumption at time t.

Suppose that the rocket is initially at rest at the origin, that is, $x_0 = 0$ and $v_0 = 0$. We wish the rocket to take off and "land softly" at unit distance from the origin after T time units, that is, $x_T = 1$ and $v_T = 0$. Furthermore, we wish to accomplish this in an economical fashion. One possibility is to minimize the total fuel $\sum_{t=0}^{T-1} |a_t|$ spent subject to the preceding constraints. Alternatively, we may wish to minimize the maximum thrust required, which is $\max_t |a_t|$. Under either alternative, the problem can be formulated as a linear programming problem (Exercise 1.6).

1.4 Graphical representation and solution

In this section, we consider a few simple examples that provide useful geometric insights into the nature of linear programming problems. Our first example involves the graphical solution of a linear programming problem with two variables.

Example 1.6 Consider the problem

minimize $\quad -x_1 - x_2$

subject to $\quad x_1 + 2x_2 \le 3$

$\qquad\qquad 2x_1 + x_2 \le 3$

$\qquad\qquad x_1, x_2 \ge 0.$

The feasible set is the shaded region in Figure 1.3. In order to find an optimal solution, we proceed as follows. For any given scalar z, we consider the set of all points whose cost $\mathbf{c}'\mathbf{x}$ is equal to z; this is the line described by the equation $-x_1 - x_2 = z$. Note that this line is perpendicular to the vector $\mathbf{c} = (-1, -1)$. Different values of z lead to different lines, all of them parallel to each other. In
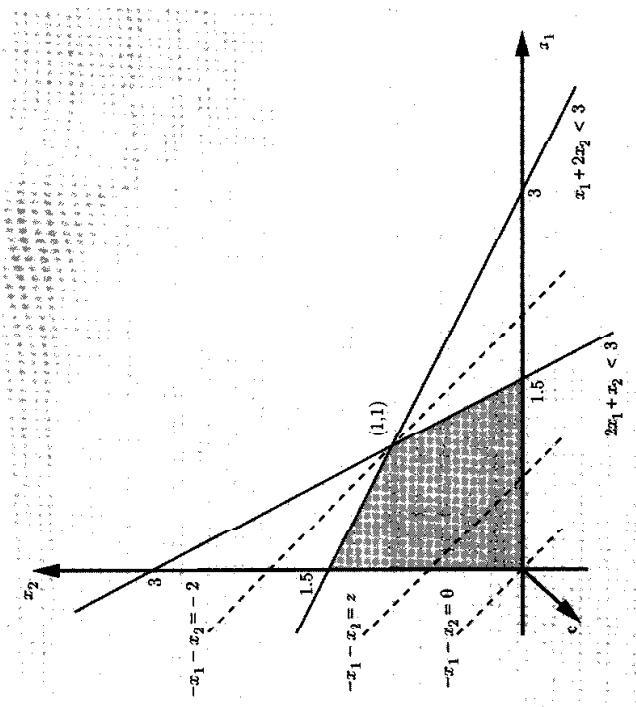
Figure 1.3: Graphical solution of the problem in Example 1.6.

particular, increasing z corresponds to moving the line $z = -x_1 - x_2$ along the direction of the vector c. Since we are interested in minimizing z, we would like to move the line as much as possible in the direction of $-c$, as long as we do not leave the feasible region. The best we can do is $z = -2$ (see Figure 1.3), and the vector $x = (1, 1)$ is an optimal solution. Note that this is a corner of the feasible set. (The concept of a "corner" will be defined formally in Chapter 2.)

For a problem in three dimensions, the same approach can be used except that the set of points with the same value of $c'x$ is a plane, instead of a line. This plane is again perpendicular to the vector c, and the objective is to slide this plane as much as possible in the direction of $-c$, as long as we do not leave the feasible set.

Example 1.7 Suppose that the feasible set is the unit cube, described by the constraints $0 \leq x_i \leq 1$, $i = 1, 2, 3$, and that $c = (-1, -1, -1)$. Then, the vector $x = (1, 1, 1)$ is an optimal solution. Once more, the optimal solution happens to be a corner of the feasible set (Figure 1.4).
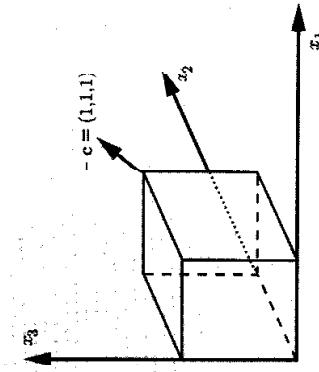
Figure 1.4: The three-dimensional linear programming problem in Example 1.7.

In both of the preceding examples, the feasible set is bounded (does not extend to infinity), and the problem has a unique optimal solution. This is not always the case and we have some additional possibilities that are illustrated by the example that follows.

Example 1.8 Consider the feasible set in $\Re^2$ defined by the constraints

$$-x_1 + x_2 \leq 1$$
$$x_1 \geq 0$$
$$x_2 \geq 0,$$

which is shown in Figure 1.5.

(a) For the cost vector $c = (1, 1)$, it is clear that $x = (0, 0)$ is the unique optimal solution.

(b) For the cost vector $c = (1, 0)$, there are multiple optimal solutions, namely, every vector x of the form $x = (0, x_2)$, with $0 \leq x_2 \leq 1$, is optimal. Note that the set of optimal solutions is bounded.

(c) For the cost vector $c = (0, 1)$, there are multiple optimal solutions, namely, every vector x of the form $x = (x_1, 0)$, with $x_1 \geq 0$, is optimal. In this case, the set of optimal solutions is unbounded (contains vectors of arbitrarily large magnitude).

(d) Consider the cost vector $c = (-1, -1)$. For any feasible solution (x_1, x_2), we can always produce another feasible solution with less cost, by increasing the value of x_1. Therefore, no feasible solution is optimal. Furthermore, by considering vectors (x_1, x_2) with ever increasing values of x_1 and x_2, we can obtain a sequence of feasible solutions whose cost converges to $-\infty$. We therefore say that the optimal cost is $-\infty$.

(e) If we impose an additional constraint of the form $x_1 + x_2 \leq -2$, it is evident that no feasible solution exists.
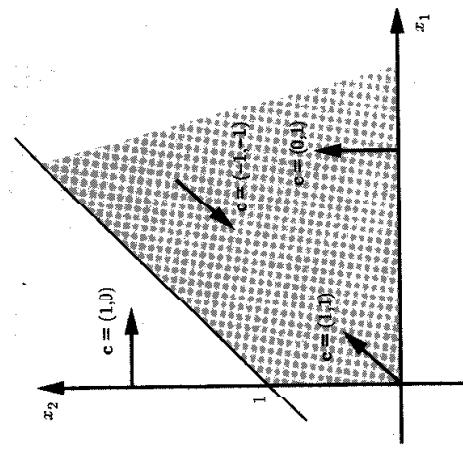
Figure 1.5: The feasible set in Example 1.8. For each choice of **c**, an optimal solution is obtained by moving as much as possible in the direction of $-\mathbf{c}$.

To summarize the insights obtained from Example 1.8, we have the following possibilities:

(a) There exists a unique optimal solution.

(b) There exist multiple optimal solutions; in this case, the set of optimal solutions can be either bounded or unbounded.

(c) The optimal cost is $-\infty$, and no feasible solution is optimal.

(d) The feasible set is empty.

In principle, there is an additional possibility: an optimal solution does not exist even though the problem is feasible and the optimal cost is not $-\infty$; this is the case, for example, in the problem of minimizing $1/x$ subject to $x > 0$ (for every feasible solution, there exists another with less cost, but the optimal cost is not $-\infty$). We will see later in this book that this possibility never arises in linear programming.
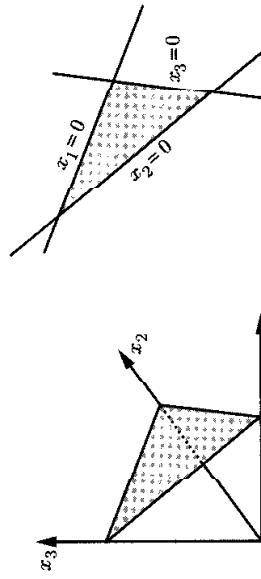
In the examples that we have considered, if the problem has at least one optimal solution, then an optimal solution can be found among the corners of the feasible set. In Chapter 2, we will show that this is a general feature of linear programming problems, as long as the feasible set has at least one corner.

Visualizing standard form problems
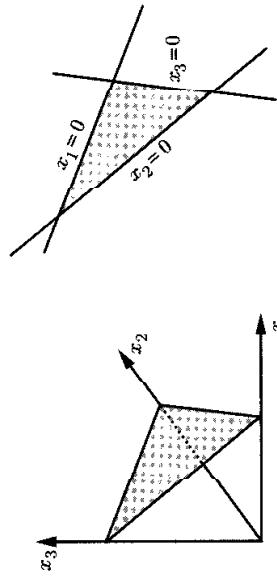
We now discuss a method that allows us to visualize standard form problems even if the dimension n of the vector **x** is greater than three. The reason for wishing to do so is that when $n \leq 3$, the feasible set of a standard form problem does not have much variety and does not provide enough insight into the general case. (In contrast, if the feasible set is described by constraints of the form $\mathbf{Ax} \geq \mathbf{b}$, enough variety is obtained even if **x** has dimension three.)

Suppose that we have a standard form problem, and that the matrix **A** has dimensions $m \times n$. In particular, the decision vector **x** is of dimension n and we have m equality constraints. We assume that $m \leq n$ and that the constraints $\mathbf{Ax} = \mathbf{b}$ force **x** to lie on an $(n - m)$-dimensional set. (Intuitively, each constraint removes one of the "degrees of freedom" of **x**.) If we "stand" on that $(n-m)$-dimensional set and ignore the m dimensions orthogonal to it, the feasible set is only constrained by the linear inequality constraints $x_i \geq 0$, $i = 1,\ldots,n$. In particular, if $n - m = 2$, the feasible set can be drawn as a two-dimensional set defined by n linear inequality constraints.

To illustrate this approach, consider the feasible set in $\Re^3$ defined by the constraints $x_1+x_2+x_3 = 1$ and $x_1,x_2,x_3 \geq 0$ [Figure 1.6(a)], and note that $n = 3$ and $m = 1$. If we stand on the plane defined by the constraint $x_1 + x_2 + x_3 = 1$, then the feasible set has the appearance of a triangle in two-dimensional space. Furthermore, each edge of the triangle corresponds to one of the constraints $x_1,x_2,x_3 \geq 0$; see Figure 1.6(b).

Figure 1.6: (a) An n-dimensional view of the feasible set. (b) An $(n - m)$-dimensional view of the same set.

1.5   Linear algebra background and notation

This section provides a summary of the main notational conventions that we will be employing. It also contains a brief review of those results from linear algebra that are used in the sequel.

Set theoretic notation

If S is a set and x is an element of S, we write $x \in S$. A set can be specified in the form $S = \{x \mid x \text{ satisfies } P\}$, as the set of all elements having property P. The cardinality of a finite set S is denoted by $|S|$. The union of two sets S and T is denoted by $S \cup T$, and their intersection by $S \cap T$. We use $S \setminus T$ to denote the set of all elements of S that do not belong to T. The notation $S \subset T$ means that S is a subset of T, i.e., every element of S is also an element of T; in particular, S could be equal to T. If in addition $S \neq T$, we say that S is a *proper* subset of T. We use $\emptyset$ to denote the empty set. The symbols $\exists$ and $\forall$ have the meanings "there exists" and "for all," respectively.

We use $\Re$ to denote the set of real numbers. For any real numbers a and b, we define the closed and open intervals $[a, b]$ and (a, b), respectively, by

$$[a, b] = \{x \in \Re \mid a \leq x \leq b\},$$

and

$$(a, b) = \{x \in \Re \mid a < x < b\}.$$

Vectors and matrices

A *matrix* of dimensions $m \times n$ is an array of real numbers a_{ij}:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & & a_{2n} \\ \vdots & & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}.$$

Matrices will be always denoted by upper case boldface characters. If $\mathbf{A}$ is a matrix, we use the notation a_{ij} or $[\mathbf{A}]_{ij}$ to refer to its (i, j)th entry. A *row vector* is a matrix with $m = 1$ and a *column vector* is a matrix with $n = 1$. The word *vector* will always mean *column vector* unless the contrary is explicitly stated. Vectors will be usually denoted by lower case boldface characters. We use the notation $\Re^n$ to indicate the set of all n-dimensional vectors. For any vector $\mathbf{x} \in \Re^n$, we use $x_1, x_2, \ldots, x_n$ to

indicate its components. Thus,

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}.$$

The more economical notation $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ will also be used even if we are referring to column vectors. We use $\mathbf{0}$ to denote the vector with all components equal to zero. The ith *unit vector* $\mathbf{e}_i$ is the vector with all components equal to zero except for the ith component which is equal to one.

The *transpose* $\mathbf{A}'$ of an $m \times n$ matrix $\mathbf{A}$ is the $n \times m$ matrix

$$\mathbf{A}' = \begin{bmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{12} & a_{22} & & a_{m2} \\ \vdots & \vdots & & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{mn} \end{bmatrix};$$

that is, $[\mathbf{A}']_{ij} = [\mathbf{A}]_{ji}$. Similarly, if $\mathbf{x}$ is a vector in $\Re^n$, its transpose $\mathbf{x}'$ is the row vector with the same entries.

If $\mathbf{x}$ and $\mathbf{y}$ are two vectors in $\Re^n$, then

$$\mathbf{x}'\mathbf{y} = \mathbf{y}'\mathbf{x} = \sum_{i=1}^n x_i y_i.$$

This quantity is called the *inner product* of $\mathbf{x}$ and $\mathbf{y}$. Two vectors are called *orthogonal* if their inner product is zero. Note that $\mathbf{x}'\mathbf{x} \geq 0$ for every vector $\mathbf{x}$, with equality holding if and only if $\mathbf{x} = \mathbf{0}$. The expression $\sqrt{\mathbf{x}'\mathbf{x}}$ is the *Euclidean norm* of $\mathbf{x}$ and is denoted by $\|\mathbf{x}\|$. The *Schwartz inequality* asserts that for any two vectors of the same dimension, we have

$$|\mathbf{x}'\mathbf{y}| \leq \|\mathbf{x}\| \cdot \|\mathbf{y}\|,$$

with equality holding if and only if one of the two vectors is a scalar multiple of the other.

If $\mathbf{A}$ is an $m \times n$ matrix, we use $\mathbf{A}_j$ to denote its jth column, that is, $\mathbf{A}_j = (a_{1j}, a_{2j}, \ldots, a_{mj})$. (This is our only exception to the rule of using lower case characters to represent vectors.) We also use $\mathbf{a}_i$ to denote the vector formed by the entries of the ith row, that is, $\mathbf{a}_i = (a_{i1}, a_{i2}, \ldots, a_{in})$. Thus,

$$\mathbf{A} = \begin{bmatrix} | & | & & | \\ \mathbf{A}_1 & \mathbf{A}_2 & \cdots & \mathbf{A}_n \\ | & | & & | \end{bmatrix} = \begin{bmatrix} - & \mathbf{a}_1' & - \\ & \vdots & \\ - & \mathbf{a}_m' & - \end{bmatrix}.$$

Given two matrices $\mathbf{A}$, $\mathbf{B}$ of dimensions $m \times n$ and $n \times k$, respectively, their *product* $\mathbf{AB}$ is a matrix of dimensions $m \times k$ whose entries are given by

$$[\mathbf{AB}]_{ij} = \sum_{\ell=1}^{n} [\mathbf{A}]_{i\ell}[\mathbf{B}]_{\ell j} = \mathbf{a}_i'\mathbf{B}_j,$$

where $\mathbf{a}_i'$ is the ith row of $\mathbf{A}$, and $\mathbf{B}_j$ is the jth column of $\mathbf{B}$. Matrix multiplication is associative, i.e., $(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$, but, in general, it is not commutative, that is, the equality $\mathbf{AB} = \mathbf{BA}$ is not always true. We also have $(\mathbf{AB})' = \mathbf{B}'\mathbf{A}'$.

Let $\mathbf{A}$ be an $m \times n$ matrix with columns $\mathbf{A}_i$. We then have $\mathbf{Ae}_i = \mathbf{A}_i$. Any vector $\mathbf{x} \in \Re^n$ can be written in the form $\mathbf{x} = \sum_{i=1}^{n} x_i \mathbf{e}_i$, which leads to

$$\mathbf{Ax} = \mathbf{A}\sum_{i=1}^{n} \mathbf{e}_i x_i = \sum_{i=1}^{n} \mathbf{Ae}_i x_i = \sum_{i=1}^{n} \mathbf{A}_i x_i.$$

A different representation of the matrix-vector product $\mathbf{Ax}$ is provided by the formula

$$\mathbf{Ax} = \begin{bmatrix} \mathbf{a}_1'\mathbf{x} \\ \mathbf{a}_2'\mathbf{x} \\ \vdots \\ \mathbf{a}_m'\mathbf{x} \end{bmatrix},$$

where $\mathbf{a}_1', \ldots, \mathbf{a}_m'$ are the rows of $\mathbf{A}$.

A matrix is called *square* if the number m of its rows is equal to the number n of its columns. We use $\mathbf{I}$ to denote the *identity* matrix, which is a square matrix whose diagonal entries are equal to one and its off-diagonal entries are equal to zero. The identity matrix satisfies $\mathbf{IA} = \mathbf{A}$ and $\mathbf{BI} = \mathbf{B}$ for any matrices $\mathbf{A}$, $\mathbf{B}$ of dimensions compatible with those of $\mathbf{I}$.

If $\mathbf{x}$ is a vector, the notation $\mathbf{x} \geq \mathbf{0}$ and $\mathbf{x} > \mathbf{0}$ means that every component of $\mathbf{x}$ is nonnegative (respectively, positive). If $\mathbf{A}$ is a matrix, the inequalities $\mathbf{A} \geq \mathbf{0}$ and $\mathbf{A} > \mathbf{0}$ have a similar meaning.

Matrix inversion

Let $\mathbf{A}$ be a square matrix. If there exists a square matrix $\mathbf{B}$ of the same dimensions satisfying $\mathbf{AB} = \mathbf{BA} = \mathbf{I}$, we say that $\mathbf{A}$ is *invertible* or *nonsingular*. Such a matrix $\mathbf{B}$, called the *inverse* of $\mathbf{A}$, is unique and is denoted by $\mathbf{A}^{-1}$. We note that $(\mathbf{A}')^{-1} = (\mathbf{A}^{-1})'$. Also, if $\mathbf{A}$ and $\mathbf{B}$ are invertible matrices of the same dimensions, then $\mathbf{AB}$ is also invertible and $(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$.

Given a finite collection of vectors $\mathbf{x}^1, \ldots, \mathbf{x}^K \in \Re^n$, we say that they are *linearly dependent* if there exist real numbers $a_1, \ldots, a_K$, not all of them zero, such that $\sum_{k=1}^{K} a_k \mathbf{x}^k = \mathbf{0}$; otherwise, they are called *linearly independent*. An equivalent definition of linear independence requires that

none of the vectors $\mathbf{x}^1, \ldots, \mathbf{x}^K$ is a linear combination of the remaining vectors (Exercise 1.18). We have the following result.

Theorem 1.2 *Let $\mathbf{A}$ be a square matrix. Then, the following statements are equivalent:*

(a) *The matrix $\mathbf{A}$ is invertible.*
(b) *The matrix $\mathbf{A}'$ is invertible.*
(c) *The determinant of $\mathbf{A}$ is nonzero.*
(d) *The rows of $\mathbf{A}$ are linearly independent.*
(e) *The columns of $\mathbf{A}$ are linearly independent.*
(f) *For every vector $\mathbf{b}$, the linear system $\mathbf{Ax} = \mathbf{b}$ has a unique solution.*
(g) *There exists some vector $\mathbf{b}$ such that the linear system $\mathbf{Ax} = \mathbf{b}$ has a unique solution.*

Assuming that $\mathbf{A}$ is an invertible square matrix, an explicit formula for the solution $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ of the system $\mathbf{Ax} = \mathbf{b}$, is given by *Cramer's rule*. Specifically, the jth component of $\mathbf{x}$ is given by

$$x_j = \frac{\det(\mathbf{A}^j)}{\det(\mathbf{A})},$$

where $\mathbf{A}^j$ is the same matrix as $\mathbf{A}$, except that its jth column is replaced by $\mathbf{b}$. Here, as well as later, the notation $\det(\mathbf{A})$ is used to denote the *determinant* of a square matrix $\mathbf{A}$.

Subspaces and bases

A nonempty subset S of $\Re^n$ is called a *subspace* of $\Re^n$ if $a\mathbf{x} + b\mathbf{y} \in S$ for every $\mathbf{x}, \mathbf{y} \in S$ and every $a, b \in \Re$. If, in addition, $S \neq \Re^n$, we say that S is a *proper* subspace. Note that every subspace must contain the zero vector.

The *span* of a finite number of vectors $\mathbf{x}^1, \ldots, \mathbf{x}^K$ in $\Re^n$ is the subspace of $\Re^n$ defined as the set of all vectors $\mathbf{y}$ of the form $\mathbf{y} = \sum_{k=1}^{K} a_k \mathbf{x}^k$, where each a_k is a real number. Any such vector $\mathbf{y}$ is called a *linear combination* of $\mathbf{x}^1, \ldots, \mathbf{x}^K$.

Given a subspace S of $\Re^n$, with $S \neq \{\mathbf{0}\}$, a *basis* of S is a collection of vectors that are linearly independent and whose span is equal to S. Every basis of a given subspace has the same number of vectors and this number is called the *dimension* of the subspace. In particular, the dimension of $\Re^n$ is equal to n and every proper subspace of $\Re^n$ has dimension smaller than n. Note that one-dimensional subspaces are lines through the origin; two-dimensional subspaces are planes through the origin. Finally, the set $\{\mathbf{0}\}$ is a subspace and its dimension is defined to be zero.

If S is a proper subspace of $\Re^n$, then there exists a nonzero vector $\mathbf{a}$ which is orthogonal to S, that is, $\mathbf{a}'\mathbf{x} = 0$ for every $\mathbf{x} \in S$. More generally, if S has dimension $m < n$, there exist $n - m$ linearly independent vectors that are orthogonal to S.

The result that follows provides some important facts regarding bases and linear independence.

Theorem 1.3 Suppose that the span S of the vectors $\mathbf{x}^1, \ldots, \mathbf{x}^K$ has dimension m. Then:

(a) There exists a basis of S consisting of m of the vectors $\mathbf{x}^1, \ldots, \mathbf{x}^K$.

(b) If $k \le m$ and $\mathbf{x}^1, \ldots, \mathbf{x}^k$ are linearly independent, we can form a basis of S by starting with $\mathbf{x}^1, \ldots, \mathbf{x}^k$, and choosing $m - k$ of the vectors $\mathbf{x}^{k+1}, \ldots, \mathbf{x}^K$.

Proof. We only prove part (b), because (a) is the special case of part (b) with $k = 0$. If every vector $\mathbf{x}^{k+1}, \ldots, \mathbf{x}^K$ can be expressed as a linear combination of $\mathbf{x}^1, \ldots, \mathbf{x}^k$, then every vector in the span of $\mathbf{x}^1, \ldots, \mathbf{x}^K$ is also a linear combination of $\mathbf{x}^1, \ldots, \mathbf{x}^k$, and the latter vectors form a basis. (In particular, $m = k$.) Otherwise, at least one of the vectors $\mathbf{x}^{k+1}, \ldots, \mathbf{x}^K$ is linearly independent from $\mathbf{x}^1, \ldots, \mathbf{x}^k$. By picking one such vector, we now have $k+1$ of the vectors $\mathbf{x}^1, \ldots, \mathbf{x}^K$ that are linearly independent. By repeating this process $m - k$ times, we end up with the desired basis of S. □

Let A be a matrix of dimensions $m \times n$. The *column space* of A is the subspace of $\Re^m$ spanned by the columns of A. The *row space* of A is the subspace of $\Re^n$ spanned by the rows of A. The dimension of the column space is always equal to the dimension of the row space, and this number is called the *rank* of A. Clearly, $\text{rank}(A) \le \min\{m, n\}$. The matrix A is said to have *full rank* if $\text{rank}(A) = \min\{m, n\}$. Finally, the set $\{\mathbf{x} \in \Re^n \mid A\mathbf{x} = 0\}$ is called the *nullspace* of A; it is a subspace of $\Re^n$ and its dimension is equal to $n - \text{rank}(A)$.

Affine subspaces

Let S_0 be a subspace of $\Re^n$ and let $\mathbf{x}^0$ be some vector. If we add $\mathbf{x}^0$ to every element of S_0, this amounts to translating S_0 by $\mathbf{x}^0$. The resulting set S can be defined formally by

$$S = S_0 + \mathbf{x}^0 = \{\mathbf{x} + \mathbf{x}^0 \mid \mathbf{x} \in S_0\}.$$

In general, S is not a subspace, because it does not necessarily contain the zero vector, and it is called an *affine subspace*. The *dimension* of S is defined to be equal to the dimension of the underlying subspace S_0.

As an example, let $\mathbf{x}^0, \mathbf{x}^1, \ldots, \mathbf{x}^k$ be some vectors in $\Re^n$, and consider the set S of all vectors of the form

$$\mathbf{x}^0 + \lambda_1 \mathbf{x}^1 + \cdots + \lambda_k \mathbf{x}^k,$$

where $\lambda_1, \ldots, \lambda_k$ are arbitrary scalars. For this case, S_0 can be identified with the span of the vectors $\mathbf{x}^1, \ldots, \mathbf{x}^k$, and S is an affine subspace. If the vectors $\mathbf{x}^1, \ldots, \mathbf{x}^k$ are linearly independent, their span has dimension k, and the affine subspace S also has dimension k.

For a second example, we are given an $m \times n$ matrix A and a vector $\mathbf{b} \in \Re^m$, and we consider the set

$$S = \{\mathbf{x} \in \Re^n \mid A\mathbf{x} = \mathbf{b}\},$$

which we assume to be nonempty. Let us fix some $\mathbf{x}^0$ such that $A\mathbf{x}^0 = \mathbf{b}$. An arbitrary vector $\mathbf{x}$ belongs to S if and only if $A\mathbf{x} = \mathbf{b} = A\mathbf{x}^0$, or $A(\mathbf{x} - \mathbf{x}^0) = 0$. Hence, $\mathbf{x} \in S$ if and only if $\mathbf{x} - \mathbf{x}^0$ belongs to the subspace $S_0 = \{\mathbf{y} \mid A\mathbf{y} = 0\}$. We conclude that $S = \{\mathbf{y} + \mathbf{x}^0 \mid \mathbf{y} \in S_0\}$; and S is an affine subspace of $\Re^n$. If A has m linearly independent rows, its nullspace S_0 has dimension $n - m$. Hence, the affine subspace S also has dimension $n - m$. Intuitively, if $\mathbf{a}'_i$ are the rows of A, each one of the constraints $\mathbf{a}'_i\mathbf{x} = b_i$ removes one degree of freedom from $\mathbf{x}$, thus reducing the dimension from n to $n - m$; see Figure 1.7 for an illustration.
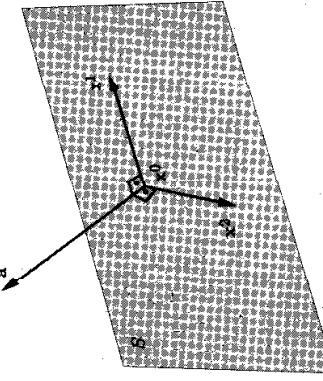
Figure 1.7: Consider a set S in $\Re^3$ defined by a single equality constraint $\mathbf{a}'\mathbf{x} = b$. Let $\mathbf{x}^0$ be an element of S. The vector $\mathbf{a}$ is perpendicular to S. If $\mathbf{x}^1$ and $\mathbf{x}^2$ are linearly independent vectors that are orthogonal to $\mathbf{a}$, then every $\mathbf{x} \in S$ is of the form $\mathbf{x} = \mathbf{x}^0 + \lambda_1 \mathbf{x}^1 + \lambda_2 \mathbf{x}^2$. In particular, S is a two-dimensional affine subspace.

1.6   Algorithms and operation counts

Optimization problems such as linear programming and, more generally, all computational problems are solved by *algorithms*. Loosely speaking, an algorithm is a finite set of instructions of the type used in common programming languages (arithmetic operations, conditional statements, read and write statements, etc.). Although the running time of an algorithm may depend substantially on clever programming or on the computer hardware available, we are interested in comparing algorithms without having to examine the details of a particular implementation. As a first approximation, this can be accomplished by counting the number of arithmetic operations (additions, multiplications, divisions, comparisons) required by an algorithm. This approach is often adequate even though it ignores the fact that adding or multiplying large integers or high-precision floating point numbers is more demanding than adding or multiplying single-digit integers. A more refined approach will be discussed briefly in Chapter 8.

Example 1.9

(a) Let **a** and **b** be vectors in $\Re^n$. The natural algorithm for computing $\mathbf{a}'\mathbf{b}$ requires n multiplications and $n-1$ additions, for a total of $2n-1$ arithmetic operations.

(b) Let **A** and **B** be matrices of dimensions $n \times n$. The traditional way of computing **AB** forms the inner product of a row of **A** and a column of **B** to obtain an entry of **AB**. Since there are n^2 entries to be evaluated, a total of $(2n-1)n^2$ arithmetic operations are involved.

In Example 1.9, an exact operation count was possible. However, for more complicated problems and algorithms, an exact count is usually very difficult. For this reason, we will settle for an estimate of the rate of growth of the number of arithmetic operations, as a function of the problem parameters. Thus, in Example 1.9, we might be content to say that the number of operations in the computation of an inner product increases linearly with n, and the number of operations in matrix multiplication increases cubically with n. This leads us to the order of magnitude notation that we define next.

Definition 1.2 *Let f and g be functions that map positive numbers to positive numbers.*

(a) *We write $f(n) = O(g(n))$ if there exist positive numbers n_0 and c such that $f(n) \le cg(n)$ for all $n \ge n_0$.*

(b) *We write $f(n) = \Omega(g(n))$ if there exist positive numbers n_0 and c such that $f(n) \ge cg(n)$ for all $n \ge n_0$.*

(c) *We write $f(n) = \Theta(g(n))$ if both $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ hold.*

For example, we have $3n^3 + n^2 + 10 = \Theta(n^3)$, $n \log n = O(n^2)$, and $n \log n = \Omega(n)$.

While the running time of the algorithms considered in Example 1.9 is predictable, the running time of more complicated algorithms often depends on the numerical values of the input data. In such cases, instead of trying to estimate the running time for each possible choice of the input, it is customary to estimate the running time for the *worst possible input data* of a given "size." For example, if we have an algorithm for linear programming, we might be interested in estimating its worst-case running time over all problems with a given number of variables and constraints. This emphasis on the worst case is somewhat conservative and, in practice, the "average" running time of an algorithm might be more relevant. However, the average running time is much more difficult to estimate or even to define, and for this reason, the worst-case approach is widely used.

Example 1.10 (Operation count of linear system solvers and matrix inversion) Consider the problem of solving a system of n linear equations in n unknowns. The classical method that eliminates one variable at a time (Gaussian elimination) is known to require $O(n^3)$ arithmetic operations in order to either compute a solution or to decide that no solution exists. Practical methods for matrix inversion also require $O(n^3)$ arithmetic operations. These facts will be of use later on.

Is the $O(n^3)$ running time of Gaussian elimination good or bad? Some perspective into this question is provided by the following observation: each time that technological advances lead to computer hardware that is faster by a factor of 8 (presumably every few years), we can solve problems of twice the size than earlier possible. A similar argument applies to algorithms whose running time is $O(n^k)$ for some positive integer k. Such algorithms are said to run in *polynomial time*.

Algorithms also exist whose running time is $\Omega(2^{cn})$, where n is a parameter representing problem size and c is a constant; these are said to take at least *exponential time*. For such algorithms and if $c = 1$, each time that computer hardware becomes faster by a factor of 2, we can increase the value of n that we can handle only by 1. It is then reasonable to expect that no matter how much technology improves, problems with truly large values of n will always be difficult to handle.

Example 1.11 Suppose that we have a choice of two algorithms. The running time of the first is $10^n/100$ (exponential) and the running time of the second is $10n^3$ (polynomial). For very small n. e.g., for $n = 3$, the exponential time algorithm is preferable. To gain some perspective as to what happens for larger n, suppose that we have access to a workstation that can execute 10^7 arithmetic operations per second and that we are willing to let it run for 1000 seconds. Let us figure out what size problems can each algorithm handle within this time frame. The equation $10^n/100 = 10^7 \times 1000$ yields $n = 12$, whereas the equation

$10n^3 = 10^7 \times 1000$ yields $n = 1000$, indicating that the polynomial time algorithm allows us to solve much larger problems.

The point of view emerging from the above discussion is that, as a first cut, it is useful to juxtapose polynomial and exponential time algorithms, the former being viewed as relatively fast and efficient, and the latter as relatively slow. This point of view is justified in many – but not all – contexts and we will be returning to it later in this book.

1.7 Exercises

Exercise 1.1* Suppose that a function $f : \Re^n \mapsto \Re$ is both concave and convex. Prove that f is an affine function.

Exercise 1.2 Suppose that $f_1, \ldots, f_m$ are convex functions from $\Re^n$ into $\Re$ and let $f(\mathbf{x}) = \sum_{i=1}^m f_i(\mathbf{x})$.

(a) Show that if each f_i is convex, so is f.

(b) Show that if each f_i is piecewise linear and convex, so is f.

Exercise 1.3 Consider the problem of minimizing a cost function of the form $\mathbf{c'x} + f(\mathbf{d'x})$, subject to the linear constraints $\mathbf{Ax} \geq \mathbf{b}$. Here, $\mathbf{d}$ is a given vector and the function $f : \Re \mapsto \Re$ is as specified in Figure 1.8. Provide a linear programming formulation of this problem.

Figure 1.8: The function f of Exercise 1.3.

Exercise 1.4 Consider the problem

minimize $2x_1 + 3|x_2 - 10|$
subject to $|x_1 + 2| + |x_2| \leq 5$,

and reformulate it as a linear programming problem.

Exercise 1.5 Consider a linear optimization problem, with absolute values, of the following form:

minimize $\mathbf{c'x} + \mathbf{d'y}$
subject to $\mathbf{Ax} + \mathbf{By} \leq \mathbf{b}$
$\qquad\qquad y_i = |x_i|$, $\forall i$.

Assume that all entries of $\mathbf{B}$ and $\mathbf{d}$ are nonnegative.

(a) Provide two different linear programming formulations, along the lines discussed in Section 1.3.

(b) Show that the original problem and the two reformulations are equivalent in the sense that either all three are infeasible, or all three have the same optimal cost

(c) Provide an example to show that if $\mathbf{B}$ has negative entries, the problem may have a local minimum that is not a global minimum. (It will be seen in Chapter 2 that this is never the case in linear programming problems. Hence, in the presence of such negative entries, a linear programming reformulation is implausible.)

Exercise 1.6 Provide linear programming formulations of the two variants of the rocket control problem discussed at the end of Section 1.3.

Exercise 1.7 (**The moment problem**) Suppose that Z is a random variable taking values in the set $0, 1, \ldots, K$, with probabilities $p_0, p_1, \ldots, p_K$, respectively. We are given the values of the first two moments $E[Z] = \sum_{k=0}^K k p_k$ and $E[Z^2] = \sum_{k=0}^K k^2 p_k$ of Z and we would like to obtain upper and lower bounds on the value of the fourth moment $E[Z^4] = \sum_{k=0}^K k^4 p_k$ of Z. Show how linear programming can be used to approach this problem.

Exercise 1.8 (**Road lighting**) Consider a road divided into n segments that is illuminated by m lamps. Let p_j be the power of the jth lamp. The illumination I_i of the ith segment is assumed to be $\sum_{j=1}^m a_{ij} p_j$, where a_{ij} are known coefficients. Let I_i^* be the desired illumination of road i.

We are interested in choosing the lamp powers p_j so that the illuminations I_i are close to the desired illuminations I_i^*. Provide a reasonable linear programming formulation of this problem. Note that the wording of the problem is loose and there is more than one possible formulation.

Exercise 1.9 Consider a school district with I neighborhoods, J schools, and G grades at each school. Each school j has a capacity of C_{jg} for grade g. In each neighborhood i, the student population of grade i is S_{ig}. Finally, the distance of school j from neighborhood i is d_{ij}. Formulate a linear programming problem whose objective is to assign all students to schools while minimizing the total distance traveled by all students. (You may ignore the fact that numbers of students must be integer.)

Exercise 1.10 (**Production and inventory planning**) A company must deliver d_i units of its product at the end of the ith month. Material produced during

a month can be delivered either at the end of the same month or can be stored as inventory and delivered at the end of a subsequent month; however, there is a storage cost of c_1 dollars per month for each unit of product held in inventory. The year begins with zero inventory. If the company produces x_i units in month i and x_{i+1} units in month $i+1$, it incurs a cost of $c_2|x_{i+1} - x_i|$ dollars, reflecting the cost of switching to a new production level. Formulate a linear programming problem whose objective is to minimize the total cost of the production and inventory schedule over a period of twelve months. Assume that inventory left at the end of the year has no value and does not incur any storage costs.

Exercise 1.11 (Optimal currency conversion) Suppose that there are N available currencies, and assume that one unit of currency i can be exchanged for r_{ij} units of currency j. (Naturally, we assume that $r_{ij} > 0$.) There also certain regulations that impose a limit u_i on the total amount of currency i that can be exchanged on any given day. Suppose that we start with B units of currency 1 and that we would like to maximize the number of units of currency N that we end up with at the end of the day, through a sequence of currency transactions. Provide a linear programming formulation of this problem. Assume that for any sequence $i_1,\ldots,i_k$ of currencies, we have $r_{i_1 i_2} r_{i_2 i_3} \cdots r_{i_{k-1} i_k} r_{i_k i_1} \le 1$, which means that wealth cannot be multiplied by going through a cycle of currencies.

Exercise 1.12 (Chebychev center) Consider a set P described by linear inequality constraints, that is, $P = \{x \in \Re^n \mid a_i'x \le b_i, \ i = 1,\ldots,m\}$. A ball with center y and radius r is defined as the set of all points within (Euclidean) distance r from y. We are interested in finding a ball with the largest possible radius, which is entirely contained within the set P. (The center of such a ball is called the *Chebychev center* of P.) Provide a linear programming formulation of this problem.

Exercise 1.13 (Linear fractional programming) Consider the problem

$$\begin{aligned} \text{minimize} \quad & \frac{c'x + d}{f'x + g} \\ \text{subject to} \quad & Ax \le b \\ & f'x + g > 0. \end{aligned}$$

Suppose that we have some prior knowledge that the optimal cost belongs to an interval $[K, L]$. Provide a procedure, that uses linear programming as a subroutine, and that allows us to compute the optimal cost within any desired accuracy. *Hint:* Consider the problem of deciding whether the optimal cost is less than or equal to a certain number.

Exercise 1.14 A company produces and sells two different products. The demand for each product is unlimited, but the company is constrained by cash availability and machine capacity.

Each unit of the first and second product requires 3 and 4 machine hours, respectively. There are 20,000 machine hours available in the current production period. The production costs are \$3 and \$2 per unit of the first and second product, respectively. The selling prices of the first and second product are \$6 and \$5.40 per unit, respectively. The available cash is \$4,000; furthermore, 45%

of the sales revenues from the first product and 30% of the sales revenues from the second product will be made available to finance operations during the current period.

(a) Formulate a linear programming problem that aims at maximizing net income subject to the cash availability and machine capacity limitations.

(b) Solve the problem graphically to obtain an optimal solution.

(c) Suppose that the company could increase its available machine hours by 2,000, after spending \$400 for certain repairs. Should the investment be made?

Exercise 1.15 A company produces two kinds of products. A product of the first type requires 1/4 hours of assembly labor, 1/8 hours of testing, and \$1.2 worth of raw materials. A product of the second type requires 1/3 hours of assembly, 1/3 hours of testing, and \$0.9 worth of raw materials. Given the current personnel of the company, there can be at most 90 hours of assembly labor and 80 hours of testing, each day. Products of the first and second type have a market value of \$9 and \$8, respectively.

(a) Formulate a linear programming problem that can be used to maximize the daily profit of the company.

(b) Consider the following two modifications to the original problem:

  (i) Suppose that up to 50 hours of overtime assembly labor can be scheduled, at a cost of \$7 per hour.

  (ii) Suppose that the raw material supplier provides a 10% discount if the daily bill is above \$300.

  Which of the above two elements can be easily incorporated into the linear programming formulation and how? If one or both are not easy to incorporate, indicate how you might nevertheless solve the problem.

Exercise 1.16 A manager of an oil refinery has 8 million barrels of crude oil A and 5 million barrels of crude oil B allocated for production during the coming month. These resources can be used to make either gasoline, which sells for \$38 per barrel, or home heating oil, which sells for \$33 per barrel. There are three production processes with the following characteristics:

	Process 1	Process 2	Process 3
Input crude A	3	1	5
Input crude B	5	1	3
Output gasoline	4	1	3
Output heating oil	3	1	4
Cost	\$51	\$11	\$40

All quantities are in barrels. For example, with the first process, 3 barrels of crude A and 5 barrels of crude B are used to produce 4 barrels of gasoline and

3 barrels of heating oil. The costs in this table refer to variable and allocated overhead costs, and there are no separate cost items for the cost of the crudes. Formulate a linear programming problem that would help the manager maximize net revenue over the next month.

Exercise 1.17 (Investment under taxation) An investor has a portfolio of n different stocks. He has bought s_i shares of stock i at price p_i, $i = 1, \ldots, n$. The current price of one share of stock i is q_i. The investor expects that the price of one share of stock i in one year will be r_i. If he sells shares, the investor pays a transaction cost at the rate of 1% of the amount transacted. In addition, the investor pays taxes at the rate of 30% on capital gains. For example, suppose that the investor sells 1,000 shares of a stock at $50 per share. He has bought these shares at $30 per share. He receives $50,000. However, he owes $0.30 \times (50,000 - 30,000) = \$6,000$ on capital gain taxes and $0.01 \times (50,000) = \$500$ on transaction costs. So, by selling 1,000 shares of this stock he nets $50,000 - 6,000 - 500 = \$43,500$. Formulate the problem of selecting how many shares the investor needs to sell in order to raise an amount of money K, net of capital gains and transaction costs, while maximizing the expected value of his portfolio next year.

Exercise 1.18 Show that the vectors in a given finite collection are linearly independent if and only if none of the vectors can be expressed as a linear combination of the others.

Exercise 1.19 Suppose that we are given a set of vectors in $\Re^n$ that form a basis, and let y be an arbitrary vector in $\Re^n$. We wish to express y as a linear combination of the basis vectors. How can this be accomplished?

Exercise 1.20

(a) Let $S = \{Ax \mid x \in \Re^n\}$, where A is a given matrix. Show that S is a subspace of $\Re^n$.

(b) Assume that S is a proper subspace of $\Re^n$. Show that there exists a matrix B such that $S = \{y \in \Re^n \mid B'y = 0\}$. *Hint:* Use vectors that are orthogonal to S to form the matrix B.

(c) Suppose that V is an m-dimensional affine subspace of $\Re^n$, with $m < n$. Show that there exist linearly independent vectors $a_1, \ldots, a_{n-m}$, and scalars $b_1, \ldots, b_{n-m}$, such that

$$V = \{y \mid a_i'y = b_i, \ i = 1, \ldots, n - m\}.$$

1.8   History, notes, and sources

The word "programming" has been used traditionally by planners to describe the process of operations planning and resource allocation. In the 1940s, it was realized that this process could often be aided by solving optimization problems involving linear constraints and linear objectives. The term "linear programming" then emerged. The initial impetus came in the aftermath of World War II, within the context of military planning problems. In 1947, Dantzig proposed an algorithm, the *simplex method*, which

made the solution of linear programming problems practical. There followed a period of intense activity during which many important problems in transportation, economics, military operations, scheduling, etc., were cast in this framework. Since then, computer technology has advanced rapidly, the range of applications has expanded, new powerful methods have been discovered, and the underlying mathematical understanding has become deeper and more comprehensive. Today, linear programming is a routinely used tool that can be found in some spreadsheet software packages.

Dantzig's development of the simplex method has been a defining moment in the history of the field, because it came at a time of growing practical needs and of advances in computing technology. But, as is the case with most "scientific revolutions," the history of the field is much richer. Early work goes back to Fourier, who in 1824 developed an algorithm for solving systems of linear inequalities. Fourier's method is far less efficient than the simplex method, but this issue was not relevant at the time. In 1910, de la Vallée Poussin developed a method, similar to the simplex method, for minimizing $\max_i |b_i - a_i'x|$, a problem that we discussed in Section 1.3.

In the late 1930s, the Soviet mathematician Kantorovich became interested in problems of optimal resource allocation in a centrally planned economy, for which he gave linear programming formulations. He also provided a solution method, but his work did not become widely known at the time. Around the same time, several models arising in classical, Walrasian, economics were studied and refined, and led to formulations closely related to linear programming. Koopmans, an economist, played an important role and eventually (in 1975) shared the Nobel Prize in economic science with Kantorovich.

On the theoretical front, the mathematical structures that underlie linear programming were independently studied, in the period 1870-1930, by many prominent mathematicians, such as Farkas, Minkowski, Carathéodory, and others. Also, in 1928, von Neumann developed an important result in game theory that would later prove to have strong connections with the deeper structure of linear programming.

Subsequent to Dantzig's work, there has been much and important research in areas such as large scale optimization, network optimization, interior point methods, integer programming, and complexity theory. We defer the discussion of this research to the notes and sources sections of later chapters. For a more detailed account of the history of linear programming, the reader is referred to Schrijver (1986), Orden (1993), and the volume edited by Lenstra, Rinnooy Kan, and Schrijver (1991) (see especially the article by Dantzig in that volume).

There are several texts that cover the general subject of linear programming, starting with a comprehensive one by Dantzig (1963). Some more recent texts are Papadimitriou and Steiglitz (1982), Chvátal (1983), Murty (1983), Luenberger (1984), Bazaraa, Jarvis, and Sherali (1990). Fi-

nally, Schrijver (1986) is a comprehensive, but more advanced reference on the subject.

1.1. The formulation of the diet problem is due to Stigler (1945).

1.2. The case study on DEC's production planning was developed by Freund and Shannahan (1992). Methods for dealing with the nurse scheduling and other cyclic problems are studied by Bartholdi, Orlin, and Ratliff (1980). More information on pattern classification can be found in Duda and Hart (1973), or Haykin (1994).

1.3. A deep and comprehensive treatment of convex functions and their properties is provided by Rockafellar (1970). Linear programming arises in control problems, in ways that are more sophisticated than what is described here; see, e.g., Dahleh and Diaz-Bobillo (1995).

1.5. For an introduction to linear algebra, see Strang (1988).

1.6. For a more detailed treatment of algorithms and their computational requirements, see Lewis and Papadimitriou (1981), Papadimitriou and Steiglitz (1982), or Cormen, Leiserson, and Rivest (1990).

1.7. Exercise 1.8 is adapted from Boyd and Vandenberghe (1995). Exercises 1.9 and 1.14 are adapted from Bradley, Hax, and Magnanti (1977). Exercise 1.11 is adapted from Ahuja, Magnanti, and Orlin (1993).

Chapter 2

The geometry of linear programming

Contents

In this chapter, we define a polyhedron as a set described by a finite number of linear equality and inequality constraints. In particular, the feasible set in a linear programming problem is a polyhedron. We study the basic geometric properties of polyhedra in some detail, with emphasis on their "corner points" (vertices). As it turns out, common geometric intuition derived from the familiar three-dimensional polyhedra is essentially correct when applied to higher-dimensional polyhedra. Another interesting aspect of the development in this chapter is that certain concepts (e.g., the concept of a vertex) can be defined either geometrically or algebraically. While the geometric view may be more natural, the algebraic approach is essential for carrying out computations. Much of the richness of the subject lies in the interplay between the geometric and the algebraic points of view.

Our development starts with a characterization of the corner points of feasible sets in the general form $\{x \mid Ax \geq b\}$. Later on, we focus on the case where the feasible set is in the standard form $\{x \mid Ax = b, x \geq 0\}$, and we derive a simple algebraic characterization of the corner points. The latter characterization will play a central role in the development of the simplex method in Chapter 3.

The main results of this chapter state that a nonempty polyhedron has at least one corner point if and only if it does not contain a line, and if this is the case, the search for optimal solutions to linear programming problems can be restricted to corner points. These results are proved for the most general case of linear programming problems using geometric arguments. The same results will also be proved in the next chapter, for the case of standard form problems, as a corollary of our development of the simplex method. Thus, the reader who wishes to focus on standard form problems may skip the proofs in Sections 2.5 and 2.6. Finally, Sections 2.7 and 2.8 can also be skipped during a first reading; any results in these sections that are needed later on will be rederived in Chapter 4, using different techniques.

2.1 Polyhedra and convex sets

In this section, we introduce some important concepts that will be used to study the geometry of linear programming, including a discussion of convexity.

Hyperplanes, halfspaces, and polyhedra

We start with the formal definition of a polyhedron.

***

Definition 2.1 A **polyhedron** *is a set that can be described in the form* $\{x \in \Re^n \mid Ax \geq b\}$, *where A is an $m \times n$ matrix and b is a vector in $\Re^m$.*

***

As discussed in Section 1.1, the feasible set of any linear programming problem can be described by inequality constraints of the form $Ax \geq b$, and is therefore a polyhedron. In particular, a set of the form $\{x \in \Re^n \mid Ax = b, x \geq 0\}$ is also a polyhedron and will be referred to as a *polyhedron in standard form*.

A polyhedron can either "extend to infinity," or can be confined in a finite region. The definition that follows refers to this distinction.

***

Definition 2.2 *A set $S \subset \Re^n$ is **bounded** if there exists a constant K such that the absolute value of every component of every element of S is less than or equal to K.*

***

The next definition deals with polyhedra determined by a single linear constraint.

***

Definition 2.3 *Let a be a nonzero vector in $\Re^n$ and let b be a scalar.*

(a) *The set $\{x \in \Re^n \mid a'x = b\}$ is called a **hyperplane**.*
(b) *The set $\{x \in \Re^n \mid a'x \geq b\}$ is called a **halfspace**.*

***

Note that a hyperplane is the boundary of a corresponding halfspace. In addition, the vector a in the definition of the hyperplane is perpendicular to the hyperplane itself. [To see this, note that if x and y belong to the same hyperplane, then $a'x = a'y$. Hence, $a'(x - y) = 0$ and therefore a is orthogonal to any direction vector confined to the hyperplane.] Finally, note that a polyhedron is equal to the intersection of a finite number of halfspaces; see Figure 2.1.

Convex Sets

We now define the important notion of a convex set.

***

Definition 2.4 *A set $S \subset \Re^n$ is **convex** if for any $x, y \in S$, and any $\lambda \in [0,1]$, we have $\lambda x + (1 - \lambda)y \in S$.*

***

Note that if $\lambda \in [0,1]$, then $\lambda x + (1 - \lambda)y$ is a weighted average of the vectors x, y, and therefore belongs to the line segment joining x and y. Thus, a set is convex if the segment joining any two of its elements is contained in the set; see Figure 2.2.

Our next definition refers to weighted averages of a finite number of vectors; see Figure 2.3.

(a)

(b)
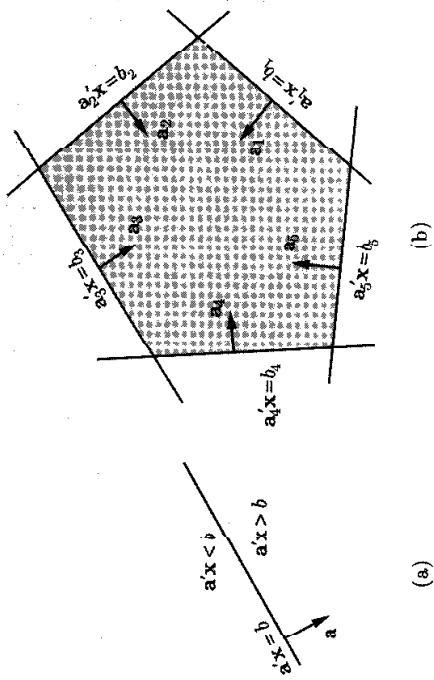
Figure 2.1: (a) A hyperplane and two halfspaces. (b) The polyhedron $\{\mathbf{x} \mid \mathbf{a}_i'\mathbf{x} \geq b_i,\ i = 1,\ldots,5\}$ is the intersection of five halfspaces. Note that each vector $\mathbf{a}_i$ is perpendicular to the hyperplane $\{\mathbf{x} \mid \mathbf{a}_i'\mathbf{x} = b_i\}$.

Definition 2.5 *Let* $\mathbf{x}^1,\ldots,\mathbf{x}^k$ *be vectors in* $\Re^n$ *and let* $\lambda_1,\ldots,\lambda_k$ *be nonnegative scalars whose sum is unity.*

(a) *The vector* $\sum_{i=1}^{k} \lambda_i \mathbf{x}^i$ *is said to be a* **convex combination** *of the vectors* $\mathbf{x}^1,\ldots,\mathbf{x}^k$.

(b) *The* **convex hull** *of the vectors* $\mathbf{x}^1,\ldots,\mathbf{x}^k$ *is the set of all convex combinations of these vectors.*

The result that follows establishes some important facts related to convexity.

Theorem 2.1

(a) *The intersection of convex sets is convex.*

(b) *Every polyhedron is a convex set.*

(c) *A convex combination of a finite number of elements of a convex set also belongs to that set.*

(d) *The convex hull of a finite number of vectors is a convex set.*
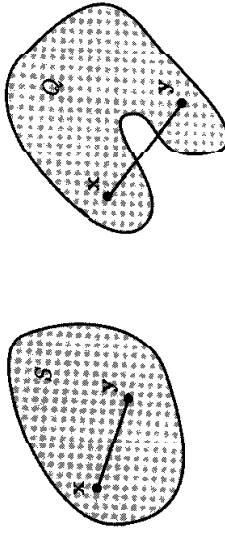
Figure 2.2: The set S is convex, but the set Q is not, because the segment joining $\mathbf{x}$ and $\mathbf{y}$ is not contained in Q.
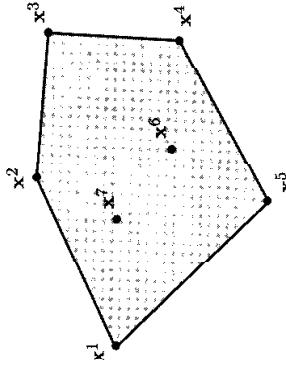
Figure 2.3: The convex hull of seven points in $\Re^2$.

Proof.

(a) Let S_i, $i \in I$, be convex sets where I is some index set, and suppose that $\mathbf{x}$ and $\mathbf{y}$ belong to the intersection $\cap_{i \in I} S_i$. Let $\lambda \in [0,1]$. Since each S_i is convex and contains $\mathbf{x}, \mathbf{y}$, we have $\lambda \mathbf{x} + (1-\lambda)\mathbf{y} \in S_i$, which proves that $\lambda \mathbf{x} + (1-\lambda)\mathbf{y}$ also belongs to the intersection of the sets S_i. Therefore, $\cap_{i \in I} S_i$ is convex.

(b) Let $\mathbf{a}$ be a vector and let b a scalar. Suppose that $\mathbf{x}$ and $\mathbf{y}$ satisfy $\mathbf{a}'\mathbf{x} \geq b$ and $\mathbf{a}'\mathbf{y} \geq b$, respectively, and therefore belong to the same halfspace. Let $\lambda \in [0,1]$. Then, $\mathbf{a}'(\lambda \mathbf{x} + (1-\lambda)\mathbf{y}) \geq \lambda b + (1-\lambda)b = b$, which proves that $\lambda \mathbf{x} + (1-\lambda)\mathbf{y}$ also belongs to the same halfspace. Therefore a halfspace is convex. Since a polyhedron is the intersection of a finite number of halfspaces, the result follows from part (a).

(c) A convex combination of two elements of a convex set lies in that

set, by the definition of convexity. Let us assume, as an induction hypothesis, that a convex combination of k elements of a convex set belongs to that set. Consider $k+1$ elements $\mathbf{x}^1,\ldots,\mathbf{x}^{k+1}$ of a convex set S and let $\lambda_1,\ldots,\lambda_{k+1}$ be nonnegative scalars that sum to 1. We assume, without loss of generality, that $\lambda_{k+1} \neq 1$. We then have

$$\sum_{i=1}^{k+1} \lambda_i \mathbf{x}^i = \lambda_{k+1}\mathbf{x}^{k+1} + (1-\lambda_{k+1})\sum_{i=1}^{k} \frac{\lambda_i}{1-\lambda_{k+1}}\mathbf{x}^i. \qquad (2.1)$$

The coefficients $\lambda_i/(1-\lambda_{k+1})$, $i=1,\ldots,k$, are nonnegative and sum to unity; using the induction hypothesis, $\sum_{i=1}^{k} \lambda_i \mathbf{x}^i/(1-\lambda_{k+1}) \in S$. Then, the fact that S is convex and Eq. (2.1) imply that $\sum_{i=1}^{k+1} \lambda_i \mathbf{x}^i \in S$, and the induction step is complete.

(d) Let S be the convex hull of the vectors $\mathbf{x}^1,\ldots,\mathbf{x}^k$ and let $\mathbf{y} = \sum_{i=1}^{k} \zeta_i \mathbf{x}^i$, $\mathbf{z} = \sum_{i=1}^{k} \theta_i \mathbf{x}^i$ be two elements of S, where $\zeta_i \geq 0$, $\theta_i \geq 0$, and $\sum_{i=1}^{k} \zeta_i = \sum_{i=1}^{k} \theta_i = 1$. Let $\lambda \in [0,1]$. Then,

$$\lambda\mathbf{y} + (1-\lambda)\mathbf{z} = \lambda \sum_{i=1}^{k} \zeta_i \mathbf{x}^i + (1-\lambda)\sum_{i=1}^{k} \theta_i \mathbf{x}^i = \sum_{i=1}^{k}(\lambda\zeta_i + (1-\lambda)\theta_i)\mathbf{x}^i.$$

We note that the coefficients $\lambda\zeta_i + (1-\lambda)\theta_i$, $i=1,\ldots,k$, are nonnegative and sum to unity. This shows that $\lambda\mathbf{y}+(1-\lambda)\mathbf{z}$ is a convex combination of $\mathbf{x}^1,\ldots,\mathbf{x}^k$ and, therefore, belongs to S. This establishes the convexity of S.   □

2.2 Extreme points, vertices, and basic feasible solutions

We observed in Section 1.4 that an optimal solution to a linear programming problem tends to occur at a "corner" of the polyhedron over which we are optimizing. In this section, we suggest three different ways of defining the concept of a "corner" and then show that all three definitions are equivalent.

Our first definition defines an *extreme point* of a polyhedron as a point that cannot be expressed as a convex combination of two other elements of the polyhedron, and is illustrated in Figure 2.4. Notice that this definition is entirely geometric and does not refer to a specific representation of a polyhedron in terms of linear constraints.

Definition 2.6 *Let P be a polyhedron. A vector $\mathbf{x} \in P$ is an extreme point of P if we cannot find two vectors $\mathbf{y}, \mathbf{z} \in P$, both different from $\mathbf{x}$, and a scalar $\lambda \in [0,1]$, such that $\mathbf{x} = \lambda\mathbf{y} + (1-\lambda)\mathbf{z}$.*
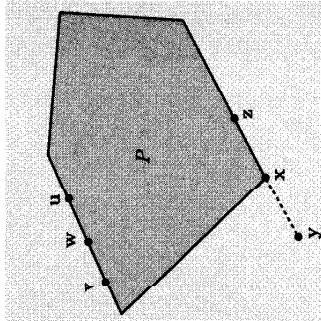
Figure 2.4: The vector $\mathbf{w}$ is not an extreme point because it is a convex combination of $\mathbf{v}$ and $\mathbf{u}$. The vector $\mathbf{x}$ is an extreme point: if $\mathbf{x} = \lambda\mathbf{y} + (1-\lambda)\mathbf{z}$ and $\lambda \in [0,1]$, then either $\mathbf{y} \notin P$, or $\mathbf{z} \notin P$, or $\mathbf{x} = \mathbf{y}$, or $\mathbf{x} = \mathbf{z}$.

An alternative geometric definition defines a *vertex* of a polyhedron P as the unique optimal solution to some linear programming problem with feasible set P.

Definition 2.7 *Let P be a polyhedron. A vector $\mathbf{x} \in P$ is a vertex of P if there exists some $\mathbf{c}$ such that $\mathbf{c}'\mathbf{x} < \mathbf{c}'\mathbf{y}$ for all $\mathbf{y}$ satisfying $\mathbf{y} \in P$ and $\mathbf{y} \neq \mathbf{x}$.*

In other words, $\mathbf{x}$ is a vertex of P if and only if P is on one side of a hyperplane (the hyperplane $\{\mathbf{y} \mid \mathbf{c}'\mathbf{y} = \mathbf{c}'\mathbf{x}\}$) which meets P only at the point $\mathbf{x}$; see Figure 2.5.

The two geometric definitions that we have given so far are not easy to work with from an algorithmic point of view. We would like to have a definition that relies on a representation of a polyhedron in terms of linear constraints and which reduces to an algebraic test. In order to provide such a definition, we need some more terminology.

Consider a polyhedron $P \subset \Re^n$ defined in terms of the linear equality and inequality constraints

$$\begin{aligned}
\mathbf{a}'_i\mathbf{x} &\geq b_i, & i &\in M_1, \\
\mathbf{a}'_i\mathbf{x} &\leq b_i, & i &\in M_2, \\
\mathbf{a}'_i\mathbf{x} &= b_i, & i &\in M_3,
\end{aligned}$$

where M_1, M_2, and M_3 are finite index sets, each $\mathbf{a}_i$ is a vector in $\Re^n$, and
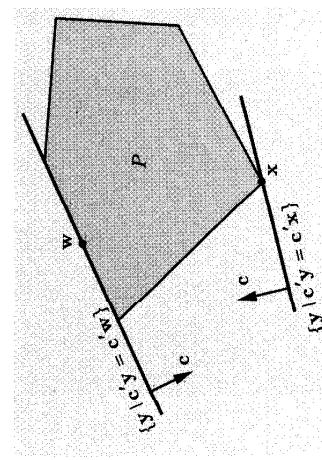
Figure 2.5: The line a: the bottom touches P at a single point and $\mathbf{x}$ is a vertex. On the other hand, $\mathbf{w}$ is not a vertex because there is no hyperplane that meets P only at $\mathbf{w}$.

each b_i is a scalar. The definition that follows is illustrated in Figure 2.6.

Definition 2.8 *If a vector $\mathbf{x}^*$ satisfies $\mathbf{a}_i'\mathbf{x}^* = b_i$ for some i in M_1, M_2, or M_3, we say that the corresponding constraint is **active** or **binding** at $\mathbf{x}^*$.*

If there are n constraints that are active at a vector $\mathbf{x}^*$, then $\mathbf{x}^*$ satisfies a certain system of n linear equations in n unknowns. This system has a unique solution if and only if these n equations are "linearly independent." The result that follows gives a precise meaning to this statement, together with a slight generalization.

Theorem 2.2 *Let $\mathbf{x}^*$ be an element of $\Re^n$ and let $I = \{i \mid \mathbf{a}_i'\mathbf{x}^* = b_i\}$ be the set of indices of constraints that are active at $\mathbf{x}^*$. Then, the following are equivalent:*

(a) *There exist n vectors in the set $\{\mathbf{a}_i \mid i \in I\}$, which are linearly independent.*

(b) *The span of the vectors $\mathbf{a}_i$, $i \in I$, is all of $\Re^n$, that is, every element of $\Re^n$ can be expressed as a linear combination of the vectors $\mathbf{a}_i$, $i \in I$.*

(c) *The system of equations $\mathbf{a}_i'\mathbf{x} = b_i$, $i \in I$, has a unique solution.*

Proof. Suppose that the vectors $\mathbf{a}_i$, $i \in I$, span $\Re^n$. Then, the span of these vectors has dimension n. By Theorem 1.3(a) in Section 1.5, n of
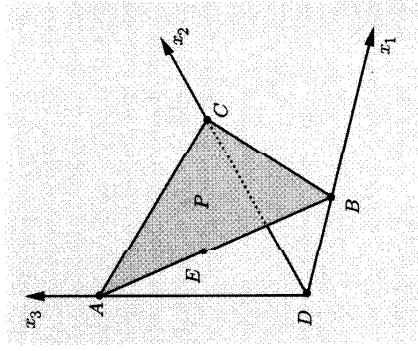
Figure 2.6: Let $P = \{(x_1, x_2, x_3) \mid x_1 + x_2 + x_3 = 1, \ x_1, x_2, x_3 \geq 0\}$. There are three constraints that are active at each one of the points A, B, C and D. There are only two constraints that are active at point E, namely $x_1 + x_2 + x_3 = 1$ and $x_2 = 0$.

these vectors form a basis of $\Re^n$, and are therefore linearly independent. Conversely, suppose that n of the vectors $\mathbf{a}_i$, $i \in I$, are linearly independent. Then, the subspace spanned by these n vectors is n-dimensional and must be equal to $\Re^n$. Hence, every element of $\Re^n$ is a linear combination of the vectors $\mathbf{a}_i$, $i \in I$. This establishes the equivalence of (a) and (b).

If the system of equations $\mathbf{a}_i'\mathbf{x} = b_i$, $i \in I$, has multiple solutions, say $\mathbf{x}^1$ and $\mathbf{x}^2$, then the nonzero vector $\mathbf{d} = \mathbf{x}^1 - \mathbf{x}^2$ satisfies $\mathbf{a}_i'\mathbf{d} = 0$ for all $i \in I$. Since $\mathbf{d}$ is orthogonal to every vector $\mathbf{a}_i$, $i \in I$, $\mathbf{d}$ is not a linear combination of these vectors and it follows that the vectors $\mathbf{a}_i$, $i \in I$, do not span $\Re^n$. Conversely, if the vectors $\mathbf{a}_i$, $i \in I$, do not span $\Re^n$, choose a nonzero vector $\mathbf{d}$ which is orthogonal to the subspace spanned by these vectors. If $\mathbf{x}$ satisfies $\mathbf{a}_i'\mathbf{x} = b_i$ for all $i \in I$, we also have $\mathbf{a}_i'(\mathbf{x} + \mathbf{d}) = b_i$ for all $i \in I$, thus obtaining multiple solutions. We have therefore established that (b) and (c) are equivalent. $\square$

With a slight abuse of language, we will often say that certain *constraints are linearly independent*, meaning that the corresponding vectors $\mathbf{a}_i$ are linearly independent. With this terminology, statement (a) in Theorem 2.2 requires that there exist n linearly independent constraints that are active at $\mathbf{x}^*$.

We are now ready to provide an algebraic definition of a corner point, as a feasible solution at which there are n linearly independent active constraints. Note that since we are interested in a feasible solution, all equality

constraints must be active. This suggests the following way of looking for corner points: first impose the equality constraints and then require that enough additional constraints be active, so that we get a total of n linearly independent active constraints. Once we have n linearly independent active constraints, a unique vector $\mathbf{x}^*$ is determined (Theorem 2.2). However, this procedure has no guarantee of leading to a feasible vector $\mathbf{x}^*$, because some of the inactive constraints could be violated; in the latter case we say that we have a basic (but not basic feasible) solution.

Definition 2.9 *Consider a polyhedron P defined by linear equality and inequality constraints, and let $\mathbf{x}^*$ be an element of $\Re^n$.*

(a) *The vector $\mathbf{x}^*$ is a basic solution if:*

    (i) *All equality constraints are active;*

    (ii) *Out of the constraints that are active at $\mathbf{x}^*$, there are n of them that are linearly independent.*

(b) *If $\mathbf{x}^*$ is a basic solution that satisfies all of the constraints, we say that it is a **basic feasible solution**.*

In reference to Figure 2.6, we note that points A, B, and C are basic feasible solutions. Point D is not a basic solution because it fails to satisfy the equality constraint. Point E is feasible, but not basic. If the equality constraint $x_1 + x_2 + x_3 = 1$ were to be replaced by the constraints $x_1 + x_2 + x_3 \leq 1$ and $x_1 + x_2 + x_3 \geq 1$, then D would be a basic solution, according to our definition. This shows that whether a point is a basic solution or not may depend on the way that a polyhedron is represented. Definition 2.9 is also illustrated in Figure 2.7.

Note that if the number m of constraints used to define a polyhedron $P \subset \Re^n$ is less than n, the number of active constraints at any given point must also be less than n, and there are no basic or basic feasible solutions.

We have given so far three different definitions that are meant to capture the same concept; two of them are geometric (extreme point, vertex) and the third is algebraic (basic feasible solution). Fortunately, all three definitions are equivalent as we prove next and, for this reason, the three terms can be used interchangeably.

Theorem 2.3 *Let P be a nonempty polyhedron and let $\mathbf{x}^* \in P$. Then, the following are equivalent:*

(a) $\mathbf{x}^*$ *is a vertex;*

(b) $\mathbf{x}^*$ *is an extreme point;*

(c) $\mathbf{x}^*$ *is a basic feasible solution.*
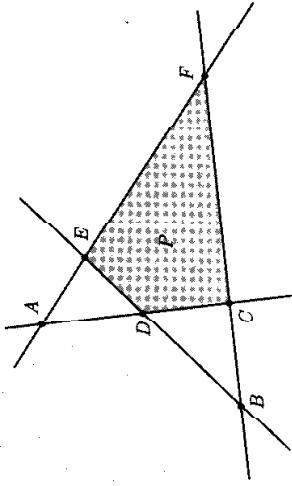
Figure 2.7: The points A, B, C, D, E, F are all basic solutions because at each one of them, there are two linearly independent constraints that are active. Points C, D, E, F are basic feasible solutions.

Proof. For the purposes of this proof and without loss of generality, we assume that P is represented in terms of constraints of the form $\mathbf{a}_i'\mathbf{x} \geq b_i$ and $\mathbf{a}_i'\mathbf{x} = b_i$.

Vertex $\Rightarrow$ Extreme point

Suppose that $\mathbf{x}^* \in P$ is a vertex. Then, by Definition 2.7, there exists some $\mathbf{c} \in \Re^n$ such that $\mathbf{c}'\mathbf{x}^* < \mathbf{c}'\mathbf{y}$ for every $\mathbf{y}$ satisfying $\mathbf{y} \in P$ and $\mathbf{y} \neq \mathbf{x}^*$. If $\mathbf{y} \in P$, $\mathbf{z} \in P$, $\mathbf{y} \neq \mathbf{x}^*$, $\mathbf{z} \neq \mathbf{x}^*$, and $0 \leq \lambda \leq 1$, then $\mathbf{c}'\mathbf{x}^* < \mathbf{c}'\mathbf{y}$ and $\mathbf{c}'\mathbf{x}^* < \mathbf{c}'\mathbf{z}$, which implies that $\mathbf{c}'\mathbf{x}^* < \mathbf{c}'(\lambda\mathbf{y} + (1-\lambda)\mathbf{z})$ and, therefore, $\mathbf{x}^* \neq \lambda\mathbf{y} + (1-\lambda)\mathbf{z}$. Thus, $\mathbf{x}^*$ cannot be expressed as a convex combination of two other elements of P and is, therefore, an extreme point (cf. Definition 2.6)

Extreme point $\Rightarrow$ Basic feasible solution

Suppose that $\mathbf{x}^* \in P$ is not a basic feasible solution. We will show that $\mathbf{x}^*$ is not an extreme point of P. Let $I = \{i \mid \mathbf{a}_i'\mathbf{x}^* = b_i\}$. Since $\mathbf{x}^*$ is not a basic feasible solution, there do not exist n linearly independent vectors in the family $\mathbf{a}_i$, $i \in I$. Thus, the vectors $\mathbf{a}_i$, $i \in I$, lie in a proper subspace of $\Re^n$, and there exists some nonzero vector $\mathbf{d} \in \Re^n$ such that $\mathbf{a}_i'\mathbf{d} = 0$, for all $i \in I$. Let ϵ be a small positive number and consider the vectors $\mathbf{y} = \mathbf{x}^* + \epsilon\mathbf{d}$ and $\mathbf{z} = \mathbf{x}^* - \epsilon\mathbf{d}$. Notice that $\mathbf{a}_i'\mathbf{y} = \mathbf{a}_i'\mathbf{x}^* = b_i$, for $i \in I$. Furthermore, for $i \notin I$, we have $\mathbf{a}_i'\mathbf{x}^* > b_i$ and, provided that ϵ is small, we will also have $\mathbf{a}_i'\mathbf{y} > b_i$. (It suffices to choose ϵ so that $\epsilon|\mathbf{a}_i'\mathbf{d} < \mathbf{a}_i'\mathbf{x}^* - b_i$ for all $i \notin I$.) Thus, when ϵ is small enough, $\mathbf{y} \in P$ and, by a similar argument, $\mathbf{z} \in P$. We finally notice that $\mathbf{x}^* = (\mathbf{y} + \mathbf{z})/2$, which implies that $\mathbf{x}^*$ is not an extreme point.

Basic feasible solution ⇒ Vertex

Let x^* be a basic feasible solution and let $I = \{i \mid a_i'x^* = b_i\}$. Let $c = \sum_{i \in I} a_i$. We then have

$$c'x^* = \sum_{i \in I} a_i'x^* = \sum_{i \in I} b_i.$$

Furthermore, for any $x \in P$ and any i, we have $a_i'x \geq b_i$, and

$$c'x = \sum_{i \in I} a_i'x \geq \sum_{i \in I} b_i. \quad (2.2)$$

This shows that x^* is an optimal solution to the problem of minimizing $c'x$ over the set P. Furthermore, equality holds in (2.2) if and only if $a_i'x = b_i$ for all $i \in I$. Since x^* is a basic feasible solution, there are n linearly independent constraints that are active at x^*, and x^* is the unique solution to the system of equations $a_i'x = b_i$, $i \in I$ (Theorem 2.2). It follows that x^* is the unique minimizer of $c'x$ over the set P and, therefore, x^* is a vertex of P.

Since a vector is a basic feasible solution if and only if it is an extreme point, and since the definition of an extreme point does not refer to any particular representation of a polyhedron, we conclude that the property of being a basic feasible solution is also independent of the representation used. (This is in contrast to the definition of a basic solution, which is representation dependent, as pointed out in the discussion that followed Definition 2.9.)

We finally note the following important fact.

> **Corollary 2.1** *Given a finite number of linear inequality constraints, there can only be a finite number of basic or basic feasible solutions.*

Proof. Consider a system of m linear inequality constraints imposed on a vector $x \in \mathfrak{R}^n$. At any basic solution, there are n linearly independent active constraints. Since any n linearly independent active constraints define a unique point, it follows that different basic solutions correspond to different sets of n linearly independent active constraints. Therefore, the number of basic solutions is bounded above by the number of ways that we can choose n constraints out of a total of m, which is finite. $\square$

Although the number of basic and, therefore, basic feasible solutions is guaranteed to be finite, it can be very large. For example, the unit cube $\{x \in \mathfrak{R}^n \mid 0 \leq x_i \leq 1, \; i = 1,\ldots,n\}$ is defined in terms of $2n$ constraints, but has 2^n basic feasible solutions.

Adjacent basic solutions

Two distinct basic solutions to a set of linear constraints in $\mathfrak{R}^n$ are said to be *adjacent* if we can find $n - 1$ linearly independent constraints that are active at both of them. In reference to Figure 2.7, D and E are adjacent to B; also, A and C are adjacent to D. If two adjacent basic solutions are also feasible, then the line segment that joins them is called an *edge* of the feasible set (see also Exercise 2.15).

2.3 Polyhedra in standard form

The definition of a basic solution (Definition 2.9) refers to general polyhedra. We will now specialize to polyhedra in standard form. The definitions and the results in this section are central to the development of the simplex method in the next chapter.

Let $P = \{x \in \mathfrak{R}^n \mid Ax = b, \; x \geq 0\}$ be a polyhedron in standard form, and let the dimensions of A be $m \times n$, where m is the number of equality constraints. In most of our discussion of standard form problems, we will make the assumption that the m rows of the matrix A are linearly independent. (Since the rows are n-dimensional, this requires that $m \leq n$.) At the end of this section, we show that when P is nonempty, linearly dependent rows of A correspond to redundant constraints that can be discarded; therefore, our linear independence assumption can be made without loss of generality.

Recall that at any basic solution there must be n linearly independent constraints that are active. Furthermore, every basic solution must satisfy the equality constraints $Ax = b$, which provides us with m active constraints; these are linearly independent because of our assumption on the rows of A. In order to obtain a total of n active constraints, we need to choose $n - m$ of the variables x_i and set them to zero, which makes the corresponding nonnegativity constraints $x_i \geq 0$ active. However, for the resulting set of n active constraints to be linearly independent, the choice of these $n - m$ variables is not entirely arbitrary, as shown by the following result.

> **Theorem 2.4** *Consider the constraints $Ax = b$ and $x \geq 0$ and assume that the $m \times n$ matrix A has linearly independent rows. A vector $x \in \mathfrak{R}^n$ is a basic solution if and only if we have $Ax = b$, and there exist indices $B(1),\ldots,B(m)$ such that:*
> *(a) The columns $A_{B(1)},\ldots,A_{B(m)}$ are linearly independent;*
> *(b) If $i \neq B(1),\ldots,B(m)$, then $x_i = 0$.*

Proof. Consider some $x \in \mathfrak{R}^n$ and suppose that there are indices $B(1),\ldots,$

$B(m)$ that satisfy (a) and (b) in the statement of the theorem. The active constraints $x_i = 0$, $i \neq B(1), \ldots, B(m)$, and $Ax = b$ imply that

$$\sum_{i=1}^{m} A_{B(i)} x_{B(i)} = \sum_{i=1}^{n} A_i x_i = Ax = b.$$

Since the columns $A_{B(i)}$, $i = 1, \ldots, m$, are linearly independent, $x_{B(1)}, \ldots, x_{B(m)}$ are uniquely determined. Thus, the system of equations formed by the active constraints has a unique solution. By Theorem 2.2, there are n linearly independent active constraints, and this implies that x is a basic solution.

For the converse, we assume that x is a basic solution and we will show that conditions (a) and (b) in the statement of the theorem are satisfied. Let $x_{B(1)}, \ldots, x_{B(k)}$ be the components of x that are nonzero. Since x is a basic solution, the system of equations formed by the active constraints $\sum_{i=1}^{n} A_i x_i = b$ and $x_i = 0$, $i \neq B(1), \ldots, B(k)$, have a unique solution (cf. Theorem 2.2); equivalently, the equation $\sum_{i=1}^{k} A_{B(i)} x_{B(i)} = b$ has a unique solution. It follows that the columns $A_{B(1)}, \ldots, A_{B(k)}$ are linearly independent. [If they were not, we could find scalars $\lambda_1, \ldots, \lambda_k$, not all of them zero, such that $\sum_{i=1}^{k} A_{B(i)} \lambda_i = 0$. This would imply that $\sum_{i=1}^{k} A_{B(i)}(x_{B(i)} + \lambda_i) = b$, contradicting the uniqueness of the solution.] We have shown that the columns $A_{B(1)}, \ldots, A_{B(k)}$ are linearly independent and this implies that $k \leq m$. Since A has m linearly independent rows, it also has m linearly independent columns which span $\Re^m$. It follows [cf. Theorem 1.3(t) in Section 1.5] that we can find $m-k$ additional columns $A_{B(k+1)}, \ldots, A_{B(m)}$ so that the columns $A_{B(i)}$, $i = 1, \ldots, m$, are linearly independent. In addition, if $i \neq B(1), \ldots, B(m)$, then $i \neq B(1), \ldots, B(k)$ (because $k \leq m$), and $x_i = 0$. Therefore, both conditions (a) and (b) in the statement of the theorem are satisfied.

In view of Theorem 2.4, all basic solutions to a standard form polyhedron can be constructed according to the following procedure.

Procedure for constructing basic solutions

1. Choose m linearly independent columns $A_{B(1)}, \ldots, A_{B(m)}$.
2. Let $x_i = 0$ for all $i \neq B(1), \ldots, B(m)$.
3. Solve the system of m equations $Ax = b$ for the unknowns $x_{B(1)}, \ldots, x_{B(m)}$.

If a basic solution constructed according to this procedure is nonnegative, then it is feasible, and it is a basic feasible solution. Conversely, since every basic feasible solution is a basic solution, it can be obtained from this procedure. If x is a basic solution, the variables $x_{B(1)}, \ldots, x_{B(m)}$ are called

basic variables; the remaining variables are called nonbasic. The columns $A_{B(1)}, \ldots, A_{B(m)}$ are called the basic columns and, since they are linearly independent, they form a basis of $\Re^m$. We will sometimes talk about two bases being distinct or different; our convention is that distinct bases involve different sets $\{B(1), \ldots, B(m)\}$ of basic indices; if two bases involve the same set of indices in a different order, they will be viewed as one and the same basis.

By arranging the m basic columns next to each other, we obtain an $m \times m$ matrix B, called a basis matrix (Note that this matrix is invertible because the basic columns are required to be linearly independent.) We can similarly define a vector x_B with the values of the basic variables. Thus,

$$B = \begin{bmatrix} | & | & & | \\ A_{B(1)} & A_{B(2)} & \cdots & A_{B(m)} \\ | & | & & | \end{bmatrix}, \qquad x_B = \begin{bmatrix} x_{B(1)} \\ \vdots \\ x_{B(m)} \end{bmatrix}.$$

The basic variables are determined by solving the equation $Bx_B = b$ whose unique solution is given by

$$x_B = B^{-1}b.$$

Example 2.1 Let the constraint $Ax = b$ be of the form

$$\begin{bmatrix} 1 & 1 & 2 & 1 & 0 & 0 & 0 \\ 0 & 1 & 6 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} x = \begin{bmatrix} 8 \\ 12 \\ 4 \\ 6 \end{bmatrix}.$$

Let us choose A_4, A_5, A_6, A_7 as our basic columns. Note that they are linearly independent and the corresponding basis matrix is the identity. We then obtain the basic solution $x = (0, 0, 0, 8, 12, 4, 6)$ which is nonnegative and, therefore, is a basic feasible solution. Another basis is obtained by choosing the columns A_3, A_5, A_6, A_7 (note that they are linearly independent). The corresponding basic solution is $x = (0, 0, 4, 0, -12, 4, 6)$, which is not feasible because $x_5 = -12 < 0$. □

Suppose now that there was an eighth column A_8, identical to A_7. Then, the two sets of columns $\{A_3, A_5, A_6, A_7\}$ and $\{A_3, A_5, A_6, A_8\}$ coincide. On the other hand the corresponding sets of basic indices, which are $\{3, 5, 6, 7\}$ and $\{3, 5, 6, 8\}$, are different and we have two different bases, according to our conventions.

For an intuitive view of basic solutions, recall our interpretation of the constraint $Ax = b$, or $\sum_{i=1}^{n} A_i x_i = b$, as a requirement to synthesize the vector $b \in \Re^m$ using the resource vectors A_i (Section 1.1). In a basic solution, we use only m of the resource vectors, those associated with the basic variables. Furthermore, in a basic feasible solution this is accomplished using a nonnegative amount of each basic vector; see Figure 2.8.

are adjacent: we have $n = 7$ and a total of six common linearly independent active constraints; these are $x_1 \geq 0$, $x_2 \geq 0$, and the four equality constraints.

The full row rank assumption on A

We close this section by showing that the full row rank assumption on the matrix A results in no loss of generality.

Theorem 2.5 *Let* $P = \{x \mid Ax = b,\ x \geq 0\}$ *be a nonempty polyhedron, where* A *is a matrix of dimensions* $m \times n$, *with rows* $a'_1, \ldots, a'_m$. *Suppose that* $\text{rank}(A) = k < m$ *and that the rows* $a'_{i_1}, \ldots, a'_{i_k}$ *are linearly independent. Consider the polyhedron*

$$Q = \{x \mid a'_{i_1} x = b_{i_1}, \ldots, a'_{i_k} x = b_{i_k},\ x \geq 0\}.$$

Then $Q = P$.

Proof. We provide the proof for the case where $i_1 = 1, \ldots, i_k = k$, that is, the first k rows of A are linearly independent. The general case can be reduced to this one by rearranging the rows of A.

Clearly $P \subset Q$ since any element of P automatically satisfies the constraints defining Q. We will now show that $Q \subset P$.

Since $\text{rank}(A) = k$, the row space of A has dimension k and the rows $a'_1, \ldots, a'_k$ form a basis of the row space. Therefore, every row a'_i of A can be expressed in the form $a'_i = \sum_{j=1}^{k} \lambda_{ij} a'_j$, for some scalars λ_{ij}. Let x be an element of P and note that

$$b_i = a'_i x = \sum_{j=1}^{k} \lambda_{ij} a'_j x = \sum_{j=1}^{k} \lambda_{ij} b_j, \qquad i = 1, \ldots, m.$$

Consider now an element y of Q. We will show that it belongs to P. Indeed, for any i,

$$a'_i y = \sum_{j=1}^{k} \lambda_{ij} a'_j y = \sum_{j=1}^{k} \lambda_{ij} b_j = b_i,$$

which establishes that $y \in P$ and $Q \subset P$.

Notice that the polyhedron Q in Theorem 2.5 is in standard form; namely, $Q = \{x \mid Dx = f,\ x \geq 0\}$ where D is a $k \times n$ submatrix of A, with rank equal to k, and f is a k-dimensional subvector of b. We conclude that as long as the feasible set is nonempty, a linear programming problem in standard form can be reduced to an equivalent standard form problem (with the same feasible set) in which the equality constraints are linearly independent. $\qquad\square$

Figure 2.8 Consider a standard form problem with $n = 4$ and $m = 2$, and let the vectors $b, A_1, \ldots, A_4$ be as shown. The vectors A_1, A_2 form a basis; the corresponding basic solution is infeasible because a negative value of x_2 is needed to synthesize b from A_1, A_2. The vectors A_1, A_3 form another basis; the corresponding basic solution is feasible. Finally, the vectors A_1, A_4 do not form a basis because they are linearly dependent.

Correspondence of bases and basic solutions

We now elaborate on the correspondence between basic solutions and bases. Different basic solutions must correspond to different bases, because a basis uniquely determines a basic solution. However, two different bases may lead to the same basic solution. (For an extreme example, if we have $b = 0$, then every basis matrix leads to the same basic solution, namely, the zero vector.) This phenomenon has some important algorithmic implications, and is closely related to degeneracy, which is the subject of the next section.

Adjacent basic solutions and adjacent bases

Recall that two distinct basic solutions are said to be adjacent if there are $n - 1$ linearly independent constraints that are active at both of them. For standard form problems, we also say that two bases are *adjacent* if they share all but one basic column. Then, it is not hard to check that adjacent basic solutions can always be obtained from two adjacent bases. Conversely, if two adjacent bases lead to distinct basic solutions, then the latter are adjacent.

Example 2.2 In reference to Example 2.1, the bases $\{A_4, A_5, A_6, A_7\}$ and $\{A_3, A_5, A_6, A_7\}$ are adjacent because all but one columns are the same. The corresponding basic solutions $x = (0, 0, 0, 8, 12, 4, 6)$ and $x = (0, 0, 4, 0, -12, 4, 6)$

Example 2.3 Consider the (nonempty) polyhedron defined by the constraints

$$2x_1 + x_2 + x_3 = 2$$
$$x_1 + x_2 = 1$$
$$x_1 + x_3 = 1$$
$$x_1, x_2, x_3 \geq 0.$$

The corresponding matrix A has rank two. This is because the last two rows $(1,1,0)$ and $(1,0,1)$ are linearly independent, but the first row is equal to the sum of the other two. Thus, the first constraint is redundant and after it is eliminated, we still have the same polyhedron.

2.4 Degeneracy

According to our definition, at a basic solution, we must have n linearly independent active constraints. This allows for the possibility that the number of active constraints is greater than n. (Of course, in n dimensions, no more than n of them can be linearly independent.) In this case, we say that we have a *degenerate* basic solution. In other words, at a degenerate basic solution, the number of active constraints is greater than the minimum necessary.

> **Definition 2.10** A basic solution $x \in \Re^n$ is said to be **degenerate** if more than n of the constraints are active at x.

In two dimensions, a degenerate basic solution is at the intersection of three or more lines; in three dimensions, a degenerate basic solution is at the intersection of four or more planes; see Figure 2.9 for an illustration. It turns out that the presence of degeneracy can strongly affect the behavior of linear programming algorithms and for this reason, we will now develop some more intuition.

Example 2.4 Consider the polyhedron P defined by the constraints

$$x_1 + x_2 + 2x_3 \leq 8$$
$$x_2 + 6x_3 \leq 12$$
$$x_1 \leq 4$$
$$x_2 \leq 6$$
$$x_1, x_2, x_3 \geq 0.$$

The vector $x = (2, 6, 0)$ is a nondegenerate basic feasible solution, because there are exactly three active and linearly independent constraints, namely, $x_1 + x_2 + 2x_3 \leq 8$, $x_2 \leq 6$, and $x_3 \geq 0$. The vector $x = (4,0,2)$ is a degenerate basic feasible solution, because there are four active constraints, three of them linearly independent, namely, $x_1 + x_2 + 2x_3 \leq 8$, $x_2 + 6x_3 \leq 12$, $x_1 \leq 4$, and $x_2 \geq 0$.

(a)

(b)
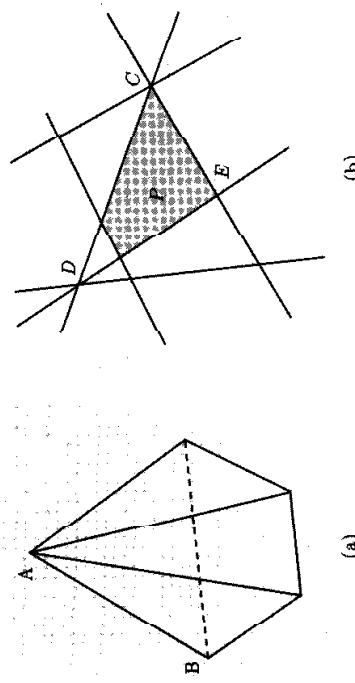
Figure 2.9: The points A and C are degenerate basic feasible solutions. The points B and E are nondegenerate basic feasible solutions. The point D is a degenerate basic solution.

Degeneracy in standard form polyhedra

At a basic solution of a polyhedron in standard form, the m equality constraints are always active. Therefore, having more than n active constraints is the same as having more than $n - m$ variables at zero level. This leads us to the next definition which is a special case of Definition 2.10.

> **Definition 2.11** Consider the standard form polyhedron $P = \{x \in \Re^n \mid Ax = b, \ x \geq 0\}$ and let x be a basic solution. Let m be the number of rows of A. The vector x is a **degenerate basic solution** if more than $n - m$ of the components of x are zero.

Example 2.5 Consider once more the polyhedron of Example 2.4. By introducing the slack variables $x_4,\ldots,x_7$, we can transform it into the standard form $P = \{x = (x_1,\ldots,x_7) \mid Ax = b, \ x \geq 0\}$, where

$$A = \begin{bmatrix} 1 & 1 & 2 & 1 & 0 & 0 & 0 \\ 0 & 1 & 6 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \qquad b = \begin{bmatrix} 8 \\ 12 \\ 4 \\ 6 \end{bmatrix}.$$

Consider the basis consisting of the linearly independent columns A_1, A_2, A_3, A_7. To calculate the corresponding basic solution, we first set the nonbasic variables x_4, x_5, and x_6 to zero, and then solve the system $Ax = b$ for the remaining variables to obtain $x = (4, 0, 2, 0, 0, 0, 6)$. This is a degenerate basic feasible solution, because we have a total of four variables that are zero, whereas

$n - m = 7 - 4 = 3$. Thus, while we initially set only the three nonbasic variables to zero, the solution to the system $Ax = b$ turned out to satisfy one more of the constraints (namely, the constraint $x_2 \geq 0$) with equality. Consider now the basis consisting of the linearly independent columns A_1, A_3, A_4, and A_7. The corresponding basic feasible solution is again $x = (4, 0, 2, 0, 0, 0, 6)$.

The preceding example suggests that we can think of degeneracy in the following terms. We pick a basic solution by picking n linearly independent constraints to be satisfied with equality, and we realize that certain other constraints are also satisfied with equality. If the entries of A or b were chosen at random, this would almost never happen. Also, Figure 2.10 illustrates that if the coefficients of the active constraints are slightly perturbed, degeneracy can disappear (cf. Exercise 2.18). In practical problems, however, the entries of A and b often have a special (nonrandom) structure, and degeneracy is more common than the preceding argument would seem to suggest.
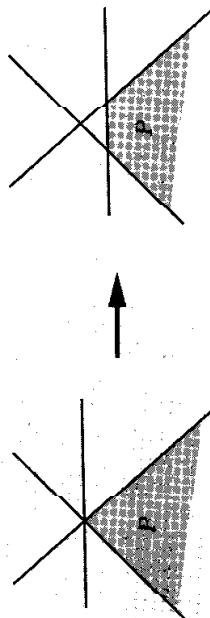
Figure 2.10: Small changes in the constraining inequalities can remove degeneracy.

In order to visualize degeneracy in standard form polyhedra, we assume that $n - m = 2$ and we draw the feasible set as a subset of the two-dimensional set defined by the equality constraints $Ax = b$; see Figure 2.11. At a nondegenerate basic solution, exactly $n - m$ of the constraints $x_i \geq 0$ are active; the corresponding variables are nonbasic. In the case of a degenerate basic solution, more than $n - m$ of the constraints $x_i \geq 0$ are active, and there are usually several ways of choosing which $n - m$ variables to call nonbasic; in that case, there are several bases corresponding to that same basic solution. (This discussion refers to the typical case. However, there are examples of degenerate basic solutions to which there corresponds only one basis.)

Degeneracy is not a purely geometric property

We close this section by pointing out that degeneracy of basic feasible solutions is not, in general, a geometric (representation independent) property,
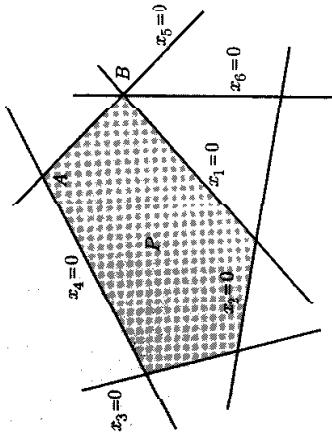
Figure 2.11: An $(n - m)$-dimensional illustration of degeneracy. Here, $n = 6$ and $m = 4$. The basic feasible solution A is nondegenerate and the basic variables are x_1, x_2, x_3, x_6. The basic feasible solution B is degenerate. We can choose x_1, x_6 as the nonbasic variables. Other possibilities are to choose x_1, x_5, or to choose x_5, x_4. Thus, there are three possible bases, for the same basic feasible solution B.

but rather it may depend on the particular representation of a polyhedron. To illustrate this point, consider the standard form polyhedron (cf. Figure 2.12)

$$P = \left\{(x_1, x_2, x_3) \mid x_1 - x_2 = 0, \ x_1 + x_2 + 2x_3 = 2, \ x_1, x_2, x_3 \geq 0\right\}.$$

We have $n = 3$, $m = 2$ and $n - m = 1$. The vector $(1, 1, 0)$ is nondegenerate because only one variable is zero. The vector $(0, 0, 1)$ is degenerate because two variables are zero. However, the same polyhedron can also be described
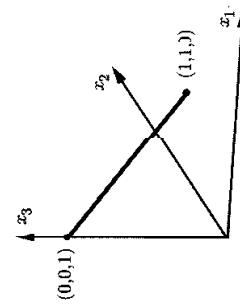
Figure 2.12: An example of degeneracy in a standard form problem.
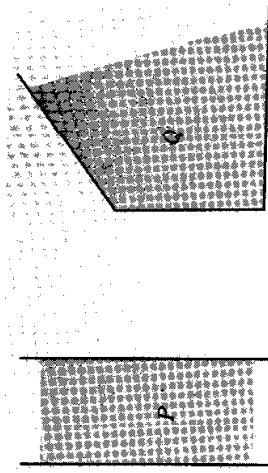
Figure 2.13: The polyhedron P contains a line and does not have an extreme point, while Q does not contain a line and has extreme points.

in the (nonstandard) form

$$P = \left\{ (x_1, x_2, x_3) \mid x_1 - x_2 = 0,\ x_1 + x_2 + 2x_3 = 2,\ x_1 \geq 0,\ x_3 \geq 0 \right\}.$$

The vector $(0,0,1)$ is now a nondegenerate basic feasible solution, because there are only three active constraints.

For another example, consider a nondegenerate basic feasible solution x^* of a standard form polyhedron $P = \{x \mid Ax = b,\ x \geq 0\}$, where A is of dimensions $m \times n$. In particular, exactly $n - m$ of the variables x_i^* are equal to zero. Let us now represent P in the form $P = \{x \mid Ax \geq b,\ -Ax \geq -b,\ x \geq 0\}$. Then, at the basic feasible solution x^*, we have $n - m$ variables set to zero and an additional $2m$ inequality constraints are satisfied with equality. We therefore have $n + m$ active constraints and x^* is degenerate. Hence, under the second representation, every basic feasible solution is degenerate.

We have established that a degenerate basic feasible solution under one representation could be nondegenerate under another representation. Still, it can be shown that if a basic feasible solution is degenerate under one particular standard form representation, then it is degenerate under every standard form representation of the same polyhedron (Exercise 2.19).

2.5 Existence of extreme points

We obtain in this section necessary and sufficient conditions for a polyhedron to have at least one extreme point. We first observe that not every polyhedron has this property. For example, if $n > 1$, a halfspace in $\Re^n$ is a polyhedron without extreme points. Also, as argued in Section 2.2 (cf. the discussion after Definition 2.9), if the matrix A has fewer than n rows, then the polyhedron $\{x \in \Re^n \mid Ax \geq b\}$ does not have a basic feasible solution.

It turns out that the existence of an extreme point depends on whether a polyhedron contains an infinite line or not; see Figure 2.13. We need the following definition.

Definition 2.12 *A polyhedron $P \subset \Re^n$ contains a line if there exists a vector $x \in P$ and a nonzero vector $d \in \Re^n$ such that $x + \lambda d \in P$ for all scalars λ.*

We then have the following result.

Theorem 2.6 *Suppose that the polyhedron $P = \{x \in \Re^n \mid a_i'x \geq b_i,\ i = 1, \ldots, m\}$ is nonempty. Then, the following are equivalent:*

(a) The polyhedron P has at least one extreme point.

(b) The polyhedron P does not contain a line.

(c) There exist n vectors out of the family $a_1, \ldots, a_n$, which are linearly independent.

Proof.

(b) $\Rightarrow$ (a)

We first prove that if P does not contain a line, then it has a basic feasible solution and, therefore, an extreme point. A geometric interpretation of this proof is provided in Figure 2.14.

Let x be an element of P and let $I = \{i \mid a_i'x = b_i\}$. If n of the vectors a_i, $i \in I$, corresponding to the active constraints are linearly independent, then x is, by definition, a basic feasible solution and, therefore, a basic feasible solution exists. If this is not the case, then all of the vectors a_i, $i \in I$, lie in a proper subspace of $\Re^n$ and there exists a nonzero vector $d \in \Re^n$ such that $a_i'd = 0$, for every $i \in I$. Let us consider the line consisting of all points of the form $y = x + \lambda d$, where λ is an arbitrary scalar. For $i \in I$, we have $a_i'y = a_i'x + \lambda a_i'd = a_i'x = b_i$. Thus, those constraints that were active at x remain active at all points on the line. However, since the polyhedron is assumed to contain no lines, it follows that as we vary λ, some constraint will be eventually violated. At the point where some constraint is about to be violated, a new constraint must become active, and we conclude that there exists some λ^* and some $j \notin I$ such that $a_j'(x + \lambda^*d) = b_j$.

We claim that a_j is not a linear combination of the vectors a_i, $i \in I$. Indeed, we have $a_j'x \neq b_j$ (because $j \notin I$) and $a_j'(x + \lambda^*d) = b_j$ (by the definition of λ^*). Thus, $a_j'd \neq 0$. On the other hand, $a_i'd = 0$ for every $i \in I$ (by the definition of d) and therefore, d is orthogonal to any linear combination of the vectors a_i, $i \in I$. Since d is not orthogonal to a_j, we
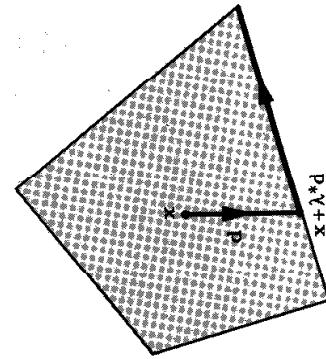
Figure 2.14: Starting from an arbitrary point of a polyhedron, we choose a direction along which all currently active constraints remain active. We then move along that direction until a new constraint is about to be violated. At that point, the number of linearly independent active constraints has increased by at least one. We repeat this procedure until we end up with n linearly independent active constraints, at which point we have a basic feasible solution.

conclude that a_j is a not a linear combination of the vectors a_i, $i \in I$. Thus, by moving from x to $x + \lambda^* d$, the number of linearly independent active constraints has been increased by at least one. By repeating the same argument, as many times as needed, we eventually end up with a point at which there are n linearly independent active constraints. Such a point is, by definition, a basic solution; it is also feasible since we have stayed within the feasible set.

(a) ⇒ (c)

If P has an extreme point x, then x is also a basic feasible solution (cf. Theorem 2.3), and there exist n constraints that are active at x, with the corresponding vectors a_i being linearly independent.

(c) ⇒ (b)

Suppose that n of the vectors a_i are linearly independent and, without loss of generality, let us assume that $a_1, \ldots, a_n$ are linearly independent. Suppose that P contains a line $x + \lambda d$, where d is a nonzero vector. We then have $a_i'(x + \lambda d) \geq b_i$ for all i and all λ. We conclude that $a_i' d = 0$ for all i. (If $a_i' d < 0$, we can violate the constraint by picking λ very large; a symmetric argument applies if $a_i' d > 0$.) Since the vectors a_i, $i = 1, \ldots, n$, are linearly independent, this implies that $d = 0$. This is a contradiction and establishes that P does not contain a line □

Notice that a bounded polyhedron does not contain a line. Similarly,

---

the positive orthant $\{x \mid x \geq 0\}$ does not contain a line. Since a polyhedron in standard form is contained in the positive orthant, it does not contain a line either. These observations establish the following important corollary of Theorem 2.6.

Corollary 2.2 *Every nonempty bounded polyhedron and every nonempty polyhedron in standard form has at least one basic feasible solution.*

2.6 Optimality of extreme points

Having established the conditions for the existence of extreme points, we will now confirm the intuition developed in Chapter 1: as long as a linear programming problem has an optimal solution and as long as the feasible set has at least one extreme point, we can always find an optimal solution within the set of extreme points of the feasible set. Later in this section, we prove a somewhat stronger result, at the expense of a more complicated proof.

Theorem 2.7 *Consider the linear programming problem of minimizing $c'x$ over a polyhedron P. Suppose that P has at least one extreme point and that there exists an optimal solution. Then, there exists an optimal solution which is an extreme point of P.*

Proof. (See Figure 2.15 for an illustration.) Let Q be the set of all optimal solutions, which we have assumed to be nonempty. Let P be of the form $P = \{x \in \Re^n \mid Ax \geq b\}$ and let v be the optimal value of the cost $c'x$. Then, $Q = \{x \in \Re^n \mid Ax \geq b, \; c'x = v\}$, which is also a polyhedron Since

Figure 2.15: Illustration of the proof of Theorem 2.7. Here, Q is the set of optimal solutions and an extreme point x^* of Q is also an extreme point of P.

$Q \subset P$, and since P contains no lines (cf. Theorem 2.6), Q contains no lines either. Therefore, Q has an extreme point.

Let $\mathbf{x}^*$ be an extreme point of Q. We will show that $\mathbf{x}^*$ is also an extreme point of P. Suppose, in order to derive a contradiction, that $\mathbf{x}^*$ is not an extreme point of P. Then, there exist $\mathbf{y} \in P$, $\mathbf{z} \in P$, such that $\mathbf{y} \neq \mathbf{x}^*$, $\mathbf{z} \neq \mathbf{x}^*$, and some $\lambda \in [0,1]$ such that $\mathbf{x}^* = \lambda \mathbf{y} + (1-\lambda)\mathbf{z}$. It follows that $v = \mathbf{c}'\mathbf{x}^* = \lambda \mathbf{c}'\mathbf{y} + (1-\lambda)\mathbf{c}'\mathbf{z}$. Furthermore, since v is the optimal cost, $\mathbf{c}'\mathbf{y} \geq v$ and $\mathbf{c}'\mathbf{z} \geq v$. This implies that $\mathbf{c}'\mathbf{y} = v$ and therefore $\mathbf{z} \in Q$ and $\mathbf{y} \in Q$. But this contradicts the fact that $\mathbf{x}^*$ is an extreme point of Q. The contradiction establishes that $\mathbf{x}^*$ is an extreme point of P. In addition, since $\mathbf{x}^*$ belongs to Q, it is optimal.

The above theorem applies to polyhedra in standard form, as well as to bounded polyhedra, since they do not contain a line.

Our next result is stronger than Theorem 2.7. It shows that the existence of an optimal solution can be taken for granted, as long as the optimal cost is finite.

Theorem 2.8 *Consider the linear programming problem of minimizing $\mathbf{c}'\mathbf{x}$ over a polyhedron P. Suppose that P has at least one extreme point. Then, either the optimal cost is equal to $-\infty$, or there exists an extreme point which is optimal.*

Proof. The proof is essentially a repetition of the proof of Theorem 2.6. The difference is that as we move towards a basic feasible solution, we will also make sure that the costs do not increase. We will use the following terminology: an element $\mathbf{x}$ of P has *rank* k if we can find k, but not more than k, linearly independent constraints that are active at $\mathbf{x}$.

Let us assume that the optimal cost is finite. Let $P = \{\mathbf{x} \in \Re^n \mid A\mathbf{x} \geq \mathbf{b}\}$ and consider some $\mathbf{x} \in P$ of rank $k < n$. We will show that there exists some $\mathbf{y} \in P$ which has greater rank and satisfies $\mathbf{c}'\mathbf{y} \leq \mathbf{c}'\mathbf{x}$. Let $I = \{i \mid \mathbf{a}_i'\mathbf{x} = b_i\}$, where $\mathbf{a}_i'$ is the ith row of A. Since $k < n$, the vectors $\mathbf{a}_i$, $i \in I$, lie in a proper subspace of $\Re^n$, and we can choose some nonzero $\mathbf{d} \in \Re^n$ orthogonal to every $\mathbf{a}_i$, $i \in I$. Furthermore, by possibly taking the negative of $\mathbf{d}$, we can assume that $\mathbf{c}'\mathbf{d} \leq 0$.

Suppose that $\mathbf{c}'\mathbf{d} < 0$. Let us consider the half-line $\mathbf{y} = \mathbf{x} + \lambda\mathbf{d}$, where λ is a positive scalar. As in the proof of Theorem 2.6, all points on this half-line satisfy the relations $\mathbf{a}_i'\mathbf{y} = b_i$, $i \in I$. If the entire half-line were contained in P, the optimal cost would be $-\infty$ which we have assumed not to be the case. Therefore, the half-line eventually exits P. When this is about to happen, we have some $\lambda^* > 0$ and $j \notin I$ such that $\mathbf{a}_j'(\mathbf{x} + \lambda^*\mathbf{d}) = b_j$, and note that $\mathbf{c}'\mathbf{y} < \mathbf{c}'\mathbf{x}$. As in the proof of Theorem 2.6, $\mathbf{a}_j$ is linearly independent from $\mathbf{a}_i$, $i \in I$, and the rank of $\mathbf{y}$ is at least $k+1$.

Suppose now that $\mathbf{c}'\mathbf{d} = 0$. We consider the line $\mathbf{y} = \mathbf{x} + \lambda\mathbf{d}$, where λ is an arbitrary scalar. Since P contains no lines, the line must eventually exit P and when that is about to happen, we are again at a vector $\mathbf{y}$ of rank greater than that of $\mathbf{x}$. Furthermore, since $\mathbf{c}'\mathbf{d} = 0$, we have $\mathbf{c}'\mathbf{y} = \mathbf{c}'\mathbf{x}$.

In either case, we have found a new point $\mathbf{y}$ such that $\mathbf{c}'\mathbf{y} \leq \mathbf{c}'\mathbf{x}$, and whose rank is greater than that of $\mathbf{x}$. By repeating this process as many times as needed, we end up with a vector $\mathbf{w}$ of rank n (thus, $\mathbf{w}$ is a basic feasible solution) such that $\mathbf{c}'\mathbf{w} \leq \mathbf{c}'\mathbf{x}$.

Let $\mathbf{w}^1, \ldots, \mathbf{w}^r$ be the basic feasible solutions in P and let $\mathbf{w}^*$ be a basic feasible solution such that $\mathbf{c}'\mathbf{w}^* \leq \mathbf{c}'\mathbf{w}^i$ for all i. We have already shown that for every $\mathbf{x}$ there exists some i such that $\mathbf{c}'\mathbf{w}^i \leq \mathbf{c}'\mathbf{x}$. It follows that $\mathbf{c}'\mathbf{w}^* \leq \mathbf{c}'\mathbf{x}$ for all $\mathbf{x} \in P$, and the basic feasible solution $\mathbf{w}^*$ is optimal. □

For a general linear programming problem, if the feasible set has no extreme points, then Theorem 2.8 does not apply directly. On the other hand, any linear programming problem can be transformed into an equivalent problem in standard form to which Theorem 2.8 does apply. This establishes the following corollary:

Corollary 2.3 *Consider the linear programming problem of minimizing $\mathbf{c}'\mathbf{x}$ over a nonempty polyhedron. Then, either the optimal cost is equal to $-\infty$ or there exists an optimal solution.*

The result in Corollary 2.3 should be contrasted with what may happen in optimization problems with a nonlinear cost function. For example, in the problem of minimizing $1/x$ subject to $x \geq 1$, the optimal cost is not $-\infty$, but an optimal solution does not exist.

2.7 Representation of bounded polyhedra*

So far, we have been representing polyhedra in terms of their defining inequalities. In this section, we provide an alternative, by showing that a bounded polyhedron can also be represented as the convex hull of its extreme points. The proof that we give here is elementary and constructive, and its main idea is summarized in Figure 2.16. There is a similar representation of unbounded polyhedra involving extreme points and "extreme rays" (edges that extend to infinity). This representation can be developed using the tools that we already have, at the expense of a more complicated proof. A more elegant argument, based on duality theory, will be presented in Section 4.9 and will also result in an alternative proof of Theorem 2.9 below.

Figure 2.16: Given the vector z, we express it as a convex combination of y and u. The vector u belongs to the polyhedron Q whose dimension is lower than that of P. Using induction on dimension, we can express the vector u as a convex combination of extreme points of Q. These are also extreme points of P.

Theorem 2.9 *A nonempty and bounded polyhedron is the convex hull of its extreme points.*

Proof. Every convex combination of extreme points is an element of the polyhedron, since polyhedra are convex sets. Thus, we only need to prove the converse result and show that every element of a bounded polyhedron can be represented as a convex combination of extreme points.

We define the *dimension* of a polyhedron $P \subset \Re^n$ as the smallest integer k such that P is contained in some k-dimensional affine subspace of $\Re^n$. (Recall from Section 1.5, that a k-dimensional affine subspace is a translation of a k-dimensional subspace.) Our proof proceeds by induction on the dimension of the polyhedron P. If P is zero-dimensional, it consists of a single point. This point is an extreme point of P and the result is true.

Let us assume that the result is true for all polyhedra of dimension less than k. Let $P = \{x \in \Re^n \mid a_i'x \geq b_i, i = 1, \ldots, m\}$ be a nonempty bounded k-dimensional polyhedron. Then, P is contained in a k-dimensional affine subspace S of $\Re^n$, which can be assumed to be of the form

$$S = \{x^0 + \lambda_1 x^1 + \cdots + \lambda_k x^k \mid \lambda_1, \ldots, \lambda_k \in \Re\},$$

where $x^1, \ldots, x^k$ are some vectors in $\Re^n$. Let $f_1, \ldots, f_{n-k}$ be $n-k$ linearly independent vectors that are orthogonal to $x^1, \ldots, x^k$. Let $g_i = f_i'x^0$, for

$i = 1, \ldots, n-k$. Then, every element x of S satisfies

$$f_i'x = g_i, \quad i = 1, \ldots, n-k. \quad (2.3)$$

Since $P \subset S$, the same must be true for every element of P.

Let z be an element of P. If z is an extreme point of P, then z is a trivial convex combination of the extreme points of P and there is nothing more to be proved. If z is not an extreme point of P, let us choose an arbitrary extreme point y of P and form the half-line consisting of all points of the form $z + \lambda(z - y)$, where λ is a nonnegative scalar. Since P is bounded, this half-line must eventually exit P and violate one of the constraints, say the constraint $a_{i^*}'x \geq b_{i^*}$. By considering what happens when this constraint is just about to be violated, we find some $\lambda^* \geq 0$ and $u \in P$, such that

$$u = z + \lambda^*(z - y),$$

and

$$a_{i^*}'u = b_{i^*}.$$

Since the constraint $a_{i^*}'x \geq b_{i^*}$ is violated if λ grows beyond λ^*, it follows that $a_{i^*}'(z - y) < 0$.

Let Q be the polyhedron defined by

$$Q = \{x \in P \mid a_{i^*}'x = b_{i^*}\}$$
$$= \{x \in \Re^n \mid a_i'x \geq b_i, i = 1, \ldots, m, a_{i^*}'x = b_{i^*}\}.$$

Since $z, y \in P$, we have $f_i'z = g_i = f_i'y$ which shows that $z - y$ is orthogonal to each vector f_i, for $i = 1, \ldots, n-k$. On the other hand, we have shown that $a_{i^*}'(z - y) < 0$, which implies that the vector a_{i^*} is not a linear combination of, and is therefore linearly independent from, the vectors f_i. Note that

$$Q \subset \{x \in \Re^n \mid a_i'x = b_i, f_i'x = g_i, i = 1, \ldots, n-k\},$$

since Eq. (2.3) holds for every element of P. The set on the right is defined by $n-k+1$ linearly independent equality constraints. Hence, it is an affine subspace of dimension $k-1$ (see the discussion at the end of Section 1.5). Therefore, Q has dimension at most $k-1$.

By applying the induction hypothesis to Q and u, we see that u can be expressed as a convex combination

$$u = \sum_i \lambda_i v^i$$

of the extreme points v^i of Q, where λ_i are nonnegative scalars that sum to one. Note that at an extreme point v of Q, we must have $a_i'v = b_i$ for n linearly independent vectors a_i; therefore, v must also be an extreme point of P. Using the definition of λ^*, we also have

$$z = \frac{u + \lambda^* y}{1 + \lambda^*}.$$

Therefore,
$$z = \frac{\lambda^* \mathbf{y}}{1+\lambda^*} + \sum_i \frac{\lambda_i}{1+\lambda^*} \mathbf{v}^i,$$

which shows that z is a convex combination of the extreme points of P. □

Example 2.6 Consider the polyhedron
$$P = \{(x_1, x_2, x_3) \mid x_1 + x_2 + x_3 \le 1, \ x_1, x_2, x_3 \ge 0\}.$$

It has four extreme points, namely, $\mathbf{x}^1 = (1,0,0)$, $\mathbf{x}^2 = (0,1,0)$, $\mathbf{x}^3 = (0,0,1)$, and $\mathbf{x}^4 = (0,0,0)$. The vector $\mathbf{x} = (1/3, 1/3, 1/4)$ belongs to P. It can be represented as
$$\mathbf{x} = \frac{1}{3}\mathbf{x}^1 + \frac{1}{3}\mathbf{x}^2 + \frac{1}{4}\mathbf{x}^3 + \frac{1}{12}\mathbf{x}^4.$$

There is a converse to Theorem 2.9 asserting that the convex hull of a finite number of points is a polyhedron. This result is proved in the next section and again in Section 4.9.

2.8 Projections of polyhedra: Fourier-Motzkin elimination*

In this section, we present perhaps the oldest method for solving linear programming problems. This method is not practical because it requires a very large number of steps, but it has some interesting theoretical corollaries.

The key to this method is the concept of a *projection*, defined as follows: if $\mathbf{x} = (x_1, \ldots, x_n)$ is a vector in $\Re^n$ and $k \le n$, the projection mapping $\pi_k : \Re^n \mapsto \Re^k$ projects $\mathbf{x}$ onto its first k coordinates:
$$\pi_k(\mathbf{x}) = \pi_k(x_1, \ldots, x_n) = (x_1, \ldots, x_k).$$

We also define the projection $\Pi_k(S)$ of a set $S \subset \Re^n$ by letting
$$\Pi_k(S) = \{\pi_k(\mathbf{x}) \mid \mathbf{x} \in S\};$$

see Figure 2.17 for an illustration. Note that S is nonempty if and only if $\Pi_k(S)$ is nonempty. An equivalent definition is
$$\Pi_k(S) = \Big\{(x_1, \ldots, x_k) \mid \text{there exist } x_{k+1}, \ldots, x_n \text{ s.t. } (x_1, \ldots, x_n) \in S\Big\}.$$

Suppose now that we wish to decide whether a given polyhedron $P \subset \Re^n$ is nonempty. If we can somehow eliminate the variable x_n and construct the set $\Pi_{n-1}(P) \subset \Re^{n-1}$, we can instead consider the presumably easier problem of deciding whether $\Pi_{n-1}(P)$ is nonempty. If we keep eliminating variables one by one, we eventually arrive at the set $\Pi_1(P)$ that
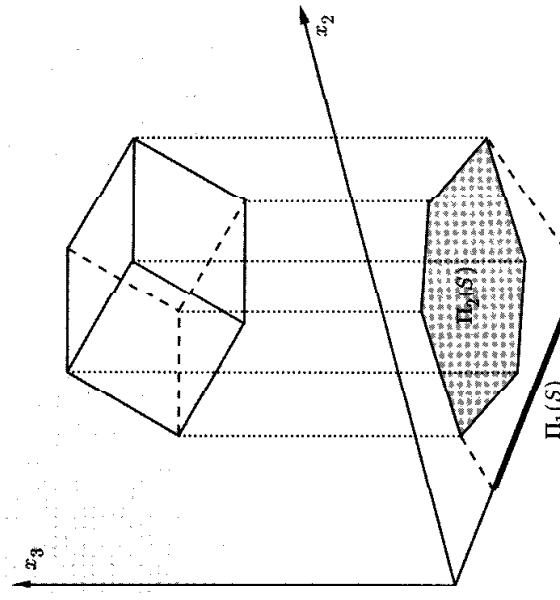
Figure 2.17: The projections $\Pi_1(S)$ and $\Pi_2(S)$ of a rotated three-dimensional cube.

involves a single variable, and whose emptiness is easy to check. The main disadvantage of this method is that while each step reduces the dimension by one, a large number of constraints is usually added. Exercise 2.20 deals with a family of examples in which the number of constraints increases exponentially with the problem dimension.

We now describe the elimination method. We are given a polyhedron P in terms of linear inequality constraints of the form
$$\sum_{j=1}^n a_{ij} x_j \ge b_i, \quad i = 1, \ldots, m.$$

We wish to eliminate x_n and construct the projection $\Pi_{n-1}(P)$.

Elimination algorithm

1. Rewrite each constraint $\sum_{j=1}^n a_{ij}x_j \geq b_i$ in the form

$$a_{in}x_n \geq -\sum_{j=1}^{n-1} a_{ij}x_j + b_i, \quad i = 1, \ldots, m;$$

if $a_{in} \neq 0$, divide both sides by a_{in}. By letting $\bar{x} = (x_1, \ldots, x_{n-1})$, we obtain an equivalent representation of P involving the following constraints:

$$x_n \geq d_i + \mathbf{f}_i'\bar{x}, \qquad \text{if } a_{in} > 0, \qquad (2.4)$$
$$d_j + \mathbf{f}_j'\bar{x} \geq x_n, \qquad \text{if } a_{jn} < 0, \qquad (2.5)$$
$$0 \geq d_k + \mathbf{f}_k'\bar{x}, \qquad \text{if } a_{kn} = 0. \qquad (2.6)$$

Here, each d_i, d_j, d_k is a scalar, and each $\mathbf{f}_i$, $\mathbf{f}_j$, $\mathbf{f}_k$ is a vector in $\Re^{n-1}$.

2. Let Q be the polyhedron in $\Re^{n-1}$ defined by the constraints

$$d_j + \mathbf{f}_j'\bar{x} \geq d_i + \mathbf{f}_i'\bar{x}, \qquad \text{if } a_{in} > 0 \text{ and } a_{jn} < 0, \qquad (2.7)$$
$$0 \geq d_k + \mathbf{f}_k'\bar{x}, \qquad \text{if } a_{kn} = 0. \qquad (2.8)$$

Example 2.7 Consider the polyhedron defined by the constraints

$$x_1 + x_2 \geq 1$$
$$x_1 + x_2 + 2x_3 \geq 2$$
$$2x_1 + 3x_3 \geq 3$$
$$x_1 - 4x_3 \geq 4$$
$$-2x_1 + x_2 - x_3 \geq 5.$$

We rewrite these constraints in the form

$$0 \geq 1 - x_1 - x_2$$
$$x_3 \geq 1 - (x_1/2) - (x_2/2)$$
$$x_3 \geq 1 - (2x_1/3)$$
$$-1 + (x_1/4) \geq x_3$$
$$-5 - 2x_1 + x_2 \geq x_3.$$

Then, the set Q is defined by the constraints

$$0 \geq 1 - x_1 - x_2$$
$$-1 + x_1/4 \geq 1 - (x_1/2) - (x_2/2)$$

$$-1 + x_1/4 \geq 1 - (2x_1/3)$$
$$-5 - 2x_1 + x_2 \geq 1 - (x_1/2) - (x_2/2)$$
$$-5 - 2x_1 + x_2 \geq 1 - (2x_1/3).$$

Theorem 2.10 *The polyhedron Q constructed by the elimination algorithm is equal to the projection $\Pi_{n-1}(P)$ of P.*

Proof. If $\bar{x} \in \Pi_{n-1}(P)$, there exists some x_n such that $(\bar{x}, x_n) \in P$. In particular, the vector $x = (\bar{x}, x_n)$ satisfies Eqs. (2.4)-(2.6), from which it follows immediately that $\bar{x}$ satisfies Eqs. (2.7)-(2.3), and $\bar{x} \in Q$. This shows that $\Pi_{n-1}(P) \subset Q$.

We will now prove that $Q \subset \Pi_{n-1}(P)$. Let $\bar{x} \in Q$. It follows from Eq. (2.7) that

$$\min_{\{j|a_{jn}<0\}} (d_j + \mathbf{f}_j'\bar{x}) \geq \max_{\{i|a_{in}>0\}} (d_i + \mathbf{f}_i'\bar{x}).$$

Let x_n be any number between the two sides of the above inequality. It then follows that $(\bar{x}, x_n)$ satisfies Eqs. (2.4)-(2.6) and, therefore, belongs to the polyhedron P. $\square$

Notice that for any vector $x = (x_1, \ldots, x_n)$, we have

$$\pi_{n-2}(\pi_{n-1}(x)) = (x_1, \ldots, x_{n-2}) = \pi_{n-2}(x).$$

Accordingly, for any polyhedron P, we also have

$$\Pi_{n-2}(\Pi_{n-1}(P)) = \Pi_{n-2}(P).$$

By generalizing this observation, we see that if we apply the elimination algorithm k times, we end up with the set $\Pi_{n-k}(P)$; if we apply it $n-1$ times, we end up with $\Pi_1(P)$. Unfortunately, each application of the elimination algorithm can increase the number of constraints substantially, leading to a polyhedron $\Pi_1(P)$ described by a very large number of constraints. Of course, since $\Pi_1(P)$ is one-dimensional, almost all of these constraints will be redundant, but this is of no help: in order to decide which ones are redundant, we must, in general, enumerate them.

The elimination algorithm has an important theoretical consequence: since the projection $\Pi_k(P)$ can be generated by repeated application of the elimination algorithm, and since the elimination algorithm always produces a polyhedron, it follows that a projection $\Pi_k(P)$ of a polyhedron is also a polyhedron. This fact might be considered obvious, but a proof simpler than the one we gave is not apparent. We now restate it in somewhat different language

Corollary 2.4 *Let $P \subset \Re^{n+k}$ be a polyhedron. Then, the set*
$$\{\mathbf{x} \in \Re^n \mid \text{there exists } \mathbf{y} \in \Re^k \text{ such that } (\mathbf{x},\mathbf{y}) \in P\}$$
is also a polyhedron.

A variation of Corollary 2.4 states that the image of a polyhedron under a linear mapping is also a polyhedron.

Corollary 2.5 *Let $P \subset \Re^n$ be a polyhedron and let $\mathbf{A}$ be an $m \times n$ matrix. Then, the set $Q = \{\mathbf{A}\mathbf{x} \mid \mathbf{x} \in P\}$ is also a polyhedron.*

Proof. We have $Q = \{\mathbf{y} \in \Re^m \mid \text{there exists } \mathbf{x} \in \Re^n \text{ such that } \mathbf{A}\mathbf{x} = \mathbf{y},\ \mathbf{x} \in P\}$. Therefore, Q is the projection of the polyhedron $\{(\mathbf{x},\mathbf{y}) \in \Re^{n+m} \mid \mathbf{A}\mathbf{x} = \mathbf{y},\ \mathbf{x} \in P\}$ onto the $\mathbf{y}$ coordinates. □

Corollary 2.6 *The convex hull of a finite number of vectors is a polyhedron.*

Proof. The convex hull
$$\left\{ \sum_{i=1}^{k} \lambda_i \mathbf{x}^i \ \middle|\ \sum_{i=1}^{k} \lambda_i = 1,\ \lambda_i \geq 0 \right\}$$
of a finite number of vectors $\mathbf{x}^1,\dots,\mathbf{x}^k$ is the image of the polyhedron
$$\left\{ (\lambda_1,\dots,\lambda_k) \ \middle|\ \sum_{i=1}^{k} \lambda_i = 1,\ \lambda_i \geq 0 \right\}$$
under the linear mapping that maps $(\lambda_1,\dots,\lambda_k)$ to $\sum_{i=1}^{k} \lambda_i \mathbf{x}^i$ and is, therefore, a polyhedron. □

We finally indicate how the elimination algorithm can be used to solve linear programming problems. Consider the problem of minimizing $\mathbf{c}'\mathbf{x}$ subject to $\mathbf{x}$ belonging to a polyhedron P. We define a new variable x_0 and introduce the constraint $x_0 = \mathbf{c}'\mathbf{x}$. If we use the elimination algorithm n times to eliminate the variables $x_1,\dots,x_n$, we are left with the set
$$Q = \{x_0 \mid \text{there exists } \mathbf{x} \in P \text{ such that } x_0 = \mathbf{c}'\mathbf{x}\},$$
and the optimal cost is equal to the smallest element of Q. An optimal solution $\mathbf{x}$ can be recovered by backtracking (Exercise 2.21).

2.9 Summary

We summarize our main conclusions so far regarding the solutions to linear programming problems.

(a) If the feasible set is nonempty and bounded, there exists an optimal solution. Furthermore, there exists an optimal solution which is an extreme point.

(b) If the feasible set is unbounded, there are the following possibilities:

(i) There exists an optimal solution which is an extreme point.

(ii) There exists an optimal solution, but no optimal solution is an extreme point. (This can only happen if the feasible set has no extreme points; it never happens when the problem is in standard form.)

(iii) The optimal cost is $-\infty$.

Suppose now that the optimal cost is finite and that the feasible set contains at least one extreme point. Since there are only finitely many extreme points, the problem can be solved in a finite number of steps, by enumerating all extreme points and evaluating the cost of each one. This is hardly a practical algorithm because the number of extreme points can increase exponentially with the number of variables and constraints. In the next chapter, we will exploit the geometry of the feasible set and develop the *simplex method*, a systematic procedure that moves from one extreme point to another, without having to enumerate all extreme points.

An interesting aspect of the material in this chapter is the distinction between geometric (representation independent) properties of a polyhedron and those properties that depend on a particular representation. In that respect, we have established the following:

(a) Whether or not a point is an extreme point (equivalently, vertex, or basic feasible solution) is a geometric property.

(b) Whether or not a point is a basic solution may depend on the way that a polyhedron is represented.

(c) Whether or not a basic or basic feasible solution is degenerate may depend on the way that a polyhedron is represented.

2.10 Exercises

Exercise 2.1 For each one of the following sets, determine whether it is a polyhedron.

(a) The set of all $(x,y) \in \Re^2$ satisfying the constraints

$$x \cos \theta + y \sin \theta \leq 1, \qquad \forall\, \theta \in [0, \pi/2],$$
$$x \geq 0,$$
$$y \geq 0.$$

(b) The set of all $x \in \Re$ satisfying the constraint $x^2 - 8x + 15 \leq 0$.

(c) The empty set.

Exercise 2.2 Let $f : \Re^n \mapsto \Re$ be a convex function and let c be some constant. Show that the set $S = \{\mathbf{x} \in \Re^n \mid f(\mathbf{x}) \leq c\}$ is convex.

Exercise 2.3 (Basic feasible solutions in standard form polyhedra with upper bounds) Consider a polyhedron defined by the constraints $\mathbf{Ax} = \mathbf{b}$ and $\mathbf{0} \leq \mathbf{x} \leq \mathbf{u}$, and assume that the matrix $\mathbf{A}$ has linearly independent rows. Provide a procedure analogous to the one in Section 2.3 for constructing basic solutions, and prove an analog of Theorem 2.4.

Exercise 2.4 We know that every linear programming problem can be converted to an equivalent problem in standard form. We also know that nonempty polyhedra in standard form have at least one extreme point. We are then tempted to conclude that every nonempty polyhedron has at least one extreme point. Explain what is wrong with this argument.

Exercise 2.5 (Extreme points of isomorphic polyhedra) A mapping f is called *affine* if it is of the form $f(\mathbf{x}) = \mathbf{Ax} + \mathbf{b}$, where $\mathbf{A}$ is a matrix and $\mathbf{b}$ is a vector. Let P and Q be polyhedra in $\Re^n$ and $\Re^m$, respectively. We say that P and Q are *isomorphic* if there exist affine mappings $f : P \mapsto Q$ and $g : Q \mapsto P$ such that $g(f(\mathbf{x})) = \mathbf{x}$ for all $\mathbf{x} \in P$, and $f(g(\mathbf{y})) = \mathbf{y}$ for all $\mathbf{y} \in Q$. (Intuitively, isomorphic polyhedra have the same shape.)

(a) If P and Q are isomorphic, show that there exists a one-to-one correspondence between their extreme points. In particular, if f and g are as above, show that $\mathbf{x}$ is an extreme point of P if and only if $f(\mathbf{x})$ is an extreme point of Q.

(b) **(Introducing slack variables leads to an isomorphic polyhedron)** Let $P = \{\mathbf{x} \in \Re^n \mid \mathbf{Ax} \geq \mathbf{b}, \mathbf{x} \geq \mathbf{0}\}$, where $\mathbf{A}$ is a matrix of dimensions $k \times n$. Let $Q = \{(\mathbf{x}, \mathbf{z}) \in \Re^{n+k} \mid \mathbf{Ax} - \mathbf{z} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}, \mathbf{z} \geq \mathbf{0}\}$. Show that P and Q are isomorphic.

Exercise 2.6 (Carathéodory's theorem) Let $\mathbf{A}_1, \ldots, \mathbf{A}_n$ be a collection of vectors in $\Re^m$.

(a) Let

$$C = \left\{ \sum_{i=1}^{n} \lambda_i \mathbf{A}_i \mid \lambda_1, \ldots, \lambda_n \geq 0 \right\}.$$

Show that any element of C can be expressed in the form $\sum_{i=1}^{n} \lambda_i \mathbf{A}_i$, with $\lambda_i \geq 0$, and with at most m of the coefficients λ_i being nonzero. *Hint:* Consider the polyhedron

$$\Lambda = \left\{ (\lambda_1, \ldots, \lambda_r) \in \Re^n \mid \sum_{i=1}^{n} \lambda_i \mathbf{A}_i = \mathbf{y}, \lambda_1, \ldots, \lambda_n \geq 0 \right\}.$$

(b) Let P be the convex hull of the vectors $\mathbf{A}_i$:

$$P = \left\{ \sum_{i=1}^{n} \lambda_i \mathbf{A}_i \mid \sum_{i=1}^{n} \lambda_i = 1, \lambda_1, \ldots, \lambda_n \geq 0 \right\}.$$

Show that any element of P can be expressed in the form $\sum_{i=1}^{n} \lambda_i \mathbf{A}_i$, where $\sum_{i=1}^{n} \lambda_i = 1$ and $\lambda_i \geq 0$ for all i, with at most $m+1$ of the coefficients λ_i being nonzero.

Exercise 2.7 Suppose that $\{\mathbf{x} \in \Re^n \mid \mathbf{a}_i'\mathbf{x} \geq b_i, i = 1, \ldots, m\}$ and $\{\mathbf{x} \in \Re^n \mid \mathbf{g}_i'\mathbf{x} \geq h_i, i = 1, \ldots, k\}$ are two representations of the same nonempty polyhedron. Suppose that the vectors $\mathbf{a}_1, \ldots, \mathbf{a}_m$ span $\Re^n$. Show that the same must be true for the vectors $\mathbf{g}_1, \ldots, \mathbf{g}_k$.

Exercise 2.8 Consider the standard form polyhedron $\{\mathbf{x} \mid \mathbf{Ax} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}\}$, and assume that the rows of the matrix $\mathbf{A}$ are linearly independent. Let $\mathbf{x}$ be a basic solution, and let $J = \{i \mid x_i \neq 0\}$. Show that a basis is associated with the basic solution $\mathbf{x}$ if and only if every column $\mathbf{A}_i$, $i \in J$, is in the basis.

Exercise 2.9 Consider the standard form polyhedron $\{\mathbf{x} \mid \mathbf{Ax} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}\}$, and assume that the rows of the matrix $\mathbf{A}$ are linearly independent.

(a) Suppose that two different bases lead to the same basic solution. Show that the basic solution is degenerate.

(b) Consider a degenerate basic solution. Is it true that it corresponds to two or more distinct bases? Prove or give a counterexample.

(c) Suppose that a basic solution is degenerate. Is it true that there exists an adjacent basic solution which is degenerate? Prove or give a counterexample.

Exercise 2.10 Consider the standard form polyhedron $P = \{\mathbf{x} \mid \mathbf{Ax} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}\}$. Suppose that the matrix $\mathbf{A}$ has dimensions $m \times n$ and that its rows are linearly independent. For each one of the following statements, state whether it is true or false. If true, provide a proof, else, provide a counterexample.

(a) If $n = m + 1$, then P has at most two basic feasible solutions.

(b) The set of all optimal solutions is bounded.

(c) At every optimal solution, no more than m variables can be positive.

(d) If there is more than one optimal solution, then there are uncountably many optima solutions.

(e) If there are several optimal solutions, then there exist at least two basic feasible solutions that are optimal.

(f) Consider the problem of minimizing $\max\{\mathbf{c}'\mathbf{x}, \mathbf{d}'\mathbf{x}\}$ over the set P. If this problem has an optimal solution, it must have an optimal solution which is an extreme point of P.

Exercise 2.11 Let $P = \{\mathbf{x} \in \Re^n \mid \mathbf{Ax} \geq \mathbf{b}\}$. Suppose that at a particular basic feasible solution, there are k active constraints, with $k > n$. Is it true that there exist exactly $\binom{k}{n}$ bases that lead to this basic feasible solution? Here $\binom{k}{n} = k!/(n!(k-n)!)$ is the number of ways that we can choose n out of k given items.

Exercise 2.12 Consider a nonempty polyhedron P and suppose that for each variable x_i we have either the constraint $x_i \geq 0$ or the constraint $x_i \leq 0$. Is it true that P has at least one basic feasible solution?

Exercise 2.13 Consider the standard form polyhedron $P = \{x \mid Ax = b, \ x \geq 0\}$. Suppose that the matrix A, of dimensions $m \times n$, has linearly independent rows, and that all basic feasible solutions are nondegenerate. Let x be an element of P that has exactly m positive components.

(a) Show that x is a basic feasible solution.

(b) Show that the result of part (a) is false if the nondegeneracy assumption is removed.

Exercise 2.14 Let P be a bounded polyhedron in $\Re^n$, let a be a vector in $\Re^n$, and let b be some scalar. We define

$$Q = \{x \in P \mid a'x = b\}.$$

Show that every extreme point of Q is either an extreme point of P or a convex combination of two adjacent extreme points of P.

Exercise 2.15 (Edges joining adjacent vertices) Consider the polyhedron $P = \{x \in \Re^n \mid a_i'x \geq b_i, \ i = 1, \ldots, m\}$. Suppose that u and v are distinct basic feasible solutions that satisfy $a_i'u = a_i'v = b_i, \ i = 1, \ldots, n-1$, and that the vectors $a_1, \ldots, a_{n-1}$ are linearly independent. (In particular, u and v are adjacent.) Let $L = \{\lambda u + (1-\lambda)v \mid 0 \leq \lambda \leq 1\}$ be the segment that joins u and v. Prove that $L = \{z \in P \mid a_i'z = b_i, \ i = 1, \ldots, n-1\}$.

Exercise 2.16 Consider the set $\{x \in \Re^n \mid x_1 = \cdots = x_{n-1} = 0, \ 0 \leq x_n \leq 1\}$. Could this be the feasible set of a problem in standard form?

Exercise 2.17 Consider the polyhedron $\{x \in \Re^n \mid Ax \leq b, \ x \geq 0\}$ and a nondegenerate basic feasible solution x^*. We introduce slack variables z and construct a corresponding polyhedron $\{(x, z) \mid Ax + z = b, \ x \geq 0, \ z \geq 0\}$ in standard form. Show that $(x^*, b - Ax^*)$ is a nondegenerate basic feasible solution for the new polyhedron.

Exercise 2.18 Consider a polyhedron $P = \{x \mid Ax \geq b\}$. Given any $\epsilon > 0$, show that there exists some $\bar{b}$ with the following two properties:

(a) The absolute value of every component of $b - \bar{b}$ is bounded by ϵ.

(b) Every basic feasible solution in the polyhedron $P = \{x \mid Ax \geq \bar{b}\}$ is nondegenerate.

Exercise 2.19* Let $P \subset \Re^n$ be a polyhedron in standard form whose definition involves m linearly independent equality constraints. Its dimension is defined as the smallest integer k such that P is contained in some k-dimensional affine subspace of $\Re^n$.

(a) Explain why the dimension of P is at most $n - m$.

(b) Suppose that P has a nondegenerate basic feasible solution. Show that the dimension of P is equal to $n - m$.

(c) Suppose that x is a degenerate basic feasible solution. Show that x is degenerate under every standard form representation of the same polyhedron (in the same space $\Re^n$). *Hint:* Using parts (a) and (b), compare the number of equality constraints in two representations of P under which x is degenerate and nondegenerate, respectively. Then, count active constraints.

Exercise 2.20* Consider the Fourier-Motzkin elimination algorithm.

(a) Suppose that the number m of constraints defining a polyhedron P is even. Show, by means of an example, that the elimination algorithm may produce a description of the polyhedron $\Pi_{n-1}(P)$ involving as many as $m^2/4$ linear constraints, but no more than that.

(b) Show that the elimination algorithm produces a description of the one-dimensional polyhedron $\Pi_1(P)$ involving no more than $m^{2^{n-1}}/2^{2^{n}-2}$ constraints.

(c) Let $n = 2^p + p + 2$, where p is a nonnegative integer. Consider a polyhedron in $\Re^n$ defined by the $8\binom{r}{3}$ constraints

$$\pm x_i \pm x_j \pm x_k \leq 1, \quad 1 \leq i < j < k \leq r,$$

where all possible combinations are present. Show that after p eliminations, we have at least

$$2^{2^p+2}$$

constraints. (Note that this number increases exponentially with n.)

Exercise 2.21 Suppose that Fourier-Motzkin elimination is used in the manner described at the end of Section 2.8 to find the optimal cost in a linear programming problem. Show how this approach can be augmented to obtain an optimal solution as well.

Exercise 2.22 Let P and Q be polyhedra in $\Re^n$. Let $P + Q = \{x + y \mid x \in P, \ y \in Q\}$.

(a) Show that $P + Q$ is a polyhedron.

(b) Show that every extreme point of $P + Q$ is the sum of an extreme point of P and an extreme point of Q.

2.11 Notes and sources

The relation between algebra and geometry goes far back in the history of mathematics, but was limited to two and three-dimensional spaces. The insight that the same relation goes through in higher dimensions only came in the middle of the nineteenth century.

2.2. Our algebraic definition of basic (feasible) solutions for general polyhedra, in terms of the number of linearly independent active constraints, is not common. Nevertheless, we consider it to be quite central, because it provides the main bridge between the algebraic and geometric viewpoint, it allows for a unified treatment, and shows that there is not much that is special about standard form problems.

2.8. Fourier-Motzkin elimination is due to Fourier (1827), Dines (1918), and Motzkin (1936).

Chapter 3

The simplex method

Contents

We saw in Chapter 2, that if a linear programming problem in standard form has an optimal solution, then there exists a basic feasible solution that is optimal. The simplex method is based on this fact and searches for an optimal solution by moving from one basic feasible solution to another, along the edges of the feasible set, always in a cost reducing direction. Eventually, a basic feasible solution is reached at which none of the available edges leads to a cost reduction; such a basic feasible solution is optimal and the algorithm terminates. In this chapter, we provide a detailed development of the simplex method and discuss a few different implementations, including the simplex tableau and the revised simplex method. We also address some difficulties that may arise in the presence of degeneracy. We provide an interpretation of the simplex method in terms of column geometry, and we conclude with a discussion of its running time, as a function of the dimension of the problem being solved.

Throughout this chapter, we consider the standard form problem

$$\begin{aligned} \text{minimize} \quad & \mathbf{c}'\mathbf{x} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0}, \end{aligned}$$

and we let P be the corresponding feasible set. We assume that the dimensions of the matrix $\mathbf{A}$ are $m \times n$ and that its rows are linearly independent. We continue using our previous notation: $\mathbf{A}_i$ is the ith column of the matrix $\mathbf{A}$, and $\mathbf{a}'_i$ is its ith row.

3.1   Optimality conditions

Many optimization algorithms are structured as follows: given a feasible solution, we search its neighborhood to find a nearby feasible solution with lower cost. If no nearby feasible solution leads to a cost improvement, the algorithm terminates and we have a *locally optimal* solution. For general optimization problems, a locally optimal solution need not be (globally) optimal. Fortunately, in linear programming, local optimality implies global optimality; this is because we are minimizing a convex function over a convex set (cf. Exercise 3.1). In this section, we concentrate on the problem of searching for a direction of cost decrease in a neighborhood of a given basic feasible solution, and on the associated optimality conditions.

Suppose that we are at a point $\mathbf{x} \in P$ and that we contemplate moving away from $\mathbf{x}$, in the direction of a vector $\mathbf{d} \in \Re^n$. Clearly, we should only consider those choices of $\mathbf{d}$ that do not immediately take us outside the feasible set. This leads to the following definition, illustrated in Figure 3.1.

Figure 3.1: Feasible directions at different points of a polyhedron.

Definition 3.1 *Let* $\mathbf{x}$ *be an element of a polyhedron* P. *A vector* $\mathbf{d} \in \Re^n$ *is said to be a* **feasible direction** *at* $\mathbf{x}$, *if there exists a positive scalar* θ *for which* $\mathbf{x} + \theta\mathbf{d} \in P$.

Let $\mathbf{x}$ be a basic feasible solution to the standard form problem, let $B(1),\ldots,B(m)$ be the indices of the basic variables and let $\mathbf{B} = [\mathbf{A}_{B(1)} \cdots \mathbf{A}_{B(m)}]$ be the corresponding basis matrix. In particular, we have $x_i = 0$ for every nonbasic variable, while the vector $\mathbf{x}_B = (x_{B(1)},\ldots,x_{B(m)})$ of basic variables is given by

$$\mathbf{x}_B = \mathbf{B}^{-1}\mathbf{b}.$$

We consider the possibility of moving away from $\mathbf{x}$, to a new vector $\mathbf{x} + \theta\mathbf{d}$, by selecting a nonbasic variable x_j (which is initially at zero level), and increasing it to a positive value θ, while keeping the remaining nonbasic variables at zero. Algebraically, $d_j = 1$, and $d_i = 0$ for every nonbasic index i other than j. At the same time, the vector $\mathbf{x}_B$ of basic variables changes to $\mathbf{x}_B + \theta\mathbf{d}_B$, where $\mathbf{d}_B = (d_{B(1)}, d_{B(2)},\ldots,d_{B(m)})$ is the vector with those components of $\mathbf{d}$ that correspond to the basic variables.

Given that we are only interested in feasible solutions, we require $\mathbf{A}(\mathbf{x}+\theta\mathbf{d}) = \mathbf{b}$, and since $\mathbf{x}$ is feasible, we also have $\mathbf{A}\mathbf{x} = \mathbf{b}$. Thus, for the equality constraints to be satisfied for $\theta > 0$, we need $\mathbf{A}\mathbf{d} = \mathbf{0}$. Recall now that $d_j = 1$, and that $d_i = 0$ for all other nonbasic indices i. Then,

$$\mathbf{0} = \mathbf{A}\mathbf{d} = \sum_{i=1}^n \mathbf{A}_i d_i = \sum_{i=1}^m \mathbf{A}_{B(i)} d_{B(i)} + \mathbf{A}_j = \mathbf{B}\mathbf{d}_B + \mathbf{A}_j.$$

Since the basis matrix $\mathbf{B}$ is invertible, we obtain

$$\mathbf{d}_B = -\mathbf{B}^{-1}\mathbf{A}_j. \tag{3.1}$$

The direction vector **d** that we have just constructed will be referred to as the jth *basic direction*. We have so far guaranteed that the equality constraints are respected as we move away from **x** along the basic direction **d**. How about the nonnegativity constraints? We recall that the variable x_j is increased, and all other nonbasic variables stay at zero level. Thus, we need only worry about the basic variables. We distinguish two cases:

(a) Suppose that **x** is a nondegenerate basic feasible solution. Then, $x_B > 0$, from which it follows that $x_B + \theta d_B \geq 0$, and feasibility is maintained, when θ is sufficiently small. In particular, **d** is a feasible direction.

(b) Suppose now that **x** is degenerate. Then, **d** is not always a feasible direction. Indeed, it is possible that a basic variable $x_{B(i)}$ is zero, while the corresponding component $d_{B(i)}$ of $d_B = -\mathbf{B}^{-1}\mathbf{A}_j$ is negative. In that case, if we follow the jth basic direction, the nonnegativity constraint for $x_{B(i)}$ is immediately violated, and we are led to infeasible solutions; see Figure 3.2.

We now study the effects on the cost function if we move along a basic direction. If **d** is the jth basic direction, then the rate $\mathbf{c'd}$ of cost change along the direction **d** is given by $\mathbf{c}'_B\mathbf{d}_B + c_j$, where $\mathbf{c}_B = (c_{B(1)},\ldots,c_{B(m)})$. Using Eq. (3.1), this is the same as $c_j - \mathbf{c}'_B\mathbf{B}^{-1}\mathbf{A}_j$. This quantity is important enough to warrant a definition. For an intuitive interpretation, c_j is the cost per unit increase in the variable x_j, and the term $-\mathbf{c}'_B\mathbf{B}^{-1}\mathbf{A}_j$ is the cost of the compensating change in the basic variables necessitated by the constraint $\mathbf{Ax} = \mathbf{b}$.

> **Definition 3.2** *Let **x** be a basic solution, let **B** be an associated basis matrix, and let $\mathbf{c}_B$ be the vector of costs of the basic variables. For each j, we define the reduced cost $\bar{c}_j$ of the variable x_j according to the formula*
>
> $$\bar{c}_j = c_j - \mathbf{c}'_B\mathbf{B}^{-1}\mathbf{A}_j.$$

Example 3.1 Consider the linear programming problem

minimize $c_1 x_1 + c_2 x_2 + c_3 x_3 + c_4 x_4$

subject to $x_1 + x_2 + \quad\;\; x_4 = 2$
$\qquad\qquad 2x_1 + \quad\; x_3 + 4x_4 = 2$
$\qquad\qquad x_1, x_2, x_3, x_4 \geq 0.$

The first two columns of the matrix **A** are $\mathbf{A}_1 = (1, 2)$ and $\mathbf{A}_2 = (1, 0)$. Since they are linearly independent, we can choose x_1 and x_2 as our basic variables. The corresponding basis matrix is
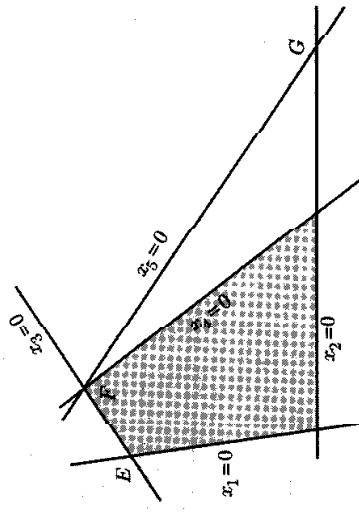
$$\mathbf{B} = \begin{bmatrix} 1 & 1 \\ 2 & 0 \end{bmatrix}.$$

Figure 3.2: Let $n = 5$, $n - m = 2$. As discussed in Section 1.4, we can visualize the feasible set by standing on the two-dimensional set defined by the constraint $\mathbf{Ax} = \mathbf{b}$, in which case, the edges of the feasible set are associated with the nonnegativity constraints $x_i \geq 0$. At the nondegenerate basic feasible solution E, the variables x_1 and x_3 are at zero level (nonbasic) and x_2, x_4, x_5 are positive basic variables. The first basic direction is obtained by increasing x_1, while keeping the other nonbasic variable x_3 at zero level. This is the direction corresponding to the edge EF. Consider now the degenerate basic feasible solution F and let x_3, x_5 be the nonbasic variables. Note that x_4 is a basic variable at zero level. A basic direction is obtained by increasing x_3, while keeping the other nonbasic variable x_5 at zero level. This is the direction corresponding to the line FG and it takes us outside the feasible set. Thus, this basic direction is not a feasible direction.

We set $x_3 = x_4 = 0$, and solve for x_1, x_2, to obtain $x_1 = 1$ and $x_2 = 1$. We have thus obtained a nondegenerate basic feasible solution.

A basic direction corresponding to an increase in the nonbasic variable x_3, is constructed as follows. We have $d_3 = 1$ and $d_4 = 0$. The direction of change of the basic variables is obtained using Eq. (3.1):

$$\begin{bmatrix} d_1 \\ d_2 \end{bmatrix} = \begin{bmatrix} d_{B(1)} \\ d_{B(2)} \end{bmatrix} = \mathbf{d}_B = -\mathbf{B}^{-1}\mathbf{A}_3 = -\begin{bmatrix} 0 & 1/2 \\ 1 & -1/2 \end{bmatrix}\begin{bmatrix} 1 \\ 3 \end{bmatrix} = \begin{bmatrix} -3/2 \\ 1/2 \end{bmatrix}.$$

The cost of moving along this basic direction is $\mathbf{c'd} = -3c_1/2 + c_2/2 + c_3$. This is the same as the reduced cost of the variable x_3.

Consider now Definition 3.2 for the case of a basic variable. Since **B** is the matrix $[\mathbf{A}_{B(1)} \cdots \mathbf{A}_{B(m)}]$, we have $\mathbf{B}^{-1}[\mathbf{A}_{B(1)} \cdots \mathbf{A}_{B(m)}] = \mathbf{I}$, where

I is the $m \times m$ identity matrix. In particular, $\mathbf{B}^{-1}\mathbf{A}_{B(i)}$ is the ith column of the identity matrix, which is the ith unit vector $\mathbf{e}_i$. Therefore, for every basic variable $x_{B(i)}$, we have

$$\bar{c}_{B(i)} = c_{B(i)} - \mathbf{c}'_B \mathbf{B}^{-1}\mathbf{A}_{B(i)} = c_{B(i)} - \mathbf{c}'_B \mathbf{e}_i = c_{B(i)} - c_{B(i)} = 0,$$

and we see that the reduced cost of every basic variable is zero.

Our next result provides us with optimality conditions. Given our interpretation of the reduced costs as rates of cost change along certain directions, this result is intuitive.

Theorem 3.1 *Consider a basic feasible solution $\mathbf{x}$ associated with a basis matrix $\mathbf{B}$, and let $\bar{\mathbf{c}}$ be the corresponding vector of reduced costs.*

(a) *If $\bar{\mathbf{c}} \geq \mathbf{0}$, then $\mathbf{x}$ is optimal.*
(b) *If $\mathbf{x}$ is optimal and nondegenerate, then $\bar{\mathbf{c}} \geq \mathbf{0}$.*

Proof.

(a) We assume that $\bar{\mathbf{c}} \geq \mathbf{0}$, we let $\mathbf{y}$ be an arbitrary feasible solution, and we define $\mathbf{d} = \mathbf{y} - \mathbf{x}$. Feasibility implies that $\mathbf{Ax} = \mathbf{Ay} = \mathbf{b}$ and, therefore, $\mathbf{Ad} = \mathbf{0}$. The latter equality can be rewritten in the form

$$\mathbf{Bd}_B + \sum_{i \in N} \mathbf{A}_i d_i = \mathbf{0},$$

where N is the set of indices corresponding to the nonbasic variables under the given basis. Since $\mathbf{B}$ is invertible, we obtain

$$\mathbf{d}_B = -\sum_{i \in N} \mathbf{B}^{-1}\mathbf{A}_i d_i,$$

and

$$\mathbf{c}'\mathbf{d} = \mathbf{c}'_B \mathbf{d}_B + \sum_{i \in N} c_i d_i = \sum_{i \in N}(c_i - \mathbf{c}'_B \mathbf{B}^{-1}\mathbf{A}_i)d_i = \sum_{i \in N} \bar{c}_i d_i.$$

For any nonbasic index $i \in N$, we must have $x_i = 0$ and, since $\mathbf{y}$ is feasible, $y_i \geq 0$. Thus, $d_i \geq 0$ and $\bar{c}_i d_i \geq 0$, for all $i \in N$. We conclude that $\mathbf{c}'(\mathbf{y} - \mathbf{x}) = \mathbf{c}'\mathbf{d} \geq 0$, and since $\mathbf{y}$ was an arbitrary feasible solution, $\mathbf{x}$ is optimal.

(b) Suppose that $\mathbf{x}$ is a nondegenerate basic feasible solution and that $\bar{c}_j < 0$ for some j. Since the reduced cost of a basic variable is always zero, x_j must be a nonbasic variable and $\bar{c}_j$ is the rate of cost change along the jth basic direction. Since $\mathbf{x}$ is nondegenerate, the jth basic direction is a feasible direction of cost decrease, as discussed earlier. By moving in that direction, we obtain feasible solutions whose cost is less than that of $\mathbf{x}$, and $\mathbf{x}$ is not optimal. □

Note that Theorem 3.1 allows the possibility that $\mathbf{x}$ is a (degenerate) optimal basic feasible solution, but that $\bar{c}_j < 0$ for some nonbasic index j. There is an analog of Theorem 3.1 that provides conditions under which a basic feasible solution $\mathbf{x}$ is a unique optimal solution; see Exercise 3.6. A related view of the optimality conditions is developed in Exercises 3.2 and 3.3.

According to Theorem 3.1, in order to decide whether a nondegenerate basic feasible solution is optimal, we need only check whether all reduced costs are nonnegative, which is the same as examining the $n - m$ basic directions. If $\mathbf{x}$ is a degenerate basic feasible solution, an equally simple computational test for determining whether $\mathbf{x}$ is optimal is not available (see Exercises 3.7 and 3.8). Fortunately, the simplex method, as developed in subsequent sections, manages to get around this difficulty in an effective manner.

Note that in order to use Theorem 3.1 and assert that a certain basic solution is optimal, we need to satisfy two conditions: feasibility, and nonnegativity of the reduced costs. This leads us to the following definition.

Definition 3.3 *A basis matrix $\mathbf{B}$ is said to be optimal if:*
(a) $\mathbf{B}^{-1}\mathbf{b} \geq \mathbf{0}$, *and*
(b) $\bar{\mathbf{c}}' = \mathbf{c}' - \mathbf{c}'_B \mathbf{B}^{-1}\mathbf{A} \geq \mathbf{0}'.$

Clearly, if an optimal basis is found, the corresponding basic solution is feasible, satisfies the optimality conditions, and is therefore optimal. On the other hand, in the degenerate case, having an optimal basic feasible solution does not necessarily mean that the reduced costs are nonnegative.

3.2 Development of the simplex method

We will now complete the development of the simplex method. Our main task is to work out the details of how to move to a better basic feasible solution whenever a profitable basic direction is discovered.

Let us assume that every basic feasible solution is nondegenerate. This assumption will remain in effect until it is explicitly relaxed later in this section. Suppose that we are at a basic feasible solution $\mathbf{x}$ and that we have computed the reduced costs $\bar{c}_j$ of the nonbasic variables. If all of them are nonnegative, Theorem 3.1 shows that we have an optimal solution and we stop. If on the other hand, the reduced cost $\bar{c}_j$ of a nonbasic variable x_j is negative, the jth basic direction $\mathbf{d}$ is a feasible direction of cost decrease. [This is the direction obtained by letting $d_j = 1$, $d_i = 0$ for $i \neq B(1), \ldots, B(m), j$, and $\mathbf{d}_B = -\mathbf{B}^{-1}\mathbf{A}_j$.] While moving along this direction $\mathbf{d}$, the nonbasic variable x_j becomes positive and all other nonbasic

variables remain at zero. We describe this situation by saying that x_j (or A_j) *enters or is brought into the basis.*

Once we start moving away from x along the direction d, we are tracing points of the form $x + \theta d$, where $\theta \geq 0$. Since costs decrease along the direction d, it is desirable to move as far as possible. This takes us to the point $x + \theta^* d$, where

$$\theta^* = \max\{\theta \geq 0 \mid x + \theta d \in P\}.$$

The resulting cost change is $\theta^* c'd$, which is the same as $\theta^* \bar{c}_j$.

We now derive a formula for θ^*. Given that $Ad = 0$, we have $A(x + \theta d) = Ax = b$ for all θ, and the equality constraints will never be violated. Thus, $x + \theta d$ can become infeasible only if one of its components becomes negative. We distinguish two cases:

(a) If $d \geq 0$, then $x + \theta d \geq 0$ for all $\theta \geq 0$, the vector $x + \theta d$ never becomes infeasible, and we let $\theta^* = \infty$.

(b) If $d_i < 0$ for some i, the constraint $x_i + \theta d_i \geq 0$ becomes $\theta \leq -x_i/d_i$. This constraint on θ must be satisfied for every i with $d_i < 0$. Thus, the largest possible value of θ is

$$\theta^* = \min_{\{i \mid d_i < 0\}} \left(-\frac{x_i}{d_i}\right).$$

Recall that if x_i is a nonbasic variable, then either x_i is the entering variable and $d_i = 1$, or else $d_i = 0$. In either case, d_i is nonnegative. Thus, we only need to consider the basic variables and we have the equivalent formula

$$\theta^* = \min_{\{i=1,\ldots,m \mid d_{B(i)} < 0\}} \left(-\frac{x_{B(i)}}{d_{B(i)}}\right). \quad (3.2)$$

Note that $\theta^* > 0$, because $x_{B(i)} > 0$ for all i, as a consequence of nondegeneracy.

Example 3.2 This is a continuation of Example 3.1 from the previous section, dealing with the linear programming problem

minimize $c_1 x_1 + c_2 x_2 + c_3 x_3 + c_4 x_4$
subject to $x_1 + x_2 + x_3 + x_4 = 2$
          $2x_1 + 3x_3 + 4x_4 = 2$
          $x_1, x_2, x_3, x_4 \geq 0.$

Let us again consider the basic feasible solution $x = (1, 1, 0, 0)$ and recall that the reduced cost $\bar{c}_3$ of the nonbasic variable x_3 was found to be $-3c_1/2 + c_2/2 + c_3$. Suppose that $c = (2, 0, 0, 0)$, in which case, we have $\bar{c}_3 = -3$. Since $\bar{c}_3$ is negative, we form the corresponding basic direction, which is $d = (-3/2, 1/2, 1, 0)$, and consider vectors of the form $x + \theta d$, with $\theta \geq 0$. As θ increases, the only component of x that decreases is the first one (because $d_1 < 0$). The largest possible value

of θ is given by $\theta^* = -(x_1/d_1) = 2/3$. This takes us to the point $y = x + 2d/3 = (0, 4/3, 2/3, 0)$. Note that the columns A_2 and A_3 corresponding to the nonzero variables at the new vector y are $(1, 0)$ and $(1, 3)$, respectively, and are linearly independent. Therefore, they form a basis and the vector y is a new basic feasible solution. In particular, the variable x_3 has entered the basis and the variable x_1 has exited the basis.

Once θ^* is chosen, and assuming it is finite, we move to the new feasible solution $y = x + \theta^* d$. Since $x_j = 0$ and $d_j = 1$, we have $y_j = \theta^* > 0$. Let ℓ be a minimizing index in Eq. (3.2), that is,

$$\frac{x_{B(\ell)}}{-d_{B(\ell)}} = \min_{\{i=1,\ldots,m \mid d_{B(i)} < 0\}} \left(-\frac{x_{E(i)}}{d_{E(i)}}\right) = \theta^*;$$

in particular,

$$d_{B(\ell)} < 0,$$

and

$$x_{B(\ell)} + \theta^* d_{B(\ell)} = 0.$$

We observe that the basic variable $x_{B(\ell)}$ has become zero, whereas the nonbasic variable x_j has now become positive, which suggests that x_j should replace $x_{B(\ell)}$ in the basis. Accordingly, we take the old basis matrix B and replace $A_{B(\ell)}$ with A_j, thus obtaining the matrix

$$\bar{B} = \left[A_{B(1)} \cdots A_{B(\ell-1)} \; A_j \; A_{B(\ell+1)} \cdots A_{B(m)} \right]. \quad (3.3)$$

Equivalently, we are replacing the set $\{B(1),\ldots,B(m)\}$ of basic indices by a new set $\{\bar{B}(1),\ldots,\bar{B}(m)\}$ of indices given by

$$\bar{B}(i) = \begin{cases} B(i), & i \neq \ell, \\ j, & i = \ell. \end{cases} \quad (3.4)$$

Theorem 3.2

(a) The columns $A_{\bar{B}(i)}$, $i \neq \ell$, and A_j are linearly independent and, therefore, $\bar{B}$ is a basis matrix.

(b) The vector $y = x + \theta^* d$ is a basic feasible solution associated with the basis matrix $\bar{B}$.

Proof.

(a) If the vectors $A_{\bar{B}(i)}$, $i = 1,\ldots,m$, are linearly dependent, then there exist coefficients $\lambda_1,\ldots,\lambda_m$, not all of them zero, such that

$$\sum_{i=1}^m \lambda_i A_{\bar{B}(i)} = 0,$$

which implies that

$$\sum_{i=1}^{n} \lambda_i B^{-1} A_{\overline{B}(i)} = 0,$$

and the vectors $B^{-1}A_{\overline{B}(i)}$ are also linearly dependent. To show that this is not the case, we will prove that the vectors $B^{-1}A_{B(i)}$, $i \neq \ell$, and $B^{-1}A_j$ are linearly independent. We have $B^{-1}B = I$. Since $A_{B(i)}$ is the ith column of B, it follows that the vectors $B^{-1}A_{B(i)}$, $i \neq \ell$, are all the unit vectors except for the ℓth unit vector. In particular, they are linearly independent and their ℓth component is zero. On the other hand, $B^{-1}A_j$ is equal to $-d_B$. Its ℓth entry, $-d_{B(\ell)}$, is nonzero by the definition of ℓ. Thus, $B^{-1}A_j$, $i \neq \ell$, is linearly independent from the unit vectors $B^{-1}A_{B(i)}$, $i \neq \ell$.

(b) We have $y \geq 0$, $Ay = b$, and $y_i = 0$ for $i \neq \overline{B}(1), \ldots, \overline{B}(m)$. Furthermore, the columns $A_{\overline{B}(1)}, \ldots, A_{\overline{B}(m)}$ have just been shown to be linearly independent. It follows that y is a basic feasible solution associated with the basis matrix $\overline{B}$. □

Since θ^* is positive, the new basic feasible solution $x + \theta^* d$ is distinct from x; since d is a direction of cost decrease, the cost of this new basic feasible solution is strictly smaller. We have therefore accomplished our objective of moving to a new basic feasible solution with lower cost. We can now summarize a typical iteration of the simplex method, also known as a pivot (see Section 3.6 for a discussion of the origins of this term). For our purposes, it is convenient to define a vector $u = (u_1, \ldots, u_m)$ by letting

$$u = -d_B = B^{-1}A_j,$$

where A_j is the column that enters the basis; in particular, $u_i = -d_{B(i)}$, for $i = 1, \ldots, m$.

An iteration of the simplex method

1. In a typical iteration, we start with a basis consisting of the basic columns $A_{B(1)}, \ldots, A_{B(m)}$ and an associated basic feasible solution x.

2. Compute the reduced costs $\bar{c}_j = c_j - c_B' B^{-1}A_j$ for all nonbasic indices j. If they are all nonnegative, the current basic feasible solution is optimal, and the algorithm terminates; else, choose some j for which $\bar{c}_j < 0$.

3. Compute $u = B^{-1}A_j$. If no component of u is positive, the optimal cost is $-\infty$, and the algorithm terminates.

4. If some component of u is positive, let

$$\theta^* = \min_{\{i=1,\ldots,m | u_i > 0\}} \frac{x_{B(i)}}{u_i}.$$

5. Let ℓ be such that $\theta^* = x_{B(\ell)}/u_\ell$. Form a new basis by replacing $A_{B(\ell)}$ with A_j. If y is the new basic feasible solution, the values of the new basic variables are $y_j = \theta^*$ and $y_{B(i)} = x_{B(i)} - \theta^* u_i$, $i \neq \ell$.

The simplex method is initialized with an arbitrary basic feasible solution, which, for feasible standard form problems, is guaranteed to exist. The following theorem states that, in the nondegenerate case, the simplex method works correctly and terminates after a finite number of iterations.

Theorem 3.3 *Assume that the feasible set is nonempty and that every basic feasible solution is nondegenerate. Then, the simplex method terminates after a finite number of iterations. At termination, there are the following two possibilities:*

(a) *We have an optimal basis B and an associated basic feasible solution which is optimal.*

(b) *We have found a vector d satisfying $Ad = 0$, $d \geq 0$, and $c'd < 0$, and the optimal cost is $-\infty$.*

Proof. If the algorithm terminates due to the stopping criterion in Step 2, then the optimality conditions in Theorem 3.1 have been met, B is an optimal basis, and the current basic feasible solution is optimal.

If the algorithm terminates because the criterion in Step 3 has been met, then we are at a basic feasible solution x and we have discovered a nonbasic variable x_j such that $\bar{c}_j < 0$ and such that the corresponding basic direction d satisfies $Ad = 0$ and $d \geq 0$. In particular, $x + \theta d \in P$ for all $\theta > 0$. Since $c'd = \bar{c}_j < 0$, by taking θ arbitrarily large, the cost can be made arbitrarily negative, and the optimal cost is $-\infty$.

At each iteration, the algorithm moves by a positive amount θ^* along a direction d that satisfies $c'd < 0$. Therefore, the cost of every successive basic feasible solution visited by the algorithm is strictly less than the cost of the previous one, and no basic feasible solution can be visited twice. Since there is a finite number of basic feasible solutions, the algorithm must eventually terminate. □

Theorem 3.3 provides an independent proof of some of the results of Chapter 2 for nondegenerate standard form problems. In particular, it shows that for feasible and nondegenerate problems, either the optimal

cost is $-\infty$, or there exists a basic feasible solution which is optimal (cf. Theorem 2.8 in Section 2.6). While the proof given here might appear more elementary, its extension to the degenerate case is not as simple.

The simplex method for degenerate problems

We have been working so far under the assumption that all basic feasible solutions are nondegenerate. Suppose now that the exact same algorithm is used in the presence of degeneracy. Then, the following new possibilities may be encountered in the course of the algorithm.

(a) If the current basic feasible solution $\mathbf{x}$ is degenerate, θ^* can be equal to zero, in which case, the new basic feasible solution $\mathbf{y}$ is the same as $\mathbf{x}$. This happens if some basic variable $x_{B(\ell)}$ is equal to zero and the corresponding component $d_{B(\ell)}$ of the direction vector $\mathbf{d}$ is negative. Nevertheless, we can still define a new basis $\overline{\mathbf{B}}$, by replacing $\mathbf{A}_{B(\ell)}$ with $\mathbf{A}_j$ [cf. Eqs. (3.3)–(3.4)], and Theorem 3.2 is still valid.

(b) Even if θ^* is positive, it may happen that more than one of the original basic variables becomes zero at the new point $\mathbf{x} + \theta^* \mathbf{d}$. Since only one of them exits the basis, the others remain in the basis at zero level, and the new basic feasible solution is degenerate.

Basis changes while staying at the same basic feasible solution are not in vain. As illustrated in Figure 3.3, a sequence of such basis changes may lead to the eventual discovery of a cost reducing feasible direction. On the other hand, a sequence of basis changes might lead back to the initial basis, in which case the algorithm may loop indefinitely. This undesirable phenomenon is called *cycling*. An example of cycling is given in Section 3.3, after we develop some bookkeeping tools for carrying out the mechanics of the algorithm. It is sometimes maintained that cycling is an exceptionally rare phenomenon. However, for many highly structured linear programming problems, most basic feasible solutions are degenerate, and cycling is a real possibility. Cycling can be avoided by judiciously choosing the variables that will enter or exit the basis (see Section 3.4). We now discuss the freedom available in this respect.

Pivot Selection

The simplex algorithm, as we described it, has certain degrees of freedom: in Step 2, we are free to choose any j whose reduced cost $\overline{c}_j$ is negative; also, in Step 5, there may be several indices ℓ that attain the minimum in the definition of θ^*, and we are free to choose any one of them. Rules for making such choices are called *pivoting rules*.

Regarding the choice of the entering column, the following rules are some natural candidates:
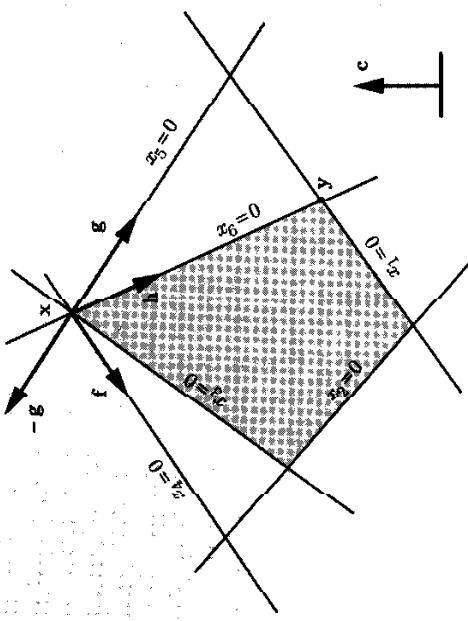
Figure 3.3: We visualize a problem in standard form, with $n - m = 2$, by standing on the two-dimensional plane defined by the equality constraints $\mathbf{Ax} = \mathbf{b}$. The basic feasible solution $\mathbf{x}$ is degenerate. If x_4 and x_5 are the nonbasic variables, then the two corresponding basic directions are the vectors $\mathbf{g}$ and $\mathbf{f}$. For either of these two basic directions, we have $\theta^* = 0$. However, if we perform a change of basis, with x_4 entering the basis and x_6 exiting, the new nonbasic variables are x_5 and x_6, and the two basic directions are $\mathbf{h}$ and $-\mathbf{g}$. (The direction $-\mathbf{g}$ is the one followed if x_6 is increased while x_5 is kept at zero.) In particular, we can now follow direction $\mathbf{h}$ to reach a new basic feasible solution $\mathbf{y}$ with lower cost.

(a) *Choose a column $\mathbf{A}_j$, with $\overline{c}_j < 0$, whose reduced cost is the most negative.* Since the reduced cost is the rate of change of the cost function, this rule chooses a direction along which costs decrease at the fastest rate. However, the actual cost decrease depends on how far we move along the chosen direction. This suggests the next rule.

(b) *Choose a column with $\overline{c}_j < 0$ for which the corresponding cost decrease $\theta^* |\overline{c}_j|$ is largest.* This rule offers the possibility of reaching optimality after a smaller number of iterations. On the other hand, the computational burden at each iteration is larger, because we need to compute θ^* for each column with $\overline{c}_j < 0$. The available empirical evidence suggests that the overall running time does not improve.

For large problems, even the rule that chooses the most negative $\overline{c}_j$ can be computationally expensive, because it requires the computation of the reduced cost of every variable. In practice, simpler rules are sometimes

used, such as the *smallest subscript* rule, that chooses the smallest j for which $\bar{c}_j$ is negative. Under this rule, once a negative reduced cost is discovered, there is no reason to compute the remaining reduced costs. Other criteria that have been found to improve the overall running time are the *Devex* (Harris, 1973) and the *steepest edge* rule (Goldfarb and Reid, 1977). Finally, there are methods based on *candidate lists* whereby one examines the reduced costs of nonbasic variables by picking them one at a time from a prioritized list. There are different ways of maintaining such prioritized lists, depending on the rule used for adding, removing, or reordering elements of the list.

Regarding the choice of the exiting column, the simplest option is again the *smallest subscript* rule: out of all variables eligible to exit the basis, choose one with the smallest subscript. It turns out that by following the smallest subscript rule for both the entering and the exiting column, cycling can be avoided (cf. Section 3.4).

3.3   Implementations of the simplex method

In this section, we discuss some ways of carrying out the mechanics of the simplex method. It should be clear from the statement of the algorithm that the vectors $B^{-1}A_j$ play a key role. If these vectors are available, the reduced costs, the direction of motion, and the stepsize θ^* are easily computed. Thus, the main difference between alternative implementations lies in the way that the vectors $B^{-1}A_j$ are computed and on the amount of related information that is carried from one iteration to the next.

When comparing different implementations, it is important to keep the following facts in mind (cf. Section 1.6). If B is a given $m \times m$ matrix and $b \in \Re^m$ is a given vector, computing the inverse of B or solving a linear system of the form $Bx = b$ takes $O(m^3)$ arithmetic operations. Computing a matrix-vector product Bb takes $O(m^2)$ operations. Finally, computing an inner product $p'b$ of two m-dimensional vectors takes $O(m)$ arithmetic operations.

Naive implementation

We start by describing the most straightforward implementation in which no auxiliary information is carried from one iteration to the next. At the beginning of a typical iteration, we have the indices $B(1), \ldots, B(m)$ of the current basic variables. We form the basis matrix B and compute $p' = c_B' B^{-1}$, by solving the linear system $p'B = c_B'$ for the unknown vector p. (This vector p is called the vector of *simplex multipliers* associated with the basis B.) The reduced cost $\bar{c}_j = c_j - c_B' B^{-1}A_j$ of any variable x_j is then obtained according to the formula

$$\bar{c}_j = c_j - p'A_j.$$

Depending on the pivoting rule employed, we may have to compute all of the reduced costs or we may compute them one at a time until a variable with a negative reduced cost is encountered. Once a column A_j is selected to enter the basis, we solve the linear system $Bu = A_j$ in order to determine the vector $u = B^{-1}A_j$. At this point, we can form the direction along which we will be moving away from the current basic feasible solution. We finally determine θ^* and the variable that will exit the basis, and construct the new basic feasible solution.

We note that we need $O(m^3)$ arithmetic operations to solve the systems $p'B = c_B'$ and $Bu = A_j$. In addition, computing the reduced costs of all variables requires $O(mn)$ arithmetic operations, because we need to form the inner product of the vector p with each one of the nonbasic columns A_j. Thus, the total computational effort per iteration is $O(m^3 + mn)$ We will see shortly that alternative implementations require only $O(m^2 + mn)$ arithmetic operations. Therefore, the implementation described here is rather inefficient, in general. On the other hand, for certain problems with a special structure, the linear systems $p'B = c_B'$ and $Bu = A_j$ can be solved very fast, in which case this implementation can be of practical interest. We will revisit this point in Chapter 7, when we apply the simplex method to network flow problems.

Revised simplex method

Much of the computational burden in the naive implementation is due to the need for solving two linear systems of equations. In an alternative implementation, the matrix B^{-1} is made available at the beginning of each iteration, and the vectors $c_B'B^{-1}$ and $B^{-1}A_j$ are computed by a matrix-vector multiplication. For this approach to be practical, we need an efficient method for updating the matrix B^{-1} each time that we effect a change of basis. This is discussed next.

Let

$$E = \begin{bmatrix} A_{B(1)} \cdots A_{B(m)} \end{bmatrix}$$

be the basis matrix at the beginning of an iteration and let

$$\bar{B} = \begin{bmatrix} A_{B(1)} \cdots A_{B(\ell-1)} & A_j & A_{B(\ell+1)} \cdots A_{B(m)} \end{bmatrix}$$

be the basis matrix at the beginning of the next iteration. These two basis matrices have the same columns except that the ℓth column $A_{B(\ell)}$ (the one that exits the basis) has been replaced by A_j. It is then reasonable to expect that B^{-1} contains information that can be exploited in the computation of $\bar{B}^{-1}$. After we develop some needed tools and terminology, we will see that this is indeed the case. An alternative explanation and line of development is outlined in Exercise 3.13.

Definition 3.4 Given a matrix, not necessarily square, the operation of adding a constant multiple of one row to the same or to another row is called an elementary row operation.

The example that follows indicates that performing an elementary row operation on a matrix C is equivalent to forming the matrix QC, where Q is a suitably constructed square matrix.

Example 3.3 Let

$$Q = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad C = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix},$$

and note that

$$QC = \begin{bmatrix} 11 & 14 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}.$$

In particular, multiplication from the left by the matrix Q has the effect of multiplying the third row of C by two and adding it to the first row.

Generalizing Example 3.3, we see that multiplying the jth row (for $i \neq j$) is the same as left-multiplying by and adding it to the ith row the matrix $Q = I + D_{ij}$, where D_{ij} is a matrix with all entries equal to zero, except for the (i,j)th entry which is equal to β. The determinant of such a matrix Q is equal to 1 and, therefore, Q is invertible.

Suppose now that we apply a *sequence* of K elementary row operations and that the kth such operation corresponds to left-multiplication by a certain invertible matrix Q_k. Then, the sequence of these elementary row operations is the same as left-multiplication by the invertible matrix $Q_K Q_{K-1} \cdots Q_2 Q_1$. We conclude that performing a *sequence* of elementary row operations on a given matrix is equivalent to left-multiplying that matrix by a certain invertible matrix.

Since $B^{-1}B = I$, we see that $B^{-1}A_{B(i)}$ is the ith unit vector e_i. Using this observation, we have

$$B^{-1}\bar{B} = \begin{bmatrix} | & & | & | & | & & | \\ e_1 & \cdots & e_{\ell-1} & u & e_{\ell+1} & \cdots & e_m \\ | & & | & | & | & & | \end{bmatrix}$$

$$= \begin{bmatrix} 1 & & & u_1 & & & \\ & \ddots & & \vdots & & & \\ & & & u_\ell & & & \\ & & & \vdots & & \ddots & \\ & & & u_m & & & 1 \end{bmatrix}$$

where $u = B^{-1}A_j$. Let us apply a sequence of elementary row operations that will change the above matrix to the identity matrix. In particular, consider the following sequence of elementary row operations.

(a) For each $i \neq \ell$, we add the ℓth row times $-u_i/u_\ell$ to the ith row. (Recall that $u_\ell > 0$.) This replaces u_i by zero.

(b) We divide the ℓth row by u_ℓ. This replaces u_ℓ by one.

In words, we are adding to each row a multiple of the ℓth row to replace the ℓth column u by the ℓth unit vector e_ℓ. This sequence of elementary row operations is equivalent to left-multiplying $B^{-1}\bar{B}$ by a certain invertible matrix Q. Since the result is the identity, we have $QB^{-1}\bar{B} = I$, which yields $QB^{-1} = \bar{B}^{-1}$. The last equation shows that if we apply the same sequence of row operations to the matrix B^{-1} (equivalently, left-multiply by Q), we obtain $\bar{B}^{-1}$. We conclude that all it takes to generate $\bar{B}^{-1}$, is to start with B^{-1} and apply the sequence of elementary row operations described above.

Example 3.4 Let

$$B^{-1} = \begin{bmatrix} 1 & 2 & 3 \\ -2 & 3 & 1 \\ 4 & -3 & -2 \end{bmatrix}, \quad u = \begin{bmatrix} -4 \\ 2 \\ 2 \end{bmatrix},$$

and suppose that $\ell = 3$. Thus, our objective is to transform the vector u to the unit vector $e_3 = (0, 0, 1)$. We multiply the third row by 2 and add it to the first row. We subtract the third row from the second row. Finally, we divide the third row by 2. We obtain

$$\bar{B}^{-1} = \begin{bmatrix} 9 & -4 & -1 \\ -6 & 6 & 3 \\ 2 & -1.5 & -1 \end{bmatrix}.$$

When the matrix B^{-1} is updated in the manner we have described, we obtain an implementation of the simplex method known as the *revised simplex method*, which we summarize below.

An iteration of the revised simplex method

1. In a typical iteration, we start with a basis consisting of the basic columns $A_{B(1)}, \dots, A_{B(m)}$, an associated basic feasible solution x, and the inverse B^{-1} of the basis matrix.

2. Compute the row vector $p' = c'_B B^{-1}$ and then compute the reduced costs $\bar{c}_j = c_j - p'A_j$. If they are all nonnegative, the current basic feasible solution is optimal, and the algorithm terminates; else, choose some j for which $\bar{c}_j < 0$.

3. Compute $u = B^{-1}A_j$. If no component of u is positive, the optimal cost is $-\infty$, and the algorithm terminates.

4. If some component of u is positive, let

$$\theta^* = \min_{\{i=1,\ldots,m\,|\,u_i>0\}} \frac{x_{B(i)}}{u_i}.$$

5. Let ℓ be such that $\theta^* = x_{B(\ell)}/u_\ell$. Form a new basis by replacing $A_{B(\ell)}$ with A_j. If y is the new basic feasible solution, the values of the new basic variables are $y_j = \theta^*$ and $y_{B(i)} = x_{B(i)} - \theta^* u_i$, $i \neq \ell$.

6. Form the $m \times (m+1)$ matrix $[B^{-1} \mid u]$. Add to each one of its rows a multiple of the ℓth row to make the last column equal to the unit vector e_ℓ. The first m columns of the result is the matrix $\overline{B}^{-1}$.

The full tableau implementation

We finally describe the implementation of simplex method in terms of the so-called *full tableau*. Here, instead of maintaining and updating the matrix B^{-1}, we maintain and update the $m \times (n+1)$ matrix

$$B^{-1}[b \mid A]$$

with columns $B^{-1}b$ and $B^{-1}A_1,\ldots,B^{-1}A_n$. This matrix is called the *simplex tableau*. Note that the column $B^{-1}b$, called the *zeroth column*, contains the values of the basic variables. The column $B^{-1}A_i$ is called the ith column of the tableau. The column $u = B^{-1}A_j$ corresponding to the variable that enters the basis is called the *pivot column*. If the ℓth basic variable exits the basis, the ℓth row of the tableau is called the *pivot row*. Finally, the element belonging to both the pivot row and the pivot column is called the *pivot element*. Note that the pivot element is u_ℓ and is always positive (unless $u \leq 0$, in which case the algorithm has met the termination condition in Step 3).

The information contained in the rows of the tableau admits the following interpretation. The equality constraints are initially given to us in the form $b = Ax$. Given the current basis matrix B, these equality constraints can also be expressed in the equivalent form

$$B^{-1}b = B^{-1}Ax,$$

which is precisely the information in the tableau. In other words, the rows of the tableau provide us with the coefficients of the equality constraints $B^{-1}b = B^{-1}Ax$.

At the end of each iteration, we need to update the tableau $B^{-1}[b \mid A]$, and compute $\overline{B}^{-1}[b \mid A]$. This can be accomplished by left-multiplying the

simplex tableau with a matrix Q satisfying $QB^{-1} = \overline{B}^{-1}$. As explained earlier, this is the same as performing those elementary row operations that turn B^{-1} to $\overline{B}^{-1}$; that is, we add to each row a multiple of the pivot row to set all entries of the pivot column to zero, with the exception of the pivot element which is set to one.

Regarding the determination of the exiting column $A_{B(\ell)}$ and the stepsize θ^*, Steps 4 and 5 in the summary of the simplex method amount to the following: $x_{B(i)}/u_i$ is the ratio of the ith entry in the zeroth column of the tableau to the ith entry in the pivot column of the tableau. We only consider those i for which u_i is positive. The smallest ratio is equal to θ^* and determines ℓ.

It is customary to augment the simplex tableau by including a top row, to be referred to as the *zeroth row*. The entry at the top left corner contains the value $-c_B'x_B$, which is the negative of the current cost. (The reason for the minus sign is that it allows for a simple update rule, as will be seen shortly.) The rest of the zeroth row is the row vector of reduced costs, that is, the vector $\bar{c}' = c' - c_B'B^{-1}A$. Thus, the structure of the tableau is:

$-c_B'B^{-1}b$	$c' - c_B'B^{-1}A$
$B^{-1}b$	$B^{-1}A$

or, in more detail,

$-c_B'x_B$	$\bar{c}_1$	$\cdots$	$\bar{c}_n$
$x_{B(1)}$			
$\vdots$	$B^{-1}A_1$	$\cdots$	$B^{-1}A_n$
$x_{B(m)}$			

The rule for updating the zeroth row turns out to be identical to the rule used for the other rows of the tableau: add a multiple of the pivot row to the zeroth row to set the reduced cost of the entering variable to zero. We will now verify that this update rule produces the correct results for the zeroth row.

At the beginning of a typical iteration, the zeroth row is of the form

$$[0 \mid c'] - g'[b \mid A]$$

where $g' = c_B'B^{-1}$. Hence, the zeroth row is equal to $[0 \mid c']$ plus a linear combination of the rows of $[b \mid A]$. Let column j be the pivot column, and row ℓ be the pivot row. Note that the pivot row is of the form $h'[b \mid A]$, where the vector h' is the ℓth row of B^{-1}. Hence, after a multiple of the

pivot row is added to the zeroth row, that row is again equal to $[0 \mid \mathbf{c}']$ plus a (different) linear combination of the rows of $[\mathbf{b} \mid \mathbf{A}]$, and is of the form

$$[0 \mid \mathbf{c}'] - \mathbf{p}'[\mathbf{b} \mid \mathbf{A}],$$

for some vector $\mathbf{p}$. Recall that our update rule is such that the pivot column entry of the zeroth row becomes zero, that is,

$$c_{\overline{B}(\ell)} - \mathbf{p}'\mathbf{A}_{\overline{B}(\ell)} = c_j - \mathbf{p}'\mathbf{A}_j = 0.$$

Consider now the $\overline{B}(i)$th column for $i \neq \ell$. (This is a column corresponding to a basic variable that stays in the basis.) The zeroth row entry of that column is zero, before the change of basis, since it is the reduced cost of a basic variable. Because $\mathbf{B}^{-1}\mathbf{A}_{B(i)}$ is the ith unit vector and $i \neq \ell$, the entry in the pivot row for that column is also equal to zero. Hence, adding a multiple of the pivot row to the zeroth row of the tableau does not affect the zeroth row entry of that column, which is left at zero. We conclude that the vector $\mathbf{p}$ satisfies $c_{\overline{B}(i)} - \mathbf{p}'\mathbf{A}_{\overline{B}(i)} = 0$ for every column $\mathbf{A}_{\overline{B}(i)}$ in the new basis. This implies that $c'_{\overline{B}} - \mathbf{p}'\overline{\mathbf{B}} = \mathbf{0}$, and $\mathbf{p}' = c'_{\overline{B}}\overline{\mathbf{B}}^{-1}$. Hence, with our update rule, the updated zeroth row of the tableau is equal to

$$[0 \mid \mathbf{c}'] - c'_{\overline{B}}\overline{\mathbf{B}}^{-1}[\mathbf{b} \mid \mathbf{A}],$$

as desired.

We can now summarize the mechanics of the full tableau implementation.

An iteration of the full tableau implementation

1. A typical iteration starts with the tableau associated with a basis matrix $\mathbf{B}$ and the corresponding basic feasible solution $\mathbf{x}$.

2. Examine the reduced costs in the zeroth row of the tableau. If they are all nonnegative, the current basic feasible solution is optimal, and the algorithm terminates; else, choose some j for which $\bar{c}_j < 0$.

3. Consider the vector $\mathbf{u} = \mathbf{B}^{-1}\mathbf{A}_j$, which is the jth column (the pivot column) of the tableau. If no component of $\mathbf{u}$ is positive, the optimal cost is $-\infty$, and the algorithm terminates.

4. For each i for which u_i is positive, compute the ratio $x_{B(i)}/u_i$. Let ℓ be the index of a row that corresponds to the smallest ratio. The column $\mathbf{A}_{B(\ell)}$ exits the basis and the column $\mathbf{A}_j$ enters the basis.

5. Add to each row of the tableau a constant multiple of the ℓth row (the pivot row) so that u_ℓ (the pivot element) becomes one and all other entries of the pivot column become zero.
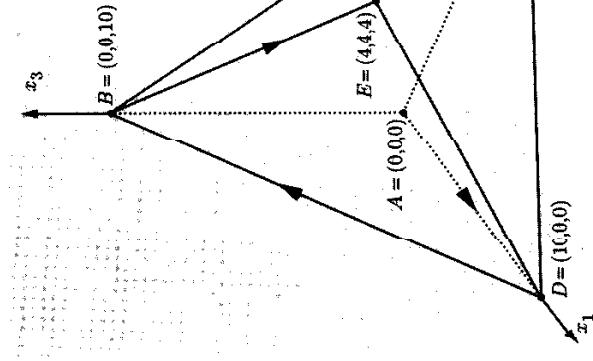
Figure 3.4: The feasible set in Example 3.5. Note that we have five extreme points. These are $A = (0,0,0)$ with cost 0, $B = (0,0,10)$ with cost -120. $C = (0,10,0)$ with cost -120, $D = (10,0,0)$ with cost -100, and $E = (4,4,4)$ with cost -136. In particular, E is the unique optimal solution.

Example 3.5 Consider the problem

$$
\begin{aligned}
\text{minimize} \quad & -10x_1 - 12x_2 - 12x_3 \\
\text{subject to} \quad & x_1 + 2x_2 + 2x_3 \le 20 \\
& 2x_1 + x_2 + 2x_3 \le 20 \\
& 2x_1 + 2x_2 + x_3 \le 20 \\
& x_1, x_2, x_3 \ge 0.
\end{aligned}
$$

The feasible set is shown in Figure 3.4.

After introducing slack variables, we obtain the following standard form problem

$$
\begin{aligned}
\text{minimize} \quad & -10x_1 - 12x_2 - 12x_3 \\
\text{subject to} \quad & x_1 + 2x_2 + 2x_3 + x_4 \qquad\qquad = 20 \\
& 2x_1 + x_2 + 2x_3 \qquad + x_5 \qquad = 20 \\
& 2x_1 + 2x_2 + x_3 \qquad\qquad + x_6 = 20 \\
& x_1, \ldots, x_6 \ge 0.
\end{aligned}
$$

Note that $\mathbf{x} = (0,0,0,20,20,20)$ is a basic feasible solution and can be used to start the algorithm. Let accordingly, $B(1) = 4$, $B(2) = 5$, and $B(3) = 6$. The

corresponding basis matrix is the identity matrix I. To obtain the zeroth row of the initial tableau, we note that $c_B = 0$ and, therefore, $c'_B x_B = 0$ and $\bar{c} = c$. Hence, we have the following initial tableau:

	x_1	x_2	x_3	x_4	x_5	x_6
0	-10	-12	-12	0	0	0
$x_4 =$ 20	1	2	2	1	0	0
$x_5 =$ 20	2*	1	2	0	1	0
$x_6 =$ 20	2	2	1	0	0	1

We note a few conventions in the format of the above tableau: the label x_i on top of the ith column indicates the variable associated with that column. The labels "$x_i =$" to the left of the tableau tell us which are the basic variables and in what order. For example, the first basic variable $x_{B(1)}$ is x_4, and its value is 20. Similarly, $x_{B(2)} = x_5 = 20$, and $x_{B(3)} = x_6 = 20$. Strictly speaking, these labels are not quite necessary. We know that the column in the tableau associated with the first basic variable must be the first unit vector. Once we observe that the column associated with the variable x_4 is the first unit vector, it follows that x_4 is the first basic variable.

We continue with our example. The reduced cost of x_1 is negative and we let that variable enter the basis. The pivot column is $u = (1, 2, 2)$. We form the ratios $x_{B(i)}/u_i$, $i = 1, 2, 3$; the smallest ratio corresponds to $i = 2$ and $i = 3$. We break this tie by choosing $\ell = 2$. This determines the pivot element, which we indicate by an asterisk. The second basic variable $x_{B(2)}$, which is x_5, exits the basis. The new basis is given by $\overline{B}(1) = 4, \overline{B}(2) = 1$, and $\overline{B}(3) = 6$. We multiply the pivot row by 5 and add it to the zeroth row. We multiply the pivot row by 1/2 and subtract it from the first row. We subtract the pivot row from the third row. Finally, we divide the pivot row by 2. This leads us to the new tableau:

	x_1	x_2	x_3	x_4	x_5	x_6
100	0	-7	-2	0	5	0
$x_4 =$ 10	0	1.5	1*	1	-0.5	0
$x_1 =$ 10	1	0.5	1	0	0.5	0
$x_6 =$ 0	0	1	-1	0	-1	1

The corresponding basic feasible solution is $x = (10, 0, 0, 10, 0, 0)$. In terms of the original variables x_1, x_2, x_3, we have moved to point $D = (10, 0, 0)$ in Figure 3.4. Note that this is a degenerate basic feasible solution, because the basic variable x_6 is equal to zero. This agrees with Figure 3.4 where we observe that there are four active constraints at point D.

We have mentioned earlier that the rows of the tableau (other than the zeroth row) amount to a representation of the equality constraints $B^{-1}Ax = B^{-1}b$, which are equivalent to the original constraints $Ax = b$. In our current

example, the tableau indicates that the equality constraints can be written in the equivalent form:

$$10 = 1.5x_2 + x_3 + x_4 - 0.5x_5$$
$$10 = x_1 + 0.5x_2 + x_3 + 0.5x_5$$
$$0 = x_2 - x_3 - x_5 + x_6.$$

We now return to the simplex method. With the current tableau, the variables x_2 and x_3 have negative reduced costs. Let us choose x_3 to be the one that enters the basis. The pivot column is $u = (1, 1, -1)$. Since $u_3 < 0$, we only form the ratios $x_{B(i)}/u_i$, for $i = 1, 2$. There is again a tie, which we break by letting $\ell = 1$, and the first basic variable, x_4, exits the basis. The pivot element is again indicated by an asterisk. After carrying out the necessary elementary row operations, we obtain the following new tableau:

	x_1	x_2	x_3	x_4	x_5	x_6
120	0	-4	0	2	4	0
$x_3 =$ 10	0	1.5	1	1	-0.5	0
$x_1 =$ 0	1	-1	0	-1	1	0
$x_6 =$ 10	0	2.5*	0	1	-1.5	1

In terms of Figure 3.4, we have moved to point $B = (0, 0, 10)$, and the cost has been reduced to −120. At this point, x_2 is the only variable with negative reduced cost. We bring x_2 into the basis, x_6 exits, and the resulting tableau is:

	x_1	x_2	x_3	x_4	x_5	x_6
136	4	0	0	3.3	1.6	1.6
$x_3 =$ 4	0	0	1	0.4	0.4	-0.6
$x_1 =$ 4	1	0	0	-0.3	0.4	0.4
$x_2 =$ 4	0	1	0	0.4	-0.6	0.4

We have now reached point E in Figure 3.4. Its optimality is confirmed by observing that all reduced costs are nonnegative.

In this example, the simplex method took three changes of basis to reach the optimal solution, and it traced the path $A - D - B - E$ in Figure 3.4. With different pivoting rules, a different path would have been traced. Could the simplex method have solved the problem by tracing the path $A - D - E$, which involves only two edges, with only two iterations? The answer is no. The initial and final bases differ in three columns, and therefore at least three basis changes are required. In particular, if the method were to trace the path $A - D - E$, there would be a degenerate change of basis at point D (with no edge being traversed) which would again bring the total to three.

Example 3.6 This example shows that the simplex method can indeed cycle. We consider a problem described in terms of the following initial tableau.

		x_1	x_2	x_3	x_4	x_5	x_6	x_7
	3	-3/4	20	-1/2	6	0	0	0
$x_5 =$	0	1/4*	-8	-1	9	1	0	0
$x_6 =$	0	1/2	-12	-1/2	3	0	1	0
$x_7 =$	1	0	0	1	0	0	0	1

We use the following pivoting rules:

(a) We select a nonbasic variable with the most negative reduced cost $\bar{c}_j$ to be the one that enters the basis.

(b) Out of all basic variables that are eligible to exit the basis, we select the one with the smallest subscript.

We then obtain the following sequence of tableaux (the pivot element is indicated by an asterisk):

		x_1	x_2	x_3	x_4	x_5	x_6	x_7
	3	0	-4	-7/2	33	3	0	0
$x_1 =$	0	1	-32	-4	36	4	0	0
$x_6 =$	0	0	4*	3/2	-15	-2	1	0
$x_7 =$	1	0	0	1	0	0	0	1

		x_1	x_2	x_3	x_4	x_5	x_6	x_7
	3	0	0	-2	18	1	1	0
$x_1 =$	0	1	0	8*	-84	-12	8	0
$x_2 =$	0	0	1	3/8	-15/4	-1/2	1/4	0
$x_7 =$	1	0	0	1	0	0	0	1

		x_1	x_2	x_3	x_4	x_5	x_6	x_7
	3	1/4	0	0	-3	-2	3	0
$x_3 =$	0	1/8	0	1	-21/2	-3/2	1	0
$x_2 =$	0	-3/64	1	0	3/16*	1/16	-1/8	0
$x_7 =$	1	-1/8	0	0	21/2	3/2	-1	1

		x_1	x_2	x_3	x_4	x_5	x_6	x_7
	3	-1/2	16	0	0	-1	1	0
$x_3 =$	0	-5/2	56	1	0	2*	-6	0
$x_4 =$	0	-1/4	16/3	0	1	1/3	-2/3	0
$x_7 =$	1	5/2	-56	0	0	-2	6	1

		x_1	x_2	x_3	x_4	x_5	x_6	x_7
	3	-7/4	44	1/2	0	0	-2	0
$x_5 =$	0	-5/4	28	1/2	0	1	-3	0
$x_4 =$	0	1/6	-4	-1/6	1	0	1/3*	0
$x_7 =$	1	0	0	1	0	0	0	1

		x_1	x_2	x_3	x_4	x_5	x_6	x_7
	3	-3/4	20	-1/2	6	0	0	0
$x_5 =$	0	1/4	-8	-1	9	1	0	0
$x_6 =$	0	1/2	-12	-1/2	3	0	1	0
$x_7 =$	1	0	0	1	0	0	0	1

After six pivots, we have the same basis and the same tableau that we started with. At each basis change, we had $\theta^* = 0$. In particular, for each intermediate tableau, we had the same feasible solution and the same cost. The same sequence of pivots can be repeated over and over, and the simplex method never terminates.

Comparison of the full tableau and the revised simplex methods

Let us pretend that the problem is changed to

$$\text{minimize} \quad c'x + 0'y$$
$$\text{subject to} \quad Ax + Iy = b$$
$$x, y \geq 0.$$

We implement the simplex method on this new problem, except that we never allow any of the components of the vector y to become basic. Then, the simplex method performs basis changes as if the vector y were entirely

absent. Note also that the vector of reduced costs in the augmented problem is

$$[c' \mid 0'] - c'_B B^{-1}[A \mid I] = [\bar{c}' \mid -c'_B B^{-1}].$$

Thus, the simplex tableau for the augmented problem takes the form

$-c'_B B^{-1}b$	$\bar{c}'$	$-c'_B B^{-1}$
$B^{-1}b$	$B^{-1}A$	B^{-1}

In particular, by following the mechanics of the full tableau method on the above tableau, the inverse basis matrix B^{-1} is made available at each iteration. We can now think of the revised simplex method as being essentially the same as the full tableau method applied to the above augmented problem, except that the part of the tableau containing $B^{-1}A$ is never formed explicitly; instead, once the entering variable x_j is chosen, the pivot column $B^{-1}A_j$ is computed on the fly. Thus, the revised simplex method is just a variant of the full tableau method, with more efficient bookkeeping. If the revised simplex method also updates the zeroth row entries that lie on top of B^{-1} (by the usual elementary operations), the simplex multipliers $p' = c'_B B^{-1}$ become available, thus eliminating the need for solving the linear system $p'B = c'_B$ at each iteration.

We now discuss the relative merits of the two methods. The full tableau method requires a constant (and small) number of arithmetic operations for updating each entry of the tableau. Thus, the amount of computation per iteration is proportional to the size of the tableau, which is $O(mn)$. The revised simplex method uses similar computations to update B^{-1} and $c'_B B^{-1}$, and since only $O(m^2)$ entries are updated, the computational requirements per iteration are $O(m^2)$. In addition, the reduced cost of each variable x_j can be computed by forming the inner product $p'A_j$, which requires $O(m)$ operations. In the worst case, the reduced cost of every variable is computed, for a total of $O(mn)$ computations per iteration. Since $m \leq n$, the worst-case computational effort per iteration is $O(mn + m^2) = O(mn)$, under either implementation. On the other hand, if we consider a pivoting rule that evaluates one reduced cost at a time, until a negative reduced cost is found, a typical iteration of the revised simplex method might require a lot less work. In the best case, if the first reduced cost computed is negative, and the corresponding variable is chosen to enter the basis, the total computational effort is only $O(m^2)$. The conclusion is that the revised simplex method cannot be slower than the full tableau method, and could be much faster during most iterations.

Another important element in favor of the revised simplex method is that memory requirements are reduced from $O(mn)$ to $O(m^2)$. As n is often much larger than m, this effect can be quite significant. It could be counterargued that the memory requirements of the revised simplex method

are also $O(mn)$ because of the need to store the matrix A. However, in most large scale problems that arise in applications, the matrix A is very sparse (has many zero entries) and can be stored compactly. (Note that the sparsity of A does not usually help in the storage of the full simplex tableau because even if A and B are sparse, $B^{-1}A$ is not sparse, in general.) We summarize this discussion in the following table:

	Full tableau	Revised simplex
Memory	$O(mn)$	$O(m^2)$
Worst-case time	$O(mn)$	$O(mn)$
Best-case time	$O(mn)$	$O(m^2)$

Table 3.1: Comparison of the full tableau method and revised simplex. The time requirements refer to a single iteration.

Practical performance enhancements

Practical implementations of the simplex method aimed at solving problems of moderate or large size incorporate a number of additional ideas from numerical linear algebra which we briefly mention.

The first idea is related to *reinversion*. Recall that at each iteration of the revised simplex method, the inverse basis matrix B^{-1} is updated according to certain rules. Each such iteration may introduce roundoff or truncation errors which accumulate and may eventually lead to highly inaccurate results. For this reason, it is customary to recompute the matrix B^{-1} from scratch once in a while. The efficiency of such reinversions can be greatly enhanced by using suitable data structures and certain techniques from computational linear algebra.

Another set of ideas is related to the way that the inverse basis matrix B^{-1} is represented. Suppose that a reinversion has been just carried out and B^{-1} is available. Subsequent to the current iteration of the revised simplex method, we have the option of generating explicitly and storing the new inverse basis matrix $\overline{B}^{-1}$. An alternative that carries the same information, is to store a matrix Q such that $QB^{-1} = \overline{B}^{-1}$. Note that Q basically prescribes which elementary row operations need to be applied to B^{-1} in order to produce $\overline{B}^{-1}$. It is not a full matrix, and can be completely specified in terms of m coefficients: for each row, we need to know what multiple of the pivot row must be added to it.

Suppose now that we wish to solve the system $\overline{B}u = A_j$ for u, where A_j is the entering column, as is required by the revised simplex method. We have $u = \overline{B}^{-1}A_j = QB^{-1}A_j$, which shows that we can first compute

absent. Note also that the vector of reduced costs in the augmented problem is

$$[c' \mid 0'] - c_B'B^{-1}[A \mid I] = [\bar{c}' \mid -c_B'B^{-1}].$$

Thus, the simplex tableau for the augmented problem takes the form

$-c_B'B^{-1}b$	$\bar{c}'$	$-c_B'B^{-1}$
$B^{-1}b$	$B^{-1}A$	B^{-1}

In particular, by following the mechanics of the full tableau method on the above tableau, the inverse basis matrix B^{-1} is made available at each iteration. We can now think of the revised simplex method as being essentially the same as the full tableau method applied to the above augmented problem, except that the part of the tableau containing $B^{-1}A$ is never formed explicitly; instead, once the entering variable x_j is chosen, the pivot column $B^{-1}A_j$ is computed on the fly. Thus, the revised simplex method is just a variant of the full tableau method, with more efficient bookkeeping. If the revised simplex method also updates the zeroth row entries that lie on top of B^{-1} (by the usual elementary operations), the simplex multipliers $p' = c_B'B^{-1}$ become available, thus eliminating the need for solving the linear system $p'B = c_B'$ at each iteration.

We now discuss the relative merits of the two methods. The full tableau method requires a constant (and small) number of arithmetic operations for updating each entry of the tableau. Thus, the amount of computation per iteration is proportional to the size of the tableau, which is $O(mn)$. The revised simplex method uses similar computations to update B^{-1} and $c_B'B^{-1}$, and since only $O(m^2)$ entries are updated, the computational requirements per iteration are $O(m^2)$. In addition, the reduced cost of each variable x_j can be computed by forming the inner product pA_j, which requires $O(m)$ operations. In the worst case, the reduced cost of every variable is computed, for a total of $O(mn)$ computations per iteration. Since $m \le n$, the worst-case computational effort per iteration is $O(mn + m^2) = O(mn)$, under either implementation. On the other hand, if we consider a pivoting rule that evaluates one reduced cost at a time, until a negative reduced cost is found, a typical iteration of the revised simplex method might require a lot less work. In the best case, if the first reduced cost computed is negative, and the corresponding variable is chosen to enter the basis, the total computational effort is only $O(m^2)$. The conclusion is that the revised simplex method cannot be slower than the full tableau method, and could be much faster during most iterations.

Another important element in favor of the revised simplex method is that memory requirements are reduced from $O(mn)$ to $O(m^2)$. As n is often much larger than m, this effect can be quite significant. It could be counterargued that the memory requirements of the revised simplex method

are also $O(mn)$ because of the need to store the matrix A. However, in most large scale problems that arise in applications, the matrix A is very sparse (has many zero entries) and can be stored compactly. (Note that the sparsity of A does not usually help in the storage of the full simplex tableau because even if A and B are sparse, $B^{-1}A$ is not sparse, in general.) We summarize this discussion in the following table:

	Full tableau	Revised simplex
Memory	$O(mn)$	$O(m^2)$
Worst-case time	$O(mn)$	$O(mn)$
Best-case time	$O(mn)$	$O(m^2)$

Table 3.1: Comparison of the full tableau method and revised simplex. The time requirements refer to a single iteration.

Practical performance enhancements

Practical implementations of the simplex method aimed at solving problems of moderate or large size incorporate a number of additional ideas from numerical linear algebra which we briefly mention.

The first idea is related to *reinversion*. Recall that at each iteration of the revised simplex method, the inverse basis matrix B^{-1} is updated according to certain rules. Each such iteration may introduce roundoff or truncation errors which accumulate and may eventually lead to highly inaccurate results. For this reason, it is customary to recompute the matrix B^{-1} from scratch once in a while. The efficiency of such reinversions can be greatly enhanced by using suitable data structures and certain techniques from computational linear algebra.

Another set of ideas is related to the way that the inverse basis matrix B^{-1} is represented. Suppose that a reinversion has been just carried out and B^{-1} is available. Subsequent to the current iteration of the revised simplex method, we have the option of generating explicitly and storing the new inverse basis matrix $\overline{B}^{-1}$. An alternative that carries the same information, is to store a matrix Q such that $QB^{-1} = \overline{B}^{-1}$. Note that Q basically prescribes which elementary row operations need to be applied to B^{-1} in order to produce $\overline{B}^{-1}$. It is not a full matrix, and can be completely specified in terms of m coefficients: for each row, we need to know what multiple of the pivot row must be added to it

Suppose now that we wish to solve the system $\overline{B}u = A_j$ for u, where A_j is the entering column, as is required by the revised simplex method. We have $u = \overline{B}^{-1}A_j = QB^{-1}A_j$, which shows that we can first compute

absent. Note also that the vector of reduced costs in the augmented problem is

$$[c' \mid 0'] - c_B' B^{-1}[A \mid I] = [\bar{c}' \mid -c_B' B^{-1}].$$

Thus, the simplex tableau for the augmented problem takes the form

$-c_B' B^{-1} b$	$\bar{c}'$	$-c_B' B^{-1}$
$B^{-1} b$	$B^{-1} A$	B^{-1}

In particular, by following the mechanics of the full tableau method on the above tableau, the inverse basis matrix B^{-1} is made available at each iteration. We can now think of the revised simplex method as being essentially the same as the full tableau method applied to the above augmented problem except that the part of the tableau containing $B^{-1}A$ is never formed explicitly; instead, once the entering variable x_j is chosen, the pivot column $B^{-1}A_j$ is computed on the fly. Thus, the revised simplex method is just a variant of the full tableau method, with more efficient bookkeeping. If the revised simplex method also updates the zeroth row entries that lie on top of B^{-1} (by the usual elementary operations), the simplex multipliers $p' = c_B' B^{-1}$ become available, thus eliminating the need for solving the linear system $p'B = c_B'$ at each iteration.

We now discuss the relative merits of the two methods. The full tableau method requires a constant (and small) number of arithmetic operations for updating each entry of the tableau. Thus, the amount of computation per iteration is proportional to the size of the tableau, which is $O(mn)$. The revised simplex method uses similar computations to update B^{-1} and $c_B' B^{-1}$, and since only $O(m^2)$ entries are updated, the computational requirements per iteration are $O(m^2)$. In addition, the reduced cost of each variable $\bar{c}_j$ can be computed by forming the inner product $p'A_j$, which requires $O(m)$ operations. In the worst case, the reduced cost of every variable is computed, for a total of $O(mn)$ computations per iteration. Since $m \leq n$, the worst-case computational effort per iteration is $O(mn+m^2) = O(mn)$ under either implementation. On the other hand, if we consider a pivoting rule that evaluates one reduced cost at a time, until a negative reduced cost is found, a typical iteration of the revised simplex method might require a lot less work. In the best case, if the first reduced cost computed is negative, and the corresponding variable is chosen to enter the basis, the total computational effort is only $O(m^2)$. The conclusion is that the revised simplex method cannot be slower than the full tableau method, and could be much faster during most iterations.

Another important element in favor of the revised simplex method is that memory requirements are reduced from $O(mn)$ to $O(m^2)$. As n is often much larger than m, this effect can be quite significant. It could be counterargued that the memory requirements of the revised simplex method

are also $O(mn)$ because of the need to store the matrix A. However, in most large scale problems that arise in applications, the matrix A is very sparse (has many zero entries) and can be stored compactly. (Note that the sparsity of A does not usually help in the storage of the full simplex tableau because even if A and B are sparse, $B^{-1}A$ is not sparse, in general.) We summarize this discussion in the following table:

	Full tableau	Revised simplex
Memory	$O(mn)$	$O(m^2)$
Worst-case time	$O(mn)$	$O(mn)$
Best-case time	$O(mn)$	$O(m^2)$

Table 3.1: Comparison of the full tableau method and revised simplex. The time requirements refer to a single iteration.

Practical performance enhancements

Practical implementations of the simplex method aimed at solving problems of moderate or large size incorporate a number of additional ideas from numerical linear algebra which we briefly mention.

The first idea is related to *reinversion*. Recall that at each iteration of the revised simplex method, the inverse basis matrix B^{-1} is updated according to certain rules. Each such iteration may introduce roundoff or truncation errors which accumulate and may eventually lead to highly inaccurate results. For this reason, it is customary to recompute the matrix B^{-1} from scratch once in a while. The efficiency of such reinversions can be greatly enhanced by using suitable data structures and certain techniques from computational linear algebra.

Another set of ideas is related to the way that the inverse basis matrix B^{-1} is represented. Suppose that a reinversion has been just carried out and B^{-1} is available. Subsequent to the current iteration of the revised simplex method, we have the option of generating explicitly and storing the new inverse basis matrix $\overline{B}^{-1}$. An alternative that carries the same information, is to store a matrix Q such that $QB^{-1} = \overline{B}^{-1}$. Note that Q basically prescribes which elementary row operations need to be applied to B^{-1} in order to produce $\overline{B}^{-1}$. It is not a full matrix, and can be completely specified in terms of m coefficients: for each row, we need to know what multiple of the pivot row must be added to it.

Suppose now that we wish to solve the system $\overline{B}u = A_j$ for u, where A_j is the entering column, as is required by the revised simplex method. We have $u = \overline{B}^{-1}A_j = QB^{-1}A_j$, which shows that we can first compute

$B^{-1}A_j$ and then left-multiply by Q (equivalently, apply a sequence of elementary row operations) to produce u. The same idea can also be used to represent the inverse basis matrix after several simplex iterations, as a product of the initial inverse basis matrix and several sparse matrices like Q.

The last idea we mention is the following. Subsequent to a "reinversion," one does not usually compute B^{-1} explicitly, but B^{-1} is instead represented in terms of sparse triangular matrices with a special structure.

The methods discussed in this subsection are designed to accomplish two objectives: improve numerical stability (minimize the effect of roundoff errors) and exploit sparsity in the problem data to improve both running time and memory requirements. These methods have a critical effect in practice. Besides having a better chance of producing numerically trustworthy results, they can also speed up considerably the running time of the simplex method. These techniques lie much closer to the subject of numerical linear algebra, as opposed to optimization, and for this reason we do not pursue them in any greater depth.

3.4 Anticycling: lexicography and Bland's rule

In this section, we discuss anticycling rules under which the simplex method is guaranteed to terminate, thus extending Theorem 3.3 to degenerate problems. As an important corollary, we conclude that if the optimal cost is finite, then there exists an optimal basis, that is, a basis satisfying $B^{-1}b \geq 0$ and $\bar{c}' = c' - c_B'B^{-1}A \geq 0'$.

Lexicography

We present here the lexicographic pivoting rule and prove that it prevents the simplex method from cycling. Historically, this pivoting rule was derived by analyzing the behavior of the simplex method on a nondegenerate problem obtained by means of a small perturbation of the right-hand side vector b. This connection is pursued in Exercise 3.15.

We start with a definition.

Definition 3.5 A vector $u \in \Re^n$ is said to be *lexicographically larger (or smaller) than another vector* $v \in \Re^n$ *if* $u \neq v$ *and the first nonzero component of* $u - v$ *is positive (or negative, respectively). Symbolically, we write*
$$u \overset{L}{>} v \qquad or \qquad u \overset{L}{<} v.$$

For example,

$$(0, 2, 3, 0) \overset{L}{>} (0, 2, 1, 4),$$

$$(0, 4, 5, 0) \overset{L}{<} (1, 2, 1, 2).$$

Lexicographic pivoting rule

1. Choose an entering column A_j arbitrarily, as long as its reduced cost $\bar{c}_j$ is negative. Let $u = B^{-1}A_j$ be the jth column of the tableau.

2. For each i with $u_i > 0$, divide the ith row of the tableau (including the entry in the zeroth column) by u_i and choose the lexicographically smallest row. If row ℓ is lexicographically smallest, then the ℓth basic variable $x_{B(\ell)}$ exits the basis.

Example 3.7 Consider the following tableau (the zeroth row is omitted), and suppose that the pivot column is the third one ($j = 3$).

1	0	5	3	...
2	4	6	-1	...
3	0	7	9	...

Note that there is a tie in trying to determine the exiting variable because $x_{B(1)}/u_1 = 1/3$ and $x_{B(3)}/u_3 = 3/9 = 1/3$. We divide the first and third rows of the tableau by $u_1 = 3$ and $u_3 = 9$, respectively, to obtain:

1/3	0	5/3	1	...
*	*	*	*	...
1/3	0	7/9	1	...

The tie between the first and third rows is resolved by performing a lexicographic comparison. Since $7/9 < 5/3$, the third row is chosen to be the pivot row, and the variable $x_{B(3)}$ exits the basis.

We note that the lexicographic pivoting rule always leads to a unique choice for the exiting variable. Indeed, if this were not the case, two of the rows in the tableau would have to be proportional. But if two rows of the matrix $B^{-1}A$ are proportional, the matrix $B^{-1}A$ has rank smaller than m and, therefore, A also has rank less than m, which contradicts our standing assumption that A has linearly independent rows.

Theorem 3.4 *Suppose that the simplex algorithm starts with all the rows in the simplex tableau, other than the zeroth row, exicographically positive. Suppose that the lexicographic pivoting rule is followed. Then:*

(a) *Every row of the simplex tableau, other than the zeroth row, remains lexicographically positive throughout the algorithm.*

(b) *The zeroth row strictly increases lexicographically at each iteration.*

(c) *The simplex method terminates after a finite number of iterations.*

Proof.

(a) Suppose that all rows of the simplex tableau, other than the zeroth row, are lexicographically positive at the beginning of a simplex iteration. Suppose that x_j enters the basis and that the pivot row is the ℓth row. According to the lexicographic pivoting rule, we have $u_\ell > 0$ and

$$\frac{(\ell\text{th row})}{u_\ell} \prec \frac{(i\text{th row})}{u_i}, \qquad \text{if } i \neq \ell \text{ and } u_i > 0. \qquad (3.5)$$

To determine the new tableau, the ℓth row is divided by the positive pivot element u_ℓ and, therefore, remains lexicographically positive. Consider the ith row and suppose that $u_i < 0$. In order to zero the (i,j)th entry of the tableau, we need to add a positive multiple of the pivot row to the ith row. Due to the lexicographic positivity of both rows, the ith row will remain lexicographically positive after this addition. Finally, consider the ith row for the case where $u_i > 0$ and $i \neq \ell$. We have

$$(\text{new } i\text{th row}) = (\text{old } i\text{th row}) - \frac{u_i}{u_\ell}(\text{old } \ell\text{th row}).$$

Because of the lexicographic inequality (3.5), which is satisfied by the old rows, the new ith row is also lexicographically positive.

(b) At the beginning of an iteration, the reduced cost in the pivot column is negative. In order to make it zero, we need to add a positive multiple of the pivot row. Since the latter row is lexicographically positive, the zeroth row increases lexicographically.

(c) Since the zeroth row increases lexicographically at each iteration, it never returns to a previous value. Since the zeroth row is determined completely by the current basis, no basis can be repeated twice and the simplex method must terminate after a finite number of iterations. □

The lexicographic pivoting rule is straightforward to use if the simplex method is implemented in terms of the full tableau. It can also be used

in conjunction with the revised simplex method, provided that the inverse basis matrix $\mathbf{B}^{-1}$ is formed explicitly (see Exercise 3.16). On the other hand, in sophisticated implementations of the revised simplex method, the matrix $\mathbf{B}^{-1}$ is never computed explicitly, and the lexicographic rule is not really suitable.

We finally note that in order to apply the lexicographic pivoting rule, an initial tableau with lexicographically positive rows is required. Let us assume that an initial tableau is available (methods for obtaining an initial tableau are discussed in the next section). We can then rename the variables so that the basic variables are the first m ones. This is equivalent to rearranging the tableau so that the first m columns of $\mathbf{B}^{-1}\mathbf{A}$ are the m unit vectors. The resulting tableau has lexicographically positive rows, as desired.

Bland's rule

The smallest subscript pivoting rule, also known as Bland's rule, is as follows.

Smallest subscript pivoting rule

1. Find the smallest j for which the reduced cost $\bar{c}_j$ is negative and have the column $\mathbf{A}_j$ enter the basis.

2. Out of all variables x_i that are tied in the test for choosing an exiting variable, select the one with the smallest value of i.

This pivoting rule is compatible with an implementation of the revised simplex method in which the reduced costs of the nonbasic variables are computed one at a time, in the natural order, until a negative one is discovered. Under this pivoting rule, it is known that cycling never occurs and the simplex method is guaranteed to terminate after a finite number of iterations.

3.5 Finding an initial basic feasible solution

In order to start the simplex method, we need to find an initial basic feasible solution. Sometimes this is straightforward. For example, suppose that we are dealing with a problem involving constraints of the form $\mathbf{Ax} \leq \mathbf{b}$, where $\mathbf{b} \geq \mathbf{0}$. We can then introduce nonnegative slack variables $\mathbf{s}$ and rewrite the constraints in the form $\mathbf{Ax} + \mathbf{s} = \mathbf{b}$. The vector $(\mathbf{x}, \mathbf{s})$ defined by $\mathbf{x} = \mathbf{0}$ and $\mathbf{s} = \mathbf{b}$ is a basic feasible solution and the corresponding basis matrix is the identity. In general, however, finding an initial basic feasible solution is not easy and requires the solution of an auxiliary linear programming problem, as will be seen shortly.

Consider the problem

$$\begin{array}{ll} \text{minimize} & \mathbf{c'x} \\ \text{subject to} & \mathbf{Ax = b} \\ & \mathbf{x \ge 0}. \end{array}$$

By possibly multiplying some of the equality constraints by -1, we can assume, without loss of generality, that $\mathbf{b \ge 0}$. We now introduce a vector $\mathbf{y} \in \Re^m$ of *artificial variables* and use the simplex method to solve the auxiliary problem

$$\begin{array}{ll} \text{minimize} & y_1 + y_2 + \cdots + y_m \\ \text{subject to} & \mathbf{Ax + y = b} \\ & \mathbf{x \ge 0} \\ & \mathbf{y \ge 0}. \end{array}$$

Initialization is easy for the auxiliary problem: by setting $\mathbf{x = 0}$ and $\mathbf{y = b}$, we have a basic feasible solution and the corresponding basis matrix is the identity.

If $\mathbf{x}$ is a feasible solution to the original problem, this choice of $\mathbf{x}$ together with $\mathbf{y = 0}$, yields a zero cost solution to the auxiliary problem. Therefore, if the optimal cost in the auxiliary problem is nonzero, we conclude that the original problem is infeasible. If on the other hand, we obtain a zero cost solution to the auxiliary problem, it must satisfy $\mathbf{y = 0}$, and $\mathbf{x}$ is a feasible solution to the original problem.

At this point, we have accomplished our objectives only partially. We have a method that either detects infeasibility or finds a feasible solution to the original problem. However, in order to initialize the simplex method for the original problem, we need a basic feasible solution, an associated basis matrix $\mathbf{B}$, and – depending on the implementation – the corresponding tableau. All this is straightforward if the simplex method, applied to the auxiliary problem, terminates with a basis matrix $\mathbf{B}$ consisting exclusively of columns of $\mathbf{A}$. We can simply drop the columns that correspond to the artificial variables and continue with the simplex method on the original problem, using $\mathbf{B}$ as the starting basis matrix.

Driving artificial variables out of the basis

The situation is more complex if the original problem is feasible, the simplex method applied to the auxiliary problem terminates with a feasible solution $\mathbf{x}^*$ to the original problem, but some of the artificial variables are in the final basis. (Since the final value of the artificial variables is zero, this implies that we have a degenerate basic feasible solution to the auxiliary problem.) Let k be the number of columns of $\mathbf{A}$ that belong to the final basis ($k < m$) and, without loss of generality, assume that these are the columns $\mathbf{A}_{B(1)}, \ldots, \mathbf{A}_{B(k)}$. (In particular, $x_{B(1)}, \ldots, x_{B(k)}$ are the only variables

that can be at nonzero level.) Note that the columns $\mathbf{A}_{B(1)}, \ldots, \mathbf{A}_{B(k)}$ must be linearly independent since they are part of a basis. Under our standard assumption that the matrix $\mathbf{A}$ has full rank, the columns of $\mathbf{A}$ span $\Re^m$, and we can choose $m - k$ additional columns $\mathbf{A}_{B(k+1)}, \ldots, \mathbf{A}_{B(m)}$ of $\mathbf{A}$, to obtain a set of m linearly independent columns, that is, a basis consisting exclusively of columns of $\mathbf{A}$. With this basis, all nonbasic components of $\mathbf{x}^*$ are at zero level, and it follows that $\mathbf{x}^*$ is the basic feasible solution associated with this new basis as well. At this point, the artificial variables and the corresponding columns of the tableau can be dropped.

The procedure we have just described is called *driving the artificial variables out of the basis*, and depends crucially on the assumption that the matrix $\mathbf{A}$ has rank m. After all, if $\mathbf{A}$ has rank less than m, constructing a basis for $\Re^m$ using the columns of $\mathbf{A}$ is impossible and there exist redundant equality constraints that must be eliminated, as described by Theorem 2.5 in Section 2.3. All of the above can be carried out mechanically, in terms of the simplex tableau, in the following manner:

Suppose that the ℓth basic variable is an artificial variable, which is in the basis at zero level. We examine the ℓth row of the tableau and find some j such that the ℓth entry of $\mathbf{B}^{-1}\mathbf{A}_j$ is nonzero. We claim that $\mathbf{A}_j$ is linearly independent from the columns $\mathbf{A}_{B(1)}, \ldots, \mathbf{A}_{B(k)}$. To see this, note that $\mathbf{B}^{-1}\mathbf{A}_{B(i)} = \mathbf{e}_i$, $i = 1, \ldots, k$, and since $k < \ell$, the ℓth entry of these vectors is zero. It follows that the ℓth entry of any linear combination of the vectors $\mathbf{B}^{-1}\mathbf{A}_{B(1)}, \ldots, \mathbf{B}^{-1}\mathbf{A}_{B(k)}$ is also equal to zero. Since the ℓth entry of $\mathbf{B}^{-1}\mathbf{A}_j$ is nonzero, this vector is not a linear combination of the vectors $\mathbf{B}^{-1}\mathbf{A}_{B(1)}, \ldots, \mathbf{B}^{-1}\mathbf{A}_{B(k)}$. Equivalently, $\mathbf{A}_j$ is not a linear combination of the vectors $\mathbf{A}_{B(1)}, \ldots, \mathbf{A}_{B(k)}$, which proves our claim. We now bring $\mathbf{A}_j$ into the basis and have the ℓth basic variable exit the basis. This is accomplished in the usual manner: perform those elementary row operations that replace $\mathbf{B}^{-1}\mathbf{A}_j$ by the ℓth unit vector. The only difference from the usual mechanics of the simplex method is that the pivot element (the ℓth entry of $\mathbf{B}^{-1}\mathbf{A}_j$) could be negative. Because the ℓth basic variable was zero, adding a multiple of the ℓth row to the other rows does not change the values of the basic variables. This means that after the change of basis, we are still at the same basic feasible solution to the auxiliary problem, but we have reduced the number of basic artificial variables by one. We repeat this procedure as many times as needed until all artificial variables are driven out of the basis.

Let us now assume that the ℓth row of $\mathbf{B}^{-1}\mathbf{A}$ is zero, in which case the above described procedure fails. Note that the ℓth row of $\mathbf{B}^{-1}\mathbf{A}$ is equal to $\mathbf{g'A}$, where $\mathbf{g'}$ is the ℓth row of $\mathbf{B}^{-1}$. Hence, $\mathbf{g'A} = \mathbf{0'}$ for some nonzero vector $\mathbf{g}$, and the matrix $\mathbf{A}$ has linearly dependent rows. Since we are dealing with a feasible problem, we must also have $\mathbf{g'b} = 0$. Thus, the constraint $\mathbf{g'Ax = g'b}$ is redundant and can be eliminated (cf. Theorem 2.5 in Section 2.3). Since this constraint is the information provided by the ℓth row of the tableau, we can eliminate that row and continue from there.

Example 3.8 Consider the linear programming problem:

$$\begin{aligned}
\text{minimize} \quad & x_1 + x_2 + x_3 \\
\text{subject to} \quad & x_1 + 2x_2 + 3x_3 &= 3 \\
& -x_1 + 2x_2 + 6x_3 &= 2 \\
& 4x_2 + 9x_3 &= 5 \\
& 3x_3 + x_4 &= 1 \\
& x_1, \ldots, x_4 \geq 0.
\end{aligned}$$

In order to find a feasible solution, we form the auxiliary problem

$$\begin{aligned}
\text{minimize} \quad & & & & x_5 + x_6 + x_7 + x_8 \\
\text{subject to} \quad & x_1 + 2x_2 + 3x_3 & & + x_5 & &= 3 \\
& -x_1 + 2x_2 + 6x_3 & & + x_6 & &= 2 \\
& 4x_2 + 9x_3 & & + x_7 & &= 5 \\
& 3x_3 + x_4 & & + x_8 & &= 1 \\
& x_1, \ldots, x_8 \geq 0.
\end{aligned}$$

A basic feasible solution to the auxiliary problem is obtained by letting $(x_5, x_6, x_7, x_8) = \mathbf{b} = (3, 2, 5, 1)$. The corresponding basis matrix is the identity. Furthermore, we have $c_B = (1, 1, 1, 1)$. We evaluate the reduced cost of each one of the original variables x_i, which is $-c_B' A_i$, and form the initial tableau:

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
-11	0	-8	-21	-1	0	0	0	0
$x_5 = 3$	1	2	3	0	1	0	0	0
$x_6 = 2$	-1	2	6	0	0	1	0	0
$x_7 = 5$	0	4	9	0	0	0	1	0
$x_8 = 1$	0	0	3	1^*	0	0	0	1

We bring x_4 into the basis and have x_8 exit the basis. The basis matrix B is still the identity and only the zeroth row of the tableau changes. We obtain:

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
-10	0	-8	-18	0	0	0	0	1
$x_5 = 3$	1	2	3	0	1	0	0	0
$x_6 = 2$	-1	2	6	0	0	1	0	0
$x_7 = 5$	0	4	9	0	0	0	1	0
$x_4 = 1$	0	0	3^*	1	0	0	0	1

We now bring x_3 into the basis and have x_4 exit the basis. The new tableau is:

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
-4	0	-8	0	6	0	0	0	7
$x_5 = 2$	1	2	0	-1	1	0	0	-1
$x_6 = 0$	-1	2^*	0	-2	0	1	0	-2
$x_7 = 2$	0	4	0	-3	0	0	1	-3
$x_3 = 1/3$	0	0	1	1/3	0	0	0	1/3

We now bring x_2 into the basis and x_6 exits. Note that this is a degenerate pivot with $\theta^* = 0$. The new tableau is:

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
-4	-4	0	0	-2	0	4	0	-1
$x_5 = 2$	2^*	0	0	1	1	-1	0	1
$x_2 = 0$	$-1/2$	1	0	-1	0	1/2	0	-1
$x_7 = 2$	2	0	0	1	0	-2	1	1
$x_3 = 1/3$	0	0	1	1/3	0	0	0	1/3

We now have x_1 enter the basis and x_5 exit the basis. We obtain the following tableau:

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
0	0	0	0	0	2	2	0	1
$x_1 = 1$	1	0	0	1/2	1/2	$-1/2$	0	1/2
$x_2 = 1/2$	0	1	0	$-3/4$	1/4	1/4	0	$-3/4$
$x_7 = 0$	0	0	0	0	-1	-1	1	0
$x_3 = 1/3$	0	0	1	1/3	0	0	0	1/3

Note that the cost in the auxiliary problem has dropped to zero, indicating that we have a feasible solution to the original problem. However, the artificial variable x_7 is still in the basis, at zero level. In order to obtain a basic feasible solution to the original problem, we need to drive x_7 out of the basis. Note that x_7 is the third basic variable and that the third entry of the columns $B^{-1}A_j$, $j = 1, \ldots, 4$, associated with the original variables, is zero. This indicates that the matrix A has linearly dependent rows. At this point, we remove the third row of the tableau, because it corresponds to a redundant constraint, and also remove all of the artificial variables. This leaves us with the following initial tableau for the

original problem:

		x_1	x_2	x_3	x_4
	*	*	*	*	*
$x_1 =$	1	1	0	0	1/2
$x_2 =$	1/2	0	1	0	-3/4
$x_3 =$	1/3	0	0	1	1/3

We may now compute the reduced costs of the original variables, fill in the zeroth row of the tableau, and start executing the simplex method on the original problem.

We observe that in this example, the artificial variable x_8 was unnecessary. Instead of starting with $x_8 = 1$, we could have started with $x_4 = 1$ thus eliminating the need for the first pivot. More generally, whenever there is a variable that appears in a single constraint and with a positive coefficient (slack variables being the typical example), we can always set that variable be in the initial basis and we do not have to associate an artificial variable with that constraint.

The two-phase simplex method

We can now summarize a complete algorithm for linear programming problems in standard form.

Phase I:

1. By multiplying some of the constraints by -1, change the problem so that $\mathbf{b} \geq \mathbf{0}$.

2. Introduce artificial variables $y_1, \ldots, y_m$, if necessary, and apply the simplex method to the auxiliary problem with cost $\sum_{i=1}^{m} y_i$.

3. If the optimal cost in the auxiliary problem is positive, the original problem is infeasible and the algorithm terminates.

4. If the optimal cost in the auxiliary problem is zero, a feasible solution to the original problem has been found. If no artificial variable is in the final basis, the artificial variables and the corresponding columns are eliminated, and a feasible basis for the original problem is available.

5. If the ℓth basic variable is an artificial one, examine the ℓth entry of the columns $\mathbf{B}^{-1}\mathbf{A}_j$, $j = 1, \ldots, n$. If all of these entries are zero, the ℓth row represents a redundant constraint and is eliminated. Otherwise, if the ℓth entry of the jth column is nonzero, apply a change of basis (with this entry serving as the pivot

element): the ℓth basic variable exits and x_j enters the basis. Repeat this operation until all artificial variables are driven out of the basis.

Phase II:

1. Let the final basis and tableau obtained from Phase I be the initial basis and tableau for Phase II.

2. Compute the reduced costs of all variables for this initial basis, using the cost coefficients of the original problem.

3. Apply the simplex method to the original problem.

The above two-phase algorithm is a complete method, in the sense that it can handle all possible outcomes. As long as cycling is avoided (due to either nondegeneracy, an anticycling rule, or luck), one of the following possibilities will materialize:

(a) If the problem is infeasible, this is detected at the end of Phase I.

(b) If the problem is feasible but the rows of $\mathbf{A}$ are linearly dependent, this is detected and corrected at the end of Phase I, by eliminating redundant equality constraints.

(c) If the optimal cost is equal to $-\infty$, this is detected while running Phase II.

(d) Else, Phase II terminates with an optimal solution.

The big-M method

We close by mentioning an alternative approach, the *big-M method*, that combines the two phases into a single one. The idea is to introduce a cost function of the form

$$\sum_{j=1}^{n} c_j x_j + M \sum_{i=1}^{m} y_i,$$

where M is a large positive constant, and where y_i are the same artificial variables as in Phase I simplex. For a sufficiently large choice of M, if the original problem is feasible and its optimal cost is finite, all of the artificial variables are eventually driven to zero (Exercise 3.26), which takes us back to the minimization of the original cost function. In fact, there is no reason for fixing a numerical value for M. We can leave M as an undetermined parameter and let the reduced costs be functions of M. Whenever M is compared to another number (in order to determine whether a reduced cost is negative), M will be always treated as being larger.

Example 3.9 We consider the same linear programming problem as in Example 3.8:

$$\begin{aligned}
\text{minimize} \quad & x_1 + x_2 + x_3 \\
\text{subject to} \quad & x_1 + 2x_2 + 3x_3 = 3 \\
& -x_1 + 2x_2 + 6x_3 = 2 \\
& \quad\;\; 4x_2 + 9x_3 = 5 \\
& \quad\;\; 3x_3 + x_4 = 1 \\
& x_1, \ldots, x_4 \geq 0.
\end{aligned}$$

We use the big-M method in conjunction with the following auxiliary problem, in which the unnecessary artificial variable x_8 is omitted.

$$\begin{aligned}
\text{minimize} \quad & x_1 + x_2 + x_3 + Mx_5 + Mx_6 + Mx_7 \\
\text{subject to} \quad & x_1 + 2x_2 + 3x_3 + x_5 = 3 \\
& -x_1 + 2x_2 + 6x_3 + x_6 = 2 \\
& \quad\;\; 4x_2 + 9x_3 + x_7 = 5 \\
& \quad\;\; 3x_3 + x_4 = 1 \\
& x_1, \ldots, x_7 \geq 0.
\end{aligned}$$

A basic feasible solution to the auxiliary problem is obtained by letting $(x_5, x_6, x_7, x_4) = \mathbf{b} = (3, 2, 5, 1)$. The corresponding basis matrix is the identity. Furthermore, we have $\mathbf{c}_B = (M, M, M, 0)$. We evaluate the reduced cost of each one of the original variables x_i, which is $c_i - \mathbf{c}_B'\mathbf{A}_i$, and form the initial tableau:

	x_1	x_2	x_3	x_4	x_5	x_6	x_7
$-10M$	1	$-8M+1$	$-18M+1$	0	0	0	0
$x_5 =$ 3	1	2	3	0	1	0	0
$x_6 =$ 2	-1	2	6	0	0	1	0
$x_7 =$ 5	0	4	9	0	0	0	1
$x_4 =$ 1	0	0	3*	1	0	0	0

The reduced cost of x_3 is negative when M is large enough. We therefore bring x_3 into the basis and have x_4 exit. Note that in order to set the reduced cost of x_3 to zero, we need to multiply the pivot row by $6M - 1/3$ and add it to the zeroth row. The new tableau is:

	x_1	x_2	x_3	x_4	x_5	x_6	x_7
$-4M-1/3$	1	$-8M+1$	0	$6M-1/3$	0	0	0
$x_5 =$ 2	1	2	0	-1	1	0	0
$x_6 =$ 0	-1	2*	0	-2	0	1	0
$x_7 =$ 2	0	4	0	-3	0	0	1
$x_3 =$ 1/3	0	0	1	1/3	0	0	0

The reduced cost of x_2 is negative when M is large enough. We therefore bring x_2 into the basis and x_6 exits. Note that this is a degenerate pivot with $\theta^* = 0$.

The new tableau is:

	x_1	x_2	x_3	x_4	x_5	x_6	x_7
$-4M-\dfrac{1}{3}$	$-4M+\dfrac{3}{2}$	0	0	$-2M+\dfrac{2}{3}$	0	$4M-\dfrac{1}{2}$	0
$x_5 =$ 2	2*	0	0	1	1	-1	0
$x_2 =$ 0	$-1/2$	1	0	-1	0	1/2	0
$x_7 =$ 2	2	0	0	1	0	-2	1
$x_3 =$ 1/3	0	0	1	1/3	0	0	0

We now have x_1 enter and x_5 exit the basis. We obtain the following tableau:

	x_1	x_2	x_3	x_4	x_5	x_6	x_7
$-11/6$	0	0	0	$-1/12$	$2M-3/4$	$2M+1/4$	0
$x_1 =$ 1	1	0	0	1/2	1/2	$-1/2$	0
$x_2 =$ 1/2	0	1	0	$-3/4$	1/4	1/4	0
$x_7 =$ 0	0	0	0	0	-1	-1	1
$x_3 =$ 1/3	0	0	1	1/3*	0	0	0

We now bring x_4 into the basis and x_3 exits. The new tableau is:

	x_1	x_2	x_3	x_4	x_5	x_6	x_7
$-7/4$	0	0	1/4	0	$2M-3/4$	$2M+1/4$	0
$x_1 =$ 1/2	1	0	$-3/2$	0	1/2	$-1/2$	0
$x_2 =$ 5/4	0	1	9/4	0	1/4	1/4	0
$x_7 =$ 0	0	0	0	0	-1	-1	1
$x_4 =$ 1	0	0	3	1	0	0	0

With M large enough, all of the reduced costs are nonnegative and we have an optimal solution to the auxiliary problem. In addition, all of the artificial variables have been driven to zero, and we have an optimal solution to the original problem.

3.6 Column geometry and the simplex method

In this section, we introduce an alternative way of visualizing the workings of the simplex method. This approach provides some insights into why the

simplex method appears to be efficient in practice.

We consider the problem

$$\text{minimize} \quad \mathbf{c}'\mathbf{x}$$
$$\text{subject to} \quad \mathbf{A}\mathbf{x} = \mathbf{b} \qquad (3.6)$$
$$\mathbf{e}'\mathbf{x} = 1$$
$$\mathbf{x} \geq \mathbf{0},$$

where $\mathbf{A}$ is an $m \times n$ matrix and $\mathbf{e}$ is the n-dimensional vector with all components equal to one. Although this might appear to be a special type of a linear programming problem, it turns out that every problem with a bounded feasible set can be brought into this form (Exercise 3.28). The constraint $\mathbf{e}'\mathbf{x} = 1$ is called the *convexity constraint*. We also introduce an auxiliary variable z defined by $z = \mathbf{c}'\mathbf{x}$. If $\mathbf{A}_1, \mathbf{A}_2, \ldots, \mathbf{A}_n$ are the n columns of $\mathbf{A}$, we are dealing with the problem of minimizing z subject to the nonnegativity constraints $\mathbf{x} \geq \mathbf{0}$, the convexity constraint $\sum_{i=1}^n x_i = 1$, and the constraint

$$x_1 \begin{bmatrix} \mathbf{A}_1 \\ c_1 \end{bmatrix} + x_2 \begin{bmatrix} \mathbf{A}_2 \\ c_2 \end{bmatrix} + \cdots + x_n \begin{bmatrix} \mathbf{A}_n \\ c_n \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ z \end{bmatrix}.$$

In order to capture this problem geometrically, we view the horizontal plane as an m-dimensional space containing the columns of $\mathbf{A}$, and we view the vertical axis as the one-dimensional space associated with the cost components c_i. Then, each point in the resulting three-dimensional space corresponds to a point $(\mathbf{A}_i, c_i)$; see Figure 3.5.

In this geometry, our objective is to construct a vector $(\mathbf{b}, z)$, which is a convex combination of the vectors $(\mathbf{A}_i, c_i)$, such that z is as small as possible. Note that the vectors of the form $(\mathbf{b}, z)$ lie on a vertical line, which we call the *requirement line*, and which intersects the horizontal plane at $(\mathbf{b}, c_i)$; the requirement line does not intersect the convex hull of the points $(\mathbf{A}_i, c_i)$, the problem is infeasible. If it does intersect it, the problem is feasible and an optimal solution corresponds to the lowest point in the intersection of the convex hull and the requirement line. For example, in Figure 3.6, the requirement line intersects the convex hull of the points $(\mathbf{A}_i, c_i)$; the point G corresponds to an optimal solution, and its height is the optimal cost.

We now need some terminology.

Definition 3.6

(a) A collection of vectors $\mathbf{y}^1, \ldots, \mathbf{y}^{k+1}$ in $\Re^n$ are said to be *affinely independent* if the vectors $\mathbf{y}^1 - \mathbf{y}^{k+1}, \mathbf{y}^2 - \mathbf{y}^{k+1}, \ldots, \mathbf{y}^k - \mathbf{y}^{k+1}$ are linearly independent. *(Note that we must have $k \leq n$.)*

(b) The convex hull of $k + 1$ affinely independent vectors in $\Re^n$ is called a *k-dimensional simplex*.
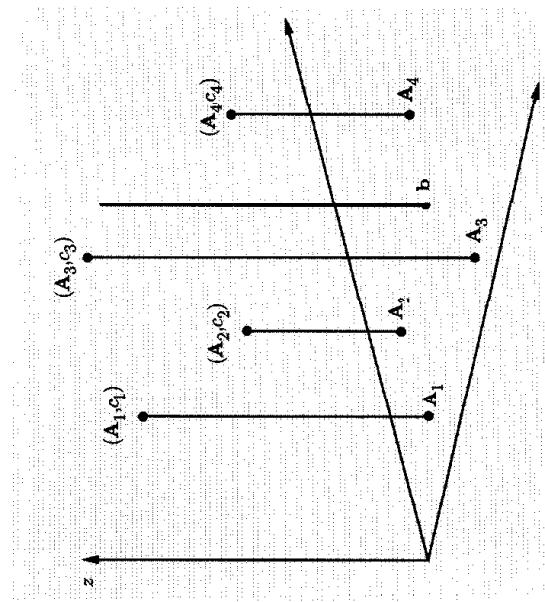

Figure 3.5: The column geometry.

Thus, three points are either collinear or they are affinely independent and determine a two-dimensional simplex (a triangle). Similarly, four points either lie on the same plane, or they are affinely independent and determine a three-dimensional simplex (a pyramid).

Let us now give an interpretation of basic feasible solutions to problem (3.6) in this geometry. Since we have added the convexity constraint, we have a total of $m + 1$ equality constraints. Thus, a basic feasible solution is associated with a collection of $m + 1$ linearly independent columns $(\mathbf{A}_i, 1)$ of the linear programming problem (3.6). These are in turn associated with $m + 1$ of the points $(\mathbf{A}_i, c_i)$, which we call *basic points*; the remaining points $(\mathbf{A}_i, c_i)$ are called the *nonbasic points*. It is not hard to show that the $m + 1$ basic points are affinely independent (Exercise 3.29) and, therefore, their convex hull is an m-dimensional simplex, which we call the *basic simplex*. Let the requirement line intersect the m-dimensional basic simplex at some point $(\mathbf{b}, z)$. The vector of weights x_i used in expressing $(\mathbf{b}, z)$ as a convex combination of the basic points, is the current basic feasible solution, and z represents its cost. For example, in Figure 3.6, the shaded triangle CDF is the basic simplex, and the point H corresponds to a basic feasible solution associated with the basic points C, D, and F.

Let us now interpret a change of basis geometrically. In a change of basis, a new point $(\mathbf{A}_j, c_j)$ becomes basic, and one of the currently basic points is to become nonbasic. For example, in Figure 3.6, if C, D, F, are the current basic points, we could make point B basic, replacing F
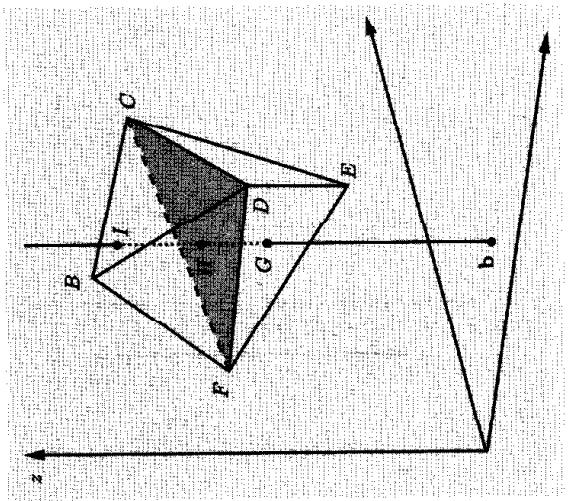
Figure 3.6: Feasibility and optimality in the column geometry.

(even though this turns out not to be profitable). The new basic simplex would be the convex hull of B, C, D, and the new basic feasible solution would correspond to point I. Alternatively, we could make point E basic, replacing C, and the new basic feasible solution would now correspond to point G. After a change of basis, the intercept of the requirement line with the new basic simplex is lower, and hence the cost decreases, if and only if the new basic point is below the plane that passes through the old basic points; we refer to the latter plane as the *dual plane*. For example, point E is below the dual plane and having it enter the basis is profitable; this is not the case for point B. In fact, the vertical distance from the dual plane to a point $(\mathbf{A}_j, c_j)$ is equal to the reduced cost of the associated variable x_j (Exercise 3.30); requiring the new basic point to be below the dual plane is therefore equivalent to requiring the entering column to have negative reduced cost.

We discuss next the selection of the basic point that will exit the basis. Each possible choice of the exiting point leads to a different basic simplex. These m basic simplices, together with the original basic simplex (before the change of basis) form the boundary (the faces) of an $(m+1)$-dimensional simplex. The requirement line exits this $(m+1)$-dimensional simplex through its top face and must therefore enter it by crossing some other face. This determines which one of the potential basic simplices will be obtained after the change of basis. In reference to Figure 3.6, the basic

points C, D, F, determine a two-dimensional basic simplex. If point E is to become basic, we obtain a three-dimensional simplex (pyramid) with vertices C, D, E, F. The requirement line exits the pyramid through its top face with vertices C, D, F. It enters the pyramid through the face with vertices D, E, F; this is the new basic simplex.

We can now visualize pivoting through the following physical analogy. Think of the original basic simplex with vertices C, D, F, as a solid object anchored at its vertices. Grasp the corner of the basic simplex at the vertex C leaving the basis, and pull the corner down to the new basic point E. While so moving, the simplex will hinge, or *pivot*, on its anchor and stretch down to the lower position. The somewhat peculiar terms (e.g., "simplex", "pivot") associated with the simplex method have their roots in this column geometry.

Example 3.10 Consider the problem illustrated in Figure 3.7, in which $m = 1$, and the following pivoting rule: choose a point $(\mathbf{A}_i, c_i)$ below the dual plane to become basic, whose vertical distance from the dual plane is largest. According to Exercise 3.30, this is identical to the pivoting rule that selects an entering variable with the most negative reduced cost. Starting from the initial basic simplex consisting of the points $(\mathbf{A}_3, c_3)$, $(\mathbf{A}_6, c_6)$, the next basic simplex is determined by the points $(\mathbf{A}_3, c_3)$, $(\mathbf{A}_5, c_5)$, and the next one by the points $(\mathbf{A}_5, c_5)$, $(\mathbf{A}_8, c_8)$. In particular, the simplex method only takes two pivots in this case. This example indicates why the simplex method may require a rather small number of pivots, even when the number of underlying variables is large.
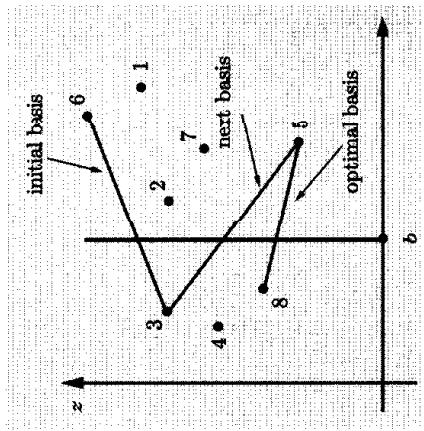

Figure 3.7: The simplex method finds the optimal basis after two iterations. Here, the point indicated by a number i corresponds to the vector $(\mathbf{A}_i, c_i)$.

required by the algorithm, when applied to a random problem drawn according to the postulated probability distribution. Unfortunately, there is no natural probability distribution over the set of linear programming problems. Nevertheless, a fair number of positive results have been obtained for a few different types of probability distributions. In one such result, a set of vectors $c, a_1, \ldots, a_m \in \Re^n$ and scalars $b_1, \ldots, b_m$ is given For $i = 1, \ldots, m$, we introduce either constraint $a_i'x \le b_i$ or $a_i'x \ge b_i$, with equal probability. We then have 2^m possible linear programming problems, and suppose that L of them are feasible. Haimovich (1983) has established that under a rather special pivoting rule, the simplex method requires no more than $n/2$ iterations, on the average over those L feasible problems. This linear dependence on the size of the problem agrees with observed behavior; some empirical evidence is discussed in Chapter 12.

3.8 Summary

This chapter was centered or the development of the simplex method, which is a complete algorithm for solving linear programming problems in standard form. The cornerstones of the simplex method are:

(a) the optimality conditions (nonnegativity of the reduced costs) that allow us to test whether the current basis is optimal;

(b) a systematic method for performing basis changes whenever the optimality conditions are violated.

At a high level, the simplex method simply moves from one extreme point of the feasible set to another, each time reducing the cost, until an optimal solution is reached. However, the lower level details of the simplex method, relating to the organization of the required computations and the associated bookkeeping, play an important role. We have described three different implementations: the naive one, the revised simplex method, and the full tableau implementation. Abstractly, they are all equivalent, but their mechanics are quite different. Practical implementations of the simplex method follow our general description of the revised simplex method, but the details are different, because an explicit computation of the inverse basis matrix is usually avoided.

We have seen that degeneracy can cause substantial difficulties, including the possibility of nonterminating behavior (cycling). This is because in the presence of degeneracy, a change of basis may keep us at the same basic feasible solution, with no cost improvement resulting. Cycling can be avoided if suitable rules for choosing the entering and exiting variables (pivoting rules) are applied (e.g., Bland's rule or the lexicographic pivoting rule).

Starting the simplex method requires an initial basic feasible solution, and an associated tableau. These are provided by the Phase I simplex algorithm, which is nothing but the simplex method applied to an auxiliary

problem. We saw that the changeover from Phase I to Phase II involves some delicate steps whenever some artificial variables are in the final basis constructed by the Phase I algorithm.

The simplex method is a rather efficient algorithm and is incorporated in most of the commercial codes for linear programming. While the number of pivots can be an exponential function of the number of variables and constraints in the worst case, its observed behavior is a lot better, hence the practical usefulness of the method.

3.9 Exercises

Exercise 3.1 (Local minima of convex functions) Let $f : \Re^n \to \Re$ be a convex function and let $S \subset \Re^n$ be a convex set. Let x^* be an element of S. Suppose that x^* is a local optimum for the problem of minimizing $f(x)$ over S; that is, there exists some $\epsilon > 0$ such that $f(x^*) \le f(x)$ for all $x \in S$ for which $\|x - x^*\| \le \epsilon$. Prove that x^* is globally optimal; that is, $f(x^*) \le f(x)$ for all $x \in S$.

Exercise 3.2 (Optimality conditions) Consider the problem of minimizing $c'x$ over a polyhedron P. Prove the following:

(a) A feasible solution x is optimal if and only if $c'd \ge 0$ for every feasible direction d at x.

(b) A feasible solution x is the unique optimal solution if and only if $c'd > 0$ for every nonzero feasible direction d at x.

Exercise 3.3 Let x be an element of the standard form polyhedron $P = \{x \in \Re^n \mid Ax = b, x \ge 0\}$. Prove that a vector $d \in \Re^n$ is a feasible direction at x if and only if $Ad = 0$ and $d_i \ge 0$ for every i such that $x_i = 0$.

Exercise 3.4 Consider the problem of minimizing $c'x$ over the set $P = \{x \in \Re^n \mid Ax = b, Dx \le f, Ex \le g\}$. Let x^* be an element of P that satisfies $Dx^* = f, Ex^* < g$. Show that the set of feasible directions at the point x^* is the set

$$\{d \in \Re^n \mid Ad = 0, Dd \le 0\}.$$

Exercise 3.5 Let $P = \{x \in \Re^3 \mid x_1 + x_2 + x_3 = 1, x \ge 0\}$ and consider the vector $x = (0, 0, 1)$. Find the set of feasible directions at x.

Exercise 3.6 (Conditions for a unique optimum) Let x be a basic feasible solution associated with some basis matrix B. Prove the following:

(a) If the reduced cost of every nonbasic variable is positive, then x is the unique optimal solution.

(b) If x is the unique optimal solution and is nondegenerate, then the reduced cost of every nonbasic variable is positive.

3.7. The example showing that the simplex method can take an exponential number of iterations is due to Klee and Minty (1972). The Hirsch conjecture was made by Hirsch in 1957. The first results on the average case behavior of the simplex method were obtained by Borgwardt (1982) and Smale (1983). Schrijver (1986) contains an overview of the early research in this area, as well as proof of the $n/2$ bound on the number of pivots due to Haimovich (1983).

3.9. The results in Exercises 3.10 and 3.11, which deal with the smallest examples of cycling, are due to Marshall and Suurballe (1969). The matrix inversion lemma [Exercise 3.13(a)] is known as the Sherman-Morrison formula.

Chapter 4
Duality theory

Contents

In this chapter, we start with a linear programming problem, called the primal, and introduce another linear programming problem, called the dual. Duality theory deals with the relation between these two problems and uncovers the deeper structure of linear programming. It is a powerful theoretical tool that has numerous applications, provides new geometric insights, and leads to another algorithm for linear programming (the dual simplex method).

4.1 Motivation

Duality theory can be motivated as an outgrowth of the Lagrange multiplier method, often used in calculus to minimize a function subject to equality constraints. For example, in order to solve the problem

$$\text{minimize} \quad x^2 + y^2$$
$$\text{subject to} \quad x + y = 1,$$

we introduce a Lagrange multiplier p and form the Lagrangean $L(x,y,p)$ defined by

$$L(x,y,p) = x^2 + y^2 + p(1 - x - y).$$

While keeping p fixed, we minimize the Lagrangean over all x and y, subject to no constraints, which can be done by setting $\partial L/\partial x$ and $\partial L/\partial y$ to zero. The optimal solution to this unconstrained problem is

$$x = y = \frac{p}{2}$$

and depends on p. The constraint $x + y = 1$ gives us the additional relation $p = 1$, and the optimal solution to the original problem is $x = y = 1/2$.

The main idea in the above example is the following. Instead of enforcing the hard constraint $x + y = 1$, we allow it to be violated and associate a Lagrange multiplier, or price, p with the amount $1 - x - y$ by which it is violated. This leads to the unconstrained minimization of $x^2 + y^2 + p(1 - x - y)$. When the price is properly chosen ($p = 1$, in our example), the optimal solution to the constrained problem is also optimal for the unconstrained problem. In particular, under that specific value of p, the presence or absence of the hard constraint does not affect the optimal cost.

The situation in linear programming is similar: we associate a price variable with each constraint and start searching for prices under which the presence or absence of the constraints does not affect the optimal cost. It turns out that the right prices can be found by solving a new linear programming problem, called the *dual* of the original. We now motivate the form of the dual problem.

Consider the standard form problem

$$\text{minimize} \quad \mathbf{c}'\mathbf{x}$$
$$\text{subject to} \quad \mathbf{A}\mathbf{x} = \mathbf{b}$$
$$\mathbf{x} \geq \mathbf{0},$$

which we call the *primal* problem, and let $\mathbf{x}^*$ be an optimal solution, assumed to exist. We introduce a *relaxed* problem in which the constraint $\mathbf{A}\mathbf{x} = \mathbf{b}$ is replaced by a penalty $\mathbf{p}'(\mathbf{b} - \mathbf{A}\mathbf{x})$, where $\mathbf{p}$ is a price vector of the same dimension as $\mathbf{b}$. We are then faced with the problem

$$\text{minimize} \quad \mathbf{c}'\mathbf{x} + \mathbf{p}'(\mathbf{b} - \mathbf{A}\mathbf{x})$$
$$\text{subject to} \quad \mathbf{x} \geq \mathbf{0}.$$

Let $g(\mathbf{p})$ be the optimal cost for the relaxed problem, as a function of the price vector $\mathbf{p}$. The relaxed problem allows for more options than those present in the primal problem, and we expect $g(\mathbf{p})$ to be no larger than the optimal primal cost $\mathbf{c}'\mathbf{x}^*$. Indeed,

$$g(\mathbf{p}) = \min_{\mathbf{x} \geq 0} \left[\mathbf{c}'\mathbf{x} + \mathbf{p}'(\mathbf{b} - \mathbf{A}\mathbf{x}) \right] \leq \mathbf{c}'\mathbf{x}^* + \mathbf{p}'(\mathbf{b} - \mathbf{A}\mathbf{x}^*) = \mathbf{c}'\mathbf{x}^*,$$

where the last inequality follows from the fact that $\mathbf{x}^*$ is a feasible solution to the primal problem, and satisfies $\mathbf{A}\mathbf{x}^* = \mathbf{b}$. Thus, each $\mathbf{p}$ leads to a lower bound $g(\mathbf{p})$ for the optimal cost $\mathbf{c}'\mathbf{x}^*$. The problem

$$\text{maximize} \quad g(\mathbf{p})$$
$$\text{subject to} \quad \text{no constraints}$$

can be then interpreted as a search for the tightest possible lower bound of this type, and is known as the *dual* problem. The main result in duality theory asserts that the optimal cost in the dual problem is equal to the optimal cost $\mathbf{c}'\mathbf{x}^*$ in the primal. In other words, when the prices are chosen according to an optimal solution for the dual problem, the option of violating the constraints $\mathbf{A}\mathbf{x} = \mathbf{b}$ is of no value.

Using the definition of $g(\mathbf{p})$, we have

$$g(\mathbf{p}) = \min_{\mathbf{x} \geq 0} \left[\mathbf{c}'\mathbf{x} + \mathbf{p}'(\mathbf{b} - \mathbf{A}\mathbf{x}) \right]$$
$$= \mathbf{p}'\mathbf{b} + \min_{\mathbf{x} \geq 0}(\mathbf{c}' - \mathbf{p}'\mathbf{A})\mathbf{x}.$$

Note that

$$\min_{\mathbf{x} \geq 0}(\mathbf{c}' - \mathbf{p}'\mathbf{A})\mathbf{x} = \begin{cases} 0, & \text{if } \mathbf{c}' - \mathbf{p}'\mathbf{A} \geq 0', \\ -\infty, & \text{otherwise.} \end{cases}$$

In maximizing $g(\mathbf{p})$, we only need to consider those values of $\mathbf{p}$ for which $g(\mathbf{p})$ is not equal to $-\infty$. We therefore conclude that the dual problem is the same as the linear programming problem

$$\text{maximize} \quad \mathbf{p}'\mathbf{b}$$
$$\text{subject to} \quad \mathbf{p}'\mathbf{A} \leq \mathbf{c}'.$$

In the preceding example, we started with the equality constraint $\mathbf{Ax} = \mathbf{b}$ and we ended up with no constraints on the sign of the price vector $\mathbf{p}$. If the primal problem had instead inequality constraints of the form $\mathbf{Ax} \geq \mathbf{b}$, they could be replaced by $\mathbf{Ax} - \mathbf{s} = \mathbf{b}$, $\mathbf{s} \geq \mathbf{0}$. The equality constraint can be written in the form

$$[\mathbf{A} \,|\, -\mathbf{I}]\begin{bmatrix}\mathbf{x}\\\mathbf{s}\end{bmatrix} = \mathbf{0},$$

which leads to the dual constraints

$$\mathbf{p}'[\mathbf{A} \,|\, -\mathbf{I}] \leq [\mathbf{c}' \,|\, \mathbf{0}'],$$

or, equivalently,

$$\mathbf{p}'\mathbf{A} \leq \mathbf{c}', \qquad \mathbf{p} \geq \mathbf{0}.$$

Also, if the vector $\mathbf{x}$ is free rather than sign-constrained, we use the fact

$$\min_{\mathbf{x}}(\mathbf{c}' - \mathbf{p}'\mathbf{A})\mathbf{x} = \begin{cases} 0, & \text{if } \mathbf{c}' - \mathbf{p}'\mathbf{A} = \mathbf{0}' \\ -\infty, & \text{otherwise,} \end{cases}$$

to end up with the constraint $\mathbf{p}'\mathbf{A} = \mathbf{c}'$ in the dual problem. These considerations motivate the general form of the dual problem which we introduce in the next section.

In summary, the construction of the dual of a primal minimization problem can be viewed as follows. We have a vector of parameters (dual variables) $\mathbf{p}$, and for every $\mathbf{p}$ we have a method for obtaining a lower bound on the optimal primal cost. The dual problem is a maximization problem that looks for the tightest such lower bound. For some vectors $\mathbf{p}$, the corresponding lower bound is equal to $-\infty$, and does not carry any useful information. Thus, we only need to maximize over those $\mathbf{p}$ that lead to nontrivial lower bounds, and this is what gives rise to the dual constraints.

4.2   The dual problem

Let $\mathbf{A}$ be a matrix with rows $\mathbf{a}'_i$ and columns $\mathbf{A}_j$. Given a *primal* problem with the structure shown on the left, its *dual* is defined to be the maximization problem shown on the right:

$$
\begin{array}{ll}
\text{minimize} & \mathbf{c}'\mathbf{x} \\
\text{subject to} & \mathbf{a}'_i\mathbf{x} \geq b_i, \quad i \in M_1, \\
& \mathbf{a}'_i\mathbf{x} \leq b_i, \quad i \in M_2, \\
& \mathbf{a}'_i\mathbf{x} = b_i, \quad i \in M_3, \\
& x_j \geq 0, \quad j \in N_1, \\
& x_j \leq 0, \quad j \in N_2, \\
& x_j \text{ free}, \quad j \in N_3,
\end{array}
\qquad
\begin{array}{ll}
\text{maximize} & \mathbf{p}'\mathbf{b} \\
\text{subject to} & p_i \geq 0, \quad i \in M_1, \\
& p_i \leq 0, \quad i \in M_2, \\
& p_i \text{ free}, \quad i \in M_3, \\
& \mathbf{p}'\mathbf{A}_j \leq c_j, \quad j \in N_1, \\
& \mathbf{p}'\mathbf{A}_j \geq c_j, \quad j \in N_2, \\
& \mathbf{p}'\mathbf{A}_j = c_j, \quad j \in N_3.
\end{array}
$$

Notice that for each constraint in the primal (other than the sign constraints), we introduce a variable in the dual problem; for each variable in the primal, we introduce a constraint in the dual. Depending on whether the primal constraint is an equality or inequality constraint, the corresponding dual variable is either free or sign-constrained, respectively. In addition, depending on whether a variable in the primal problem is free or sign-constrained, we have an equality or inequality constraint, respectively, in the dual problem. We summarize these relations in Table 4.1.

PRIMAL	minimize	maximize	DUAL
constraints	$\geq b_i$	≥ 0	**variables**
	$\leq b_i$	≤ 0	
	$= b_i$	free	
variables	≥ 0	$\leq c_j$	**constraints**
	≤ 0	$\geq c_j$	
	free	$= c_j$	

Table 4.1: Relation between primal and dual variables and constraints.

If we start with a maximization problem, we can always convert it into an equivalent minimization problem, and then form its dual according to the rules we have described. However, to avoid confusion, we will adhere to the convention that the primal is a minimization problem, and its dual is a maximization problem. Finally, we will keep referring to the objective function in the dual problem as a "cost" that is being maximized.

A problem and its dual can be stated more compactly, in matrix notation, if a particular form is assumed for the primal. We have, for example, the following pairs of primal and dual problems:

$$
\begin{array}{ll}
\text{minimize} & \mathbf{c}'\mathbf{x} \\
\text{subject to} & \mathbf{Ax} = \mathbf{b} \\
& \mathbf{x} \geq \mathbf{0},
\end{array}
\qquad
\begin{array}{ll}
\text{maximize} & \mathbf{p}'\mathbf{b} \\
\text{subject to} & \mathbf{p}'\mathbf{A} \leq \mathbf{c}',
\end{array}
$$

and

$$
\begin{array}{ll}
\text{minimize} & \mathbf{c}'\mathbf{x} \\
\text{subject to} & \mathbf{Ax} \geq \mathbf{b},
\end{array}
\qquad
\begin{array}{ll}
\text{maximize} & \mathbf{p}'\mathbf{b} \\
\text{subject to} & \mathbf{p}'\mathbf{A} = \mathbf{c}' \\
& \mathbf{p} \geq \mathbf{0}.
\end{array}
$$

Example 4.1 Consider the primal problem shown on the left and its dual shown

on the right:

minimize　　$x_1 + 2x_2 + 3x_3$
subject to　$-x_1 + 3x_2 = 5$
　　　　　　$2x_1 - x_2 + 3x_3 \geq 6$
　　　　　　　　　　　$x_3 \leq 4$
　　　　　　$x_1 \geq 0$
　　　　　　$x_2 \leq 0$
　　　　　　x_3 free,

maximize　　$5p_1 + 6p_2 + 4p_3$
subject to　p_1 free
　　　　　　$p_2 \geq 0$
　　　　　　$p_3 \leq 0$
　　　　　　$-p_1 + 2p_2 \leq 1$
　　　　　　$3p_1 - p_2 \geq 2$
　　　　　　$3p_2 + p_3 = 3.$

We transform the dual into an equivalent minimization problem, rename the variables from p_1, p_2, p_3 to x_1, x_2, x_3, and multiply the three last constraints by -1. The resulting problem is shown on the left. Then, on the right, we show its dual:

minimize　　$-5x_1 - 6x_2 - 4x_3$
subject to　x_1 free
　　　　　　$x_2 \geq 0$
　　　　　　$x_3 \leq 0$
　　　　　　$x_1 - 2x_2 \geq -1$
　　　　　　$-3x_1 + x_2 \leq -2$
　　　　　　$-3x_2 - x_3 = -3,$

maximize　　$-p_1 - 2p_2 - 3p_3$
subject to　$p_1 - 3p_2 = -5$
　　　　　　$-2p_1 + p_2 - 3p_3 \leq -6$
　　　　　　$- p_3 \geq -4$
　　　　　　$p_1 \geq 0$
　　　　　　$p_2 \leq 0$
　　　　　　p_3 free.

We observe that the latter problem is equivalent to the primal problem we started with. (The first three constraints in the latter problem are the same as the first three constraints in the original problem, multiplied by −1. Also, if the maximization in the latter problem is changed to a minimization, by multiplying the objective function by −1, we obtain the cost function in the original problem.)

The first primal problem considered in Example 4.1 had all of the ingredients of a general linear programming problem. This suggests that the conclusion reached at the end of the example should hold in general. Indeed, we have the following result. Its proof needs nothing more than the steps followed in Example 4.1, with abstract symbols replacing specific numbers, and will therefore be omitted.

Theorem 4.1 *If we transform the dual into an equivalent minimization problem and then form its dual, we obtain a problem equivalent to the original problem.*

A compact statement that is often used to describe Theorem 4.1 is that "the dual of the dual is the prima."

Any linear programming problem can be manipulated into one of several equivalent forms, for example, by introducing slack variables or by using the difference of two nonnegative variables to replace a single free variable. Each equivalent form leads to a somewhat different form for the dual problem. Nevertheless, the examples that follow indicate that the duals of equivalent problems are equivalent.

Example 4.2 Consider the primal problem shown on the left and its dual shown on the right:

minimize　　$c'x$
subject to　$Ax \geq b$
　　　　　　x free,

maximize　　$p'b$
subject to　$p \geq 0$
　　　　　　$p'A = c'.$

We transform the primal problem by introducing surplus variables and then obtain its dual:

minimize　　$c'x + 0's$
subject to　$Ax - s = b$
　　　　　　x free
　　　　　　$s \geq 0,$

maximize　　$p'b$
subject to　p free
　　　　　　$p'A = c'$
　　　　　　$-p \leq 0.$

Alternatively, if we take the original primal problem and replace x by sign-constrained variables, we obtain the following pair of problems:

minimize　　$c'x^+ - c'x^-$
subject to　$Ax^+ - Ax^- \geq b$
　　　　　　$x^+ \geq 0$
　　　　　　$x^- \geq 0,$

maximize　　$p'b$
subject to　$p \geq 0$
　　　　　　$p'A \leq c'$
　　　　　　$-p'A \leq -c'.$

Note that we have three equivalent forms of the primal. We observe that the constraint $p \geq 0$ is equivalent to the constraint $-p \leq 0$. Furthermore, the constraint $p'A = c'$ is equivalent to the two constraints $p'A \leq c'$ and $-p'A \leq -c'$. Thus, the duals of the three variants of the primal problem are also equivalent.

The next example is in the same spirit and examines the effect of removing redundant equality constraints in a standard form problem.

Example 4.3 Consider a standard form problem, assumed feasible, and its dual:

minimize　　$c'x$
subject to　$Ax = b$
　　　　　　$x \geq 0,$

maximize　　$p'b$
subject to　$p'A \leq c'.$

Let $a_1', \ldots, a_m'$ be the rows of A and suppose that $a_m = \sum_{i=1}^{m-1} \gamma_i a_i$, for some scalars $\gamma_1, \ldots, \gamma_{m-1}$. In particular, the last equality constraint is redundant and can be eliminated. By considering an arbitrary feasible solution x, we obtain

$$b_m = a_m'x = \sum_{i=1}^{m-1} \gamma_i a_i' x = \sum_{i=1}^{m-1} \gamma_i b_i. \qquad (4.1)$$

Note that the dual constraints are of the form $\sum_{i=1}^{m} p_i a_i' \leq c'$ and can be rewritten as

$$\sum_{i=1}^{m-1} (p_i + \gamma_i p_m) a_i' \leq c'.$$

Furthermore, using Eq. (4.1), the dual cost $\sum_{i=1}^{m} p_i b_i$ is equal to

$$\sum_{i=1}^{m-1} (p_i + \gamma_i p_m) b_i.$$

If we now let $q_i = p_i + \gamma_i p_m$, we see that the dual problem is equivalent to

$$\text{maximize} \quad \sum_{i=1}^{m-1} q_i b_i$$

$$\text{subject to} \quad \sum_{i=1}^{m-1} q_i a_i' \leq c'.$$

We observe that this is the exact same dual that we would have obtained if we had eliminated the last (and redundant) constraint in the primal problem, before forming the dual.

The conclusions of the preceding two examples are summarized and generalized by the following result.

Theorem 4.2 *Suppose that we have transformed a linear programming problem Π_1 to another linear programming problem Π_2, by a sequence of transformations of the following types:*

(a) *Replace a free variable with the difference of two nonnegative variables.*

(b) *Replace an inequality constraint by an equality constraint involving a nonnegative slack variable.*

(c) *If some row of the matrix A in a feasible standard form problem is a linear combination of the other rows, eliminate the corresponding equality constraint.*

Then, the duals of Π_1 and Π_2 are equivalent, i.e., they are either both infeasible, or they have the same optimal cost.

The proof of Theorem 4.2 involves a combination of the various steps in Examples 4.2 and 4.3, and is left to the reader.

4.3   The duality theorem

We saw in Section 4.1 that for problems in standard form, the cost $g(\mathbf{p})$ of any dual solution provides a lower bound for the optimal cost. We now show that this property is true in general.

Theorem 4.3 (Weak duality) *If $\mathbf{x}$ is a feasible solution to the primal problem and $\mathbf{p}$ is a feasible solution to the dual problem, then*

$$\mathbf{p}'\mathbf{b} \leq \mathbf{c}'\mathbf{x}.$$

Proof. For any vectors $\mathbf{x}$ and $\mathbf{p}$, we define

$$u_i = p_i(\mathbf{a}_i'\mathbf{x} - b_i),$$
$$v_j = (c_j - \mathbf{p}'\mathbf{A}_j)x_j.$$

Suppose that $\mathbf{x}$ and $\mathbf{p}$ are primal and dual feasible, respectively. The definition of the dual problem requires the sign of p_i to be the same as the sign of $\mathbf{a}_i'\mathbf{x} - b_i$, and the sign of $c_j - \mathbf{p}'\mathbf{A}_j$ to be the same as the sign of x_j. Thus, primal and dual feasibility imply that

$$u_i \geq 0, \qquad \forall\, i,$$

and

$$v_j \geq 0, \qquad \forall\, j.$$

Notice that

$$\sum_i u_i = \mathbf{p}'\mathbf{A}\mathbf{x} - \mathbf{p}'\mathbf{b},$$

and

$$\sum_j v_j = \mathbf{c}'\mathbf{x} - \mathbf{p}'\mathbf{A}\mathbf{x}.$$

We add these two equalities and use the nonnegativity of u_i, v_j, to obtain

$$0 \leq \sum_i u_i + \sum_j v_j = \mathbf{c}'\mathbf{x} - \mathbf{p}'\mathbf{b}.$$

□

The weak duality theorem is not a deep result, yet it does provide some useful information about the relation between the primal and the dual. We have, for example, the following corollary.

Corollary 4.1

(a) *If the optimal cost in the primal is $-\infty$, then the dual problem must be infeasible.*

(b) *If the optimal cost in the dual is $+\infty$, then the primal problem must be infeasible.*

Proof. Suppose that the optimal cost in the primal problem is $-\infty$ and that the dual problem has a feasible solution $\mathbf{p}$. By weak duality, $\mathbf{p}$ satisfies $\mathbf{p}'\mathbf{b} \leq \mathbf{c}'\mathbf{x}$ for every primal feasible $\mathbf{x}$. Taking the minimum over all primal feasible $\mathbf{x}$, we conclude that $\mathbf{p}'\mathbf{b} \leq -\infty$. This is impossible and shows that the dual cannot have a feasible solution, thus establishing part (a). Part (b) follows by a symmetrical argument.

□

Another important corollary of the weak duality theorem is the following.

Corollary 4.2 *Let* **x** *and* **p** *be feasible solutions to the primal and the dual, respectively, and suppose that* $p'b = c'x$. *Then,* **x** *and* **p** *are optimal solutions to the primal and the dual, respectively.*

Proof. Let **x** and **p** be as in the statement of the corollary. For every primal feasible solution **y**, the weak duality theorem yields $c'x = p'b \le c'y$, which proves that **x** is optimal. The proof of optimality of **p** is similar. ☐

The next theorem is the central result on linear programming duality.

Theorem 4.4 (Strong duality) *If a linear programming problem has an optimal solution, so does its dual, and the respective optimal costs are equal.*

Proof. Consider the standard form problem

$$\text{minimize} \quad c'x$$
$$\text{subject to} \quad Ax = b$$
$$x \ge 0.$$

Let us assume temporarily that the rows of **A** are linearly independent and that there exists an optimal solution. Let us apply the simplex method to this problem. As long as cycling is avoided, e.g., by using the lexicographic pivoting rule, the simplex method terminates with an optimal solution **x** and an optimal basis **B**. Let $x_B = B^{-1}b$ be the corresponding vector of basic variables. When the simplex method terminates, the reduced costs must be nonnegative and we obtain

$$c' - c'_B B^{-1}A \ge 0',$$

where c'_B is the vector with the costs of the basic variables. Let us define a vector **p** by letting $p' = c'_B B^{-1}$. We then have $p'A \le c'$, which shows that **p** is a feasible solution to the dual problem

$$\text{maximize} \quad p'b$$
$$\text{subject to} \quad p'A \le c'.$$

In addition,

$$p'b = c'_B B^{-1}b = c'_B x_B = c'x.$$

It follows that **p** is an optimal solution to the dual (cf. Corollary 4.2), and the optimal dual cost is equal to the optimal primal cost.

If we are dealing with a general linear programming problem Π_1 that has an optimal solution, we first transform it into an equivalent standard

form problem Π_2, with the same optimal cost, and in which the rows of the matrix **A** are linearly independent. Let D_1 and D_2 be the duals of Π_1 and Π_2, respectively. By Theorem 4.2, the dual problems D_1 and D_2 have the same optimal cost. We have already proved that Π_2 and D_2 have the same optimal cost. It follows that Π_1 and D_1 have the same optimal cost (see Figure 4.1). ☐

Figure 4.1

equivalent

$$\Pi_1 \longrightarrow \Pi_2$$
duality for standard form problems

$$D_1 \longleftarrow D_2$$
duals of equivalent problems are equivalent

L_2
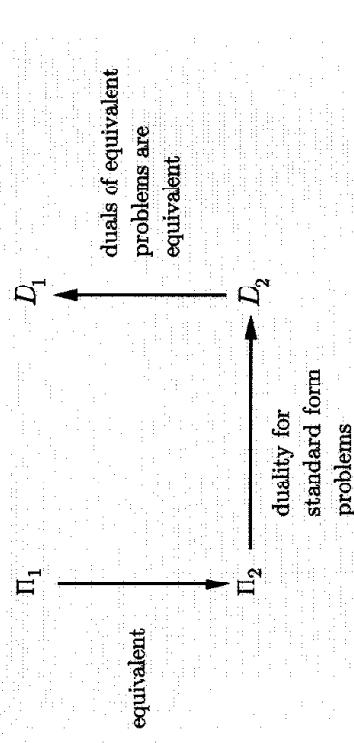
Figure 4.1: Proof of the duality theorem for general linear programming problems.

The preceding proof shows that an optimal solution to the dual problem is obtained as a byproduct of the simplex method as applied to a primal problem in standard form. It is based on the fact that the simplex method is guaranteed to terminate and this, in turn, depends on the existence of pivoting rules that prevent cycling. There is an alternative derivation of the duality theorem, which provides a geometric, algorithm-independent view of the subject, and which is developed in Section 4.7. At this point, we provide an illustration that conveys most of the content of the geometric proof.

Example 4.4 Consider a solid ball constrained to lie in a polyhedron defined by inequality constraints of the form $a_i'x \ge b_i$. If left under the influence of gravity, this ball reaches equilibrium at the lowest corner x^* of the polyhedron; see Figure 4.2. This corner is an optimal solution to the problem

$$\text{minimize} \quad c'x$$
$$\text{subject to} \quad a_i'x \ge b_i, \quad \forall\, i,$$

where **c** is a vertical vector pointing upwards. At equilibrium, gravity is counterbalanced by the forces exerted on the ball by the "walls" of the polyhedron. The latter forces are normal to the walls, that is, they are aligned with the vectors a_i. We conclude that $c = \sum_i p_i a_i$, for some nonnegative coefficients p_i; in particular,

the vector p is a feasible solution to the dual problem

$$\text{maximize} \quad p'b$$
$$\text{subject to} \quad p'A = c'$$
$$p \geq 0.$$

Given that forces can only be exerted by the walls that touch the ball, we must have $p_i = 0$, whenever $a_i'x^* > b_i$. Consequently, $p_i(b_i - a_i'x^*) = 0$ for all i. We therefore have $p'b = \sum_i p_i b_i = \sum_i p_i a_i'x^* = c'x^*$. It follows (Corollary 4.2) that p is an optimal solution to the dual, and the optimal dual cost is equal to the optimal primal cost.
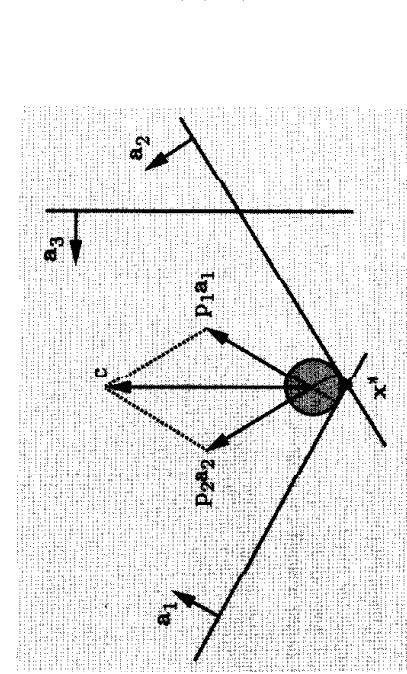

Figure 4.2: A mechanical analogy of the duality theorem.

Recall that in a linear programming problem, exactly one of the following three possibilities will occur:

(a) There is an optimal solution.

(b) The problem is "unbounded"; that is, the optimal cost is $-\infty$ (for minimization problems), or $+\infty$ (for maximization problems).

(c) The problem is infeasible.

This leads to nine possible combinations for the primal and the dual, which are shown in Table 4.2. By the strong duality theorem, if one problem has an optimal solution, so does the other. Furthermore, as discussed earlier, the weak duality theorem implies that if one problem is unbounded, the other must be infeasible. This allows us to mark some of the entries in Table 4.2 as "impossible."

	Finite optimum	Unbounded	Infeasible
Finite optimum	Possible	Impossible	Impossible
Unbounded	Impossible	Impossible	Possible
Infeasible	Impossible	Possible	Possible

Table 4.2: The different possibilities for the primal and the dual.

The case where both problems are infeasible can indeed occur, as shown by the following example.

Example 4.5 Consider the infeasible primal

$$\text{minimize} \quad x_1 + 2x_2$$
$$\text{subject to} \quad x_1 + x_2 = 1$$
$$2x_1 + 2x_2 = 3.$$

Its dual is

$$\text{maximize} \quad p_1 + 3p_2$$
$$\text{subject to} \quad p_1 + 2p_2 = 1$$
$$p_1 + 2p_2 = 2,$$

which is also infeasible.

There is another interesting relation between the primal and the dual which is known as Clark's theorem (Clark, 1961). It asserts that unless both problems are infeasible, at least one of them must have an unbounded feasible set (Exercise 4.21).

Complementary slackness

An important relation between primal and dual optimal solutions is provided by the *complementary slackness* conditions, which we present next.

> **Theorem 4.5 (Complementary slackness)** *Let* **x** *and* **p** *be feasible solutions to the primal and the dual problem, respectively. The vectors* **x** *and* **p** *are optimal solutions for the two respective problems if and only if:*
>
> $$p_i(a_i'x - b_i) = 0, \qquad \forall \, i,$$
> $$(c_j - p'A_j)x_j = 0, \qquad \forall \, j.$$

Proof. In the proof of Theorem 4.3, we defined $u_i = p_i(a_i'x - b_i)$ and $v_j = (c_j - p'A_j)x_j$, and noted that for x primal feasible and p dual feasible,

we have $u_i \geq 0$ and $v_j \geq 0$ for all i and j. In addition, we showed that

$$c'x - p'b = \sum_i u_i + \sum_j v_j.$$

By the strong duality theorem, if x and p are optimal, then $c'x = p'b$ which implies that $u_i = v_j = 0$ for all i, j. Conversely, if $u_i = v_j = 0$ for all i, j, then $c'x = p'b$, and Corollary 4.2 implies that x and p are optimal.

The first complementary slackness condition is automatically satisfied by every feasible solution to a problem in standard form. If the primal problem is not in standard form and has a constraint like $a_i'x \geq b_i$, the corresponding complementary slackness condition asserts that the dual variable p_i is zero unless the constraint is active. An intuitive explanation is that a constraint which is not active at an optimal solution can be removed from the problem without affecting the optimal cost, and there is no point in associating a nonzero price with such a constraint. Note also the analogy with Example 4.4, where "forces" were only exerted by the active constraints.

If the primal problem is in standard form and a nondegenerate optimal basic feasible solution is known, the complementary slackness conditions determine a unique solution to the dual problem. We illustrate this fact in the next example.

Example 4.6 Consider a problem in standard form and its dual:

minimize $13x_1 + 10x_2 + 6x_3$ maximize $8p_1 + 3p_2$
subject to $5x_1 + x_2 + 3x_3 = 8$ subject to $5p_1 + 3p_2 \leq 13$
$3x_1 + x_2 = 3$ $p_1 + p_2 \leq 10$
$x_1, x_2, x_3 \geq 0$, $3p_1 \leq 6$.

As will be verified shortly, the vector $x^* = (1, 0, 1)$ is a nondegenerate optimal solution to the primal problem. Assuming this to be the case, we use the complementary slackness conditions to construct the optimal solution to the dual. The condition $p_i(a_i'x^* - b_i) = 0$ is automatically satisfied for each i, since the primal is in standard form. The condition $(c_j - p'A_j)x_j^* = 0$ is clearly satisfied for $j = 2$, because $x_2^* = 0$. However, since $x_1^* > 0$ and $x_3^* > 0$, we obtain

$$5p_1 + 3p_2 = 13,$$

and

$$3p_1 = 6,$$

which we can solve to obtain $p_1 = 2$ and $p_2 = 1$. Note that this is a dual feasible solution whose cost is equal to 19, which is the same as the cost of x^*. This verifies that x^* is indeed an optimal solution as claimed earlier.

We now generalize the above example. Suppose that x_j is a basic variable in a nondegenerate optimal basic feasible solution to a primal

problem in standard form. Then, the complementary slackness condition $(c_j - p'A_j)x_j = 0$ yields $p'A_j = c_j$ for every such j. Since the basic columns A_j are linearly independent, we obtain a system of equations for p which has a unique solution, namely, $p' = c_B'B^{-1}$. A similar conclusion can also be drawn for problems not in standard form (Exercise 4.12). On the other hand, if we are given a degenerate optimal basic feasible solution to the primal, complementary slackness may be of very little help in determining an optimal solution to the dual problem (Exercise 4.17).

We finally mention that if the primal constraints are of the form $Ax \geq b$, $x \geq 0$, and the primal problem has an optimal solution, then there exist optimal solutions to the primal and the dual which satisfy *strict complementary slackness*; that is, a variable in one problem is nonzero if and only if the corresponding constraint in the other problem is active (Exercise 4.20). This result has some interesting applications in discrete optimization, but these lie outside the scope of this book.

A geometric view

We now develop a geometric view that allows us to visualize pairs of primal and dual vectors without having to draw the dual feasible set.
We consider the primal problem

minimize $c'x$
subject to $a_i'x \geq b_i$, $i = 1, \ldots, m$,

where the dimension of x is equal to n. We assume that the vectors a_i span $\Re^n$. The corresponding dual problem is

maximize $p'b$

subject to $\displaystyle\sum_{i=1}^m p_i a_i = c$

$p \geq 0$.

Let I be a subset of $\{1, \ldots, m\}$ of cardinality n, such that the vectors $a_i, i \in I$, are linearly independent. The system $a_i'x = b_i$, $i \in I$, has a unique solution, denoted by x^I, which is a basic solution to the primal problem (cf. Definition 2.9 in Section 2.2). We assume that x^I is nondegenerate, that is, $a_i'x \neq b_i$ for $i \notin I$.

Let $p \in \Re^m$ be a dual vector (not necessarily dual feasible), and let us consider what is required for x^I and p to be optimal solutions to the primal and the dual problem, respectively. We need:

(a) $a_i'x^I \geq b_i$, for all i,   (primal feasibility),

(b) $p_i = 0$, for all $i \notin I$,   (complementary slackness),

(c) $\sum_{i=1}^m p_i a_i = c$,   (dual feasibility)

(d) $p \geq 0$,   (dual feasibility)
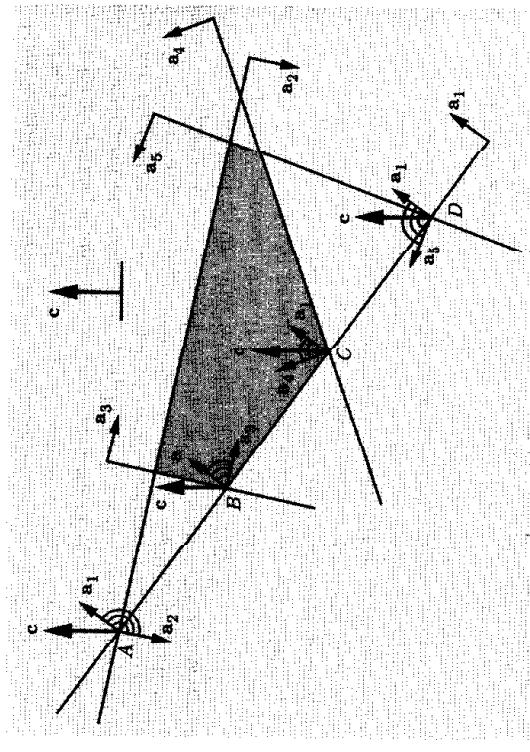
Figure 4.3: Consider a primal problem with two variables and five inequality constraints ($n = 2$, $m = 5$), and suppose that no two of the vectors $\mathbf{a}_i$ are collinear. Every two-element subset I of $\{1, 2, 3, 4, 5\}$ determines basic solutions $\mathbf{x}^I$ and $\mathbf{p}^I$ of the primal and the dual, respectively.

If $I = \{1, 2\}$, $\mathbf{x}^I$ is primal infeasible (point A) and $\mathbf{p}^I$ is dual infeasible, because $\mathbf{c}$ cannot be expressed as a nonnegative linear combination of the vectors $\mathbf{a}_1$ and $\mathbf{a}_2$.

If $I = \{1, 3\}$, $\mathbf{x}^I$ is primal feasible (point B) and $\mathbf{p}^I$ is dual infeasible.

If $I = \{1, 4\}$, $\mathbf{x}^I$ is primal feasible (point C) and $\mathbf{p}^I$ is dual feasible, because $\mathbf{c}$ can be expressed as a nonnegative linear combination of the vectors $\mathbf{a}_1$ and $\mathbf{a}_4$. In particular, $\mathbf{x}^I$ and $\mathbf{p}^I$ are optimal.

If $I = \{1, 5\}$, $\mathbf{x}^I$ is primal infeasible (point D) and $\mathbf{p}^I$ is dual feasible.

Given the complementary slackness condition (b), condition (c) becomes

$$\sum_{i \in I} p_i \mathbf{a}_i = \mathbf{c}.$$

Since the vectors $\mathbf{a}_i$, $i \in I$, are linearly independent, the latter equation has a unique solution that we denote by $\mathbf{p}^I$. In fact, it is readily seen that the vectors $\mathbf{a}_i$, $i \in I$, form a basis for the dual problem (which is in standard form) and $\mathbf{p}^I$ is the associated basic solution. For the vector $\mathbf{p}^I$ to be dual feasible, we also need it to be nonnegative. We conclude that once the complementary slackness condition (b) is enforced, feasibility of
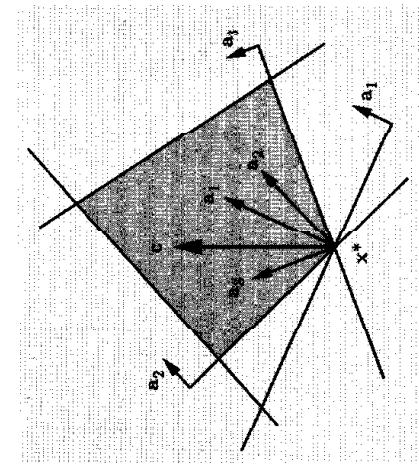
Figure 4.4: The vector $\mathbf{x}^*$ is a degenerate basic feasible solution of the primal. If we choose $I = \{1, 2\}$, the corresponding dual basic solution $\mathbf{p}^I$ is infeasible, because $\mathbf{c}$ is not a nonnegative linear combination of $\mathbf{a}_1$, $\mathbf{a}_2$. On the other hand, if we choose $I = \{1, 3\}$ or $I = \{2, 3\}$, the resulting dual basic solution $\mathbf{p}^I$ is feasible and, therefore, optimal.

the resulting dual vector $\mathbf{p}^I$ is equivalent to $\mathbf{c}$ being a nonnegative linear combination of the vectors $\mathbf{a}_i$, $i \in I$, associated with the active primal constraints. This allows us to visualize dual feasibility without having to draw the dual feasible set; see Figure 4.3.

If $\mathbf{x}^*$ is a degenerate basic solution to the primal, there can be several subsets I such that $\mathbf{x}^I = \mathbf{x}^*$. Using different choices for I, and by solving the system $\sum_{i \in I} p_i \mathbf{a}_i = \mathbf{c}$, we may obtain several dual basic solutions $\mathbf{p}^I$. It may then well be the case that some of them are dual feasible and some are not; see Figure 4.4. Still, if $\mathbf{p}^I$ is dual feasible (i.e., all p_i are nonnegative) and if $\mathbf{x}^*$ is primal feasible, then they are both optimal, because we have been enforcing complementary slackness and Theorem 4.5 applies.

4.4 Optimal dual variables as marginal costs

In this section, we elaborate on the interpretation of the dual variables as prices. This theme will be revisited, in more depth, in Chapter 5.

Consider the standard form problem

$$\text{minimize} \quad \mathbf{c}'\mathbf{x}$$
$$\text{subject to} \quad \mathbf{A}\mathbf{x} = \mathbf{b}$$
$$\mathbf{x} \geq \mathbf{0}.$$

We assume that the rows of $\mathbf{A}$ are linearly independent and that there

is a nondegenerate basic feasible solution x^* which is optimal. Let B be the corresponding basis matrix and let $x_B = B^{-1}b$ be the vector of basic variables, which is positive, by nondegeneracy. Let us now replace b by $b + d$, where d is a small perturbation vector. Since $B^{-1}b > 0$, we also have $B^{-1}(b + d) > 0$, as long as d is small. This implies that the same basis leads to a basic feasible solution of the perturbed problem as well. Perturbing the right-hand side vector b has no effect on the reduced costs associated with this basis. By the optimality of x^* in the original problem, the vector of reduced costs $c' - c'_B B^{-1}A$ is nonnegative and this establishes that the same basis is optimal for the perturbed problem as well. Thus, the optimal cost in the perturbed problem is

$$c'_B B^{-1}(b + d) = p'(b + d),$$

where $p' = c'_B B^{-1}$ is an optimal solution to the dual problem. Therefore, a small change of d in the right-hand side vector b results in a change of $p'd$ in the optimal cost. We conclude that each component p_i of the optimal dual vector can be interpreted as the *marginal cost* (or *shadow price*) per unit increase of the ith requirement b_i.

We conclude with yet another interpretation of duality, for standard form problems. In order to develop some concrete intuition, we phrase our discussion in terms of the diet problem (Example 1.3 in Section 1.1). We interpret each vector A_j as the nutritional content of the jth available food, and view b as the nutritional content of an ideal food that we wish to synthesize. Let us interpret p_i as the "fair" price per unit of the ith nutrient. A unit of the jth food has a value of c_j at the food market, but it also has a value of $p'A_j$ if priced at the nutrient market. Complementary slackness asserts that every food which is used (at a nonzero level) to synthesize the ideal food, should be consistently priced at the two markets. Thus, duality is concerned with two alternative ways of cost accounting. The value of the ideal food, as computed in the food market, is $c'x^*$, where x^* is an optimal solution to the primal problem; the value of the ideal food, as computed in the nutrient market, is $p'b$. The duality relation $c'x^* = p'b$ states that when prices are chosen appropriately, the two accounting methods should give the same results.

4.5 Standard form problems and the dual simplex method

In this section, we concentrate on the case where the primal problem is in standard form. We develop the *dual simplex method*, which is an alternative to the simplex method of Chapter 3. We also comment on the relation between the basic feasible solutions to the primal and the dual, including a discussion of dual degeneracy.

In the proof of the strong duality theorem, we considered the simplex method applied to a primal problem in standard form and defined a dual vector p by letting $p' = c'_B B^{-1}$. We then noted that the primal optimality condition $c' - c'_B B^{-1}A \geq 0'$ is the same as the dual feasibility condition $p'A \leq c'$. We can thus think of the simplex method as an algorithm that maintains primal feasibility and works towards dual feasibility. A method with this property is generally called a *primal* algorithm. An alternative is to start with a dual feasible solution and work towards primal feasibility. A method of this type is called a *dual* algorithm. In this section, we present a dual simplex method, implemented in terms of the full tableau. We argue that it does indeed solve the dual problem, and we show that it moves from one basic feasible solution of the dual problem to another. An alternative implementation that only keeps track of the matrix B^{-1}, instead of the entire tableau, is called a *revised dual simplex* method (Exercise 4.23).

The dual simplex method

Let us consider a problem in standard form, under the usual assumption that the rows of the matrix A are linearly independent. Let B be a basis matrix, consisting of m linearly independent columns of A, and consider the corresponding tableau

$-c'_B B^{-1}b$	$\bar{c}'$
$B^{-1}b$	$B^{-1}A$

or, in more detail,

$-c'_B x_B$	$\bar{c}_1$	...	$\bar{c}_n$
$x_{B(1)}$			
⋮		$B^{-1}A_1 \ \cdots \ B^{-1}A_n$	
$x_{B(m)}$			

We do not require $B^{-1}b$ to be nonnegative, which means that we have a basic, but not necessarily feasible solution to the primal problem. However, we assume that $\bar{c} \geq 0$; equivalently, the vector $p' = c'_B B^{-1}$ satisfies $p'A \leq c'$, and we have a feasible solution to the dual problem. The cost of this dual feasible solution is $p'b = c'_B B^{-1}b = c'_B x_B$, which is the negative of the entry at the upper left corner of the tableau. If the inequality $B^{-1}b \geq 0$ happens to hold, we also have a primal feasible solution with the same cost, and optimal solutions to both problems have been found. If the inequality $B^{-1}b \geq 0$ fails to hold, we perform a change of basis in a manner we describe next.

We find some ℓ such that $x_{B(\ell)} < 0$ and consider the ℓth row of the tableau, called the *pivot row*; this row is of the form $(x_{B(\ell)}, v_1, \ldots, v_n)$, where v_i is the ℓth component of $B^{-1}A_i$. For each i with $v_i < 0$ (if such i exist), we form the ratio $\bar{c}_i/|v_i|$ and let j be an index for which this ratio is smallest; that is, $v_j < 0$ and

$$\frac{\bar{c}_j}{|v_j|} = \min_{\{i|v_i<0\}} \frac{\bar{c}_i}{|v_i|}. \tag{4.2}$$

(We call the corresponding entry v_j the *pivot element*. Note that x_j must be a nonbasic variable, since the jth column in the tableau contains the negative element v_i.) We then perform a change of basis: column A_j enters the basis and column $A_{B(\ell)}$ exits. This change of basis (or *pivot*) is effected exactly as in the primal simplex method: we add to each row of the tableau a multiple of the pivot row so that all entries in the pivot column are set to zero, with the exception of the pivot element which is set to 1. In particular, in order to set the reduced cost in the pivot column to zero, we multiply the pivot row by $\bar{c}_j/|v_j|$ and add it to the zeroth row. For every i, the new value of $\bar{c}_i$ is equal to

$$\bar{c}_i + v_i \frac{\bar{c}_j}{|v_j|},$$

which is nonnegative because of the way that j was selected [cf. Eq. (4.2)]. We conclude that the reduced costs in the new tableau will also be nonnegative and dual feasibility has been maintained.

Example 4.7 Consider the tableau

	x_1	x_2	x_3	x_4	x_5
0	2	6	10	0	0
$x_4 =$ 2	-2	4	1	1	0
$x_5 =$ -1	4	-2*	-3	0	1

Since $x_{B(2)} < 0$, we choose the second row to be the pivot row. Negative entries of the pivot row are found in the second and third column. We compare the corresponding ratios $6/|-2|$ and $10/|-3|$. The smallest ratio is $6/|-2|$ and, therefore, the second column enters the basis. (The pivot element is indicated by an asterisk.) We multiply the pivot row by 3 and add it to the zeroth row. We multiply the pivot row by 2 and add it to the first row. We then divide the pivot row by -2. The new tableau is

	x_1	x_2	x_3	x_4	x_5
-3	14	0	1	0	3
$x_4 =$ 0	6	0	-5	1	2
$x_2 =$ 1/2	-2	1	3/2	0	-1/2

The cost has increased to 3. Furthermore, we now have $B^{-1}b \geq 0$, and an optimal solution has been found.

Note that the pivot element v_j is always chosen to be negative, whereas the corresponding reduced cost $\bar{c}_j$ is nonnegative. Let us temporarily assume that $\bar{c}_j$ is in fact positive. Then, in order to replace $\bar{c}_j$ by zero, we need to add a positive multiple of the pivot row to the zeroth row. Since $x_{B(\ell)}$ is negative, this has the effect of adding a negative quantity to the upper left corner. Equivalently, the dual cost increases. Thus, as long as the reduced cost of every nonbasic variable is nonzero, the dual cost increases with each basis change, and no basis will ever be repeated in the course of the algorithm. It follows that the algorithm must eventually terminate and this can happen in one of two ways:

(a) We have $B^{-1}b \geq 0$ and an optimal solution.

(b) All of the entries $v_1, \ldots, v_n$ in the pivot row are nonnegative and we are therefore unable to locate a pivot element. In full analogy with the primal simplex method, this implies that the optimal dual cost is equal to $+\infty$ and the primal problem is infeasible; the proof is left as an exercise (Exercise 4.22).

We now provide a summary of the algorithm.

An iteration of the dual simplex method

1. A typical iteration starts with the tableau associated with a basis matrix B and with all reduced costs nonnegative.

2. Examine the components of the vector $B^{-1}b$ in the zeroth column. If they are all nonnegative, we have an optimal basic feasible solution and the algorithm terminates; else, choose some ℓ such that $x_{B(\ell)} < 0$.

3. Consider the ℓth row of the tableau, with elements $x_{B(\ell)}, v_1, \ldots, v_n$ (the pivot row). If $v_i \geq 0$ for all i, then the optimal dual cost is $+\infty$ and the algorithm terminates.

4. For each i such that $v_i < 0$, compute the ratio $\bar{c}_i/|v_i|$ and let j be the index of a column that corresponds to the smallest ratio. The column $A_{B(\ell)}$ exits the basis and the column A_j takes its place.

5. Add to each row of the tableau a multiple of the ℓth row (the pivot row) so that v_j (the pivot element) becomes 1 and all other entries of the pivot column become 0.

Let us now consider the possibility that the reduced cost $\bar{c}_j$ in the pivot column is zero. In this case, the zeroth row of the tableau does not change and the dual cost $c'_B B^{-1}b$ remains the same. The proof of termina-

tion given earlier does not apply and the algorithm can cycle. This can be avoided by employing a suitable anticycling rule, such as the following.

Lexicographic pivoting rule for the dual simplex method

1. Choose any row ℓ such that $x_{B(\ell)} < 0$, to be the pivot row.
2. Determine the index j of the entering column as follows. For each column with $v_i < 0$, divide all entries by $|v_i|$, and then choose the lexicographically smallest column. If there is a tie between several lexicographically smallest columns, choose the one with the smallest index.

If the dual simplex method is initialized so that every column of the tableau [that is, each vector $(\bar{c}_j, \mathbf{B}^{-1}\mathbf{A}_j)$] is lexicographically positive, and if the above lexicographic pivoting rule is used, the method terminates in a finite number of steps. The proof is similar to the proof of the corresponding result for the primal simplex method (Theorem 3.4) and is left as an exercise (Exercise 4.24).

When should we use the dual simplex method

At this point, it is natural to ask when the dual simplex method should be used. One such case arises when a basic feasible solution of the dual problem is readily available. Suppose, for example, that we already have an optimal basis for some linear programming problem, and that we wish to solve the same problem for a different choice of the right-hand side vector $\mathbf{b}$. The optimal basis for the original problem may be primal infeasible under the new value of $\mathbf{b}$. On the other hand, a change in $\mathbf{b}$ does not affect the reduced costs and we still have a dual feasible solution. Thus, instead of solving the new problem from scratch, it may be preferable to apply the dual simplex algorithm starting from the optimal basis for the original problem. This idea will be considered in more detail in Chapter 5.

The geometry of the dual simplex method

Our development of the dual simplex method was based entirely on tableau manipulations and algebraic arguments. We now present an alternative viewpoint based on geometric considerations.

We continue assuming that we are dealing with a problem in standard form and that the matrix $\mathbf{A}$ has linearly independent rows. Let $\mathbf{B}$ be a basis matrix with columns $\mathbf{A}_{B(1)}, \ldots, \mathbf{A}_{B(m)}$. This basis matrix determines a basic solution to the primal problem with $\mathbf{x}_B = \mathbf{B}^{-1}\mathbf{b}$. The same basis can also be used to determine a dual vector $\mathbf{p}$ by means of the equations

$$\mathbf{p}'\mathbf{A}_{B(i)} = c_{B(i)}, \qquad i = 1, \ldots, m.$$

These are m equations in m unknowns; since the columns $\mathbf{A}_{B(1)}, \ldots, \mathbf{A}_{B(m)}$ are linearly independent, there is a unique solution $\mathbf{p}$. For such a vector $\mathbf{p}$, the number of linearly independent active dual constraints is equal to the dimension of the dual vector, and it follows that we have a basic solution to the dual problem. In matrix notation, the dual basic solution $\mathbf{p}$ satisfies $\mathbf{p}'\mathbf{B} = \mathbf{c}'_B$, or $\mathbf{p}' = \mathbf{c}'_B\mathbf{B}^{-1}$, which was referred to as the vector of simplex multipliers in Chapter 3. If $\mathbf{p}$ is also dual feasible, that is, if $\mathbf{p}'\mathbf{A} \le \mathbf{c}'$, then $\mathbf{p}$ is a basic feasible solution of the dual problem.

To summarize, a basis matrix $\mathbf{B}$ is associated with a basic solution to the primal problem and also with a basic solution to the dual. A basic solution to the primal (respectively, dual) which is primal (respectively, dual) feasible, is a basic feasible solution to the primal (respectively, dual).

We now have a geometric interpretation of the dual simplex method: at every iteration, we have a basic feasible solution to the dual problem. The basic feasible solutions obtained at any two consecutive iterations have $m-1$ linearly independent active constraints in common (the reduced costs of the $m-1$ variables that are common to both bases are zero); thus, consecutive basic feasible solutions are either adjacent or they coincide.

Example 4.8 Consider the following standard form problem and its dual:

$$
\begin{array}{ll}
\text{minimize} & x_1 + x_2 \\
\text{subject to} & x_1 + 2x_2 - x_3 = 2 \\
& x_1 - x_4 = 1 \\
& x_1, x_2, x_3, x_4 \ge 0,
\end{array}
\qquad
\begin{array}{ll}
\text{maximize} & 2p_1 + p_2 \\
\text{subject to} & p_1 + p_2 \le 1 \\
& 2p_1 \le 1 \\
& p_1, p_2 \ge 0.
\end{array}
$$

The feasible set of the primal problem is 4-dimensional. If we eliminate the variables x_3 and x_4, we obtain the equivalent problem

$$
\begin{array}{ll}
\text{minimize} & x_1 + x_2 \\
\text{subject to} & x_1 + 2x_2 \ge 2 \\
& x_1 \ge 1 \\
& x_1, x_2 \ge 0.
\end{array}
$$

The feasible sets of the equivalent primal problem and of the dual are shown in Figures 4.5(a) and 4.5(b), respectively.

There is a total of five different bases in the standard form primal problem and five different basic solutions. These correspond to the points A, B, C, D, and E in Figure 4.5(a). The same five bases also lead to five basic solutions to the dual problem, which are points A, B, C, D, and E in Figure 4.5(b).

For example, if we choose the columns $\mathbf{A}_3$ and $\mathbf{A}_4$ to be the basic columns, we have the infeasible primal basic solution $\mathbf{x} = (0, 0, -2, -1)$ (point A). The corresponding dual basic solution is obtained by letting $\mathbf{p}'\mathbf{A}_3 = c_3 = 0$ and $\mathbf{p}'\mathbf{A}_4 = c_4 = 0$, which yields $\mathbf{p} = (0, 0)$. This is a basic feasible solution of the dual problem and can be used to start the dual simplex method. The associated initial tableau is
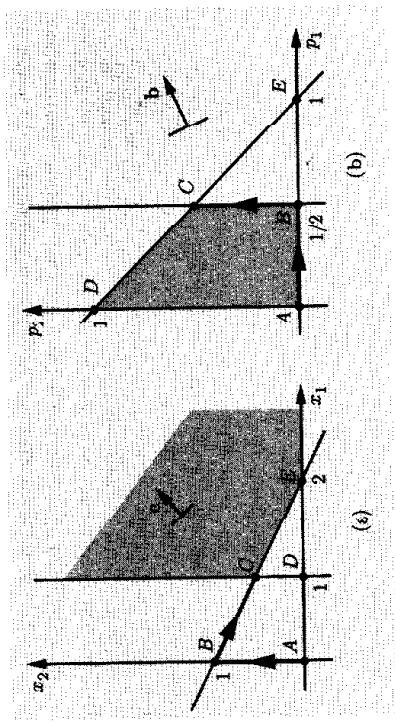
Figure 4.5: The feasible sets in Example 4.8.

We carry out two iterations of the dual simplex method to obtain the following two tableaux:

	x_1	x_2	x_3	x_4
	0	1	0	0
$x_3 = -2$	-1	-2*	1	0
$x_4 = -1$	-1	0	0	1

	x_1	x_2	x_3	x_4
	-1	1/2	0	1/2	0
$x_2 = 1$	1/2	1	-1/2	0
$x_4 = -1$	-1*	0	0	1

	x_1	x_2	x_3	x_4
	-3/2	0	0	1/2	1/2
$x_2 = 1/2$	0	1	-1/2	1/2
$x_1 = 1$	1	0	0	-1

This sequence of tableaux corresponds to the path $A - B - C$ in either figure. In the primal space, the path traces a sequence of infeasible basic solutions until, at

optimality, it becomes feasible. In the dual space, the algorithm behaves exactly like the primal simplex method: it moves through a sequence of (dual) basic feasible solutions, while at each step improving the cost function.

Having observed that the dual simplex method moves from one basic feasible solution of the dual to an adjacent one, it may be tempting to say that the dual simplex method is simply the primal simplex method applied to the dual. This is a somewhat ambiguous statement, however, because the dual problem is not in standard form. If we were to convert it to standard form and then apply the primal simplex method, the resulting method is not necessarily identical to the dual simplex method (Exercise 4.25). A more accurate statement is to simply say that the dual simplex method is a variant of the simplex method tailored to problems defined exclusively in terms of linear inequality constraints.

Duality and degeneracy

Let us keep assuming that we are dealing with a standard form problem in which the rows of the matrix $\mathbf{A}$ are linearly independent. Any basis matrix $\mathbf{B}$ leads to an associated dual basic solution given by $\mathbf{p}' = \mathbf{c}'_B \mathbf{B}^{-1}$. At this basic solution, the dual constraint $\mathbf{p}'\mathbf{A}_j = c_j$ is active if and only if $\mathbf{c}'_B \mathbf{B}^{-1} \mathbf{A}_j = c_j$, that is, if and only if the reduced cost $\bar{c}_j$ is zero. Since $\mathbf{p}$ is m-dimensional, dual degeneracy amounts to having more than m reduced costs that are zero. Given that the reduced costs of the m basic variables must be zero, dual degeneracy is obtained whenever there exists a nonbasic variable whose reduced cost is zero.

The example that follows deals with the relation between basic solutions to the primal and the dual in the face of degeneracy.

Example 4.9 Consider the following standard form problem and its dual:

minimize $3x_1 + x_2$
subject to
$x_1 + x_2 - x_3 \qquad = 2$
$2x_1 - x_2 \qquad - x_4 = 0$
$x_1, x_2, x_3, x_4 \geq 0,$

maximize $2p_1$
subject to
$p_1 + 2p_2 \leq 3$
$p_1 - p_2 \leq 1$
$p_1, p_2 \geq 0.$

We eliminate x_3 and x_4 to obtain the equivalent primal problem

minimize $3x_1 + x_2$
subject to
$x_1 + x_2 \geq 2$
$2x_1 - x_2 \geq 0$
$x_1, x_2 \geq 0$

The feasible set of the equivalent primal and of the dual is shown in Figures 4.6(a) and 4.6(b), respectively.

There is a total of six different bases in the standard form primal problem, but only four different basic solutions [points A, B C, D in Figure 4.6(a)]. In the dual problem, however, the six bases lead to six distinct basic solutions [points A, A', A'', B, C, D in Figure 4.6(b)].

(a)

(b)
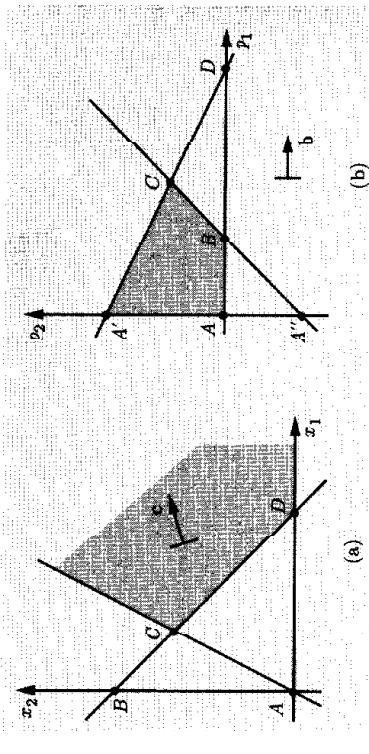
Figure 4.6: The feasible sets in Example 4.9.

For example, if we let columns A_3 and A_4 be basic, the primal basic solution has $x_1 = x_2 = 0$ and the corresponding dual basic solution is $(p_1, p_2) = (0,0)$. Note that this is a basic feasible solution of the dual problem. If we let columns A_1 and A_3 be basic, the primal basic solution has again $x_1 = x_2 = 0$. For the dual problem, however, the equations $p'A_1 = c_1$ and $p'A_3 = c_3$ yield $(p_1, p_2) = (0, 3/2)$, which is a basic feasible solution of the dual, namely, point A' in Figure 4.6(b). Finally, if we let columns A_2 and A_3 be basic, we still have the same primal solution. For the dual problem, the equations $p'A_2 = c_1$ and $p'A_3 = c_3$ yield $(p_1, p_2) = (0, -1)$, which is an infeasible basic solution to the dual, namely, point A'' in Figure 4.6(b).

Example 4.9 has established that different bases may lead to the same basic solution for the primal problem, but to different basic solutions for the dual. Furthermore, out of the different basic solutions to the dual problem, it may be that some are feasible and some are infeasible.

We conclude with a summary of some properties of bases and basic solutions, for standard form problems, that were discussed in this section.

(a) Every basis determines a basic solution to the primal, but also a corresponding basic solution to the dual, namely, $p' = c_B' B^{-1}$.

(b) This dual basic solution is feasible if and only if all of the reduced costs are nonnegative.

(c) Under this dual basic solution, the reduced costs that are equal to zero correspond to active constraints in the dual problem.

(d) This dual basic solution is degenerate if and only if some nonbasic variable has zero reduced cost.

4.6 Farkas' lemma and linear inequalities

Suppose that we wish to determine whether a given system of linear inequalities is infeasible. In this section, we approach this question using duality theory, and we show that infeasibility of a given system of linear inequalities is equivalent to the feasibility of another, related, system of linear inequalities. Intuitively, the latter system of linear inequalities can be interpreted as a search for a *certificate of infeasibility* for the former system.

To be more specific, consider a set of standard form constraints $Ax = b$ and $x \geq 0$. Suppose that there exists some vector p such that $p'A \geq 0'$ and $p'b < 0$. Then, for any $x \geq 0$, we have $p'Ax \geq 0$ and since $p'b < 0$, it follows that $p'Ax \neq p'b$. We conclude that $Ax \neq b$, for all $x \geq 0$. This argument shows that if we can find a vector p satisfying $p'A \geq 0'$ and $p'b < 0$, the standard form constraints cannot have any feasible solution, and such a vector p is a certificate of infeasibility. Farkas' lemma below states that whenever a standard form problem is infeasible, such a certificate of infeasibility p is guaranteed to exist.

Theorem 4.6 (Farkas' lemma) *Let A be a matrix of dimensions $m \times n$ and let b be a vector in $\Re^m$. Then, exactly one of the following two alternatives holds:*

(a) *There exists some $x \geq 0$ such that $Ax = b$.*

(b) *There exists some vector p such that $p'A \geq 0'$ and $p'b < 0$.*

Proof. One direction is easy. If there exists some $x \geq 0$ satisfying $Ax = b$, and if $p'A \geq 0'$, then $p'b = p'Ax \geq 0$, which shows that the second alternative cannot hold.

Let us now assume that there exists no vector $x \geq 0$ satisfying $Ax = b$. Consider the pair of problems

$$\begin{array}{ll} \text{maximize} & 0'x \\ \text{subject to} & Ax = b \\ & x \geq 0, \end{array} \qquad \begin{array}{ll} \text{minimize} & p'b \\ \text{subject to} & p'A \geq 0', \end{array}$$

and note that the first is the dual of the second. The maximization problem is infeasible, which implies that the minimization problem is either unbounded (the optimal cost is $-\infty$) or infeasible. Since $p = 0$ is a feasible solution to the minimization problem, it follows that the minimization problem is unbounded. Therefore, there exists some p which is feasible, that is, $p'A \geq 0'$, and whose cost is negative, that is, $p'b < 0$. $\Box$

We now provide a geometric illustration of Farkas' lemma (see Figure 4.7). Let $A_1, \ldots, A_n$ be the columns of the matrix A and note that $Ax = \sum_{i=1}^n A_i x_i$. Therefore, the existence of a vector $x \geq 0$ satisfying

Figure 4.7: If the vector **b** does not belong to the set of all nonnegative linear combinations of $\mathbf{A}_1,\ldots,\mathbf{A}_n$, then we can find a hyperplane $\{\mathbf{z} \mid \mathbf{p}'\mathbf{z} = 0\}$ that separates it from that set.

alternative. There are several more results of this type; see, for example, Exercises 4.26, 4.27, and 4.28.

Applications of Farkas' lemma to asset pricing

Consider a market that operates for a single period, and in which n different assets are traded. Depending on the events during that single period, there are m possible states of nature at the end of the period. If we invest one dollar in some asset i and the state of nature turns out to be s, we receive a payoff of r_{si}. Thus, each asset i is described by a payoff vector $(r_{1i},\ldots,r_{mi})$. The following $m \times n$ payoff matrix gives the payoffs of each of the n assets for each of the m states of nature:

$$\mathbf{R} = \begin{bmatrix} r_{11} & \cdots & r_{1r} \\ \vdots & \ddots & \vdots \\ r_{m1} & \cdots & r_{ma} \end{bmatrix}.$$

Let x_i be the amount held of asset i. A portfolio of assets is then a vector $\mathbf{x} = (x_1,\ldots,x_n)$. The components of a portfolio $\mathbf{x}$ can be either positive or negative. A positive value of x_i indicates that one has bought x_i units of asset i and is thus entitled to receive $r_{si}x_i$ if state s materializes. A negative value of x_i indicates a "short" position in asset i: this amounts to selling $|x_i|$ units of asset i at the beginning of the period, with a promise to buy them back at the end. Hence, one must pay out $r_{si}|x_i|$ if state s occurs, which is the same as receiving a payoff of $r_{si}x_i$.

$\mathbf{Ax} = \mathbf{b}$ is the same as requiring that $\mathbf{b}$ lies in the set of all nonnegative linear combinations of the vectors $\mathbf{A}_1,\ldots,\mathbf{A}_n$, which is the shaded region in Figure 4.7. If $\mathbf{b}$ does not belong to the shaded region (in which case the first alternative in Farkas' lemma does not hold), we expect intuitively that we can find a vector $\mathbf{p}$ and an associated hyperplane $\{\mathbf{z} \mid \mathbf{p}'\mathbf{z} = 0\}$ such that $\mathbf{b}$ lies on one side of the hyperplane while the shaded region lies on the other side. We then have $\mathbf{p}'\mathbf{b} < 0$ and $\mathbf{p}'\mathbf{A}_i \geq 0$ for all i, or, equivalently, $\mathbf{p}'\mathbf{A} \geq \mathbf{0}'$, and the second alternative holds.

Farkas' lemma predates the development of linear programming, but duality theory leads to a simple proof. A different proof, based on the geometric argument we just gave, is provided in the next section. Finally, there is an equivalent statement of Farkas' lemma which is sometimes more convenient.

Corollary 4.3 *Let* $\mathbf{A}_1,\ldots,\mathbf{A}_n$ *and* $\mathbf{b}$ *be given vectors and suppose that any vector* $\mathbf{p}$ *that satisfies* $\mathbf{p}'\mathbf{A}_i \geq \mathbf{0}$, $i = 1,\ldots,n$, *must also satisfy* $\mathbf{p}'\mathbf{b} \geq 0$. *Then,* $\mathbf{b}$ *can be expressed as a nonnegative linear combination of the vectors* $\mathbf{A}_1,\ldots,\mathbf{A}_n$.

Our next result is of a similar character.

Theorem 4.7 *Suppose that the system of linear inequalities* $\mathbf{Ax} \leq \mathbf{b}$ *has at least one solution, and let d be some scalar. Then, the following are equivalent:*
(a) *Every feasible solution to the system* $\mathbf{Ax} \leq \mathbf{b}$ *satisfies* $\mathbf{c}'\mathbf{x} \leq d$.
(b) *There exists some* $\mathbf{p} \geq \mathbf{0}$ *such that* $\mathbf{p}'\mathbf{A} = \mathbf{c}'$ *and* $\mathbf{p}'\mathbf{b} \leq d$.

Proof. Consider the following pair of problems

maximize $\mathbf{c}'\mathbf{x}$	minimize $\mathbf{p}'\mathbf{b}$
subject to $\mathbf{Ax} \leq \mathbf{b}$,	subject to $\mathbf{p}'\mathbf{A} = \mathbf{c}'$
	$\mathbf{p} \geq \mathbf{0}$,

and note that the first is the dual of the second. If the system $\mathbf{Ax} \leq \mathbf{b}$ has a feasible solution and if every feasible solution satisfies $\mathbf{c}'\mathbf{x} \leq d$, then the first problem has an optimal solution and the optimal cost is bounded above by d. By the strong duality theorem, the second problem also has an optimal solution $\mathbf{p}$ whose cost is bounded above by d. This optimal solution satisfies $\mathbf{p}'\mathbf{A} = \mathbf{c}'$, $\mathbf{p} \geq \mathbf{0}$, and $\mathbf{p}'\mathbf{b} \leq d$.

Conversely, if some $\mathbf{p}$ satisfies $\mathbf{p}'\mathbf{A} = \mathbf{c}'$, $\mathbf{p} \geq \mathbf{0}$, and $\mathbf{p}'\mathbf{b} \leq d$, then the weak duality theorem asserts that every feasible solution to the first problem must also satisfy $\mathbf{c}'\mathbf{x} \leq d$. □

Results such as Theorems 4.6 and 4.7 are often called *theorems of the*

The wealth in state s that results from a portfolio x is given by

$$w_s = \sum_{i=1}^n r_{si} x_i.$$

We introduce the vector $w = (w_1, \ldots, u_m)$, and we obtain

$$w = Rx.$$

Let p_i be the price of asset i in the beginning of the period, and let $p = (p_1, \ldots, p_n)$ be the vector of asset prices. Then, the cost of acquiring a portfolio x is given by $p'x$.

The central problem in asset pricing is to determine what the prices p_i should be. In order to address this question, we introduce the *absence of arbitrage* condition, which underlies much of finance theory: asset prices should always be such that no investor can get a guaranteed nonnegative payoff out of a negative investment. In other words, any portfolio that pays off nonnegative amounts in every state of nature, must be valuable to investors, so it must have nonnegative cost. Mathematically, the absence of arbitrage condition can be expressed as follows:

if $Rx \geq 0$, then we must have $p'x \geq 0$.

Given a particular set of assets, as described by the payoff matrix R, only certain prices p are consistent with the absence of arbitrage. What characterizes such prices? What restrictions does the assumption of no arbitrage impose on asset prices? The answer is provided by Farkas' lemma.

Theorem 4.8 *The absence of arbitrage condition holds if and only if there exists a nonnegative vector $q = (q_1, \ldots, q_m)$, such that the price of each asset i is given by*

$$p_i = \sum_{s=1}^m q_s r_{si}.$$

Proof. The absence of arbitrage condition states that there exists no vector x such that $x'R' \geq 0'$ and $x'p < 0$. This is of the same form as condition (b) in the statement of Farkas' lemma (Theorem 4.6). (Note that here p plays the role of b, and R' plays the role of A.) Therefore, by Farkas' lemma, the absence of arbitrage condition holds if and only if there exists some nonnegative vector q such that $R'q = p$, which is the same as the condition in the theorem's statement. □

Theorem 4.8 asserts that whenever the market works efficiently enough to eliminate the possibility of arbitrage, there must exist "state prices" q_s

that can be used to value the existing assets. Intuitively, it establishes a nonnegative price q_s for an elementary asset that pays one dollar if the state of nature is s, and nothing otherwise. It then requires that every asset must be consistently priced, its total value being the sum of the values of the elementary assets from which it is composed. There is an alternative interpretation of the variables q_s as being (unnormalized) probabilities of the different states s, which, however, we will not pursue. In general, the state price vector q will not be unique, unless the number of assets equals or exceeds the number of states.

The no arbitrage condition is very simple, and yet very powerful. It is the key element behind many important results in financial economics, but these lie beyond the scope of this text. (See, however, Exercise 4.33 for an application in options pricing.)

4.7　From separating hyperplanes to duality*

Let us review the path followed in our development of duality theory. We started from the fact that the simplex method, in conjunction with an anti-cycling rule, is guaranteed to terminate. We then exploited the termination conditions of the simplex method to derive the strong duality theorem. We finally used the duality theorem to derive Farkas' lemma, which we interpreted in terms of a hyperplane that separates b from the columns of A. In this section, we show that the reverse line of argument is also possible. We start from first principles and prove a general result on separating hyperplanes. We then establish Farkas' lemma, and conclude by showing that the duality theorem follows from Farkas' lemma. This line of argument is more elegant and fundamental because instead of relying on the rather complicated development of the simplex method, it only involves a small number of basic geometric concepts. Furthermore, it can be naturally generalized to nonlinear optimization problems.

Closed sets and Weierstrass' theorem

Before we proceed any further, we need to develop some background material. A set $S \subset \Re^n$ is called *closed* if it has the following property: if $x^1, x^2, \ldots$ is a sequence of elements of S that converges to some $x \in \Re^n$, then $x \in S$. In other words, S contains the limit of any sequence of elements of S. Intuitively, the set S contains its boundary.

Theorem 4.9 *Every polyhedron is closed.*

Proof. Consider the polyhedron $P = \{x \in \Re^n \mid Ax \geq b\}$. Suppose that $x^1, x^2, \ldots$ is a sequence of elements of P that converges to some x^*. We have

to show that $\mathbf{x}^* \in P$. For each k, we have $\mathbf{x}^k \in P$ and, therefore, $A\mathbf{x}^k \geq \mathbf{b}$. Taking the limit, we obtain $A\mathbf{x}^* = A(\lim_{k\to\infty} \mathbf{x}^k) = \lim_{k\to\infty}(A\mathbf{x}^k) \geq \mathbf{b}$, and $\mathbf{x}^*$ belongs to P. □

The following is a fundamental result from real analysis that provides us with conditions for the existence of an optimal solution to an optimization problem. The proof lies beyond the scope of this book and is omitted.

Theorem 4.10 (Weierstrass' theorem) *If $f : \Re^n \to \Re$ is a continuous function, and if S is a nonempty, closed, and bounded subset of $\Re^n$, then there exists some $\mathbf{x}^* \in S$ such that $f(\mathbf{x}^*) \leq f(\mathbf{x})$ for all $\mathbf{x} \in S$. Similarly, there exists some $\mathbf{y}^* \in S$ such that $f(\mathbf{y}^*) \geq f(\mathbf{x})$ for all $\mathbf{x} \in S$.*

Weierstrass' theorem is not valid if the set S is not closed. Consider, for example, the set $S = \{x \in \Re \mid x > 0\}$. This set is not closed because we can form a sequence of elements of S that converge to zero, but $x = 0$ does not belong to S. We then observe that the cost function $f(x) = x$ is not minimized at any point in S; for every $x > 0$, there exists another positive number with smaller cost, and no feasible x can be optimal. Ultimately, the reason that S is not closed is that the feasible set was defined by means of *strict* inequalities. The definition of polyhedra and linear programming problems does not allow for strict inequalities in order to avoid situations of this type.

The separating hyperplane theorem

The result that follows is "geometrically obvious" but nevertheless extremely important in the study of convex sets and functions. It states that if we are given a closed and nonempty convex set S and a point $\mathbf{x}^* \notin S$, then we can find a hyperplane, called a *separating hyperplane*, such that S and $\mathbf{x}^*$ lie in different halfspaces (Figure 4.8).

Theorem 4.11 (Separating hyperplane theorem) *Let S be a nonempty closed convex subset of $\Re^n$ and let $\mathbf{x}^* \in \Re^n$ be a vector that does not belong to S. Then, there exists some vector $\mathbf{c} \in \Re^n$ such that $\mathbf{c}'\mathbf{x}^* < \mathbf{c}'\mathbf{x}$ for all $\mathbf{x} \in S$.*

Proof. Let $\|\cdot\|$ be the Euclidean norm defined by $\|\mathbf{x}\| = (\mathbf{x}'\mathbf{x})^{1/2}$. Let us fix some element $\mathbf{w}$ of S, and let

$$B = \{\mathbf{x} \mid \|\mathbf{x} - \mathbf{x}^*\| \leq \|\mathbf{w} - \mathbf{x}^*\|\},$$

and $D = S \cap B$ [Figure 4.9(a)]. The set D is nonempty, because $\mathbf{w} \in D$.
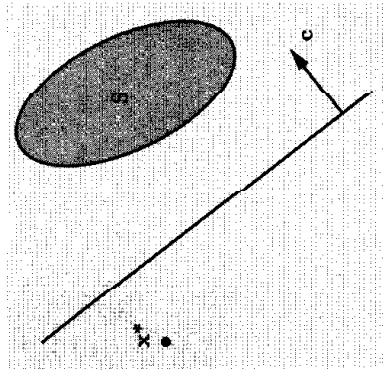
Figure 4.8: A hyperplane that separates the point $\mathbf{x}^*$ from the convex set S.

Furthermore, D is the intersection of the closed set S with the closed set B and is also closed. Finally, D is a bounded set because B is bounded. Consider the quantity $\|\mathbf{x} - \mathbf{x}^*\|$, where $\mathbf{x}$ ranges over the set D. This is a continuous function of $\mathbf{x}$. Since D is nonempty, closed, and bounded, Weierstrass' theorem implies that there exists some $\mathbf{y} \in D$ such that

$$\|\mathbf{y} - \mathbf{x}^*\| \leq \|\mathbf{x} - \mathbf{x}^*\|, \quad \forall \mathbf{x} \in D.$$

For any $\mathbf{x} \in S$ that does not belong to D, we have $\|\mathbf{x} - \mathbf{x}^*\| > \|\mathbf{w} - \mathbf{x}^*\| \geq \|\mathbf{y} - \mathbf{x}^*\|$. We conclude that $\mathbf{y}$ minimizes $\|\mathbf{x} - \mathbf{x}^*\|$ over all $\mathbf{x} \in S$.

We have so far established that there exists an element $\mathbf{y}$ of S which is closest to $\mathbf{x}^*$. We now show that the vector $\mathbf{c} = \mathbf{y} - \mathbf{x}^*$ has the desired property [see Figure 4.9(b)].

Let $\mathbf{x} \in S$. For any λ satisfying $0 < \lambda \leq 1$, we have $\mathbf{y} + \lambda(\mathbf{x} - \mathbf{y}) \in S$, because S is convex. Since $\mathbf{y}$ minimizes $\|\mathbf{x} - \mathbf{x}^*\|$ over all $\mathbf{x} \in S$, we obtain

$$\|\mathbf{y} - \mathbf{x}^*\|^2 \leq \|\mathbf{y} + \lambda(\mathbf{x} - \mathbf{y}) - \mathbf{x}^*\|^2$$
$$= \|\mathbf{y} - \mathbf{x}^*\|^2 + 2\lambda(\mathbf{y} - \mathbf{x}^*)'(\mathbf{x} - \mathbf{y}) + \lambda^2\|\mathbf{x} - \mathbf{y}\|^2,$$

which yields

$$2\lambda(\mathbf{y} - \mathbf{x}^*)'(\mathbf{x} - \mathbf{y}) + \lambda^2\|\mathbf{x} - \mathbf{y}\|^2 \geq 0.$$

We divide by λ and then take the limit as λ decreases to zero. We obtain

$$(\mathbf{y} - \mathbf{x}^*)'(\mathbf{x} - \mathbf{y}) \geq 0.$$

[This inequality states that the angle θ in Figure 4.9(b) is no larger than 90 degrees.] Thus,

$$(\mathbf{y} - \mathbf{x}^*)'\mathbf{x} \geq (\mathbf{y} - \mathbf{x}^*)'\mathbf{y}$$

(a)   (b)
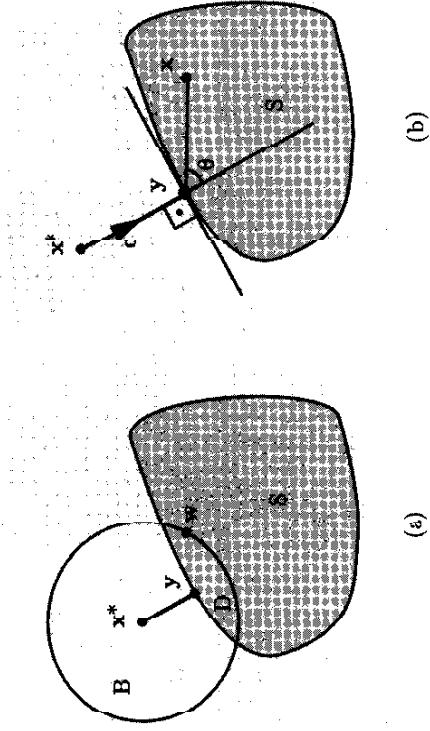
**Figure 4.9:** Illustration of the proof of the separating hyperplane theorem.

$$= (y - x^*)'x^* + (y - x^*)'(y - x^*)$$
$$> (y - x^*)'x^*.$$

Setting $c = y - x^*$ proves the theorem.   □

Farkas' lemma revisited

We now show that Farkas' lemma is a consequence of the separating hyperplane theorem.

We will only be concerned with the difficult half of Farkas' lemma. In particular, we will prove that if the system $Ax = b$, $x \geq 0$, does not have a solution, then there exists a vector p such that $p'A \geq 0'$ and $p'b < 0$.

Let

$$S = \{Ax \mid x \geq 0\}$$
$$= \{y \mid \text{there exists } x \text{ such that } y = Ax, \ x \geq 0\},$$

and suppose that the vector b does not belong to S. The set S is clearly convex; it is also nonempty because $0 \in S$. Finally, the set S is closed; this may seem obvious, but is not easy to prove. For one possible proof, note that S is the projection of the polyhedron $\{(x,y) \mid y = Ax, \ x \geq 0\}$ onto the y coordinates, is itself a polyhedron (see Section 2.8), and is therefore closed. An alternative proof is outlined in Exercise 4.37.

We now invoke the separating hyperplane theorem to separate b from S and conclude that there exists a vector p such that $p'b < p'y$ for every

$y \in S$. Since $0 \in S$, we must have $p'b < 0$. Furthermore, for every column A_i of A and every $\lambda > 0$, we have $\lambda A_i \in S$ and $p'b < \lambda p'A_i$. We divide both sides of the latter inequality by λ and then take the limit as λ tends to infinity, to conclude that $p'A_i \geq 0$. Since this is true for every i, we obtain $p'A \geq 0'$ and the proof is complete.

The duality theorem revisited

We will now derive the duality theorem as a corollary of Farkas' lemma. We only provide the proof for the case where the primal constraints are of the form $Ax \geq b$. The proof for the general case can be constructed along the same lines at the expense of more notation (Exercise 4.38). We also note that the proof given here is very similar to the line of argument used in the heuristic explanation of the duality theorem in Example 4.4.

We consider the following pair of primal and dual problems

minimize	$c'x$		maximize	$p'b$
subject to	$Ax \geq b$,		subject to	$p'A = c'$
				$p \geq 0$,

and we assume that the primal has an optimal solution x^*. We will show that the dual problem also has a feasible solution with the same cost. Once this is done, the strong duality theorem follows from weak duality (cf. Corollary 4.2).

Let $I = \{i \mid a_i'x^* = b_i\}$ be the set of indices of the constraints that are active at x^*. We will first show that any vector d that satisfies $a_i'd \geq 0$ for every $i \in I$, must also satisfy $c'd \geq 0$. Consider such a vector d and let ϵ be a positive scalar. We then have $a_i'(x^* + \epsilon d) \geq a_i'x^* = b_i$ for all $i \in I$. In addition, if $i \notin I$ and if ϵ is sufficiently small, the inequality $a_i'x^* > b_i$ implies that $a_i'(x^* + \epsilon d) > b_i$. We conclude that when ϵ is sufficiently small, $x^* + \epsilon d$ is a feasible solution. By the optimality of x^*, we obtain $c'd \geq 0$, which establishes our claim. By Farkas' lemma (cf. Corollary 4.3), c can be expressed as a nonnegative linear combination of the vectors a_i, $i \in I$, and there exist nonnegative scalars p_i, $i \in I$, such that

$$c = \sum_{i \in I} p_i a_i. \qquad (4.3)$$

For $i \notin I$, we define $p_i = 0$. We then have $p \geq 0$ and Eq. (4.3) shows that the vector p satisfies the dual constraint $p'A = c'$. In addition,

$$p'b = \sum_{i \in I} p_i b_i = \sum_{i \in I} p_i a_i'x^* = c'x^*,$$

which shows that the cost of this dual feasible solution p is the same as the optimal primal cost. The duality theorem now follows from Corollary 4.2.

Notice that if C is a nonempty cone, then $\mathbf{0} \in C$. To this see, consider an arbitrary element $\mathbf{x}$ of C and set $\lambda = 0$ in the definition of a cone; see also Figure 4.10. A polyhedron of the form $P = \{\mathbf{x} \in \Re^n \mid \mathbf{A}\mathbf{x} \geq \mathbf{0}\}$ is easily seen to be a nonempty cone and is called a *polyhedral cone*.

Let $\mathbf{x}$ be a nonzero element of a polyhedral cone C. We then have $3\mathbf{x}/2 \in C$ and $\mathbf{x}/2 \in C$. Since $\mathbf{x}$ is the average of $3\mathbf{x}/2$ and $\mathbf{x}/2$, it is not an extreme point and, therefore, the only possible extreme point is the zero vector. If the zero vector is indeed an extreme point, we say that the cone is *pointed*. Whether this will be the case or not is determined by the criteria provided by our next result.

Theorem 4.12 *Let $C \subset \Re^n$ be the polyhedral cone defined by the constraints $\mathbf{a}_i'\mathbf{x} \geq 0$, $i = 1, \ldots, m$. Then, the following are equivalent:*

(a) *The zero vector is an extreme point of C.*

(b) *The cone C does not contain a line.*

(c) *There exist n vectors out of the family $\mathbf{a}_1, \ldots, \mathbf{a}_m$, which are linearly independent.*

Proof. This result is a special case of Theorem 2.6 in Section 2.5. □

Rays and recession cones

Consider a nonempty polyhedron

$$P = \{\mathbf{x} \in \Re^n \mid \mathbf{A}\mathbf{x} \geq \mathbf{b}\},$$

and let us fix some $\mathbf{y} \in P$. We define the *recession cone at $\mathbf{y}$* as the set of all directions $\mathbf{d}$ along which we can move indefinitely away from $\mathbf{y}$, without leaving the set P. More formally, the recession cone is defined as the set

$$\{\mathbf{d} \in \Re^n \mid \mathbf{A}(\mathbf{y} + \lambda \mathbf{d}) \geq \mathbf{b}, \text{ for all } \lambda \geq 0\}.$$

It is easily seen that this set is the same as

$$\{\mathbf{d} \in \Re^n \mid \mathbf{A}\mathbf{d} \geq \mathbf{0}\},$$

and is a polyhedral cone. This shows that the recession cone is independent of the starting point $\mathbf{y}$; see Figure 4.11. The nonzero elements of the recession core are called the *rays* of the polyhedron P.

For the case of a nonempty polyhedron $F = \{\mathbf{x} \in \Re^n \mid \mathbf{A}\mathbf{x} = \mathbf{b}, \ \mathbf{x} \geq \mathbf{0}\}$ in standard form, the recession cone is seen to be the set of all vectors $\mathbf{d}$ that satisfy

$$\mathbf{A}\mathbf{d} = \mathbf{0}, \qquad \mathbf{d} \geq \mathbf{0}.$$

Figure 4.10: Examples of cones.

In conclusion, we have accomplished the goals that were set out in the beginning of this section. We proved the separating hyperplane theorem, which is a very intuitive and seemingly simple result, but with many important ramifications in optimization and other areas in mathematics. We used the separating hyperplane theorem to establish Farkas' lemma, and finally showed that the strong duality theorem is an easy consequence of Farkas' lemma.

4.8   Cones and extreme rays

We have seen in Chapter 2, that if the optimal cost in a linear programming problem is finite, then our search for an optimal solution can be restricted to finitely many points, namely, the basic feasible solutions, assuming one exists. In this section, we wish to develop a similar result for the case where the optimal cost is $-\infty$. In particular, we will show that the optimal cost can move without ever leaving the feasible set. Furthermore, our search for such a direction can be restricted to a finite set of suitably defined "extreme rays."

Cones

The first step in our development is to introduce the concept of a cone.

Definition 4.1 *A set $C \subset \Re^n$ is a cone if $\lambda \mathbf{x} \in C$ for all $\lambda \geq 0$ and all $\mathbf{x} \in C$.*

Figure 4.11: The recession cone at different elements of a polyhedron.

Extreme rays

We now define the extreme rays of a polyhedron. Intuitively, these are the directions associated with "edges" of the polyhedron that extend to infinity; see Figure 4.12 for an illustration.

Definition 4.2
(a) A nonzero element x of a polyhedral cone $C \subset \Re^n$ is called an **extreme ray** if there are $n-1$ linearly independent constraints that are active at x.

(b) An extreme ray of the recession cone associated with a nonempty polyhedron P is also called an **extreme ray of** P.

Note that a positive multiple of an extreme ray is also an extreme ray. We say that two extreme rays are *equivalent* if one is a positive multiple of the other. Note that for this to happen, they must correspond to the same $n-1$ linearly independent active constraints. Any $n-1$ linearly independent constraints define a line and can lead to at most two nonequivalent extreme rays (one being the negative of the other). Given that there is a finite number of ways that we can choose $n-1$ constraints to become active, and as long as we do not distinguish between equivalent extreme rays, we conclude that the number of extreme rays of a polyhedron is finite. A finite collection of extreme rays will be said to be a *complete set of extreme rays* if it contains exactly one representative from each equivalence class.
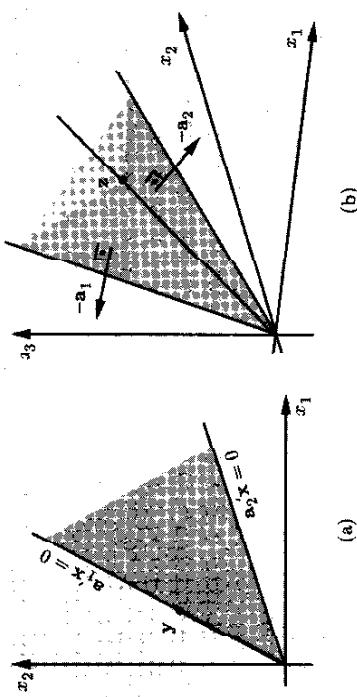
Figure 4.12: Extreme rays of polyhedral cones. (a) The vector y is an extreme ray because $n = 2$ and the constraint $a_1'x = 0$ is active at y. (b) A polyhedral cone defined by three linearly independent constraints of the form $a_i'x \geq 0$. The vector z is an extreme ray because $n = 3$ and the two linearly independent constraints $a_1'x \geq 0$ and $a_2'x \geq 0$ are active at z.

The definition of extreme rays mimics the definition of basic feasible solutions. An alternative and equivalent definition, resembling the definition of extreme points of polyhedra, is explored in Exercise 4.39.

Characterization of unbounded linear programming problems

We now derive conditions under which the optimal cost in a linear programming problem is equal to $-\infty$, first for the case where the feasible set is a cone, and then for the general case.

Theorem 4.13 *Consider the problem of minimizing $c'x$ over a pointed polyhedral cone $C = \{x \in \Re^n \mid a_i'x \geq 0, \ i = 1,\ldots,m\}$. The optimal cost is equal to $-\infty$ if and only if some extreme ray d of C satisfies $c'd < 0$.*

Proof. One direction of the result is trivial because if some extreme ray has negative cost, then the cost becomes arbitrarily negative by moving along this ray.

For the converse, suppose that the optimal cost is $-\infty$. In particular, there exists some $x \in C$ whose cost is negative and, by suitably scaling x,

we can assume that $\mathbf{c'x} = -1$. In particular, the polyhedron

$$P = \{\mathbf{x} \in \Re^n \mid \mathbf{a}_1'\mathbf{x} \geq 0, \ldots, \mathbf{a}_m'\mathbf{x} \geq 0, \mathbf{c'x} = -1\}$$

is nonempty. Since C is pointed, the vectors $\mathbf{a}_1, \ldots, \mathbf{a}_m$ span $\Re^n$ and this implies that P has at least one extreme point; let **d** be one of them. At **d**, we have n linearly independent active constraints, which means that $n-1$ linearly independent constraints of the form $\mathbf{a}_i'\mathbf{x} \geq 0$ must be active. It follows that **d** is an extreme ray of C.

By exploiting duality, Theorem 4.13 leads to a criterion for unboundedness in general linear programming problems. Interestingly enough, this criterion does not involve the right-hand side vector **b**.

Theorem 4.14 *Consider the problem of minimizing* $\mathbf{c'x}$ *subject to* $\mathbf{Ax} \geq \mathbf{b}$, *and assume that the feasible set has at least one extreme point. The optimal cost is equal to* $-\infty$ *if and only if some extreme ray* **d** *of the feasible set satisfies* $\mathbf{c'd} < 0$.

Proof. One direction of the result is trivial because if an extreme ray has negative cost, then the cost becomes arbitrarily negative by starting at a feasible solution and moving along the direction of this ray.

For the proof of the reverse direction, we consider the dual problem:

maximize $\mathbf{p'b}$
subject to $\mathbf{p'A} = \mathbf{c'}$
$\quad\quad\quad \mathbf{p} \geq 0$.

If the primal problem is unbounded, the dual problem is infeasible. Then, the related problem

maximize $\mathbf{p'0}$
subject to $\mathbf{p'A} = \mathbf{c'}$
$\quad\quad\quad \mathbf{p} \geq 0$,

is also infeasible. This implies that the associated primal problem

minimize $\mathbf{c'x}$
subject to $\mathbf{Ax} \geq 0$,

is either unbounded or infeasible. Since $\mathbf{x} = 0$ is one feasible solution, it must be unbounded. Since the primal feasible set has at least one extreme point, the rows of $\mathbf{A}$ span $\Re^n$, where n is the dimension of **x**. It follows that the recession cone $\{\mathbf{x} \mid \mathbf{Ax} \geq 0\}$ is pointed and, by Theorem 4.13, there exists an extreme ray **d** of the recession cone satisfying $\mathbf{c'd} < 0$. By definition, this is an extreme ray of the feasible set. $\quad\square$

The unboundedness criterion in the simplex method

We end this section by pointing out that if we have a standard form problem in which the optimal cost is $-\infty$, the simplex method provides us at termination with an extreme ray.

Indeed, consider what happens when the simplex method terminates with an indication that the optimal cost is $-\infty$. At that point, we have a basis matrix **B**, a nonbasic variable x_j with negative reduced cost, and the jth column $\mathbf{B}^{-1}\mathbf{A}_j$ of the tableau has no positive elements. Consider the jth basic direction **d**, which is the vector that satisfies $\mathbf{d}_B = -\mathbf{B}^{-1}\mathbf{A}_j$, $d_j = 1$, and $d_i = 0$ for every nonbasic index i other than j. Then, the vector **d** satisfies $\mathbf{Ad} = 0$ and $\mathbf{d} \geq 0$, and belongs to the recession cone. It is also a direction of cost decrease, since the reduced cost $\bar{c}_j$ of the entering variable is negative.

Out of the constraints defining the recession cone, the jth basic direction **d** satisfies $n-1$ linearly independent such constraints with equality: these are the constraints $\mathbf{Ac} = 0$ (m of them) and the constraints $d_i = 0$ for i nonbasic and different than j ($n-m-1$ of them). We conclude that **d** is an extreme ray.

4.9   Representation of polyhedra

In this section, we establish one of the fundamental results of linear programming theory. In particular, we show that any element of a polyhedron that has at least one extreme point can be represented as a convex combination of extreme points plus a nonnegative linear combination of extreme rays. A precise statement is given by our next result. A generalization to the case of general polyhedra is developed in Exercise 4.47.

Theorem 4.15 (Resolution theorem) *Let*

$$P = \{\mathbf{x} \in \Re^n \mid \mathbf{Ax} \geq \mathbf{b}\}$$

be a nonempty polyhedron with at least one extreme point. Let $\mathbf{x}^1, \ldots, \mathbf{x}^k$ *be the extreme points, and let* $\mathbf{w}^1, \ldots, \mathbf{w}^r$ *be a complete set of extreme rays of P. Let*

$$Q = \left\{ \sum_{i=1}^k \lambda_i \mathbf{x}^i + \sum_{j=1}^r \theta_j \mathbf{w}^j \;\middle|\; \lambda_i \geq 0, \theta_j \geq 0, \sum_{i=1}^k \lambda_i = 1 \right\}.$$

Then, $Q = P$.

Proof. We first prove that $Q \subset P$. Let

$$x = \sum_{i=1}^{k} \lambda_i x^i + \sum_{j=1}^{r} \theta_j w^j$$

be an element of Q, where the coefficients λ_i and θ_j are nonnegative, and $\sum_{i=1}^{k} \lambda_i = 1$. The vector $y = \sum_{i=1}^{k} \lambda_i x^i$ is a convex combination of elements of P. It therefore belongs to P and satisfies $Ay \geq b$. We also have $Aw^j \geq 0$ for every j, which implies that the vector $z = \sum_{j=1}^{r} \theta_j w^j$ satisfies $Az \geq 0$. It then follows that the vector $x = y + z$ satisfies $Ax \geq b$ and belongs to P.

For the reverse inclusion, we assume that P is not a subset of Q and we will derive a contradiction. Let z be an element of P that does not belong to Q. Consider the linear programming problem

maximize $\displaystyle\sum_{i=1}^{k} 0\lambda_i + \sum_{j=1}^{r} 0\theta_j$

subject to $\displaystyle\sum_{i=1}^{k} \lambda_i x^i + \sum_{j=1}^{r} \theta_j w^j = z$

$\displaystyle\sum_{i=1}^{k} \lambda_i = 1$

$\lambda_i \geq 0, \qquad i = 1,\ldots,k,$
$\theta_j \geq 0, \qquad j = 1,\ldots,r,$

(4.4)

which is infeasible because $z \notin Q$. This problem is the dual of the problem

minimize $p'z + q$
subject to $p'x^i + q \geq 0, \qquad i = 1,\ldots,k,$
$\qquad\qquad p'w^j \geq 0, \qquad j = 1,\ldots,r,$

(4.5)

Because the latter problem has a feasible solution, namely, $p = 0$ and $q = 0$, the optimal cost is $-\infty$, and there exists a feasible solution (p,q) whose cost $p'z + q$ is negative. On the other hand, $p'x^i + q \geq 0$ for all i and this implies that $p'z < p'x^i$ for all i. We also have $p'w^j \geq 0$ for all j. [1]

Having fixed p as above, we now consider the linear programming problem

minimize $p'x$
subject to $Ax \geq b$.

If the optimal cost is finite, there exists an extreme point x^i which is optimal. Since z is a feasible solution, we obtain $p'x^i \leq p'z$, which is a

---

[1] For an intuitive view of this proof, the purpose of this paragraph was to construct a hyperplane that separates z from Q.

contradiction. If the optimal cost is $-\infty$, Theorem 4.14 implies that there exists an extreme ray w^j such that $p'w^j < 0$, which is again a contradiction. $\square$

Example 4.10 Consider the unbounded polyhedron defined by the constraints

$x_1 - x_2 \geq -2$
$x_1 + x_2 \geq 1$
$x_1, x_2 \geq 0$

(see Figure 4.13). This polyhedron has three extreme points, namely, $x^1 = (0,2)$, $x^2 = (0,1)$, and $x^3 = (1,0)$. The recession cone C is described by the inequalities $d_1 - d_2 \geq 0$, $d_1 + d_2 \geq 0$, and $d_1, d_2 \geq 0$. We conclude that $C = \{(d_1,d_2) \mid 0 \leq d_2 \leq d_1\}$. This cone has two extreme rays, namely, $w^1 = (1,1)$ and $w^2 = (1,0)$. The vector $y = (2,2)$ is an element of the polyhedron and can be represented as

$$y = \begin{bmatrix} 2 \\ 2 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} = x^2 + w^1 + w^2.$$

However, this representation is not unique; for example, we also have

$$y = \begin{bmatrix} 2 \\ 2 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 0 \\ 1 \end{bmatrix} + \frac{1}{2}\begin{bmatrix} 1 \\ 0 \end{bmatrix} + \frac{3}{2}\begin{bmatrix} 1 \\ 1 \end{bmatrix} = \frac{1}{2}x^2 + \frac{1}{2}x^3 + \frac{3}{2}w^1.$$
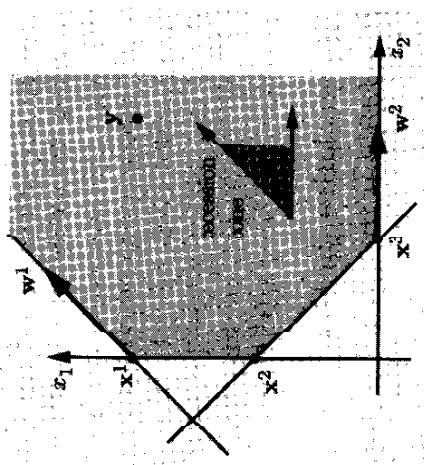

Figure 4.13: The polyhedron of Example 4.10.

We note that the set Q in Theorem 4.15 is the image of the polyhedron

$$H = \left\{ (\lambda_1,\ldots,\lambda_k,\theta_1,\ldots,\theta_r) \;\middle|\; \sum_{i=1}^{k}\lambda_i = 1, \ \lambda_i \geq 0, \ \theta_j \geq 0 \right\},$$

under the linear mapping

$$(\lambda_1,\ldots,\lambda_k,\theta_1,\ldots,\theta_r) \mapsto \sum_{i=1}^{k}\lambda_i\mathbf{x}^i + \sum_{j=1}^{r}\theta_j\mathbf{w}^j.$$

Thus, one corollary of the resolution theorem is that every polyhedron is the image, under a linear mapping, of a polyhedron H with this particular structure.

We now specialize Theorem 4.15 to the case of bounded polyhedra, to recover a result that was also proved in Section 2.7, using a different line of argument.

Corollary 4.4 *A nonempty bounded polyhedron is the convex hull of its extreme points.*

Proof. Let $P = \{\mathbf{x} \mid \mathbf{A}\mathbf{x} \geq \mathbf{b}\}$ be a nonempty bounded polyhedron. If $\mathbf{d}$ is a nonzero element of the cone $C = \{\mathbf{x} \mid \mathbf{A}\mathbf{x} \geq 0\}$ and $\mathbf{x}$ is an element of P, we have $\mathbf{x} + \lambda\mathbf{d} \in P$ for all $\lambda \geq 0$, contradicting the boundedness of P. We conclude that C consists of only the zero vector and does not have any extreme rays. The result then follows from Theorem 4.15. $\square$

There is another corollary of Theorem 4.15 that deals with cones, and which is proved by noting that a cone can have no extreme points other than the zero vector.

Corollary 4.5 *Assume that the cone $C = \{\mathbf{x} \mid \mathbf{A}\mathbf{x} \geq 0\}$ is pointed. Then, every element of C can be expressed as a nonnegative linear combination of the extreme rays of C.*

Converse to the resolution theorem

Let us say that a set Q is *finitely generated* if it is specified in the form

$$Q = \left\{\sum_{i=1}^{k}\lambda_i\mathbf{x}^i + \sum_{j=1}^{r}\theta_j\mathbf{w}^j \;\middle|\; \lambda_i \geq 0,\ \theta_j \geq 0,\ \sum_{i=1}^{k}\lambda_i = 1\right\}, \qquad (4.6)$$

where $\mathbf{x}^1,\ldots,\mathbf{x}^k$ and $\mathbf{w}^1,\ldots,\mathbf{w}^r$ are some given elements of $\Re^n$. The resolution theorem states that a polyhedron with at least one extreme point is a finitely generated set (this is also true for general polyhedra; see Exercise 4.47). We now discuss a converse result, which states that every finitely generated set is a polyhedron.

As observed earlier, a finitely generated set Q can be viewed as the image of the polyhedron

$$H = \left\{(\lambda_1,\ldots,\lambda_k,\theta_1,\ldots,\theta_r) \;\middle|\; \sum_{i=1}^{k}\lambda_i = 1,\ \lambda_i \geq 0,\ \theta_j \geq 0\right\}$$

under a certain linear mapping. Thus, the results of Section 2.8 apply and establish that a finitely generated set is indeed a polyhedron. We record this result and also present a proof based on duality.

Theorem 4.16 *A finitely generated set is a polyhedron. In particular, the convex hull of finitely many vectors is a (bounded) polyhedron.*

Proof. Consider the linear programming problem (4.4) that was used in the proof of Theorem 4.15. A given vector $\mathbf{z}$ belongs to a finitely generated set Q of the form (4.6) if and only if the problem (4.5) has a feasible solution. Using duality, this is the case if and only if problem (4.5) has finite optimal cost. We convert problem (4.5) to standard form by introducing nonnegative variables $\mathbf{p}^+,\mathbf{p}^-,q^+,q^-$, such that $\mathbf{p} = \mathbf{p}^+ - \mathbf{p}^-$, and $q = q^+ - q^-$, as well as surplus variables. Since standard form polyhedra contain no lines, Theorem 4.13 shows that the optimal cost in the standard form problem is finite if and only if

$$(\mathbf{p}^+)'\mathbf{z} - (\mathbf{p}^-)'\mathbf{z} + q^+ - q^- \geq 0,$$

for each one of its finitely many extreme rays. Hence, $\mathbf{z} \in Q$ if and only if $\mathbf{z}$ satisfies a finite collection of linear inequalities. This shows that Q is a polyhedron. $\square$

In conclusion, we have two ways of representing a polyhedron:

(a) in terms of a finite set of linear constraints;

(b) as a finitely generated set, in terms of its extreme points and extreme rays.

These two descriptions are mathematically equivalent, but can be quite different from a practical viewpoint. For example, we may be able to describe a polyhedron in terms of a small number of linear constraints. If on the other hand, this polyhedron has many extreme points, a description as a finitely generated set can be much more complicated. Furthermore, passing from one type of description to the other is, in general, a complicated computational task.

4.10   General linear programming duality*

In the definition of the dual problem (Section 4.2), we associated a dual variable p_i with each constraint of the form $\mathbf{a}_i'\mathbf{x} = b_i$, $\mathbf{a}_i'\mathbf{x} \geq b_i$, or $\mathbf{a}_i'\mathbf{x} \leq b_i$.

However, no dual variables were associated with constraints of the form $x_i \geq 0$ or $x_i \leq 0$. In the same spirit, and in a more general approach to linear programming duality, we can choose arbitrarily which constraints will be associated with price variables and which ones will not. In this section, we develop a general duality theorem that covers such a situation.

Consider the primal problem

minimize $\quad c'x$
subject to $\quad Ax \geq b$
$\qquad\qquad x \in P$,

where P is the polyhedron

$$P = \{x \mid Dx \geq d\}.$$

We associate a dual vector p with the constraint $Ax \geq b$. The constraint $x \in P$ is a generalization of constraints of the form $x_i \geq 0$ or $x_i \leq 0$ and dual variables are not associated with it.

As in Section 4.1, we define the *dual objective* $g(p)$ by

$$g(p) = \min_{x \in P} \left[c'x + p'(b - Ax) \right]. \quad (4.7)$$

The *dual problem* is then defined as

maximize $\quad g(p)$
subject to $\quad p \geq 0$.

We first provide a generalization of the weak duality theorem.

Theorem 4.17 (Weak duality) *If x is primal feasible ($Ax \geq b$ and $x \in P$), and p is dual feasible ($p \geq 0$), then $g(p) \leq c'x$.*

Proof. If x and p are primal and dual feasible, respectively, then $p'(b - Ax) \leq 0$, which implies that

$$g(p) = \min_{y \in P} \left[c'y + p'(b - Ay) \right]$$
$$\leq c'x + p'(b - Ax)$$
$$\leq c'x. \qquad \square$$

We also have the following generalization of the strong duality theorem.

Theorem 4.18 (Strong duality) *If the primal problem has an optimal solution, so does the dual, and the respective optimal costs are equal.*

Proof. Since $P = \{x \mid Dx \geq d\}$, the primal problem is of the form

minimize $\quad c'x$
subject to $\quad Ax \geq b$
$\qquad\qquad Dx \geq d$, $\qquad (4.8)$

and we assume that it has an optimal solution. Its dual, which is

maximize $\quad pb + q'd$
subject to $\quad pA + q'D = c'$
$\qquad\qquad p \geq 0$
$\qquad\qquad q \geq 0$,

must then have the same optimal cost. For any fixed p, the vector q should be chosen optimally in the problem (4.8). Thus, the dual problem (4.8) can also be written as

maximize $\quad p'b + f(p)$
subject to $\quad p \geq 0$,

where $f(p)$ is the optimal cost in the problem

maximize $\quad q'd$
subject to $\quad qD = c' - p'A$
$\qquad\qquad q \geq 0$. $\qquad (4.9)$

[If the latter problem is infeasible, we set $f(p) = -\infty$.] Using the strong duality theorem for problem (4.9), we obtain

$$f(p) = \min_{Dx \geq d} (c'x - p'Ax).$$

We conclude that the dual problem (4.8) has the same optimal cost as the problem

maximize $\quad p'b + \min_{Dx \geq d} (c'x - p'Ax)$
subject to $\quad p \geq 0$.

By comparing with Eq. (4.7), we see that this is the same as maximizing $g(p)$ over all $p \geq 0$. $\qquad \square$

The idea of selectively assigning dual variables to some of the constraints is often used in order to treat "simpler" constraints differently than more "complex" ones, and has numerous applications in large scale optimization. (Applications to integer programming are discussed in Section 11.4.) Finally, let us point out that the approach in this section extends to certain nonlinear optimization problems. For example, if we replace the

linear cost function $c'x$ by a general convex function $c(x)$, and the polyhedron P by a general convex set, we can again define the dual objective according to the formula

$$g(p) = \min_{x \in P} \left[c(x) + p'(b - Ax) \right].$$

It turns out that the strong duality theorem remains valid for such nonlinear problems, under suitable technical conditions, but this lies beyond the scope of this book.

4.11 Summary

We summarize here the main ideas that have been developed in this chapter.

Given a (primal) linear programming problem, we can associate with it another (dual) linear programming problem, by following a set of mechanical rules. The definition of the dual problem is consistent, in the sense that the duals of equivalent primal problems are themselves equivalent.

Each dual variable is associated with a particular primal constraint and can be viewed as a penalty for violating that constraint. By replacing the primal constraints with penalty terms, we increase the set of available options, and this allows us to construct primal solutions whose cost is less than the optimal cost. In particular, every dual feasible vector leads to a lower bound on the optimal cost of the primal problem (this is the essence of the weak duality theorem). The maximization in the dual problem is then a search for the tightest such lower bound. The strong duality theorem asserts that the tightest such lower bound is equal to the optimal primal cost.

An optimal dual variable can also be interpreted as a marginal cost, that is, as the rate of change of the optimal primal cost when we perform a small perturbation of the right-hand side vector b, assuming nondegeneracy.

A useful relation between optimal primal and dual solutions is provided by the complementary slackness conditions. Intuitively, these conditions require that any constraint that is inactive at an optimal solution carries a zero price, which is compatible with the interpretation of prices as marginal costs.

We saw that every basis matrix in a standard form problem determines not only a primal basic solution, but also a basic dual solution. This observation is at the heart of the dual simplex method. This method is similar to the primal simplex method in that it generates a sequence of primal basic solutions, together with an associated sequence of dual basic solutions. It is different, however, in that the dual basic solutions are dual feasible, with ever improving costs, while the primal basic solutions are infeasible (except for the last one). We developed the dual simplex method by simply describing its mechanics and by providing an algebraic justification.

Nevertheless, the dual simplex method also has a geometric interpretation. It keeps moving from one dual basic feasible solution to an adjacent one and, in this respect, it is similar to the primal simplex method applied to the dual problem.

All of duality theory can be developed by exploiting the termination conditions of the simplex method, and this was our initial approach to the subject. We also pursued an alternative line of development that proceeded from first principles and used geometric arguments. This is a more direct and more general approach, but requires more abstract reasoning.

Duality theory provided us with some powerful tools based on which we were able to enhance our geometric understanding of polyhedra. We derived a few theorems of the alternative (like Farkas' lemma), which are surprisingly powerful and have applications in a wide variety of contexts. In fact, Farkas' lemma can be viewed as the core of linear programming duality theory. Another major result that we derived is the resolution theorem, which allows us to express any element of a nonempty polyhedron with at least one extreme point as a convex combination of its extreme points plus a nonnegative linear combination of its extreme rays; in other words, every polyhedron is "finitely generated." The converse is also true, and every finitely generated set is a polyhedron (can be represented in terms of linear inequality constraints). Results of this type play a key role in confirming our intuitive geometric understanding of polyhedra and linear programming. They allow us to develop alternative views of certain situations and lead to deeper understanding. Many such results have an "obvious" geometric content and are often taken for granted. Nevertheless, as we have seen, rigorous proofs can be quite elaborate.

4.12 Exercises

Exercise 4.1 Consider the linear programming problem:

$$
\begin{aligned}
\text{minimize} \quad & x_1 - x_2 \\
\text{subject to} \quad & 2x_1 + 3x_2 - x_3 + x_4 \leq 0 \\
& 3x_1 + x_2 + 4x_3 - 2x_4 \geq 3 \\
& -x_1 - x_2 + 2x_3 + x_4 = 6 \\
& x_1 \leq 0 \\
& x_2, x_3 \geq 0.
\end{aligned}
$$

Write down the corresponding dual problem.

Exercise 4.2 Consider the primal problem

$$
\begin{aligned}
\text{minimize} \quad & c'x \\
\text{subject to} \quad & Ax \geq b \\
& x \geq 0.
\end{aligned}
$$

Form the dual problem and convert it into an equivalent minimization problem. Derive a set of conditions on the matrix A and the vectors b, c, under which the

dual is identical to the primal, and construct an example in which these conditions are satisfied.

Exercise 4.3 The purpose of this exercise is to show that solving linear programming problems is no harder than solving systems of linear inequalities.

Suppose that we are given a subroutine which, given a system of linear inequality constraints, either produces a solution or decides that no solution exists. Construct a simple algorithm that uses a single call to this subroutine and which finds an optimal solution to any linear programming problem that has an optimal solution.

Exercise 4.4 Let A be a symmetric square matrix. Consider the linear programming problem

$$\text{minimize} \quad c'x$$
$$\text{subject to} \quad Ax \geq c$$
$$x \geq 0.$$

Prove that if x^* satisfies $Ax^* \geq 0$, then x^* is an optimal solution.

Exercise 4.5 Consider a linear programming problem in standard form and assume that the rows of A are linearly independent. For each one of the following statements, provide either a proof or a counterexample.

(a) Let x^* be a basic feasible solution. Suppose that for every basis corresponding to x^*, the associated basic solution to the dual is infeasible. Then, the optimal cost must be strictly less that $c'x^*$.

(b) The dual of the auxiliary primal problem considered in Phase I of the simplex method is always feasible.

(c) Let p_i be the dual variable associated with the ith equality constraint in the primal. Eliminating the ith primal equality constraint is equivalent to introducing the additional constraint $p_i = 0$ in the dual problem.

(d) If the unboundedness criterion in the primal simplex algorithm is satisfied, then the dual problem is infeasible.

Exercise 4.6* (**Duality in Chebychev approximation**) Let A be an $m \times n$ matrix and let b be a vector in $\Re^m$. We consider the problem of minimizing $\|Ax - b\|_\infty$ over all $x \in \Re^n$. Here $\| \cdot \|_\infty$ is the vector norm defined by $\|y\|_\infty = \max_i |y_i|$. Let v be the value of the optimal cost.

(a) Let p be any vector in $\Re^m$ that satisfies $\sum_{i=1}^m |p_i| = 1$ and $p'A = 0'$. Show that $p'b \leq v$.

(b) In order to obtain the best possible lower bound of the form considered in part (a), we form the linear programming problem

$$\text{maximize} \quad p'b$$
$$\text{subject to} \quad p'A = 0'$$
$$\sum_{i=1}^m |p_i| \leq 1.$$

Show that the optimal cost in this problem is equal to v.

Exercise 4.7 (**Duality in piecewise linear convex optimization**) Consider the problem of minimizing $\max_{i=1,\ldots,m}(a_i'x - b_i)$ over all $x \in \Re^n$. Let v be the value of the optimal cost, assumed finite. Let A be the matrix with rows $a_1,\ldots,a_m$, and let b be the vector with components $b_1,\ldots,b_m$.

(a) Consider any vector $p \in \Re^m$ that satisfies $p'A = 0'$, $p \geq 0$, and $\sum_{i=1}^m p_i = 1$. Show that $-p'b \leq v$.

(b) In order to obtain the best possible lower bound of the form considered in part (a), we form the linear programming problem

$$\text{maximize} \quad -p'b$$
$$\text{subject to} \quad p'A = 0'$$
$$p'e = 1$$
$$p \geq 0,$$

where e is the vector with all components equal to 1. Show that the optimal cost in this problem is equal to v.

Exercise 4.8 Consider the linear programming problem of minimizing $c'x$ subject to $Ax = b$, $x \geq 0$. Let x^* be an optimal solution, assumed to exist, and let p^* be an optimal solution to the dual.

(a) Let $\bar{x}$ be an optimal solution to the primal, when c is replaced by some $\bar{c}$. Show that $(\bar{c} - c)'(\bar{x} - x^*) \leq 0$.

(b) Let the cost vector be fixed at c, but suppose that we now change b to $\bar{b}$, and let $\bar{x}$ be a corresponding optimal solution to the primal. Prove that $(p^*)'(\bar{b} - b) \leq c'(\bar{x} - x^*)$.

Exercise 4.9 (**Back-propagation of dual variables in a multiperiod problem**) A company makes a product that can be either sold or stored to meet future demand. Let $t = 1,\ldots,T$ denote the periods of the planning horizon. Let b_t be the production volume during period t, which is assumed to be known in advance. During each period t, a quantity x_t of the product is sold at a unit price of d_t. Furthermore, a quantity y_t can be sent to long-term storage, at a unit transportation cost of c. Alternatively, a quantity w_t can be retrieved from storage, at zero cost. We assume that when the product is prepared for long-term storage, it is partly damaged, and only a fraction f of the total survives. Demand is assumed to be unlimited. The main question is whether it is profitable to store some of the production, in anticipation of higher prices in the future. This leads us to the following problem, where z_t stands for the amount kept in long-term storage, at the end of period t

$$\text{maximize} \quad \sum_{t=1}^T \alpha^{t-1}(d_t x_t - c y_t) + \alpha^T d_{T+1} z_T$$

$$\text{subject to} \quad x_t + y_t - w_t = b_t, \qquad t = 1,\ldots,T,$$
$$z_t + w_t - z_{t-1} - f y_t = 0, \qquad t = 1,\ldots,T,$$
$$z_0 = 0,$$
$$x_t, y_t, w_t, z_t \geq 0.$$

Here, d_{T+1} is the salvage price for whatever inventory is left at the end of period T. Furthermore, α is a discount factor, with $0 < \alpha < 1$, reflecting the fact that future revenues are valued less than current ones.

linear optimization.max

(a) Let p_t and q_t be dual variables associated with the first and second equality constraint, respectively. Write down the dual problem.

(b) Assume that $0 < f < 1$, $b_t \geq 0$, and $c \geq 0$. Show that the following formulae provide an optimal solution to the dual problem:

$$q_T = c^T d_{T+1},$$
$$p_T = \max\{\alpha^{T-1} d_T,\ fq_T - \alpha^{T-1}c\},$$
$$q_t = \max\{q_{t+1}, \alpha^{t-1} d_t\}, \qquad t = 1, \ldots, T-1,$$
$$p_t = \max\{\alpha^{t-1} d_t,\ fq_t - \alpha^{t-1}c\}, \qquad t = 1, \ldots, T-1.$$

(c) Explain how the result in part (b) can be used to compute an optimal solution to the original problem. Primal and dual nondegeneracy can be assumed.

Exercise 4.10 (Saddle points of the Lagrangean) Consider the standard form problem of minimizing $c'x$ subject to $Ax = b$ and $x \geq 0$. We define the Lagrangean by

$$L(x, p) = c'x + p'(b - Ax).$$

Consider the following "game": player 1 chooses some $x \geq 0$, and player 2 chooses some p; then, player 1 pays to player 2 the amount $L(x, p)$. Player 1 would like to minimize $L(x, p)$, while player 2 would like to maximize it.

A pair (x^*, p^*), with $x^* \geq 0$, is called an *equilibrium* point (or a *saddle point*, or a *Nash equilibrium*) if

$$L(x^*, p) \leq L(x^*, p^*) \leq L(x, p^*), \qquad \forall x \geq 0, \ \forall p.$$

(Thus, we have an equilibrium if no player is able to improve her performance by unilaterally modifying her choice.)

Show that a pair (x^*, p^*) is an equilibrium if and only if x^* and p^* are optimal solutions to the standard form problem under consideration and its dual, respectively.

Exercise 4.11 Consider a linear programming problem in standard form which is infeasible, but which becomes feasible and has finite optimal cost when the last equality constraint is omitted. Show that the dual of the original (infeasible) problem is feasible and the optimal cost is infinite.

Exercise 4.12* (Degeneracy and uniqueness – I) Consider a general linear programming problem and suppose that we have a nondegenerate basic feasible solution to the primal. Show that the complementary slackness conditions lead to a system of equations for the dual vector that has a unique solution.

Exercise 4.13* (Degeneracy and uniqueness – II) Consider the following pair of problems that are duals of each other:

$$\begin{array}{ll} \text{minimize} & c'x \\ \text{subject to} & Ax = b \\ & x \geq 0, \end{array} \qquad \begin{array}{ll} \text{maximize} & p'b \\ \text{subject to} & p'A \leq c' \end{array}$$

(a) Prove that if one problem has a nondegenerate and unique optimal solution, so does the other.

(b) Suppose that we have a nondegenerate optimal basis for the primal and that the reduced cost for one of the basic variables is zero. What does the result of part (a) imply? Is it true that there must exist another optimal basis?

Exercise 4.14 (Degeneracy and uniqueness – III) Give an example in which the primal problem has a degenerate optimal basic feasible solution, but the dual has a unique optimal solution. (The example need not be in standard form.)

Exercise 4.15 (Degeneracy and uniqueness – IV) Consider the problem

$$\begin{array}{ll} \text{minimize} & x_2 \\ \text{subject to} & x_2 = 1 \\ & x_1 \geq 0 \\ & x_2 \geq 0. \end{array}$$

Write down its dual. For both the primal and the dual problem determine whether they have unique optimal solutions and whether they have nondegenerate optimal solutions. Is this example in agreement with the statement that nondegeneracy of an optimal basic feasible solution in one problem implies uniqueness of optimal solutions for the other? Explain.

Exercise 4.16 Give an example of a pair (primal and dual) of linear programming problems, both of which have multiple optimal solutions.

Exercise 4.17 This exercise is meant to demonstrate that knowledge of a primal optimal solution does not necessarily contain information that can be exploited to determine a dual optimal solution. In particular, determining an optimal solution to the dual is as hard as solving a system of linear inequalities, even if an optimal solution to the primal is available.

Consider the problem of minimizing $c'x$ subject to $Ax \geq 0$, and suppose that we are told that the zero vector is optimal. Let the dimensions of A be $m \times n$, and suppose that we have an algorithm that determines a dual optimal solution and whose running time $O((m+n)^k)$, for some constant k. (Note that if $x = 0$ is not an optimal primal solution, the dual has no feasible solution, and we assume that in this case our algorithm exits with an error message.) Assuming the availability of the above algorithm, construct a new algorithm that takes as input a system of m linear inequalities in n variables, runs for $O((m+n)^k)$ time, and either finds a feasible solution or determines that no feasible solution exists.

Exercise 4.18 Consider a problem in standard form. Suppose that the matrix A has dimensions $m \times n$ and its rows are linearly independent. Suppose that all basic solutions to the primal and to the dual are nondegenerate. Let x be a feasible solution to the primal and let p be a dual vector (not necessarily feasible), such that the pair (x, p) satisfies complementary slackness.

(a) Show that there exist m columns of A that are linearly independent and such that the corresponding components of x are all positive.

(b) Show that x and p are basic solutions to the primal and the dual, respectively.

(c) Show that the result of part (a) is false if the nondegeneracy assumption is removed.

Exercise 4.19 Let $P = \{x \in \Re^n \mid Ax = b, x \geq 0\}$ be a nonempty polyhedron, and let m be the dimension of the vector b. We call x_j a *null variable* if $x_j = 0$ whenever $x \in P$.

(a) Suppose that there exists some $p \in \Re^m$ for which $p'A \geq 0'$, $p'b = 0$, and such that the jth component of $p'A$ is positive. Prove that x_j is a null variable.

(b) Prove the converse of (a): if x_j is a null variable, then there exists some $p \in \Re^m$ with the properties stated in part (a).

(c) If x_j is not a null variable, then by definition, there exists some $y \in P$ for which $y_j > 0$. Use the results in parts (a) and (b) to prove that there exist $x \in P$ and $p \in \Re^m$ such that:

$$p'A \geq 0', \quad p'b = 0, \quad x + A'p > 0.$$

Exercise 4.20 * (Strict complementary slackness)

(a) Consider the following linear programming problem and its dual

$$\begin{array}{ll} \text{minimize} & c'x \\ \text{subject to} & Ax = b \\ & x \geq 0, \end{array} \qquad \begin{array}{ll} \text{maximize} & p'b \\ \text{subject to} & p'A \leq c', \end{array}$$

and assume that both problems have an optimal solution. Fix some j. Suppose that every optimal solution to the primal satisfies $x_j = 0$. Show that there exists an optimal solution p to the dual such that $p'A_j < c_j$. (Here, A_j is the jth column of A.) Hint: Let d be the optimal cost. Consider the problem of minimizing $-x_j$ subject to $Ax = b$, $x \geq 0$, and $-c'x \geq -d$, and form its dual.

(b) Show that there exist optimal solutions x and p to the primal and to the dual, respectively, such that for every j we have either $x_j > 0$ or $p'A_j < c_j$. Hint: Use part (a) for each j, and then take the average of the vectors obtained.

(c) Consider now the following linear programming problem and its dual:

$$\begin{array}{ll} \text{minimize} & c'x \\ \text{subject to} & Ax \geq b \\ & x \geq 0, \end{array} \qquad \begin{array}{ll} \text{maximize} & p'b \\ \text{subject to} & p'A \leq c' \\ & p \geq 0. \end{array}$$

Assume that both problems have an optimal solution. Show that there exist optimal solutions to the primal and to the dual, respectively, that satisfy *strict complementary slackness*, that is:

(i) For every j we have either $x_j > 0$ or $p'A_j < c_j$;
(ii) For every i, we have either $a_i'x > b_i$ or $p_i > 0$. (Here, a_i' is the ith row of A.) Hint: Convert the primal to standard form and apply part (b).

(d) Consider the linear programming problem

$$\begin{array}{ll} \text{minimize} & 5x_1 + 5x_2 \\ \text{subject to} & x_1 + x_2 \geq 2 \\ & 2x_1 - x_2 \geq 0 \\ & x_1, x_2 \geq 0. \end{array}$$

Does the optimal primal solution $(2/3, 4/3)$, together with the corresponding dual optimal solution, satisfy strict complementary slackness? Determine all primal and dual optimal solutions and identify the set of *all* strictly complementary pairs.

Exercise 4.21 * (Clark's theorem) Consider the following pair of linear programming problems:

$$\begin{array}{ll} \text{minimize} & c'x \\ \text{subject to} & Ax \geq b \\ & x \geq 0, \end{array} \qquad \begin{array}{ll} \text{maximize} & p'b \\ \text{subject to} & p'A \leq c' \\ & p \geq 0. \end{array}$$

Suppose that at least one of these two problems has a feasible solution. Prove that the set of feasible solutions to at least one of the two problems is unbounded. Hint: Interpret boundedness of a set in terms of the finiteness of the optimal cost of some linear programming problem.

Exercise 4.22 Consider the dual simplex method applied to a standard form problem with linearly independent rows. Suppose that we have a basis which is primal infeasible, but dual feasible, and let i be such that $x_{B(i)} < 0$. Suppose that all entries in the ith row in the tableau (other than $x_{B(i)}$) are nonnegative. Show that the optimal dual cost is $+\infty$.

Exercise 4.23 Describe in detail the mechanics of a revised dual simplex method that works in terms of the inverse basis matrix B^{-1} instead of the full simplex tableau.

Exercise 4.24 Consider the lexicographic pivoting rule for the dual simplex method and suppose that the algorithm is initialized with each column of the tableau being lexicographically positive. Prove that the dual simplex method does not cycle.

Exercise 4.25 This exercise shows that if we bring the dual problem into standard form and then apply the primal simplex method, the resulting algorithm is not identical to the dual simplex method.

Consider the following standard form problem and its dual.

$$\begin{array}{ll} \text{minimize} & x_1 + x_2 \\ \text{subject to} & x_1 = 1 \\ & x_2 = 1 \\ & x_1, x_2 \geq 0 \end{array} \qquad \begin{array}{ll} \text{maximize} & p_1 + p_2 \\ \text{subject to} & p_1 \leq 1 \\ & p_2 \leq 1. \end{array}$$

Here, there is only one possible basis and the dual simplex method must terminate immediately. Show that if the dual problem is converted into standard form and the primal simplex method is applied to it, one or more changes of basis may be required.

Exercise 4.26 Let A be a given matrix. Show that exactly one of the following alternatives must hold.

(a) There exists some $x \neq 0$ such that $Ax = 0$, $x \geq 0$.

(b) There exists some p such that $p'A > 0'$.

Exercise 4.27 Let A be a given matrix. Show that the following two statements are equivalent.

(a) Every vector such that $Ax \geq 0$ and $x \geq 0$ must satisfy $x_1 = 0$.

(b) There exists some p such that $p'A \leq 0$, $p \geq 0$, and $p'A_1 < 0$, where A_1 is the first column of A.

Exercise 4.28 Let a and $a_1, \ldots, a_m$ be given vectors in $\Re^n$. Prove that the following two statements are equivalent:

(a) For all $x \geq 0$, we have $a'x \leq \max_i a_i'x$.

(b) There exist nonnegative coefficients λ_i that sum to 1 and such that $a \leq \sum_{i=1}^m \lambda_i a_i$.

Exercise 4.29 (Inconsistent systems of linear inequalities) Let $a_1, \ldots, a_m$ be some vectors in $\Re^n$, with $m > n + 1$. Suppose that the system of inequalities $a_i'x \geq b_i$, $i = 1, \ldots, m$, does not have any solution. Show that we can choose $n + 1$ of these inequalities, so that the resulting system of inequalities has no solutions.

Exercise 4.30 (Helly's theorem)

(a) Let $\mathcal{F}$ be a finite family of polyhedra in $\Re^n$ such that every $n+1$ polyhedra in $\mathcal{F}$ have a point in common. Prove that all polyhedra in $\mathcal{F}$ have a point in common. *Hint:* Use the result in Exercise 4.29.

(b) For $n = 2$, part (a) asserts that the polyhedra $P_1, P_2, \ldots, P_K$ $(K \geq 3)$ in the plane have a point in common if and only if every three of them have a point in common. Is the result still true with "three" replaced by "two"?

Exercise 4.31 (Unit eigenvectors of stochastic matrices) We say that an $n \times n$ matrix P, with entries p_{ij}, is *stochastic* if all of its entries are nonnegative and

$$\sum_{j=1}^n p_{ij} = 1, \qquad \forall \, i,$$

that is, the sum of the entries of each row is equal to 1.

Use duality to show that if P is a stochastic matrix, then the system of equations

$$p'P = p', \qquad p \geq 0,$$

has a nonzero solution. (Note that the vector p can be normalized so that its components sum to one. Then, the result in this exercise establishes that every finite state Markov chain has an invariant probability distribution.)

Exercise 4.32 * (Leontief systems and Samuelson's substitution theorem) A *Leontief matrix* is an $m \times n$ matrix A in which every column has at most one positive element. For an interpretation, each column A_j corresponds to a production process. If a_{ij} is negative, $|a_{ij}|$ represents the amount of goods of type i consumed by the process. If a_{ij} is positive, it represents the amount of goods of type j produced by the process. If x_j is the intensity with which process j is used, then Ax represents the net output of the different goods. The matrix A is called *productive* if there exists some $x \geq 0$ such that $Ax > 0$.

(a) Let A be a square productive Leontief matrix $(m = n)$. Show that every vector z that satisfies $Az \geq 0$ must be nonnegative. *Hint:* If z satisfies $Az \geq 0$ but has a negative component, consider the smallest nonnegative θ such that some component of $x + \theta z$ becomes zero, and derive a contradiction.

(b) Show that every square productive Leontief matrix is invertible and that all entries of the inverse matrix are nonnegative. *Hint:* Use the result in part (a).

(c) We now consider the general case where $n \geq m$, and we introduce a constraint of the form $e'x \leq 1$, where $e = (1, \ldots, 1)$. (Such a constraint could capture, for example, a bottleneck due to the finiteness of the labor force.) An "output" vector $y \in \Re^m$ is said to be *achievable* if $y \geq 0$ and there exists some $x \geq 0$ such that $Ax = y$ and $e'y \leq 1$. An achievable vector y is said to be *efficient* if there exists no achievable vector z such that $z \geq y$ and $z \neq y$. (Intuitively, an output vector y which is not efficient can be improved upon and is therefore uninteresting.) Suppose that A is productive. Show that there exists a positive efficient vector y. *Hint:* Given a positive achievable vector y^*, consider maximizing $\sum_i y_i$ over all achievable vectors y that are larger than y^*.

(d) Suppose that A is productive. Show that there exists a set of m production processes that are capable of generating all possible efficient output vectors y. That is, there exist indices $E(1), \ldots, B(m)$, such that every efficient output vector y can be expressed in the form $y = \sum_{i=1}^m A_{B(i)} x_{B(i)}$, for some nonnegative coefficients $x_{B(i)}$ whose sum is bounded by 1. *Hint:* Consider the problem of minimizing $e'x$ subject to $Ax = y$, $x \geq 0$, and show that we can use the same optimal basis for all efficient vectors y.

Exercise 4.33 (Options pricing) Consider a market that operates for a single period, and which involves three assets: a stock, a bond, and an option. Let S be the price of the stock, in the beginning of the period. Its price $\overline{S}$ at the end of the period is random and is assumed to be equal to either Su, with probability β, or Sd, with probability $1 - \beta$. Here u and d are scalars that satisfy $d < 1 < u$. Bonds are assumed riskless. Investing one dollar in a bond results in a payoff of r, at the end of the period. (Here, r is a scalar greater than 1.) Finally, the option gives us the right to purchase, at the end of the period, one stock at a fixed price of K. If the realized price $\overline{S}$ of the stock is greater than K, we exercise the option and then immediately sell the stock in the stock market, for a payoff of $\overline{S} - K$. If on the other hand we have $\overline{S} < K$, there is no advantage in exercising the option, and we receive zero payoff. Thus, the value of the option at the end of the period is equal to $\max(0, \overline{S} - K)$. Since the option is itself an asset it

should have a value in the beginning of the time period. Show that under the absence of arbitrage condition, the value of the option must be equal to

$$\gamma \max\{0, Su - K\} + \delta \max\{0, Sd - K\},$$

where γ and δ are a solution to the following system of linear equations:

$$u\gamma + d\delta = 1$$
$$\gamma + \delta = \frac{1}{r}.$$

Hint: Write down the payoff matrix $\mathbf{R}$ and use Theorem 4.8.

Exercise 4.34 (Finding separating hyperplanes) Consider a polyhedron P that has at least one extreme point.

(a) Suppose that we are given the extreme points $\mathbf{x}^i$ and a complete set of extreme rays $\mathbf{w}^j$ of P. Create a linear programming problem whose solution provides us with a separating hyperplane that separates P from the origin, or allows us to conclude that none exists.

(b) Suppose now that P is given to us in the form $P = \{\mathbf{x} \mid \mathbf{a}_i'\mathbf{x} \geq b_i, \ i = 1, \ldots, m\}$. Suppose that $\mathbf{0} \notin P$. Explain how a separating hyperplane can be found.

Exercise 4.35 (Separation of disjoint polyhedra) Consider two nonempty polyhedra $P = \{\mathbf{x} \in \Re^n \mid \mathbf{Ax} \leq \mathbf{b}\}$ and $Q = \{\mathbf{x} \in \Re^n \mid \mathbf{Dx} \leq \mathbf{d}\}$. We are interested in finding out whether the two polyhedra have a point in common.

(a) Devise a linear programming problem such that: if $P \cap Q$ is nonempty, it returns a point in $P \cap Q$; if $P \cap Q$ is empty, the linear programming problem is infeasible.

(b) Suppose that $P \cap Q$ is empty. Use the dual of the problem you have constructed in part (a) to show that there exists a vector $\mathbf{c}$ such that $\mathbf{c}'\mathbf{x} < \mathbf{c}'\mathbf{y}$ for all $\mathbf{x} \in P$ and $\mathbf{y} \in Q$.

Exercise 4.36 (Containment of polyhedra)

(a) Let P and Q be two polyhedra in $\Re^n$ described in terms of linear inequality constraints. Devise an algorithm that decides whether P is a subset of Q.

(b) Repeat part (a) if the polyhedra are described in terms of their extreme points and extreme rays.

Exercise 4.37 (Closedness of finitely generated cones) Let $A_1, \ldots, A_n$ be given vectors in $\Re^m$. Consider the cone $C = \{\sum_{i=1}^n A_i x_i \mid x_i \geq 0\}$ and let $\mathbf{y}^k$, $k = 1, 2, \ldots$, be a sequence of elements of C that converges to some $\mathbf{y}$. Show that $\mathbf{y} \in C$ (and hence C is closed), using the following argument. With $\mathbf{y}$ fixed as above, consider the problem of minimizing $\|\mathbf{y} - \sum_{i=1}^n A_i x_i\|_\infty$, subject to the constraints $x_1, \ldots, x_n \geq 0$. Here $\|\cdot\|_\infty$ stands for the maximum norm, defined by $\|\mathbf{x}\|_\infty = \max_i |x_i|$. Explain why the above minimization problem has an optimal solution, find the value of the optimal cost, and prove that $\mathbf{y} \in C$.

Exercise 4.38 (From Farkas' lemma to duality) Use Farkas' lemma to prove the duality theorem for a linear programming problem involving constraints of the form $\mathbf{a}_i'\mathbf{x} = b_i$, $\mathbf{a}_i'\mathbf{x} \geq b_i$, and nonnegativity constraints for some of the variables x_j. Hint: Start by deriving the form of the set of feasible directions at an optimal solution.

Exercise 4.39 (Extreme rays of cones) Let us define a nonzero element $\mathbf{d}$ of a pointed polyhedral cone C to be an extreme ray if it has the following property: if there exist vectors $\mathbf{f} \in C$ and $\mathbf{g} \in C$ and some $\lambda \in (0,1)$ satisfying $\mathbf{d} = \mathbf{f} + \mathbf{g}$, then both $\mathbf{f}$ and $\mathbf{g}$ are scalar multiples of $\mathbf{d}$. Prove that this definition of extreme rays is equivalent to Definition 4.2.

Exercise 4.40 (Extreme rays of a cone are extreme points of its sections) Consider the cone $C = \{\mathbf{x} \in \Re^n \mid \mathbf{a}_i'\mathbf{x} \geq 0, \ i = 1, \ldots, m\}$ and assume that the first n constraint vectors $\mathbf{a}_1, \ldots, \mathbf{a}_n$ are linearly independent. For any nonnegative scalar r, we define the polyhedron P_r by

$$P_r = \left\{ \mathbf{x} \in C \ \middle|\ \sum_{i=1}^n \mathbf{a}_i'\mathbf{x} = r \right\}.$$

(a) Show that the polyhedron P_r is bounded for every $r \geq 0$.

(b) Let $r > 0$. Show that a vector $\mathbf{x} \in P_r$ is an extreme point of P_r if and only if $\mathbf{x}$ is an extreme ray of the cone C.

Exercise 4.41 (Carathéodory's theorem) Show that every element $\mathbf{x}$ of a bounded polyhedron $P \subset \Re^n$ can be expressed as a convex combination of at most $n + 1$ extreme points of P. Hint: Consider an extreme point of the set of all possible representations of $\mathbf{x}$.

Exercise 4.42 (Problems with side constraints) Consider the linear programming problem of minimizing $\mathbf{c}'\mathbf{x}$ over a bounded polyhedron $P \subset \Re^n$ and subject to additional constraints $\mathbf{a}_i'\mathbf{x} = b_i$, $i = 1, \ldots, L$. Assume that the problem has a feasible solution. Show that there exists an optimal solution which is a convex combination of $L + 1$ extreme points of P. Hint: Use the resolution theorem to represent P.

Exercise 4.43

(a) Consider the minimization of $c_1 x_1 + c_2 x_2$ subject to the constraints

$$x_2 - 5 \leq x_1 \leq 2x_2 + 2, \qquad x_1, x_2 \geq 0.$$

Find necessary and sufficient conditions on (c_1, c_2) for the optimal cost to be finite.

(b) For a general feasible linear programming problem, consider the set of all cost vectors for which the optimal cost is finite. Is it a polyhedron? Prove your answer.

Exercise 4.44

(a) Let $P = \{(x_1, x_2) \mid x_1 - x_2 = 0,\ x_1 + x_2 = 0\}$. What are the extreme points and the extreme rays of P?

(b) Let $P = \{(x_1, x_2) \mid 4x_1 + 2x_2 \geq 8,\ 2x_1 + x_2 \leq 8\}$. What are the extreme points and the extreme rays of P?

(c) For the polyhedron of part (b), is it possible to express each one of its elements as a convex combination of its extreme points plus a nonnegative linear combination of its extreme rays? Is this compatible with the resolution theorem?

Exercise 4.45 Let P be a polyhedron with at least one extreme point. Is it possible to express an arbitrary element of P as a convex combination of its extreme points plus a nonnegative multiple of a single extreme ray?

Exercise 4.46 (Resolution theorem for polyhedral cones) Let C be a nonempty polyhedral cone.

(a) Show that C can be expressed as the union of a finite number $C_1, \ldots, C_k$ of pointed polyhedral cones. Hint: Intersect with orthants.

(b) Show that an extreme ray of C must be an extreme ray of one of the cones $C_1, \ldots, C_k$.

(c) Show that there exists a finite number of elements $\mathbf{w}^1, \ldots, \mathbf{w}^r$ of C such that

$$C = \left\{ \sum_{i=1}^{r} \theta_i \mathbf{w}^i \ \middle|\ \theta_1, \ldots, \theta_r \geq 0 \right\}.$$

Exercise 4.47 (Resolution theorem for general polyhedra) Let P be a polyhedron. Show that there exist vectors $\mathbf{x}^1, \ldots, \mathbf{x}^k$ and $\mathbf{w}^1, \ldots, \mathbf{w}^r$ such that

$$P = \left\{ \sum_{i=1}^{k} \lambda_i \mathbf{x}^i + \sum_{j=1}^{r} \theta_j \mathbf{w}^j \ \middle|\ \lambda_i \geq 0,\ \theta_j \geq 0,\ \sum_{i=1}^{k} \lambda_i = 1 \right\}.$$

Hint. Generalize the steps in the preceding exercise.

Exercise 4.48 * (Polar, finitely generated, and polyhedral cones) For any cone C, we define its *polar* $C^\perp$ by

$$C^\perp = \{\mathbf{p} \mid \mathbf{p}'\mathbf{x} \leq 0,\ \text{for all } \mathbf{x} \in C\}.$$

(a) Let F be a finitely generated cone, of the form

$$F = \left\{ \sum_{i=1}^{r} \theta_i \mathbf{w}^i \ \middle|\ \theta_1, \ldots, \theta_r \geq 0 \right\}.$$

Show that $F^\perp = \{\mathbf{p} \mid \mathbf{p}'\mathbf{w}^i \leq 0,\ i = 1, \ldots, r\}$, which is a polyhedral cone.

(b) Show that the polar of $F^\perp$ is F and conclude that the polar of a polyhedral cone is finitely generated. *Hint:* Use Farkas' lemma.

(c) Show that a finitely generated pointed cone F is a polyhedron. *Hint:* Consider the polar of the polar.

(d) **(Polar cone theorem)** Let C be a closed, nonempty, and convex cone. Show that $(C^\perp)^\perp = C$. *Hint:* Mimic the derivation of Farkas' lemma using the separating hyperplane theorem (Section 4.7).

(e) Is the polar cone theorem true when C is the empty set?

Exercise 4.49 Consider a polyhedron, and let $\mathbf{x}, \mathbf{y}$ be two basic feasible solutions. If we are only allowed to make moves from any basic feasible solution to an adjacent one, show that we can go from $\mathbf{x}$ to $\mathbf{y}$ in a finite number of steps. *Hint:* Generalize the simplex method to nonstandard form problems: starting from a nonoptimal basic feasible solution, move along an extreme ray of the cone of feasible directions.

Exercise 4.50 We are interested in the problem of deciding whether a polyhedron

$$Q = \{\mathbf{x} \in \Re^n \mid A\mathbf{x} \leq \mathbf{b},\ D\mathbf{x} \geq \mathbf{d},\ \mathbf{x} \geq 0\}$$

is nonempty. We assume that the polyhedron $P = \{\mathbf{x} \in \Re^n \mid A\mathbf{x} \leq \mathbf{b},\ \mathbf{x} \geq 0\}$ is nonempty and bounded. For any vector $\mathbf{p}$, of the same dimension as $\mathbf{d}$, we define

$$g(\mathbf{p}) = -\mathbf{p}'\mathbf{d} + \max_{\mathbf{x} \in P} \mathbf{p}'D\mathbf{x}.$$

(a) Show that if Q is nonempty, then $g(\mathbf{p}) \geq 0$ for all $\mathbf{p} \geq 0$.

(b) Show that if Q is empty, then there exists some $\mathbf{p} \geq 0$, such that $g(\mathbf{p}) < 0$.

(c) If Q is empty, what is the minimum of $g(\mathbf{p})$ over all $\mathbf{p} \geq 0$?

4.13 Notes and sources

4.3. The duality theorem is due to von Neumann (1947), and Gale, Kuhn, and Tucker (1951).

4.6. Farkas' lemma is due to Farkas (1894) and Mirkowski (1896). See Schrijver (1986) for a comprehensive presentation of related results. The connection between duality theory and arbitrage was developed by Ross (1976, 1978).

4.7. Weierstrass' Theorem and its proof can be found in most texts on real analysis; see, for example, Rudin (1976). While the simplex method is only relevant to linear programming problems with a finite number of variables, the approach based on the separating hyperplane theorem leads to a generalization of duality theory that covers more general convex optimization problems, as well as infinite-dimensional linear programming problems, that is, linear programming problems with infinitely many variables and constraints; see, e.g., Luenberger (1969) and Rockafellar (1970).

4.9. The resolution theorem and its converse are usually attributed to Farkas, Minkowski, and Weyl.

in Section 11.1. The same idea can also be applied to more general convex optimization problems; see, e.g., Bertsekas (1995b).

6.4. Dantzig-Wolfe decomposition was developed by Dantzig and Wolfe (1960). Example 6.2 is adapted from Bradley, Hax, and Magnanti (1977).

6.5. Stochastic programming began with work by Dantzig in the 1950s and has been extensively studied since then. Some books on this subject are Kall and Wallace (1994), and Infanger (1993); Example 6 is adapted from the latter reference. The Benders decomposition method was developed by Benders (1962). It finds applications other contexts as well, such as discrete optimization; see, e.g., Schrijver (1986) and Nemhauser and Wolsey (1988).

Chapter 7

Network flow problems

Contents

Network flow problems (also known as *transshipment* problems) are the most frequently solved linear programming problems. They include as special cases, the assignment, transportation, maximum flow, and shortest path problems, and they arise naturally in the analysis and design of communication, transportation, and logistics networks, as well as in many other contexts.

The network flow problem is a special case of linear programming and any algorithm for linear programming can be directly applied. On the other hand, network flow problems have a special structure which results in substantial simplification of general methods (e.g., of the simplex method), as well as in new, special purpose, methods.

From a high level point of view, most of the available algorithms for network flow problems fall into one of three categories:

(a) **Primal methods.** These methods maintain and keep improving a primal feasible solution. The primal *simplex method*, presented in Section 7.3, is an important representative. An alternative algorithm is derived from first principles in Section 7.4.

(b) **Dual ascent methods.** These methods, which are discussed in Section 7.7, maintain a dual feasible solution and an auxiliary primal (usually infeasible) solution that satisfy complementary slackness. The dual variables are updated so as to increase the value of the dual objective and reduce the infeasibility of the complementary primal solution. The *Hungarian*, *primal-dual*, *relaxation*, and *dual simplex* methods fall in this general category.

(c) **Approximate dual ascent methods.** These methods are similar in spirit to the dual ascent methods, except that small decreases in the dual objective are allowed to occur and the complementary slackness conditions are only approximately enforced. The *auction* algorithm, which is discussed in Section 7.6, as well as the *ε-relaxation* and *preflow-push* methods, are of this type.

In this chapter, all three of the above mentioned algorithm types will be encountered. The chapter begins with a brief introduction to graphs (Section 7.1), that provides us with the language for studying network flow problems, and with a problem formulation (Section 7.2). We develop a number of general methods, but we also pay attention to special cases whose structure can be further exploited, such as the maximum flow problem (Section 7.5), the assignment problem (Section 7.8), and the shortest path problem (Section 7.9). We also discuss the minimum spanning tree problem (Section 7.10), which is not a network flow problem but has a similar underlying graph structure. Throughout this chapter, our focus is on major algorithmic ideas, rather than on the refinements that can lead to better complexity estimates.

7.1 Graphs

Network flow problems are defined on graphs. In this section, we introduce graphs formally and provide a number of elementary definitions and properties.

Undirected graphs

An *undirected graph* $G = (\mathcal{N}, \mathcal{E})$ consists of a set $\mathcal{N}$ of *nodes* and a set $\mathcal{E}$ of (*undirected*) *arcs* or *edges*, where an edge e is an *unordered pair* of distinct nodes, that is, a two-element subset $\{i, j\}$ of $\mathcal{N}$; see Figure 7.1. Note that
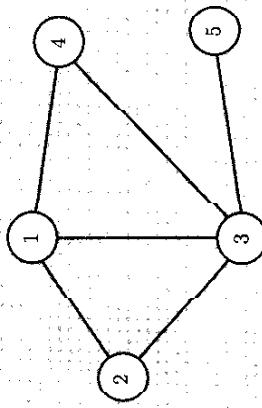

Figure 7.1: An undirected graph $G = (\mathcal{N}, \mathcal{E})$ with $\mathcal{N} = \{1, 2, 3, 4, 5\}$ and $\mathcal{E} = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 4\}, \{3, 4\}, \{3, 5\}\}$.

an undirected arc $\{i, j\}$ is one and the same object as the undirected arc $\{j, i\}$. Furthermore, "self-arcs" like $\{i, i\}$ are not allowed. We say that the arc $\{i, j\}$ is *incident* to nodes i and j, and these nodes are called the *endpoints* of the arc.

The *degree* of a node in an undirected graph is the number of arcs incident to that node. The degree of an undirected graph is defined as the maximum of the degrees of its nodes.

A *walk* from node i_1 to node i_t in an undirected graph is defined as a finite sequence of nodes $i_1, i_2, \ldots, i_t$ such that $\{i_k, i_{k+1}\} \in \mathcal{E}$, $k = 1, 2, \ldots, t-1$. A walk is called a *path* if it has no repeated nodes. A *cycle* is defined as a walk $i_1, i_2, \ldots, i_t$ such that the nodes $i_1, \ldots, i_{t-1}$ are distinct (and hence form a path) and $i_t = i_1$. In addition, we require the number $t - 1$ of distinct nodes to be at least 3. This is in order to exclude a walk of the form i, j, i, where the same arc $\{i, j\}$ is traversed back and forth. An undirected graph is said to be *connected* if for every two distinct nodes $i, j \in \mathcal{N}$, there exists a path from i to j.

As an example, the graph in Figure 7.1 is connected. The sequence 1,2,3,1,4 is a walk but not a path. The sequence 1,2,3,1 is a cycle, and the sequence 1,3,5 is a path.

For undirected graphs, we will often denote the number of nodes by $|N|$ or n, and the number of edges by $|E|$ or m.

Directed graphs

A *directed graph* $G = (N, A)$ consists of a set N of *nodes* and a set of *(directed) arcs*, where a directed arc is an *ordered pair* (i, j) of distinct nodes; see Figure 7.2. Our definition allows for both (i, j) and (j, i) to be
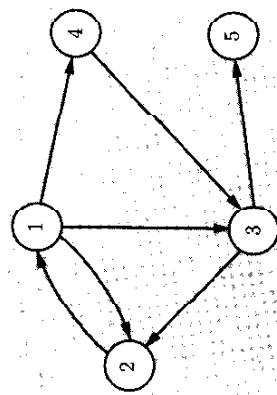

Figure 7.2: A directed graph $G = (N, A)$ with $N = \{1, 2, 3, 4, 5\}$ and $A = \{(1,2), (2,1), (1,3), (3,2), (1,4), (4,3), (3,5)\}$.

elements of the arc set A, but self-arcs like (i, i) are not allowed.

For any arc (i, j), we say that i is the *start node* and j is the *end node*. The arc (i, j) is said to be *outgoing* from node i, *incoming* to node j, and *incident* to both i and j. We define $I(i)$ and $O(i)$ as the set of start nodes (respectively, end nodes) of arcs that are incoming to (respectively, outgoing from) node i. That is,

$$I(i) = \{j \in N \mid (j, i) \in A\},$$

and

$$O(i) = \{j \in N \mid (i, j) \in A\}.$$

Starting from a directed graph, we can construct a corresponding undirected graph by ignoring the direction of the arcs and by deleting repeated arcs; for example, the directed graph in Figure 7.2 leads to the undirected graph in Figure 7.1. Under one possible interpretation, flow or movement in a directed arc is permitted only from the start node to the end node, whereas in an undirected arc, flow or movement is permitted in both directions. We say that a directed graph is *connected* if the resulting undirected graph is connected.

We now present a definition of walks in directed graphs; it is important to note that this definition allows us to traverse an arc in either direction, irrespective of the arc's direction. More specifically, a *walk* is

defined as a sequence $i_1, \ldots, i_t$ of nodes, together with an associated sequence $a_1, \ldots, a_{t-1}$ of arcs such that for $k = 1, \ldots, t-1$, we have either $a_k = (i_k, i_{k+1})$ (in which case we say that a_k is a *forward* arc) or $a_k = (i_{k+1}, i_k)$ (in which case we say that a_k is a *backward* arc). Note that if i_k and i_{k+1} are consecutive nodes in a walk and if (i_k, i_{k+1}) and (i_{k+1}, i_k) are both arcs of the underlying directed graph, then either arc can be used in the walk. The reason for including the arcs a_k in the definition of a walk is precisely to avoid such ambiguities.

A walk is said to be a *path* if all of its nodes $i_1, \ldots, i_t$ are distinct, and a *cycle* if the nodes $i_1, \ldots, i_{t-1}$ are distinct and $i_t = i_1$. Note that we allow a cycle to consist of only two distinct nodes (in contrast to our definition for the case of undirected graphs). Thus, a sequence $i, (i, j), j, (j, i), i$ is a bona fide cycle. Finally, a walk, path, or cycle is said to be *directed* if it only contains forward arcs.

For the graph shown in Figure 7.2, the sequence $1, (1, 3), 3, (3, 2), 2, (1, 2), 1, (1, 4), 4$ is a walk, but not a directed walk, because $(1, 2)$ is a backward arc. The sequence $1, (1, 3), 3, (3, 2), 2, (2, 1), 1$ is a directed cycle. The sequence $1, (1, 2), 2, (2, 1), 1$ is also a directed cycle. The sequence $4, (4, 3), 3, (1, 3), 1, (1, 2), 2$ is a path, but not a directed path, because $(1, 3)$ is a backward arc.

For directed graphs, we will often denote the number of nodes by $|N|$ or n, and the number of arcs by $|A|$ or m.

Trees

An undirected graph $G = (N, E)$ is called a *tree* if it is connected and has no cycles. If a node of a tree has degree equal to 1, it is called a *leaf*. See Figure 7.3 for an illustration.
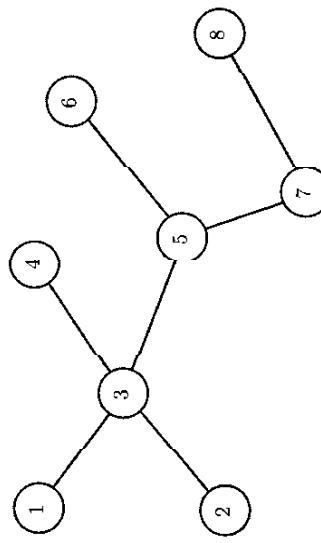

Figure 7.3: A tree with 8 nodes, 7 arcs, and 5 leaves. Note that if we were to add the arc $\{2, 7\}$, a single cycle would be created, namely, 2,3,5,7,2.

We now present some important properties of trees that will be of u[se] later on (e.g., in the development of the simplex method, in Section 7.3).

Theorem 7.1

(a) *Every tree with more than one node has at least one leaf.*

(b) *An undirected graph is a tree if and only if it is connected and has $|N| - 1$ arcs.*

(c) *For any two distinct nodes i and j in a tree, there exists a unique path from i to j.*

(d) *If we start with a tree and add a new arc, the resulting graph contains exactly one cycle (as long as we do not distinguish between cycles involving the same set of nodes).*

Proof.

(a) Consider a tree with more than one node and suppose that there are no leaves. Then, every node has degree greater than 1. (If the degree of a node were 1, that node would be a leaf, and if it were 0, the graph would not be connected.) Therefore, given a node and an arc through which we enter the node, we can find a different arc through which we can exit. By repeating such a process, we must eventually visit the same node twice, which implies that there exists a cycle, contradicting the definition of a tree.

(b) We first prove that every tree has $|N| - 1$ arcs. This is trivially true if the tree has a single node. Consider now a tree that has more than one node. Such a tree must have at least one leaf, by part (a). We delete that leaf together with the single arc incident to that node. The resulting graph is again a tree, because the deletion of a leaf cannot create a cycle or cause a graph to become disconnected. This process can be carried out $|N| - 1$ times, until we are left with a single node and, therefore, no arcs. Since at each stage there was exactly one arc deletion, we conclude that the original tree had $|N| - 1$ arcs.

In order to prove the converse statement, let us consider a connected graph with $|N| - 1$ arcs. If this graph contains a cycle, we can delete one of the arcs in the cycle and still maintain connectivity. We repeat this process as many times as needed, until we are left with a connected graph without any cycles, that is, a tree. We have already proved that a tree with $|N|$ nodes must have $|N| - 1$ arcs, and this shows that the final tree has as many arcs as the original graph. It follows that no arc was deleted and the original graph was a tree to start with.

(c) Suppose that there exist two different paths joining the same nodes i and j. By joining these two paths and by deleting any arcs that ar[e]


Figure 7.4: (a) An undirected graph. The thicker arcs form a spanning tree. (b) Another undirected graph. The arcs $\{1,2\}$, $\{2,3\}$, $\{4,6\}$ do not form any cycle. They can be augmented to form a spanning tree, e.g., by adding arcs $\{3,6\}$ and $\{5,6\}$.

common to both, we are left with one or more cycles, contradicting the definition of a tree.

(d) Consider a tree, and let us add an undirected arc $\{i,j\}$. Using part (b), the resulting graph must have $|N|$ arcs. Therefore, it cannot be a tree, and must have a cycle. Any cycle created by this addition consists of the arc $\{i,j\}$ and a path from i to j. Since there exists a unique path from i to j [part (c)], it follows that a unique cycle has been created. $\qquad\square$

Spanning trees

Given a connected undirected graph $G = (N, \mathcal{E})$, let $\mathcal{E}_1$ be a subset of $\mathcal{E}$ such that $T = (N, \mathcal{E}_1)$ is a tree. Such a tree is called a *spanning tree*. The following result will be used later on (in Sections 7.3 and 7.10) and is illustrated in Figure 7.4.

Theorem 7.2 *Let $G = (N, \mathcal{E})$ be a connected undirected graph and let $\mathcal{E}_0$ be some subset of the set $\mathcal{E}$ of arcs. Suppose that the arcs in $\mathcal{E}_0$ do not form any cycles. Then, the set $\mathcal{E}_0$ can be augmented to a set $\mathcal{E}_1 \supset \mathcal{E}_0$ so that $(N, \mathcal{E}_1)$ is a spanning tree.*

Proof. Let $G = (N, \mathcal{E})$ be a connected undirected graph. Suppose that $\mathcal{E}_0 \subset \mathcal{E}$, and that the arcs in $\mathcal{E}_0$ do not form any cycles. If G is a tree, we may let $\mathcal{E}_1 = \mathcal{E}$ and we are done. Otherwise, G contains at least one cycle. A cycle cannot consist exclusively of arcs in $\mathcal{E}_0$, because of our assumption on $\mathcal{E}_0$. Let us choose and delete an arc that lies on a cycle and that doe[s]

We now present some important properties of trees that will be of use later on (e.g., in the development of the simplex method, in Section 7.3).

Theorem 7.1

(a) Every tree with more than one node has at least one leaf.

(b) An undirected graph is a tree if and only if it is connected and has $|N| - 1$ arcs.

(c) For any two distinct nodes i and j in a tree, there exists a unique path from i to j.

(d) If we start with a tree and add a new arc, the resulting graph contains exactly one cycle (as long as we do not distinguish between cycles involving the same set of nodes).

Proof.

(a) Consider a tree with more than one node and suppose that there are no leaves. Then, every node has degree greater than 1. (If the degree of a node were 1, that node would be a leaf, and if it were 0, the graph would not be connected.) Therefore, given a node and an arc through which we enter the node, we can find a different arc through which we can exit. By repeating such a process, we must eventually visit the same node twice, which implies that there exists a cycle, contradicting the definition of a tree.

(b) We first prove that every tree has $|N| - 1$ arcs. This is trivially true if the tree has a single node. Consider now a tree that has more than one node. Such a tree must have at least one leaf by part (a). We delete that leaf together with the single arc incident to that node. The resulting graph is again a tree, because the deletion of a leaf cannot create a cycle or cause a graph to become disconnected. This process can be carried out $|N| - 1$ times, until we are left with a single node and, therefore, no arcs. Since at each stage there was exactly one arc deletion, we conclude that the original tree had $|N| - 1$ arcs.

In order to prove the converse statement, let us consider a connected graph with $|N| - 1$ arcs. If this graph contains a cycle, we can delete one of the arcs in the cycle and still maintain connectivity. We repeat this process as many times as needed, until we are left with a connected graph without any cycles, that is, a tree. We have already proved that a tree with $|N|$ nodes must have $|N| - 1$ arcs, and this shows that the final tree has as many arcs as the original graph. It follows that no arc was deleted and the original graph was a tree to start with.

(c) Suppose that there exist two different paths joining the same nodes i and j. By joining these two paths and by deleting any arcs that are

common to both, we are left with one or more cycles, contradicting the definition of a tree.

(d) Consider a tree, and let us add an undirected arc $\{i, j\}$. Using part (b), the resulting graph must have $|N|$ arcs. Therefore, it cannot be a tree, and must have a cycle. Any cycle created by this addition consists of the arc $\{i, j\}$ and a path from i to j. Since there exists a unique path from i to j [part (c)], it follows that a unique cycle has been created. $\square$

Figure 7.4: (a) An undirected graph. The thicker arcs form a spanning tree. (b) Another undirected graph. The arcs $\{1,2\}$, $\{2,3\}$, $\{4,6\}$ do not form any cycle. They can be augmented to form a spanning tree, e.g., by adding arcs $\{3,6\}$ and $\{5,6\}$.

Spanning trees

Given a connected undirected graph $G = (N, E)$, let E_1 be a subset of E such that $T = (N, E_1)$ is a tree. Such a tree is called a *spanning tree*. The following result will be used later on (in Sections 7.3 and 7.10) and is illustrated in Figure 7.4.

Theorem 7.2 *Let $G = (N, E)$ be a connected undirected graph and let E_0 be some subset of the set E of arcs. Suppose that the arcs in E_0 do not form any cycles. Then, the set E_0 can be augmented to a set $E_1 \supset E_0$ so that (N, E_1) is a spanning tree.*

Proof. Let $G = (N, E)$ be a connected undirected graph. Suppose that $E_0 \subset E$, and that the arcs in E_0 do not form any cycles. If G is a tree, we may let $E_1 = E$ and we are done. Otherwise, G contains at least one cycle. A cycle cannot consist exclusively of arcs in E_0, because of our assumption on E_0. Let us choose and delete an arc that lies on a cycle and that does

not belong to $\mathcal{E}_0$. The resulting graph is still connected. By repeating this process as many times as needed, we end up with a connected graph $(\mathcal{N}, \mathcal{E}_1)$ without any cycles, hence a tree. In addition, since the arcs in $\mathcal{E}_0$ are never deleted, we have $\mathcal{E}_0 \subset \mathcal{E}_1$. □

7.2 Formulation of the network flow problem

A *network* is a directed graph $G = (\mathcal{N}, \mathcal{A})$ together with some additional numerical information, such as numbers b_i representing the external *supply* to each node $i \in \mathcal{N}$, nonnegative (possibly infinite) numbers u_{ij} representing the *capacity* of each arc $(i,j) \in \mathcal{A}$, and numbers c_{ij} representing the cost per unit of flow along arc (i,j).

We visualize a network by thinking of some material that flows on each arc. We use f_{ij} to denote the amount of flow through arc (i,j). The supply b_i is interpreted as the amount of flow that enters the network from the outside, at node i. In particular, node i is called a *source* if $b_i > 0$, and a *sink* if $b_i < 0$. If node i is a sink, the quantity $|b_i|$ is sometimes called the *demand* at node i. We impose the following conditions on the flow variables $f_{ij}, (i,j) \in \mathcal{A}$:

$$b_i + \sum_{j \in I(i)} f_{ji} = \sum_{j \in O(i)} f_{ij}, \qquad \forall\, i \in \mathcal{N}, \qquad (7.1)$$

$$0 \le f_{ij} \le u_{ij}, \qquad \forall\, (i,j) \in \mathcal{A}. \qquad (7.2)$$

Equation (7.1) is a flow conservation law: it states that the amount of flow into a node i must be equal to the total flow out of that node. Equation (7.2) simply requires that the flow through an arc must be nonnegative and cannot exceed the capacity of the arc. Any vector with components f_{ij}, $(i,j) \in \mathcal{A}$, will be called a *flow*. If it also satisfies the constraints (7.1)-(7.2), it will be called a *feasible flow*.

By summing both sides of Eq. (7.1) over all $i \in \mathcal{N}$, we obtain

$$\sum_{i \in \mathcal{N}} b_i = 0,$$

which means that the total flow from the environment into the network (at the sources) must be equal to the total flow from the network (at the sinks) to the environment. From now on, we will always assume that the condition $\sum_{i \in \mathcal{N}} b_i = 0$ holds, because otherwise no flow vector could satisfy the flow conservation constraints, and we would have an infeasible problem.

The general minimum cost network flow problem deals with the minimization of a linear cost function of the form

$$\sum_{(i,j) \in \mathcal{A}} c_{ij} f_{ij},$$

over all feasible flows. We observe that this is a linear programming problem. If $u_{ij} = \infty$ for all $(i,j) \in \mathcal{A}$, we say that the problem is *uncapacitated*; otherwise, we say that it is *capacitated*. Note that in the uncapacitated case, we only have equality and nonnegativity constraints, and the problem is in standard form.

We now provide an overview of important special cases of the network flow problem; most of them will be studied later in this chapter.

The shortest path problem

For any directed path in a network, we define its *length* as the sum of the costs of all arcs on the path. We wish to find a *shortest path*, that is, a directed path from a given origin node to a given destination node whose length is smallest. This problem is studied in Section 7.9, where we show that it can be formulated as a network flow problem, under a certain assumption on the arc lengths.

The maximum flow problem

In the maximum flow problem, we wish to determine the largest possible amount of flow that can be sent from a given source node to a given sink node, without exceeding the arc capacities. This problem is studied in Section 7.5.

The transportation problem

Let there be m suppliers and n consumers. The ith supplier can provide s_i units of a certain good and the jth consumer has a demand for d_j units. We assume that the total supply $\sum_{i=1}^m s_i$ is equal to the total demand $\sum_{j=1}^n d_j$. Finally, we assume that the transportation of goods from the ith supplier to the jth consumer carries a cost of c_{ij} per unit of goods transported. The problem is to transport the goods from the suppliers to the consumers at minimum cost. Let f_{ij} be the amount of goods transported from the ith supplier to the jth consumer. We then have the following problem:

$$
\begin{aligned}
\text{minimize} \quad & \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} f_{ij} \\
\text{subject to} \quad & \sum_{i=1}^{n} f_{ij} = d_j, \qquad j = 1, \ldots, n, \\
& \sum_{j=1}^{n} f_{ij} = s_i, \qquad i = 1, \ldots, m, \\
& f_{ij} \ge 0, \qquad \forall\, i, j.
\end{aligned}
$$

The first equality constraint specifies that the demand d_j of each consumer must be met; the second equality constraint requires that the entire supply s_i of each supplier must be shipped. This is a special case of the uncapacitated network flow problem, where the underlying graph has a special structure; see Figure 7.5. It turns out that every network flow problem ca...
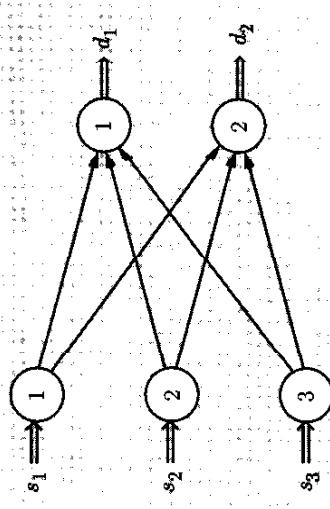

Figure 7.5: A network corresponding to a transportation problem with three suppliers and two consumers.

be transformed into an equivalent transportation problem (Exercises 7... and 7.6). Consequently, any algorithm for the transportation problem ca... be adapted and can be used to solve general network flow problems. Fo... this reason, the initial development and testing of new algorithms is often carried out for the special case of transportation problems.

The assignment problem

The assignment problem is a special case of the transportation problem, where the number of suppliers is equal to the number of consumers, each supplier has unit supply, and each consumer has unit demand. As will be proved later in this chapter, one can always find an optimal solution in which every f_{ij} is either 0 or 1. This means that for each i there will be a unique and distinct j for which $f_{ij} = 1$, and we can say that the ith supplier is assigned to the jth consumer; this justifies the name of this problem.

Variants of the network flow problem

There are several variants of the network flow problem all of which can be shown to be equivalent to each other. For example, we have already mentioned that every network flow problem is equivalent to a transportation problem. We now discuss some more examples.

(a) Every network flow problem can be reduced to one with exactly one source and exactly one sink node. This is illustrated in Figure 7.6.
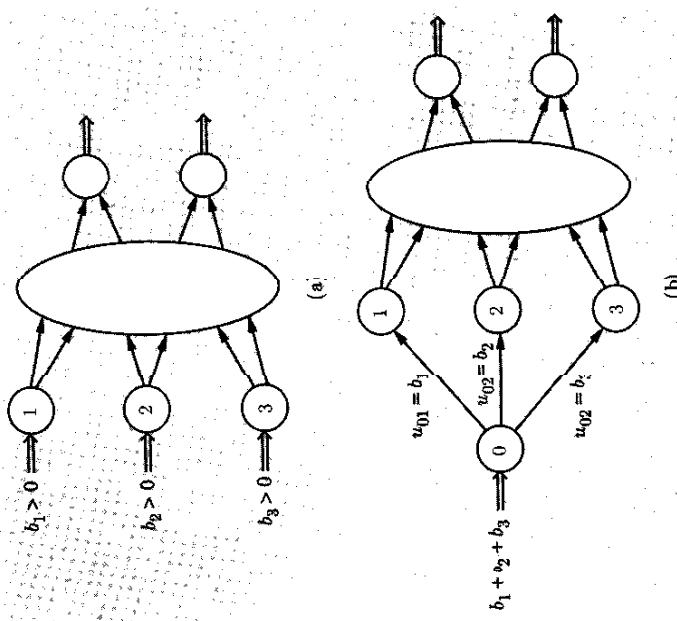
Figure 7.6: (a) A network with three source nodes. (b) A network with only one source node. The costs of the new arcs are zero. Because of the way that the arc capacities u_{0i} are chosen ($u_{0i} = b_i$, $i = 1, 2, 3$), exactly b_i units must flow on each arc $(0, i)$, $i = 1, 2, 3$. The reduction to a network with a single sink node is similar.

(b) Every network flow problem can be reduced to one *without sources or sinks*. (Problems in which all of the supplies are zero are called *circulation* problems.) Consider, without loss of generality, a network with a single source s and a single sink t. We introduce a new arc (t, s) whose capacity u_{ts} is equal to b_s and whose unit cost is $c_{ts} = -M$, where M is a large number; see Figure 7.7. Since M is large, an optimal solution to the circulation problem will try to set f_{ts} to b_s, which has the same effect as having a supply of b_s at node s. If an optimal solution to the circulation problem does not succeed in setting f_{ts} to b_s; this means that there is no way of shipping b_s units of flow from s to t, and the original problem is infeasible.

(c) *Node capacities.* Suppose that we have an upper bound of g_i on the total flow that can enter a given node i; for example, if i is a source
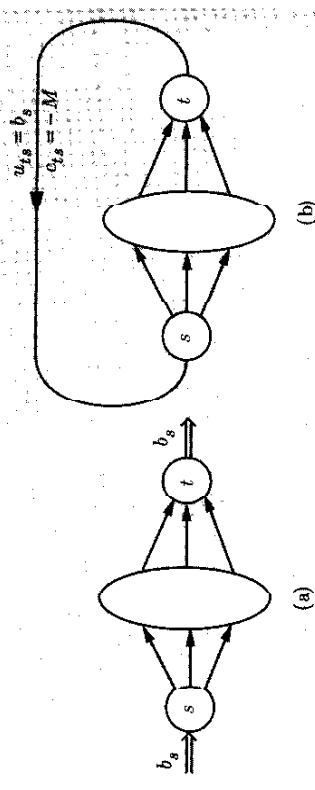
Figure 7.7: (a) A network. (b) An equivalent circulation problem.

node, we may have a constraint

$$b_i + \sum_{j \in I(i)} f_{ji} \le g_i.$$

By splitting node i into two nodes i and i', and by letting g_i be the capacity of arc (i, i'), we are back to the case where we only have arc capacities; see Figure 7.8.
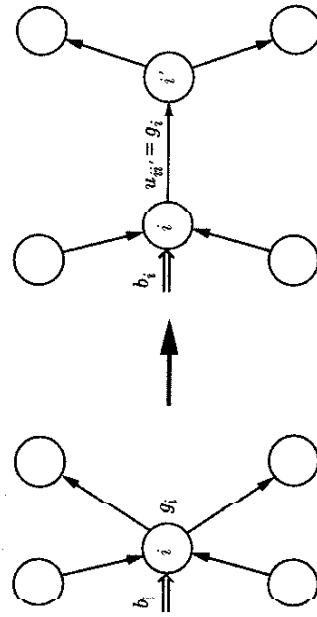

Figure 7.8: Transformation of a node capacity into an arc capacity.

(d) Lower bounds on the arc flows. Suppose that we add constraints of the form $f_{ij} \ge d_{ij}$, where d_{ij} are given scalars. The resulting problem can be reduced to an equivalent problem in which every d_{ij} is equal to zero. Exercise 7.7 provides some guidance as to how this can be accomplished.

A concise formulation

We now discuss how to rewrite the network flow problem, and especially the flow conservation constraint, in more economical matrix-vector notation. We assume that $\mathcal{N} = \{1, \ldots, n\}$ and we let m be the number of arcs. Let us fix a particular ordering of the arcs, and let $\mathbf{f}$ be the vector of flows that results when the components f_{ij} are ordered accordingly. We define the node-arc incidence matrix $\mathbf{A}$ as follows: its dimensions are $n \times m$ (each row corresponds to a node and each column to an arc) and its (i, k)th entry a_{ik} is associated with the ith node and the kth arc. We let

$$a_{ik} = \begin{cases} 1, & \text{if } i \text{ is the start node of the } k\text{th arc,} \\ -1, & \text{if } i \text{ is the end node of the } k\text{th arc,} \\ 0, & \text{otherwise.} \end{cases}$$

Thus, every column of $\mathbf{A}$ has exactly two nonzero entries, one equal to $+1$, and one equal to -1, indicating the start and the end node of the corresponding arc.

Example 7.1 Consider the directed graph of Figure 7.2 and let us use the following ordering of the arcs: $(1, 2), (2, 1), (3, 2), (4, 3), (1, 4), (1, 3), (3, 5)$. The corresponding node-arc incidence matrix is

$$\mathbf{A} = \begin{bmatrix} 1 & -1 & 0 & 0 & 1 & 1 & 0 \\ -1 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & -1 & 1 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix}.$$

Let us now focus on the ith row of $\mathbf{A}$, denoted by $\mathbf{a}_i'$ (this is the row associated with node i). Nonzero entries indicate the arcs that are incident to node i; such entries are $+1$ or -1 depending on whether the arc is outgoing or incoming, respectively. Thus,

$$\mathbf{a}_i'\mathbf{f} = \sum_{j \in O(i)} f_{ij} - \sum_{j \in I(i)} f_{ji},$$

and the flow conservation constraint at node i [cf. Eq. (7.1)] can be written as

$$\mathbf{a}_i'\mathbf{f} = b_i,$$

or, in matrix notation,

$$\mathbf{Af} = \mathbf{b},$$

where $\mathbf{b}$ is the vector $(b_1, \ldots, b_n)$.

We observe that the sum of the rows of $\mathbf{A}$ is equal to the zero vector; in particular, the rows of $\mathbf{A}$ are linearly dependent. Thus, the matrix $\mathbf{A}$ violates one of the basic assumptions underlying our development of the

simplex method. As discussed in Chapter 2 (cf. Theorem 2.5 in Section 2.3), either the problem is infeasible or we can remove some of the equality constraints, without affecting the feasible set, so that the remaining constraints are linearly independent. We revisit this issue in the next section.

Circulations

We close by introducing some elementary concepts that are central to many network flow algorithms.

Any flow vector f (feasible or infeasible) that satisfies

$$Af = 0,$$

is called a *circulation*. Intuitively, we have flow conservation within the network and zero external supply or demand which means that the flow "circulates" inside the network.

Let us now consider a cycle C. We let F and B be the set of forward and backward arcs of the cycle, respectively. The flow vector h^C with components

$$h_{ij}^C = \begin{cases} 1, & \text{if } (i,j) \in F, \\ -1, & \text{if } (i,j) \in B, \\ 0, & \text{otherwise.} \end{cases}$$

is called the *simple circulation* associated with the cycle C. It is easily seen that h^C satisfies

$$Ah^C = 0, \qquad (7.3)$$

and is indeed a circulation. The reason is that any two consecutive arcs on the cycle are either similarly oriented and carry the same amount of flow, or they have the opposite orientation and the sum of the flows that they carry is equal to 0; in either case, the net inflow to any node is zero; see Figure 7.9. We finally define the *cost of a cycle* C to be equal to

$$c'h^C = \sum_{(i,j)\in F} c_{ij} - \sum_{(i,j)\in B} c_{ij}.$$

If f is a flow vector, C is a cycle, and θ is a scalar, we say that the flow vector $f + \theta h^C$ is obtained from f by *pushing* θ units of flow around the cycle C. Note that the resulting cost change is θ times the cost $c'h^C$ of the cycle C.

7.3 The network simplex algorithm

In this section, we develop the details of the simplex method, as applied to the uncapacitated network flow problem

minimize $c'f$
subject to $Af = b$
$f \geq 0,$
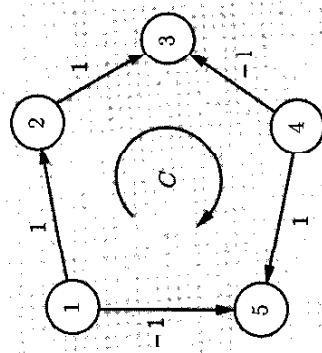
Figure 7.9: A cycle and the corresponding simple circulation. Arcs $(4,3)$ and $(1,5)$ are backward arcs and carry a flow of -1. Note that flow is conserved at each node.

where A is the node-arc incidence matrix of a directed graph $G = (N, A)$. (Capacitated problems are briefly discussed at the end of this section.) The network simplex algorithm is widely used in practice, and is included in many commercial optimization codes, due to its simplicity and efficiency. In particular, it tends to run an order of magnitude faster than a general purpose simplex code applied to a network flow problem.

Due to our restriction to uncapacitated problems, we are dealing with a linear programming problem in standard form. We let m and n be the number of arcs and nodes, respectively. We therefore have m flow variables and n equality constraints which, unfortunately, is the exact opposite of the notational conventions used in earlier chapters.

There are two different ways of developing the network simplex method. The first is to go through the mechanics of the general simplex method and specialize each step to the present context. The second is to develop the algorithm from first principles and then to point out that it is a special case of the simplex method. We take a middle ground that proceeds along two parallel tracks; each step is justified from first principles, but its relation to the simplex method is also explained. The end result is an algorithm with a fairly intuitive structure.

Throughout this section, the following assumption will be in effect.

Assumption 7.1.
(a) We have $\sum_{i\in N} b_i = 0$.
(b) The graph G is connected.

Part (a) of this assumption is natural, because otherwise the problem is infeasible. Part (b) is also natural, because if the graph is not connected,

then the problem can be decomposed into subproblems that can be treated independently.

As noted in Section 7.2, the rows of the matrix $\mathbf{A}$ sum to the zero vector and are therefore linearly dependent. In fact, the last constraint (flow conservation at node n) is a consequence of the flow conservation constraints at the other nodes, and can be omitted without affecting the feasible set. Let us define the *truncated node-arc incidence matrix* $\tilde{\mathbf{A}}$ to be the matrix of dimensions $(n-1) \times m$, which consists of the first $n-1$ rows of the matrix $\mathbf{A}$. Any column of $\tilde{\mathbf{A}}$ that corresponds to an arc of the form (i,n) has a single nonzero entry, equal to 1, at the ith row. Similarly, any column of $\tilde{\mathbf{A}}$ that corresponds to an arc of the form (n,i) has a single nonzero entry, equal to -1, at the ith row. All other columns of $\tilde{\mathbf{A}}$ have two nonzero entries. Let $\tilde{\mathbf{b}} = (b_1,\ldots,b_{n-1})$. We replace the original equality constraint $\mathbf{Af} = \mathbf{b}$ by the constraint $\tilde{\mathbf{A}}\mathbf{f} = \tilde{\mathbf{b}}$. We will see shortly that under Assumption 7.1, the matrix $\tilde{\mathbf{A}}$ has linearly independent rows.

Example 7.2 Consider the node-arc incidence matrix $\mathbf{A}$ in Example 7.1. The associated matrix $\tilde{\mathbf{A}}$ is given by

$$\tilde{\mathbf{A}} = \begin{bmatrix} 1 & -1 & 0 & 0 & \vdots & 1 & 0 \\ -1 & 1 & -1 & 0 & \vdots & 0 & 0 \\ 0 & 0 & 1 & -1 & \vdots & -1 & 1 \\ 0 & 0 & 0 & 1 & \vdots & 0 & 0 \end{bmatrix}.$$

It can be verified that the matrix $\tilde{\mathbf{A}}$ has full rank. For example, the third, fourth, sixth, and seventh columns are linearly independent.

Trees and basic feasible solutions

We now introduce an important definition.

Definition 7.1 *A flow vector* $\mathbf{f}$ *is called a* **tree solution** *if it can be constructed by the following procedure.*
(a) *Pick a set* $T \subset \mathcal{A}$ *of* $n-1$ *arcs that form a tree when their direction is ignored*
(b) *Let* $f_{ij} = 0$ *for every* $(i,j) \notin T$.
(c) *Use the flow conservation equation* $\tilde{\mathbf{A}}\mathbf{f} = \tilde{\mathbf{b}}$ *to determine the flow variables* f_{ij}, *for* $(i,j) \in T$.
A tree solution that also satisfies $\mathbf{f} \geq \mathbf{0}$, *is called a* **feasible tree solution.**

Step (c) in the above definition can be carried out using the following systematic procedure, illustrated in Figure 7.10:
(a) Call node n the *root* of the tree.
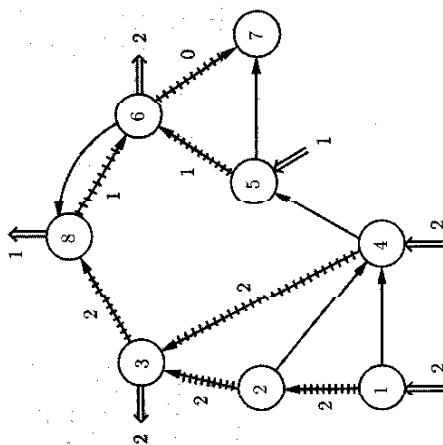
Figure 7.10: A network and a set of $n-1$ arcs (indicated by thatched lines) that form a tree. By setting the arc flows outside the tree to zero, we obtain $f_{12} = 2$, $f_{23} = 2$ and $f_{43} = 2$. We then use conservation of flow at node 3, to obtain $f_{38} = 2$. We also have $f_{56} = 1$ and $f_{67} = 0$. Using conservation of flow at node 6, we obtain $f_{86} = 1$. Note that this is a feasible tree solution.

(b) Use the flow conservation equations to determine the flows on the arcs incident to the leaves, and continue by proceeding from the leaves towards the root.

It should be pretty obvious from Figure 7.10 that once a tree is fixed, a corresponding tree solution is uniquely determined. Nevertheless, we provide a rigorous proof.

Theorem 7.3 *Let* $T \subset \mathcal{A}$ *be a set of* $n-1$ *arcs that form a tree when their direction is ignored. Then, the system of linear equations* $\tilde{\mathbf{A}}\mathbf{f} = \tilde{\mathbf{b}}$, *and* $f_{ij} = 0$ *for all* $(i,j) \notin T$ *has a unique solution.*

Proof. Let $\mathbf{B}$ be the $(n-1) \times (n-1)$ matrix that results if we only keep those $n-1$ columns of $\tilde{\mathbf{A}}$ that correspond to the arcs in T. Let $\mathbf{f}_T$ be the subvector of $\mathbf{f}$, of dimension $n-1$, whose entries are the flow variables f_{ij}, $(i,j) \in T$. We need to show that the linear system $\mathbf{B}\mathbf{f}_T = \tilde{\mathbf{b}}$ has a unique solution. For this, it suffices to show that the matrix $\mathbf{B}$ is nonsingular.

Let us assume that the nodes have been renumbered so that numbers increase along any path from a leaf to the root node n. Let us also assign

$$\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 \\
-1 & 0 & 0 & 0 & -1 & 0 & 0 \\
0 & -1 & -1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & -1 & 0 & 0 & -1
\end{bmatrix}$$
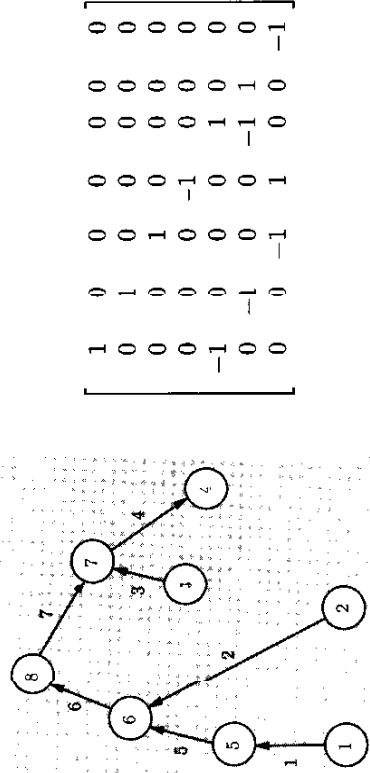
Figure 7.11: A numbering of the nodes and arcs of a tree, and the corresponding B matrix.

to every arc $(i,j) \in T$, the number $\min\{i,j\}$; see Figure 7.11. Such a renumbering of nodes and arcs amounts to a reordering of the rows and columns of **B** but does not affect whether **B** is singular or not.

With the above numbering, the ith column of **B** corresponds to the ith arc, which is an arc of the form (i,j) or (j,i), with $j > i$. Thus, any nonzero entries in the ith column will be in row i or j. Since $j > i$, no nonzero entry can be found above the diagonal. We conclude that **B** is lower triangular and has nonzero diagonal entries. This implies that **B** has nonzero determinant and is nonsingular, which completes the proof. □

We note an important corollary of the proof of the previous theorem.

Corollary 7.1 *If the graph G is connected, then the matrix $\bar{\mathbf{A}}$ has linearly independent rows.*

Proof. If the graph G is connected, then there exists a set of arcs $T \subset A$ that form a tree, when their orientation is ignored (cf Theorem 7.2). Let us pick such a set T and form the corresponding matrix **B**, as in the proof of Theorem 7.3. Since the $(n-1) \times (n-1)$ matrix **B** is nonsingular, it has linearly independent columns. Hence, the matrix $\bar{\mathbf{A}}$ has $n-1$ linearly independent columns and, therefore, has $n-1$ linearly independent rows. □

With our construction of a tree solution, the columns of **B** are the columns of $\bar{\mathbf{A}}$ corresponding to the variables f_{ij}, for $(i,j) \in T$, and are linearly independent. In general linear programming terminology, **B** is a

basis matrix. Since the remaining variables f_{ij}, $(i,j) \notin T$, are set to zero, the resulting flow vector **f** is the basic solution corresponding to this basis. Thus, a tree solution is a basic solution, and a feasible tree solution is a basic feasible solution. In fact, the converse is also true.

Theorem 7.4 *A flow vector is a basic solution if and only if it is a tree solution.*

Proof. We have already argued that a tree solution is a basic solution. Suppose now that a flow vector **f** is not a tree solution. We will show that it is not a basic solution. Note that if $\mathbf{Af} \neq \mathbf{b}$, then **f** is not a basic solution, by definition. Thus, we only need to consider the case where $\mathbf{Af} = \mathbf{b}$.

Let $S = \{(i,j) \in A \mid f_{ij} \neq 0\}$. If the arcs in the set S do not form a cycle, then there exists a set T of $n-1$ arcs such that $S \subset T$, and such that the arcs in T form a tree [cf. Assumption 7.1(b) and Theorem 7.2]. Since $f_{ij} = 0$ for all $(i,j) \notin T$, the flow vector **f** is the tree solution associated with T, which is a contradiction.

Let us now assume that the set S contains a cycle C and let $\mathbf{h}^C$ be the simple circulation associated with C. Consider the flow vector $\mathbf{f} + \mathbf{h}^C$. We have $\mathbf{Af} = \mathbf{b}$ and $\mathbf{Ah}^C = \mathbf{0}$, which implies that $\mathbf{A}(\mathbf{f} + \mathbf{h}^C) = \mathbf{b}$. Furthermore, whenever $f_{ij} = 0$ the arc (i,j) does not belong to the cycle C, and we have $h_{ij}^C = 0$. We see that all constraints that are active at the vector **f** are also active at the vector $\mathbf{f} + \mathbf{h}^C$. Thus, the constraints that are active at **f** do not have a unique solution, and **f** is not a basic solution (cf. Theorem 2.2 and Definition 2.9 in Section 2.2). See Figure 7.12 for an illustration. □
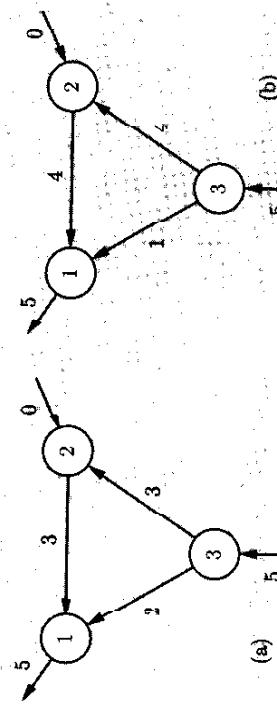
**Figure 7.12:** (a) Part of a flow vector that satisfies $\mathbf{Af} = \mathbf{b}$. This flow vector is not a tree solution because the arcs $(2,1), (3,1)$, and $(3,2)$ form a cycle C and carry nonzero flow. (b) The flow vector $\mathbf{f} + \mathbf{h}^C$. Active constraints (arcs that carry zero flow) under **f** remain active under $\mathbf{f} + \mathbf{h}^C$.

ing:

To summarize our conclusions so far, we have established the follow-

(a) Basic (feasible) solutions are (feasible) tree solutions and vice versa.

(b) Every basis matrix is triangular when its rows and columns are suit-ably reordered.

(c) Given a basis matrix $\mathbf{B}$, the vector of basic variables $\mathbf{B}^{-1}\mathbf{b}$ can be easily computed, without the need to maintain $\mathbf{B}^{-1}$ in a tableau.

As in the case of general linear programming problems, a basic feasible solution can be degenerate. This happens if the flow on some arc $(i,j) \in T$ turns out to be 0. In this case, the same basic feasible solution may correspond to several trees. For example, the tree shown in Figure 7.10 leads to a degenerate basic feasible solution, because $f_{61} = 0$. A different tree that would yield the same basic feasible solution is obtained by replacing arc $(6,7)$ by arc $(5,7)$.

Change of basis

We will now develop the mechanics of a change of basis. Recall that in a general linear programming problem, we first choose a nonbasic variable that enters the basis, find how to adjust the basic variables in order to maintain the equality constraints, and increase the value of the entering variable until one of the old basic variables is about to become negative. We specialize this procedure to the network case. Picking a nonbasic variable is the same as choosing an arc (i,j) that does not belong to T. Then, the arc (i,j) together with some of the arcs in T form a cycle. Let us choose the orientation of the cycle so that (i,j) is a forward arc. Let F and B be the sets of forward and backward arcs in the cycle, respectively. If we are to increase the value of the nonbasic variable f_{ij} to some θ, the old basic variables need to be adjusted in order not to violate the flow conservation constraints. This can be accomplished by pushing θ units of flow around the cycle. More precisely, $f_{k\ell}$ is increased (decreased) by θ for all forward (backward) arcs of the cycle. The new flow variables $\hat{f}_{k\ell}$ are given by

$$\hat{f}_{k\ell} = \begin{cases} f_{k\ell} + \theta, & \text{if } (k,\ell) \in F, \\ f_{k\ell} - \theta, & \text{if } (k,\ell) \in B, \\ f_{k\ell}, & \text{otherwise.} \end{cases} \tag{7.4}$$

We set θ as large as possible, provided that all arc flows remain nonnegative. It is clear that the largest possible value of θ is equal to

$$\theta^* = \min_{(k,\ell) \in B} f_{k\ell}, \tag{7.5}$$

except if B is empty, in which case we let $\theta^* = \infty$. A variable $f_{k\ell}$ that attains the minimum in Eq. (7.5) is set to zero and exits the basis. If $f_{k\ell} = 0$ for some arc $(k,\ell) \in B$ (which can happen if we start with a

degenerate basic feasible solution), then the change of basis occurs without any change of the arc flows. (For the example shown in Figure 7.10, if f_{57} enters the basis, f_{67} exits the basis and $\theta^* = 0$.)

Calculation of the cost change

The cost change resulting from the above described change of basis, is equal to

$$\theta^* \cdot \left(\sum_{(k,\ell) \in F} c_{k\ell} - \sum_{(k,\ell) \in B} c_{k\ell} \right). \tag{7.6}$$

Naturally, the variable f_{ij} should enter the basis only if the value of the expression (7.6) is negative.

From the development of the simplex method for general linear pro-gramming problems, we know that if the variable that enters the basis takes the value θ^*, then the cost changes by θ^* times the reduced cost of the en-tering variable. Comparing with Eq. (7.6), we see that the reduced cost $\bar{c}_{ij}$ of a nonbasic variable f_{ij} is given by

$$\bar{c}_{ij} = \sum_{(k,\ell) \in F} c_{k\ell} - \sum_{(k,\ell) \in B} c_{k\ell}, \tag{7.7}$$

which is simply the cost of the cycle around which flow is being pushed.

We will now derive an alternative formula for the reduced costs that allows for more efficient computation. Recall the general formula $\bar{\mathbf{c}}' = \mathbf{c}' - \mathbf{p}'\tilde{\mathbf{A}}$ for determining the reduced costs. where $\mathbf{p}$ is the dual vector given by $\mathbf{p}' = \mathbf{c}'_B\mathbf{B}^{-1}$, $\mathbf{B}$ is the current basis matrix, and $\mathbf{c}_B$ is the vector with the costs of the basic variables. The dimension of $\mathbf{p}$ is equal to the number of rows of $\tilde{\mathbf{A}}$, which is $n - 1$, and we have one dual variable p_i associated with each node $i \neq n$. Suppose that (i,j) is the kth arc of the graph. Then, the kth entry of the vectors $\bar{\mathbf{c}}$ and $\mathbf{c}$ is equal to $\bar{c}_{ij}$ and c_{ij}, respectively. The kth entry of $\mathbf{p}'\tilde{\mathbf{A}}$ is equal to the inner product of $\mathbf{p}$ with the kth column of $\tilde{\mathbf{A}}$. From the definition of the node-arc incidence matrix, the kth column of $\tilde{\mathbf{A}}$ has an entry equal to 1 at the ith row (if $i < n$), and an entry equal to -1 at the jth row (if $j < n$). We conclude that

$$\bar{c}_{ij} = \begin{cases} c_{ij} - (p_i - p_j), & \text{if } i, j \neq n, \\ c_{ij} - p_i, & \text{if } j = n, \\ c_{ij} + p_j, & \text{if } i = n. \end{cases} \tag{7.8}$$

Equation (7.8) can be written more concisely if we define $p_n = 0$, in which case we have

$$\bar{c}_{ij} = c_{ij} - (p_i - p_j), \quad \forall (i,j) \in \mathcal{A}. \tag{7.9}$$

It remains to compute the dual vector $\mathbf{p}' = \mathbf{c}'_B\mathbf{B}^{-1}$ associated with the current basis. Since the reduced cost of every basic variable must be

equal to zero, Eq. (7.9) yields

$$p_i - p_j = c_{ij}, \qquad \forall (i,j) \in T,$$
$$p_n = 0. \qquad (7.10)$$

The system of equations (7.10) is easily solved using the following procedure. We view node n as the root of the tree and set $p_n = 0$. We then go down the tree, proceeding from the root towards the leaves, with a new component of p being evaluated at each step; see Figure 7.13.

Overview of the algorithm

We start with a summary of the network simplex algorithm and then proceed to discuss some issues related to initialization and termination.

The simplex method for uncapacitated network flow problems

1. A typical iteration starts with a basic feasible solution f associated with a tree T.

2. To compute the dual vector p, solve the system of equations (7.10), by proceeding from the root towards the leaves.

3. Compute the reduced costs $\bar{c}_{ij} = c_{ij} - (p_i - p_j)$ of all arcs $(i,j) \notin T$. If they are all nonnegative, the current basic feasible solution is optimal and the algorithm terminates; else, choose some (i,j) with $\bar{c}_{ij} < 0$ to be brought into the basis.

4. The entering arc (i,j) and the arcs in T form a unique cycle. If all arcs in the cycle are oriented the same way as (i,j), then the optimal cost is $-\infty$ and the algorithm terminates.

5. Let B be the set of arcs in the cycle that are oriented in the opposite direction from (i,j). Let $\theta^* = \min_{(k,\ell) \in B} f_{k\ell}$, and push θ^* units of flow around the cycle. A new flow vector is determined according to Eq. (7.4). Remove from the basis one of the old basic variables whose new value is equal to zero.

In the case where finding an initial basic feasible solution is difficult, we may need to form and solve an auxiliary problem. For example, for each pair of source and sink nodes, we may introduce an auxiliary arc; finding a basic feasible solution in the presence of these arcs is straightforward. Furthermore, if the unit costs c_{ij} of the auxiliary arcs are chosen large enough, solving the auxiliary problem is equivalent to solving the original problem.

The network simplex algorithm is similar to the naive implementation described in Section 3.3. Because of the special structure of the basis matrix B, the system $c_B' = p'B$ can be solved on the fly, without the need to maintain a simplex tableau or the inverse basis matrix B^{-1}. For a rough
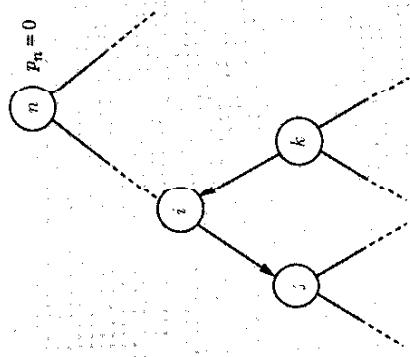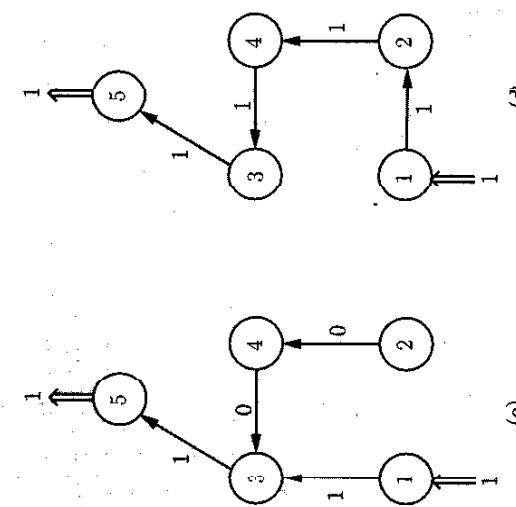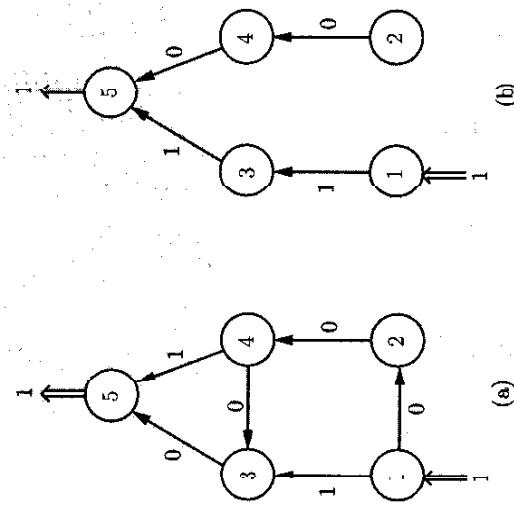

Figure 7.13: Once p_i is computed, p_j and p_k can also be computed, because we have $p_i - p_j = c_{ij}$ and $p_k - p_i = c_{ki}$. Starting from the root and continuing in this fashion, all dual variables can be computed.

count of the computational requirements of each iteration, we need $O(n)$ computations to evaluate the dual vector p, $O(m)$ computations to evaluate all of the reduced costs, and another $O(n)$ computations to effect the change of basis. Given that $m \geq n - 1$, the total is $O(m)$, which compares favorably with the $O(mn)$ computational requirements of an iteration of the simplex method for general linear programming problems. In practice, the running time of the network simplex algorithm is improved further by using a somewhat more clever way of updating the dual variables, and by using suitable data structures to organize the computation.

All of the theory in Chapters 3 and 4 applies to the network simplex method. In particular, in the absence of degeneracy, the algorithm is guaranteed to terminate after a finite number of steps. In the presence of degeneracy, the algorithm may cycle. Cycling can be avoided by using either a general purpose anticycling rule or special methods. If the optimal cost is $-\infty$, the algorithm terminates with a negative cost directed cycle. (The simple circulation h^C associated with that cycle is an extreme ray of the feasible set, and $c'h^C < 0$.) If the optimal cost is finite, the algorithm terminates with an optimal flow vector f and an optimal dual vector p. In practice, the number of iterations is often $O(m)$, but there exist examples involving an exponential number of basis changes.

Example 7.3 Consider the uncapacitated network problem shown in Figure 7.14(a); the numbers next to each arc are the corresponding costs. Figure 7.14(b) shows a tree and a corresponding feasible tree solution. Arc $(4, 3)$ forms a cycle consisting of nodes 4, 3, and 5. The reduced cost $\bar{c}_{43}$ of f_{43} is equal to the cost of

that cycle which is $c_{43} + c_{35} - c_{45} = -1$. We let arc $(4,3)$ enter the tree. Pushing flow along the cycle attempts to reduce the flow along the arc $(4,5)$. Since this was zero to start with (degeneracy), we have $\theta^* = 0$; the arc $(4,5)$ leaves the tree and we obtain the feasible tree solution indicated in Figure 7.14(c). The reduced cost associated with arc $(1,2)$ is $c_{12} + c_{24} + c_{43} - c_{13} = -1$, and we let that arc enter the tree. We can push up to one unit of flow along the cycle 1,2,4,3,1, that is, until the flow along arc $(1,3)$ is set to zero. Thus, $\theta^* = 1$, the arc $(1,3)$ leaves the tree, and we obtain the feasible tree solution indicated in Figure 7.14(d). It is not hard to verify that all reduced costs are nonnegative and we have an optimal solution.

Integrality of optimal solutions

An important feature of network flow problems is that when the problem data are integer, most quantities of interest are also integer and the simplex method can be implemented using integer (as opposed to floating point) arithmetic. This allows for faster computation and, equally important, the issues of finite precision and truncation error disappear. The theorem that follows provides a summary of integrality properties.

Theorem 7.5 *Consider an uncapacitated network flow problem and assume that the underlying graph is connected.*

(a) *For every basis matrix $\mathbf{B}$, the matrix $\mathbf{B}^{-1}$ has integer entries.*

(b) *If the supplies b_i are integer, then every basic solution has integer coordinates.*

(c) *If the cost coefficients c_{ij} are integer, then every dual basic solution has integer coordinates.*

Proof.

(a) As shown in the proof of Theorem 7.3, we can reorder the rows and columns of a basis matrix $\mathbf{B}$ so that it becomes lower triangular and its diagonal entries are either 1 or -1. Therefore, the determinant of $\mathbf{B}$ is equal to 1 or -1. By Cramer's rule, $\mathbf{B}^{-1}$ has integer entries.

(b) This follows by inspecting the nature of the algorithm that determines the values of the basic variables (see the proof of Theorem 7.3), or from the formula $\mathbf{f}_T = \mathbf{B}^{-1}\bar{\mathbf{b}}$.

(c) This follows by inspecting the nature of the algorithm that determines the values of the dual variables, or from the formula $\mathbf{p}' = \mathbf{c}_B'\mathbf{B}^{-1}$. $\quad\square$

We now have the following important corollary of Theorem 7.5.

Figure 7.14: (a) An uncapacitated network flow problem. Arc costs are indicated next to each arc. (b) An initial feasible tree solution. The arc flows are indicated next to each arc. (c)-(d) Feasible tree solutions obtained after the first and the second change of basis, respectively.

> **Corollary 7.2** *Consider an uncapacitated network flow problem, and assume that the optimal cost is finite.*
>
> (a) *If all supplies b_i are integer, there exists an integer optimal flow vector.*
>
> (b) *If all cost coefficients c_{ij} are integer, there exists an integer optimal solution to the dual problem.*

The simplex method for capacitated problems

We will now generalize the simplex method to the case where some of the arc capacities are finite and we have constraints of the form

$$d_{ij} \le f_{ij} \le u_{ij}, \qquad (i,j) \in A.$$

There are only some minor differences from the discussion earlier in this section. For this reason, our development will be less formal.

Consider a set $T \subset A$ of $n-1$ arcs that form a tree when their direction is ignored. We partition the remaining arcs into two disjoint subsets D and U. We let $f_{ij} = d_{ij}$ for every $(i,j) \in D$, $f_{ij} = u_{ij}$ for every $(i,j) \in U$, and then solve the flow conservation equations for the remaining variables f_{ij}, $(i,j) \in T$. The resulting flow vector is easily shown to be a basic solution, and all basic solutions can be obtained in this manner; the argument is similar to the proofs of Theorems 7.3 and 7.4.

Given a basic feasible solution associated with the sets T, D, and U, we evaluate the vector of reduced costs using the same formulae as before, and then examine the arcs outside T. If we find an arc $(i,j) \in D$ whose reduced cost is negative, we push as much flow as possible around the cycle created by that arc. (This is the same as in our previous development.) Alternatively, if we can find an arc $(i,j) \in U$ with positive reduced cost, we push as much flow as possible around the cycle created by that arc, but in the opposite direction. In either case, we are dealing with a direction of cost decrease. Determining how much flow can be pushed is done as follows. Let F be the set of arcs whose reduced cost is to increase due to the contemplated flow push; let B be the set of arcs whose flow is to decrease. Then, the flow increment is limited by θ^*, defined as follows

$$\theta^* = \min\left\{ \min_{(k,\ell)\in B} \{f_{k\ell} - d_{k\ell}\}, \; \min_{(k,\ell)\in F} \{u_{k\ell} - f_{k\ell}\} \right\}. \qquad (7.11)$$

By pushing θ^* units of flow around the cycle, there will be at least one arc (k,ℓ) whose flow is set to either $d_{k\ell}$ or $u_{k\ell}$. If the arc (k,ℓ) belongs to T, it is removed from the tree and is replaced by (i,j). The other possibility is that $(k,\ell) = (i,j)$. (For example, pushing flow around the cycle may result in f_{ij} being reduced from u_{ij} to d_{ij}.) In that case, the set T remains the

same, but (i,j) is moved from U to D, or vice versa. In any case, we obtain a new basic feasible solution. (In the presence of degeneracy, it is possible that the new basic feasible solution coincides with the old one, and only the sets T, D, or U change.) To summarize, the network simplex algorithm for capacitated problems is as follows.

> **The simplex method for capacitated network flow problems**
>
> 1. A typical iteration starts with a basic feasible solution f associated with a tree T, and a partition of the remaining arcs into two sets D, U, such that $f_{ij} = d_{ij}$ for $(i,j) \in D$, and $f_{ij} = u_{ij}$ for $(i,j) \in U$.
>
> 2. Solve the system of equations (7.10) for $p_1, \ldots, p_n$, by proceeding from the root towards the leaves.
>
> 3. Compute the reduced costs $\bar{c}_{ij} = c_{ij} - (p_i - p_j)$ of all arcs $(i,j) \notin T$. If $\bar{c}_{ij} \ge 0$ for all $(i,j) \in D$, and $\bar{c}_{ij} \le 0$ for all $(i,j) \in U$, the current basic feasible solution is optimal and the algorithm terminates.
>
> 4. Let (i,j) be an arc such that $\bar{c}_{ij} < 0$ and $(i,j) \in D$, or such that $\bar{c}_{ij} > 0$ and $(i,j) \in U$. This arc (i,j) together with the tree T forms a unique cycle. Choose the orientation of the cycle as follows. If $(i,j) \in D$, then (i,j) should be a forward arc. If $(i,j) \in U$, then (i,j) should be a backward arc.
>
> 5. Let F and B be the forward and backward arcs, respectively, in the cycle. Determine θ^* according to Eq. (7.11). Compute a new flow vector, with components $\hat{f}_{k\ell}$, by letting
>
> $$\hat{f}_{k\ell} = \begin{cases} f_{k\ell} - \theta^*, & \text{if } (k,\ell) \in F, \\ f_{k\ell} - \theta^*, & \text{if } (k,\ell) \in B, \\ f_{k\ell}, & \text{otherwise.} \end{cases}$$
>
> Finally, update the sets T, D, U.

7.4 The negative cost cycle algorithm

The network simplex algorithm incorporates a basic idea, which is present in practically every primal method for network flow problems: given a current primal feasible solution, find an improved one by identifying a negative cost cycle along which flow can be pushed. One advantage of the simplex method is that it searches for negative cost cycles using a streamlined and efficient mechanism. A potential disadvantage is that a change of basis can be degenerate, with no flow being pushed, and without any cost improvement.

In this section, we present a related, but different, algorithm, where every iteration aims at a nonzero cost improvement. In particular, at every

$f_{12}=0$
$u_{12}=2$
$c_{12}=3$

$f_{32}=3$
$u_{32}=3$
$c_{32}=2$

$f_{14}=2$
$u_{14}=3$
$c_{14}=5$

$f_{34}=2$
$u_{34}=4$
$c_{34}=3$

C

(a)

$f_{12}=\delta$

$f_{32}=3-\delta$

$f_{14}=2-\delta$

$f_{34}=2+\delta$

(b)
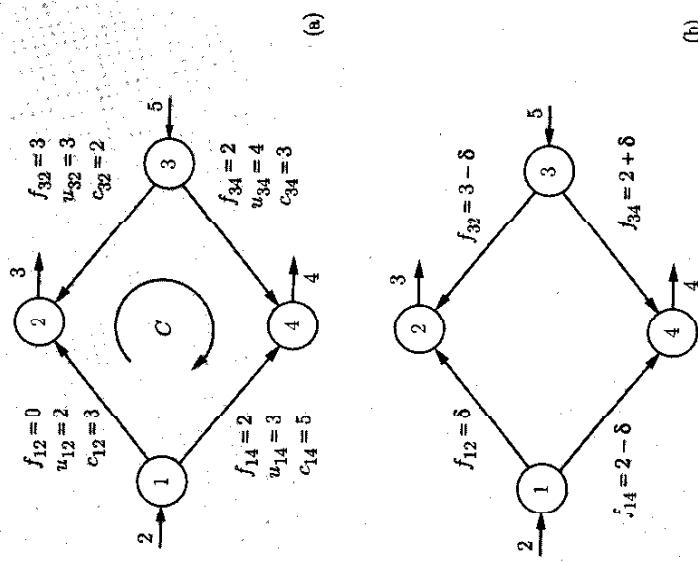
Figure 7.15: (a) A portion of a network, together with the values of some of the flow variables. (b) The new arc flows after pushing t units of flow around the cycle C.

iteration we push some flow around a negative cost cycle. The algorithm terminates when no profitable cycle can be identified. The method is justified by a key result that relates the absence of profitable cycles with optimality.

Motivation

Consider the portion of a network shown in Figure 7.15(a). Could the flow vector f given in the figure be optimal? The answer is no, for the following reason. Suppose that we push δ units of flow along the indicated cycle, where δ is a positive scalar. Taking into account the direction of the arcs, the new flow variables take the values indicated in Figure 7.15(b). In particular, the flow on every forward arc is increased by δ and the flow on every backward arc is reduced by δ. Flow conservation is preserved, and as long as $\delta \le 2$, the constraints $0 \le f_{ij} \le u_{ij}$ are respected, and the new

flow is feasible. The charge in costs is

$$c_{12}\delta + c_{32}(-\delta) + c_{34}\delta + c_{14}(-\delta) = -\delta,$$

which is negative, and f cannot be optimal. As this example illustrates, a flow f can be improved if we can identify a cycle along which flow can be profitably pushed.

Description of the algorithm

In this subsection, we present the algorithm of interest after developing some of its elements. We assume that we have a network described by a directed graph $G = (N, A)$, supplies b, arc capacities u_{ij}, and cost coefficients c_{ij}. Let C be a cycle, and let F and B be the sets of forward and backward arcs of the cycle, respectively. Let $\mathbf{h}^C$ be the simple circulation associated with this cycle, that is,

$$h_{ij}^C = \begin{cases} 1, & \text{if } (i,j) \in F, \\ -1, & \text{if } (i,j) \in B, \\ 0, & \text{otherwise.} \end{cases}$$

Let **f** be a feasible flow vector and let δ be a nonnegative scalar. If we charge **f** to $\mathbf{f} + \delta\mathbf{h}^C$, we say that we are *pushing δ units of flow along the cycle C*. Since **f** is feasible, we have $\mathbf{Af} = \mathbf{b}$; since $\mathbf{Ah}^C = \mathbf{0}$, we obtain $\mathbf{A}(\mathbf{f} + \delta\mathbf{h}^C) = \mathbf{b}$, and the flow conservation constraint is still satisfied. In order to maintain feasibility, we also need

$$0 \le f_{ij} + \delta h_{ij}^C \le u_{ij},$$

that is,

$$0 \le f_{ij} + \delta \le u_{ij}, \quad \text{if } (i,j) \in F,$$
$$0 \le f_{ij} - \delta \le u_{ij}, \quad \text{if } (i,j) \in B.$$

Since $\delta \ge 0$ and $0 \le f_{ij} \le u_{ij}$, this is equivalent to

$$\delta \le u_{ij} - f_{ij}, \quad \text{if } (i,j) \in F,$$
$$\delta \le f_{ij}, \quad \text{if } (i,j) \in B.$$

Thus, the maximum amount of flow that can be pushed along the cycle, which we denote by $\delta(C)$, is given by

$$\delta(C) = \min \left\{ \min_{(i,j)\in F}(u_{ij} - f_{ij}), \ \min_{(i,j)\in B} f_{ij} \right\}. \quad (7.12)$$

If the set B is empty and if $u_{ij} = \infty$ for every arc in the cycle, then there are no restrictions on δ, and we set $\delta(C) = \infty$. If $f_{ij} < u_{ij}$ for all forward arcs and $f_{ij} > 0$ for all backward arcs, then $\delta(C) > 0$, and we say that the

cycle is *unsaturated*. For the cycle considered in Figure 7.15(a), we have $\delta(C) = 2$.

We now calculate the cost change when we push a unit of flow along a cycle C. Using the definition of $\mathbf{h}^C$, the cost change is

$$c'\mathbf{h}^C = \sum_{(i,j)\in F} c_{ij} - \sum_{(i,j)\in B} c_{ij},$$

the *cost of cycle C*.

We can now propose an algorithm which at each iteration looks for a negative cost unsaturated cycle and pushes as much flow as possible along that cycle.

Negative cost cycle algorithm

1. Start with a feasible flow **f**.
2. Search for an unsaturated cycle with negative cost.
3. If no unsaturated cycle with negative cost can be found, the algorithm terminates.
4. If a negative cost unsaturated cycle C is found, then:
   (a) If $\delta(C) < \infty$, construct the new feasible flow $\mathbf{f} + \delta(C)\mathbf{h}^C$, and go to Step 2.
   (b) If $\delta(C) = \infty$, the algorithm terminates and the optimal cost is $-\infty$.

There are a few different issues that need to be discussed:

(a) How do we start the algorithm?

(b) How do we search for an unsaturated cycle with negative cost?

(c) If the algorithm terminates, does it provide us with an optimal solution?

(d) Is the algorithm guaranteed to terminate?

These issues are addressed, one at a time, in the subsections that follow.

Starting the algorithm

As discussed in Section 7.2, every network flow problem can be converted into an equivalent problem with no sources or sinks. For the latter problem, the zero flow is a feasible solution that provides a starting point. As an alternative, a feasible flow (if one exists) can be constructed by solving a suitable maximum flow problem (Exercise 7.21).

The residual network

Suppose that we have a network $G = (\mathcal{N}, \mathcal{A})$ and a feasible flow $\mathbf{f}$. The *residual network* is an auxiliary network $\tilde{G} = (\mathcal{N}, \tilde{\mathcal{A}})$ with the same set of nodes, but with different arcs and arc capacities. It is a convenient device to keep track of the amount of flow that can be pushed along the arcs of the original network.

Consider an arc (i,j) with capacity u_{ij}, and let f_{ij} be the current flow through that arc. Then, f_{ij} can be increased by up to $u_{ij} - f_{ij}$, or can be decreased by up to f_{ij}. We represent these options in the residual network by introducing an arc (i,j) with capacity $u_{ij} - f_{ij}$, and an arc (j,i), with capacity f_{ij}. Any flow on the arc (i,j) in the residual network is to be interpreted as a corresponding reduction of the flow on the arc (i,j) of the original network.

We assign costs to the arcs of the residual network in a way that reflects the cost changes in the original network. In particular, we associate a cost of c_{ij} with the arc (i,j) of the residual network, and a cost of $-c_{ij}$ with the arc (j,i) of the residual network. [This is because a unit of flow on the arc (j,i) corresponds to a unit reduction of the flow on the arc (i,j) of the original network, and a cost change of $-c_{ij}$.] All supplies in the residual network are set to zero, which implies that every feasible flow is a circulation. Finally, we delete those arcs of the residual network that have zero capacity.

The construction of the residual network is shown in Figure 7.16. As seen in the figure, the residual network may contain two arcs with the same start node and the same end node. In particular, the presence of two arcs from i to j indicates that we can push flow from i to j either by increasing the value of f_{ij} or by decreasing the value of f_{ji}. Strictly speaking, this violates our original definition of a graph, but this turns out not to be a problem.

Let f be a feasible flow in the original network and let $\mathbf{f} + \bar{\mathbf{f}}$ be another feasible flow in the original network. The flow increment $\bar{\mathbf{f}}$ can be associated with a flow vector $\tilde{\mathbf{f}}$ in the residual network as follows.

(a) If $\bar{f}_{ij} > 0$, we let the flow $\tilde{f}_{ij}$ on the corresponding arc (i,j) in the residual network be equal to $\bar{f}_{ij}$. Feasibility in the original network implies that $\bar{f}_{ij} \le u_{ij} - f_{ij}$, and $\tilde{f}_{ij}$ satisfies the capacity constraint in the residual network.

(b) If $\bar{f}_{ij} < 0$, we let the flow $\tilde{f}_{ji}$ on the corresponding arc (j,i) in the residual network be equal to $-\bar{f}_{ij}$. Feasibility in the original network implies that $-\bar{f}_{ij} \le f_{ij}$ and therefore $\tilde{f}_{ji}$ satisfies the capacity constraint in the residual network.

All variables $\tilde{f}_{ij}$ that are not set by either (a) or (b) above are left at zero value. See Figure 7.17 for an illustration.
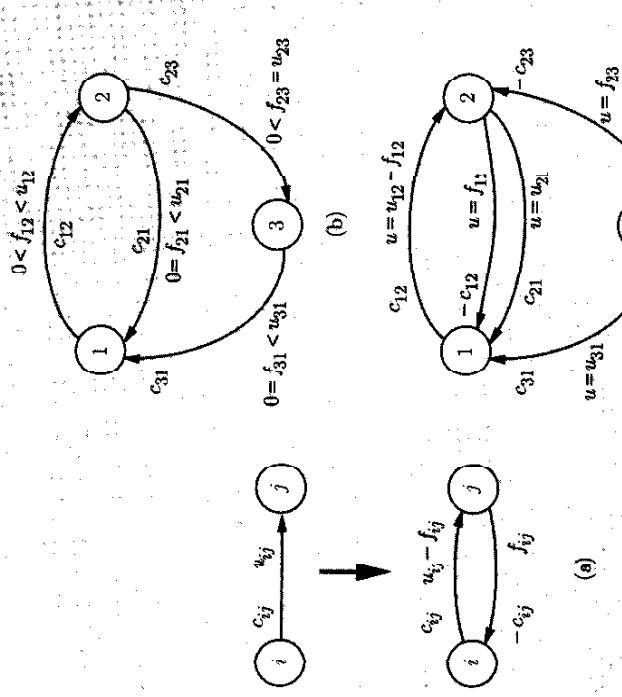
Figure 7.16: (a) Each arc of the original network leads to two arcs in the residual network. (b) A network and an associated feasible flow. (c) The corresponding residual network. Note that zero capacity arcs have been deleted.

We make the following observations:

(a) We have $\tilde{f}_{ij} \geq 0$ for all arcs in the residual network.

(b) The flow $\tilde{f}$ in the residual network is a circulation. This is because in the original network, we have $Af = b = A(f + \bar{f})$. Hence, $A\tilde{f} = 0$, which means that with the flow vector $\tilde{f}$, the net flow into any node i is zero. Because of the way $\tilde{f}$ was constructed, the net flow into any node of the residual network must also be zero.

(c) The cost of $\tilde{f}$ in the residual network is equal to $\sum_{(i,j)} c_{ij}\tilde{f}_{ij}$, which is the cost of $\bar{f}$ in the original network. This is because for each arc with $\tilde{f}_{ij} > 0$, we have an equal flow $\tilde{f}_{ij}$ in a corresponding arc (i,j) in the residual network, and the latter arc has unit cost c_{ij}. Furthermore, for each arc with $\tilde{f}_{ij} < 0$ in the original network, we have a flow $\tilde{f}_{ji} = -\tilde{f}_{ij}$ in a corresponding arc (j,i) in the residual network, and
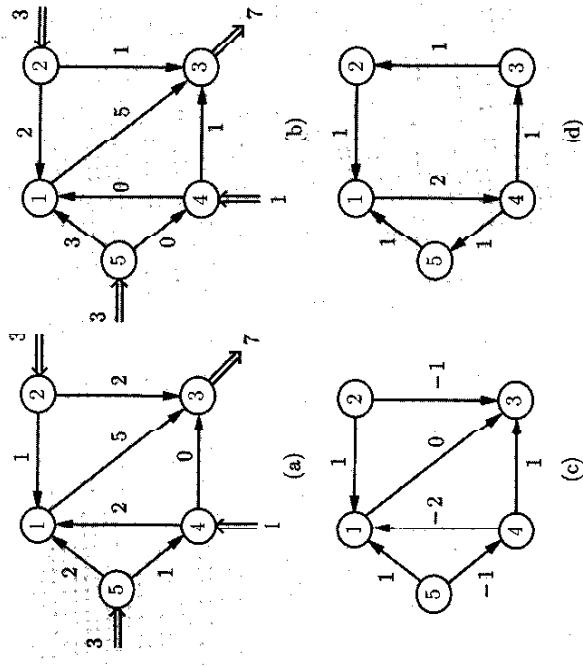
Figure 7.17: In this figure, the numbers next to each arc indicate arc flows. (a) A feasible flow **f** in a network. (b) Another feasible flow $\mathbf{f} + \bar{\mathbf{f}}$. (c) The flow increment $\bar{\mathbf{f}}$. Note that it is a circulation. (d) The flow $\tilde{\mathbf{f}}$ in the residual network (only arcs with nonzero flows are shown).

the latter arc has unit cost $-c_{ij}$. Since $(-c_{ij})\tilde{f}_{ji} = c_{ij}\tilde{f}_{ij}$, we see that $\tilde{f}_{ij}$ and $\tilde{f}_{ji}$ incur the same cost.

The preceding arguments can be reversed. That is, if we start with a feasible circulation $\tilde{\mathbf{f}}$ in the residual network, we can construct a circulation $\bar{\mathbf{f}}$ in the original network such that $\mathbf{f} + \bar{\mathbf{f}}$ is feasible and such that $\mathbf{c}'\bar{\mathbf{f}}$ is equal to the cost of $\tilde{\mathbf{f}}$ in the residual network.

We finally note that every unsaturated cycle in the original network corresponds to a directed cycle in the residual network in which all arcs have positive capacity and vice versa. Furthermore, the costs of these cycles in their respective networks are equal. We conclude that the search for negative cost unsaturated cycles in the original network can be accomplished by searching for a negative cost directed cycle in the residual network. In Section 7.9, we show that the problem of finding negative cost directed cycles in a graph can be solved in time $O(n^3)$; hence, the computational requirements of each iteration of the negative cost cycle algorithm are also $O(n^3)$.

Optimality conditions

We now investigate what happens at termination of the negative cost cycle algorithm. If the algorithm terminates because it discovered a negative cost cycle with $\delta(C) = \infty$, then the optimal cost is $-\infty$. In particular, the flow $\mathbf{f} + \delta \mathbf{h}^C$ is feasible for every $\delta > 0$, and by letting δ become arbitrarily large, the cost of such feasible solutions is unbounded below.

The algorithm may also terminate because no unsaturated negative cost cycle can be found. In that case, we have an optimal solution, as shown by the next result.

> **Theorem 7.6** *A feasible flow* $\mathbf{f}$ *is optimal if and only if there is no unsaturated cycle with negative cost.*

Proof. One direction is easy. If C is an unsaturated cycle with negative cost, then $\mathbf{f} + \delta(C) \mathbf{h}^C$ is a feasible flow whose cost is less than the cost of $\mathbf{f}$, and so $\mathbf{f}$ is not optimal.

For the converse, we argue by contradiction. Suppose that $\mathbf{f}$ is a feasible flow that is not optimal. Then, there exists another feasible flow $\mathbf{f} + \bar{\mathbf{f}}$ whose cost is less, and in particular, $\mathbf{c}'\bar{\mathbf{f}} < 0$. As discussed in the preceding subsection, it follows that there exists a feasible (in particular, nonnegative) circulation $\tilde{\mathbf{f}}$ in the residual network whose cost is negative. To prove that this circulation implies the existence of a negative cost directed cycle in the residual network, we need the following important result.

> **Lemma 7.1 (Flow decomposition theorem)** *Let* $\mathbf{f} \geq \mathbf{0}$ *be a nonzero circulation. Then, there exist simple circulations* $\mathbf{f}^1, \ldots, \mathbf{f}^k$, *involving only forward arcs, and positive scalars* $a_1, \ldots, a_k$, *such that*
> $$\mathbf{f} = \sum_{i=1}^{k} a_i \mathbf{f}^i.$$
> *Furthermore, if* $\mathbf{f}$ *is an integer vector, then each* a_i *can be chosen to be an integer.*

Proof. (See Figure 7.18 for an illustration.) If $\mathbf{f}$ is the zero vector, the result is trivially true, with $k = 0$. Suppose that $\mathbf{f}$ is nonzero. Then, there exists some arc (i, j) for which $f_{ij} > 0$. Let us traverse arc (i, j). Because of flow conservation at node j, there exists some arc (j, k) for which $f_{jk} > 0$. We then traverse arc (j, k) and keep repeating the same process. Since there are finitely many nodes, some node will be eventually visited for a second time. At that point, we have found a directed cycle with each arc in the cycle carrying a positive amount of flow. Let $\mathbf{f}^1$ be the simple circulation
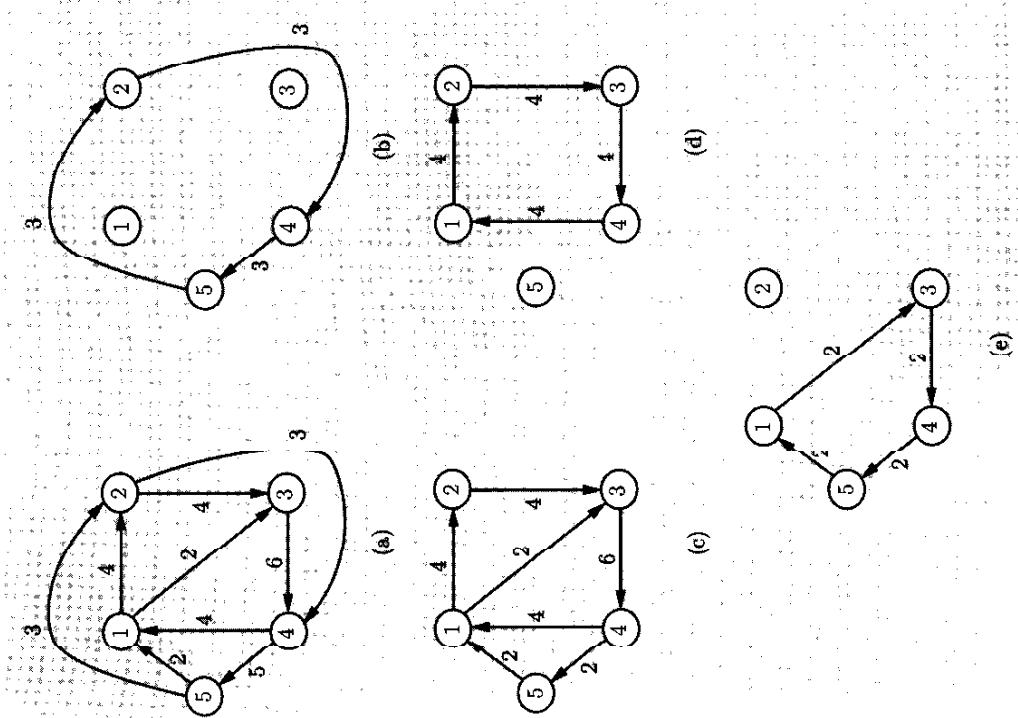

Figure 7.18: Illustration of the flow decomposition theorem. The numbers next to each arc indicate the value of the corresponding arc flows. Arcs with zero flow are not shown. (a) A nonnegative circulation $\mathbf{f}$. (b) The circulation $a_1 \mathbf{f}^1$. (c) The remaining flow $\mathbf{f} - a_1 \mathbf{f}^1$. (d) The circulation $a_2 \mathbf{f}^2$. (e) The remaining flow $\mathbf{f} - a_1 \mathbf{f}^1 - a_2 \mathbf{f}^2$ is a simple circulation and we let $a_3 \mathbf{f}^3$ be equal to it.

corresponding to that cycle. Let a_1 be the minimum value of f_{ij}, where the minimum is taken over all arcs in the cycle, and consider the vector $\hat{f} = f - a_1 f^1$. This vector is nonnegative because of the way that a_1 was chosen. In addition, we have $A\hat{f} = 0$ and $Af^1 = 0$, which implies that $A\hat{f} = 0$ and $\hat{f}$ is a circulation. By the definition of a_1, there exists some arc (k,ℓ) on the cycle for which $f_{k\ell} = a_1$ and $\hat{f}_{k\ell} = 0$. Therefore, the number of positive components of $\hat{f}$ is smaller than the number of positive components of f. We can now apply the same procedure to $\hat{f}$, to obtain a new simple circulation f^2, and continue similarly. Each time, the number of arcs that carry positive flow is reduced by at least one. Thus, after repeating this procedure a finite number of times, we end up with the zero flow. When this happens, we have succeeded in decomposing f as a nonnegative linear combination of simple circulations. Furthermore, since all of the cycles constructed were directed, these simple circulations involve only forward arcs.

If f is integer, then a_1 is integer, and $\hat{f}$ is also an integer vector. It follows, by induction, that if we start with an integer flow vector f, all flows produced in the course of the above procedure are integer, and all coefficients a_i are also integer. This concludes the proof of Lemma 7.1. ☐

We now apply Lemma 7.1 to the residual network. The circulation $\tilde{f}$ can be decomposed in the form

$$\tilde{f} = \sum_i a_i \tilde{f}^i,$$

where each $\tilde{f}^i$ is a simple circulation involving only forward arcs, and each a_i is positive. Since $\tilde{f}$ has negative cost, at least one of the circulations $\tilde{f}^i$ must also have negative cost; hence, the residual network has a negative cost directed cycle. As discussed in the preceding subsection, this implies that the original network contains a negative cost unsaturated cycle, and the proof of Theorem 7.6 is now complete. ☐

Termination of the algorithm

Before concluding that the algorithm is correct, we need a guarantee that it will eventually terminate. This is the subject of our next theorem.

Theorem 7.7 *Suppose that all arc capacities u_{ij} are integer or infinite, and that the negative cost cycle algorithm is initialized with an integer feasible flow. Then, the arc flow variables remain integer throughout the algorithm and, if the optimal cost is finite, the algorithm terminates with an integer optimal solution.*

Proof. If the current flow f is integer, then $\delta(C)$ is integer or infinite, for every cycle C. Hence, the flow obtained after one iteration of the algorithm must also be integer, and integrality is preserved.

At each iteration, before the algorithm terminates, we have a cost reduction of $\delta(C)|c'h^C|$, where C is the negative cost cycle along which flow is pushed. Since $\delta(C) \geq 1$, this is no smaller than $v = \min_D |c'h^D|$, where the minimum is taken over all negative cost cycles D. Thus, each iteration of the algorithm reduces the cost by at least v, which is positive. It follows that if the optimal cost is finite, the algorithm must terminate after a finite number of iterations. ☐

Note that Theorem 7.7 establishes an integrality property of optimal solutions. This is the same conclusion that was reached in Corollary 7.2(a), for standard form problems.

Surprisingly, and unlike the simplex method, if the arc capacities are not integer, the algorithm is not guaranteed to terminate, even if the optimal cost is finite. One possibility is that the algorithm makes an infinite number of steps, each step results in lower costs, but the cost reductions become smaller and smaller, and the cost of the current flow does not converge to the optimal cost. It turns out that finite termination can be guaranteed under special rules for choosing between negative cost cycles. Two possible rules that are known to lead to finite termination are the following:

(a) **Largest improvement rule:** Choose a negative cost cycle for which the cost improvement $\delta(C) c'h^C$ is largest. Unfortunately, finding such a cycle is difficult. See Exercise 7.16 for an upper bound on the number of iterations.

(b) **Mean cost rule:** Choose a negative cost cycle for which $|c'h^C|/k(C)$ is largest, where $k(C)$ is the number of arcs in cycle C. It turns out that the search for such a cycle is not too difficult (Exercise 7.37).

When the optimal cost is $-\infty$, the algorithm may fail to terminate after a finite number of iterations, even if the arc capacities are integer. For this reason, one should verify that the optimal cost is finite before starting the algorithm; a simple criterion is developed in Exercise 7.17.

7.5 The maximum flow problem

In the maximum flow problem, we are given a directed graph $G = (N, A)$ and an arc capacity bound $u_{ij} \in [0, \infty]$ for each arc $(i,j) \in A$. Let s and t be two special nodes, called the source and sink node, respectively. The problem is to find the largest possible amount of flow that can be sent through the network, from s to t. We will see shortly that this is a special case of the general network flow problem. On the other hand, special purpose algorithms are possible, because of the simple structure of the problem. The

maximum flow problem arises in a variety of applications. Some are rather obvious (e.g., maximizing throughput in a logistics network), while others are less expected; see the example that follows.

Example 7.4 (Preemptive scheduling) We are given m identical machines and n jobs. Each job j must be processed for a total of p_j periods. (We assume that each p_j is an integer.) However, we allow *preemption*. That is, the processing of a job can be broken down and can be carried out by different machines in different periods. Each machine can only process one job at a time, and a job can only be processed by a single machine at a time. In addition, each job j is associated with a release time r_j and a deadline d_j: processing cannot start before period r_j, and must be completed before period d_j. Naturally, we assume that $r_j + p_j \le d_j$ for all jobs j. We wish to determine a schedule whereby all jobs are processed, without violating the release times and deadlines, or show that no such schedule exists.

We will now construct a maximum flow formulation of the problem. The first step is to rank all the release times and deadlines in ascending order. The resulting ordered list of numbers divides the time horizon into a number of nonoverlapping intervals. Let T_{kl} be the interval that starts in the beginning of period k and ends in the beginning of period l. Note that during each interval T_{kl}, the set of jobs that can be processed does not change. In particular, we can process any job j that has been released ($r_j \le k$) and its deadline has not yet been reached ($l \le d_j$). For a concrete example, suppose that we have four jobs with release times 3, 1, 3, 5, and deadlines 5, 4, 7, 9. The ascending list of release times and deadlines is 1, 3, 4, 5, 7, 9. We then obtain five intervals, namely, T_{13}, T_{34}, T_{45}, T_{57}, and T_{79}.

We construct a network involving a source node s, a sink node t, a node corresponding to each job j, and a node corresponding to each interval T_{kl}. The arcs and their capacities are as follows. For every job j, we have an arc (s, j), with capacity p_j. We interpret the flow along this arc as the number of periods of processing that job j receives. For every node T_{kl}, we introduce an arc (T_{kl}, t), with capacity $m(l - k)$. The flow along this arc represents the total number of machine-periods of processing during the interval T_{kl}. Finally, if a job j is available for processing during the interval T_{kl}, that is, if $r_j \le k \le l \le d_j$, we introduce an arc (j, T_{kl}), with capacity $l - k$. The flow along this arc represents the number of periods that job j is processed during this interval. See Figure 7.19 for an illustration. It is not hard to show that every feasible schedule corresponds to a flow through this network, with value $\sum_{j=1}^{n} p_j$, and conversely. Therefore, the scheduling problem can be solved by solving a maximum flow problem, and checking whether the resulting maximum flow value is equal to $\sum_{j=1}^{n} p_j$.

Mathematically, the maximum flow problem can be formulated as follows

$$\begin{array}{ll}
\text{maximize} & b_s \\
\text{subject to} & \mathbf{Af} = \mathbf{b} \\
& b_t = -b_s \\
& b_i = 0, & \forall\, i \ne s, t, \\
& 0 \le f_{ij} \le u_{ij}, & \forall\, (i,j) \in \mathcal{A}.
\end{array}$$
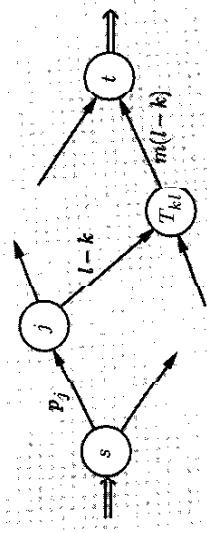
Figure 7.19: The structure of the network associated with the preemptive scheduling problem. The number next to each arc indicates its capacity. The arc from node j to node T_{kl} is present only if $r_j \le k \le l \le d_j$.

Note that, in contrast to the network flow problems considered earlier, b_s is a variable to be optimized. Any flow vector $\mathbf{f}$ satisfying the above constraints is called a feasible flow and the corresponding value of b_s is called the *value* of that flow.

The maximum flow problem can be reformulated as a network flow problem, as follows (see Figure 7.20 for an illustration). We let the cost of every arc be equal to zero and we introduce a new infinite capacity arc (t, s), with cost $c_{ts} = -1$. Minimizing $\sum_{(i,j)} c_{ij} f_{ij}$ in the new network is the same as maximizing the flow f_{ts} on the new arc. Since the flow on the arc (t, s) must return from s to t through the original network, maximizing f_{ts} is the same as solving the original maximum flow problem.
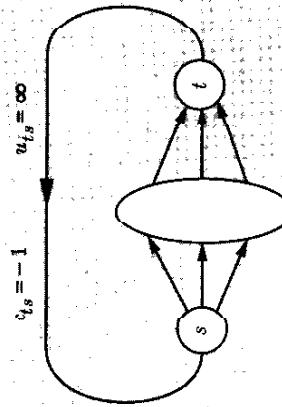

Figure 7.20: Reformulation of the maximum flow problem as a network flow problem

Once the maximum flow problem is formulated as a network flow problem, the negative cost cycle algorithm of Section 7.4 can be applied, and this is one way of deriving the main algorithm in this section (Exercise 7.18). However, our derivation will be self-contained.
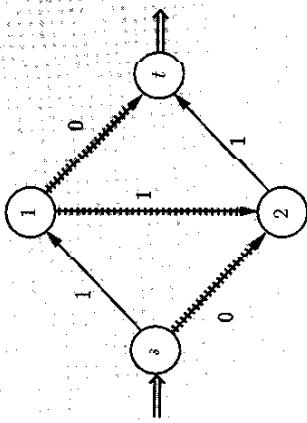
Figure 7.21: Let all arc capacities be equal to 1. The numbers next to each arc indicate the values of the arc flows. Note that up to one unit of additional flow can be pushed along the path indicated by thatched arcs.

Consider the flow illustrated in Figure 7.21. Its value can be increased by pushing additional flow along the path consisting of the arcs $(s, 2)$, $(1, 2)$, $(1, t)$. Note that arc $(1, 2)$ is a backward arc of that path; pushing δ units of flow along the path, reduces the flow along arc $(1\ 2)$ by δ. The definition that follows deals with paths of this type, through which additional flow can be pushed.

Definition 7.2 *Let f be a feasible flow vector. An augmenting path is a path from s to t such that $f_{ij} < u_{ij}$ for all forward arcs, and $f_{ij} > 0$ for all backward arcs on the path.*

Suppose that we have a feasible flow and that we have found an augmenting path P. We can then increase the flow along every forward arc, decrease the flow along every backward arc by the same amount, and still satisfy all of the problem constraints; we then say that we are *pushing flow along the path P*, or that we have a *flow augmentation*. The amount of flow pushed along P can be no more than $\delta(P)$, defined by

$$\delta(P) = \min\left\{ \min_{(i,j)\in F}(u_{ij} - f_{ij}), \ \min_{(i,j)\in B} f_{ij} \right\}, \qquad (7.13)$$

where F and E are the sets of forward and backward arcs, respectively, in the augmenting path. If the augmenting path consists exclusively of forward arcs, and if all arcs on the path have infinite capacity, then there is no limit on the amount of flow that can be pushed, and we have $\delta(P) = \infty$. For the example in Figure 7.21, we have $\delta(P) = 1$.

We now introduce a natural algorithm for the maximum flow problem.

The Ford-Fulkerson algorithm

1. Start with a feasible flow f.
2. Search for an augmenting path.
3. If no augmenting path can be found, the algorithm terminates.
4. If an augmenting path P is found, then:
   (a) If $\delta(P) < \infty$, push $\delta(P)$ units of flow along P, and go to Step 2.
   (b) If $\delta(P) = \infty$, the algorithm terminates

If the algorithm terminates because $\delta(P) = \infty$, we have found an augmenting path without capacity limitations and, using that path, an arbitrarily large amount of flow can be sent to the sink.

We now address the termination properties of the algorithm.

Theorem 7.8 *Suppose that all arc capacities u_{ij} are integer or infinite, and that the Ford-Fulkerson algorithm is initialized with an integer flow vector. Then, the arc flow variables remain integer throughout the algorithm and, if the optimal value is finite, the algorithm terminates after a finite number of steps.*

Proof. This result can be derived as a corollary of Theorem 7.7 in Section 7.4. For a self-contained proof, note that if we have an integer feasible flow, and if all arc capacities are integer or infinite, then $\delta(P)$ is integer or infinite. Thus, integrality of flows is maintained throughout the algorithm. Every iteration of the algorithm increases the value of the flow by at least 1 [since $\delta(P)$ is integer]. Hence, either the value of the flow increases to infinity, or the algorithm must terminate. □

Example 7.5 Consider the network shown in Figure 7.22(a) and let us start with the zero flow. The path consisting of the thatched arcs in Figure 7.22(b) is an augmenting path, with $\delta(P) = 1$. After a flow augmentation, we obtain the flow indicated. The path consisting of the thatched arcs in Figure 7.22(c) is an augmenting path, with $\delta(P) = 1$. By continuing similarly, and after a total of four flow augmentations, we obtain the flow shown in Figure 7.22(e), whose value is equal to 6. At this point, no augmenting path can be found. In fact, this flow must be optimal because the total capacity of the arcs leaving node s is equal to 6, and this is a bottleneck that cannot be overcome.

If the arc capacities are rational numbers, the algorithm is again guaranteed to terminate after a finite number of iterations. This is because we can multiply all arc capacities by their least common denominator, and
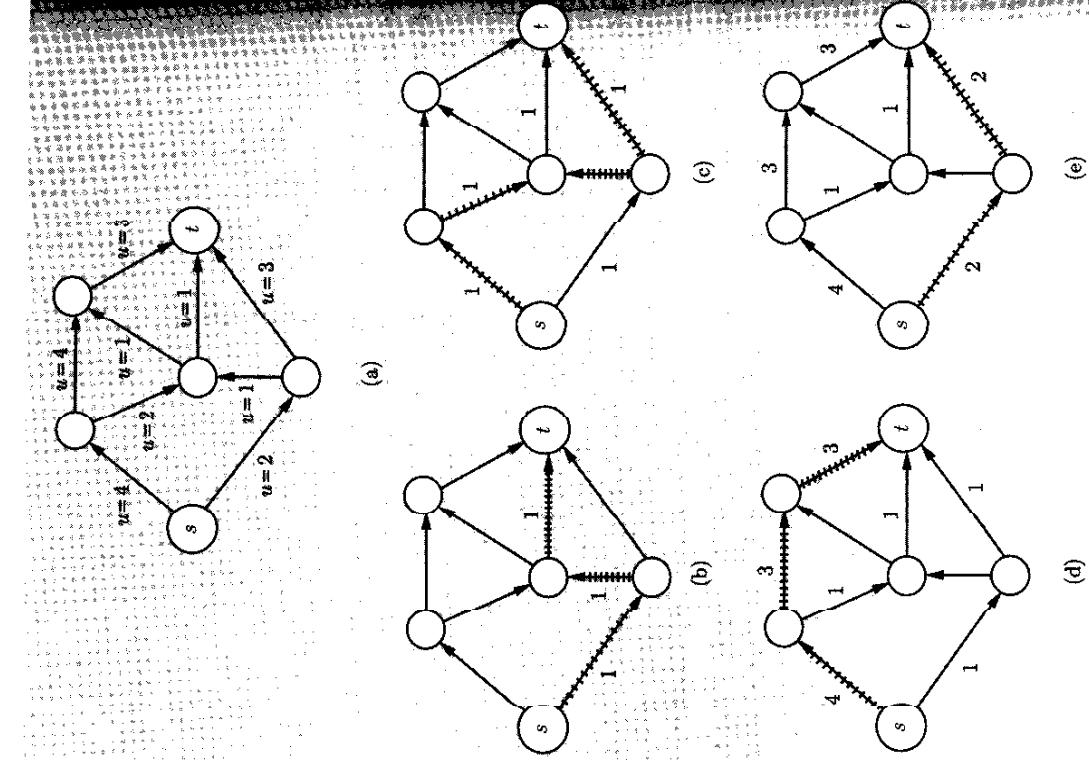
Figure 7.22: Illustration of the Ford-Fulkerson algorithm. The numbers next to the arcs in part (a) are arc capacities. We start with the zero flow. (b)–(e) In each case, we identify the augmenting path indicated in the figure, and push as much flow as possible. The numbers next to the arcs correspond to the arc flows after the flow augmentation. The flow indicated in part (e) is optimal.

obtain an equivalent problem with integer arc capacities. However, if the arc capacities are not rational, there exist examples for which the algorithm never terminates. In particular, even though the value of the flow is nonmonotonically increasing, its limit can be strictly less than the optimal.

For the non-rational case, the Ford-Fulkerson algorithm can be made to terminate after a finite number of iterations, if one uses special methods for choosing an augmenting path. For example, if one looks for an augmenting path with the least possible number of arcs, then the algorithm can be shown to terminate after $O(|A| \cdot |N|)$ iterations.

If the algorithm does terminate, it provides us with an optimal solution. This fact can be obtained as a corollary of the optimality conditions in Section 7.4. A self-contained proof using different ideas will be provided shortly. However, we will first discuss some issues related to the search for an augmenting path.

Searching for an augmenting path

The search for an augmenting path can be carried out in a fairly simple manner, using a method known as the *labeling algorithm*.

Suppose that we have a feasible flow **f**. Consider a path from the source s to some node k, such that $f_{ij} < u_{ij}$ for all forward arcs on the path, and $f_{ij} > 0$ for all backward arcs on the path; we say that this is an *unsaturated* path from s to k. Such a path can be used to push additional flow from node s to node k, without violating the capacity constraints. Note that an unsaturated path from s to t is the same as an augmenting path.

Let us say that a node i is *labeled* if we have determined that there exists an unsaturated path from s to i.

(a) Suppose that node i is labeled, that we have an unsaturated path P from s to i, and that (i, j) is an arc for which $f_{ij} < u_{ij}$. We may then append arc (i, j) to the path P, and obtain an unsaturated path from s to j. Thus, node j can also be labeled.

(b) Similarly, if we have an unsaturated path P from s to i, and if (j, i) is an arc for which $f_{ji} > 0$, we may append arc (j, i) to P (as a backward arc) and obtain an unsaturated path from s to i. Then, node j can be labeled.

The process of examining all nodes j neighboring a given labeled node i, to determine whether they can also be labeled, is called *scanning* node i. We now have the following algorithm, where I is the set of nodes that have been labeled but not yet scanned.

The labeling algorithm

1. The algorithm is initialized with $I = \{s\}$, and with s being the only labeled node.

2. A typical iteration starts with a set I of labeled, but not yet scanned nodes. If $t \in I$ or if $I = \emptyset$, the algorithm terminates. Otherwise, choose a node $i \in I$ to be scanned, and remove it from the set I. Examine all arcs of the form (i,j) or (j,i).

3. If $(i,j) \in A$, $f_{ij} < u_{ij}$, and j is unlabeled, then label j, and add j to the set I.

4. If $(j,i) \in A$, $f_{ji} > 0$, and j is unlabeled, then label j, and add j to the set I.

Note that a node enters the set I only if it changes from unlabeled to labeled. Therefore, a node can enter the set I at most once. Since each iteration removes a node from the set I, the algorithm must eventually terminate. We distinguish between two different possibilities.

(a) Suppose that the algorithm terminates because node t has been labeled. Then, there exists an unsaturated path from s to t, that is, an augmenting path. That path can be easily recovered if we do some extra bookkeeping in the course of the labeling algorithm, as follows. Whenever a node j is labeled while scanning a previously labeled node i, we record node i as the *parent* of j. At the end of the algorithm, we may start at node t, go to its parent, then to its parent's parent, etc., until we reach node s; the resulting path is an augmenting path from s to t.

(b) The second possibility is that the algorithm terminates because the set I is empty. We will now argue that this implies that there exists no augmenting path. Let S be the set of labeled nodes at termination, and suppose that there exists an augmenting path. Since $s \in S$ and $t \notin S$, it follows that there exist two consecutive nodes i and j on the augmenting path, such that $i \in S$ and $j \notin S$. Since i and j are consecutive nodes of an augmenting path, we have either $(i,j) \in A$ and $f_{ij} < u_{ij}$, or $(j,i) \in A$ and $f_{ji} > 0$. In either case, we see that node j should have been labeled at the time that node i was scanned. This is a contradiction and shows that no augmenting path exists.

Example 7.6 Consider the network shown in Figure 7.23. The labeling algorithm operates as follows:

1. $I = \{s\}$. Node s is scanned. Nodes 1, 2 are labeled.
2. $I = \{1, 2\}$. Node 1 is scanned. Node 4 is labeled.
3. $I = \{2, 4\}$. Node 4 is scanned. No node is labeled
4. $I = \{2\}$. Node 2 is scanned. Node 3 is labeled.
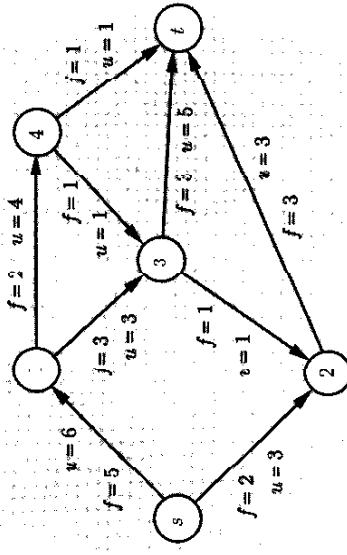
Figure 7.23: The network in Example 7.6 together, with a feasible flow.

5. $I = \{3\}$. Node 3 is scanned. Node t is labeled.

Since node t is labeled, we conclude that there exists an augmenting path, which can be obtained by backtracking, as follows. Node t was labeled while scanning node 3. Node 3 was labeled while scanning node 2. Node 2 was labeled while scanning node s. This leads us to the augmenting path $s, 2, 3, t$.

We conclude our analysis of the labeling algorithm with a brief discussion of its complexity. Every node is scanned at most once, and every arc is examined only when one of its end nodes is scanned. Thus, each arc is examined at most twice. Examining an arc entails only a constant (and small) number of arithmetic operations. We conclude that the computational complexity of the algorithm is proportional to the number of arcs.

We now formally record our conclusions so far.

Theorem 7.9 *The labeling algorithm runs in time $O(|A|)$. At termination, the node t is labeled if and only if there exists an augmenting path.*

Cuts

We define an *s-t cut* as a subset S of the set of nodes N, such that $s \in S$ and $t \notin S$. In our context, the nodes s and t are fixed, and we refer to S as simply a *cut*. We define the *capacity* $C(S)$ of a cut S as the sum of the capacities of the arcs that cross from S to its complement, that is,

$$C(S) = \sum_{\{(i,j) \in A \mid i \in S,\, j \notin S\}} u_{ij}$$
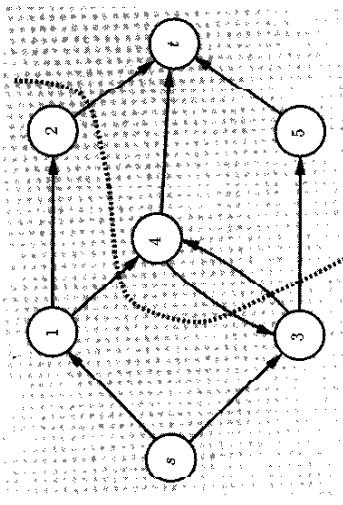
**Figure 7.24:** The set $S = \{s, 1, 2, 3\}$ is a cut. The capacity of this cut is $u_{2t} + u_{14} + u_{35}$.

(see Figure 7.24). Any flow from s to t must at some point cross an arc (i, j) with $i \in S$ and $j \notin S$. For this reason, the value v of any feasible flow satisfies

$$v \leq C(S), \qquad (7.14)$$

for every cut. In essence, each cut provides a potential bottleneck for the maximum flow. Our next result shows that the value of the maximum flow is equal to the tightest of these bottlenecks.

Theorem 7.10

(a) *If the Ford-Fulkerson algorithm terminates because no augmenting path can be found, then the current flow is optimal.*

(b) (**Max-flow min-cut theorem**) *The value of the maximum flow is equal to the minimum cut capacity.*

Proof. (a) Suppose that the Ford-Fulkerson algorithm has terminated because it failed to find an augmenting path. Let S be the set of labeled nodes at termination. These are the nodes i for which there exists an unsaturated path from s to i. Since the search for an augmenting path starts by labeling node s, we have $s \in S$. On the other hand, since no augmenting path was found, node t is not labeled. Therefore, the set S is a cut. For every arc $(i, j) \in \mathcal{A}$, with $i \in S$ and $j \notin S$, we must have $f_{ij} = u_{ij}$. (Otherwise, node j would have been labeled by the labeling algorithm.) Thus, the total amount of flow that exits the set S is equal to $C(S)$. In addition, if $(i, j) \in \mathcal{A}$, with $i \notin S$ and $j \in S$, then $f_{ij} = 0$. (Otherwise, node i would have been labeled by the labeling algorithm.) Thus, the flow crossing from S to its complement cannot return to S, and must exit at the sink node t; see Figure 7.25. This establishes that the value of the

Figure 7.25: Let S and $\overline{S}$ be the sets of labeled and unlabeled nodes, respectively, at termination of the Ford-Fulkerson algorithm. Since j is not labeled, we must have $f_{ij} = u_{ij}$. Since k is not labeled, we must have $f_{kl} = 0$. In particular, all flow moves from s to the rest of S, then to nodes in $\overline{S}$, and finally exits at t.

flow from s to t, when the Ford-Fulkerson algorithm terminates, is equal to $C(S)$. Since the value of the maximum flow can be no higher than $C(S)$ [cf. Eq. (7.14)], we conclude that at termination of the Ford-Fulkerson algorithm, an optimal flow is obtained.

(b) If the optimal value of the flow is infinite, it is not hard to see that there must exist a directed path P from s to t (consisting only of forward arcs), such that every arc in P has infinite capacity. For every cut S, there is an arc (i, j) on the path P such that $i \in S$ and $j \notin S$. Since that arc has infinite capacity, we conclude that $C(S) = \infty$. Since this is true for every cut, we conclude that the minimum cut capacity is infinite and equal to the maximum flow value.

Suppose now that the optimal value, denoted by v^*, is finite. This implies that there exists an optimal solution, that is, a flow whose value is v^*. Let us apply the Ford-Fulkerson algorithm, starting with an optimal flow. Due to optimality of the initial flow, no flow augmentation is possible, and the algorithm terminates with the first iteration. Let S be the set of labeled nodes at termination, as in part (a). From the argument in the proof of part (a), it follows that $C(S) = v^*$. On the other hand, we have $v^* \leq C(S')$ for every cut S'. It follows that $C(S)$ is the minimum cut capacity and is equal to the value of a maximum flow. □

The proof of the max-flow min-cut theorem did rely on the details of the Ford-Fulkerson algorithm. On the other hand, since this theorem relates the optimal values of two optimization problems, one being a minimization and the other being a maximization problem, it is reminiscent of

the duality theorem. Indeed, the max-flow min-cut theorem can be proved by constructing a suitable pair of linear programming problems, dual to each other, and then appealing to the duality theorem (Exercise 7.20).

The complexity of the Ford-Fulkerson algorithm

We close with a discussion of the computational complexity of the Ford-Fulkerson algorithm. We assume that every arc capacity is either integer or infinite, and that the maximum flow value is finite. Let U be the largest of those arc capacities that are finite. The capacity of any cut is either infinite or bounded above by $|A| \cdot U$. If the maximum flow value is finite, there exists at least one cut with finite capacity, and the value is bounded above by $|A| \cdot U$. Therefore, there can be at most $|A| \cdot U$ flow augmentations. Since each flow augmentation involves $O(|A|)$ computations (to run the labeling algorithm), the overall complexity of the algorithm is $O(|A|^2 \cdot U)$. Under the stronger assumption that all arcs outgoing from node s have finite capacity, the maximum flow value can be bounded above by $|N| \cdot U$, by focusing on these arcs. The complexity bound then becomes $O(|A| \cdot |N| \cdot U)$.

The linear dependence of our complexity estimate on U is unappealing, especially if U is a large number. Exercise 7.25 develops a related algorithm whose complexity is proportional to the *logarithm* of U. The key idea is to *scale* the arc capacities, leading to a new problem with smaller arc capacities, which is easier to solve, and whose optimal solution provides a near-optimal solution to the original problem.

There is an alternative method that eliminates the dependence on U altogether. As mentioned earlier, if we always choose an augmenting path with the least possible number of arcs, the number of iterations is $O(|A| \cdot |N|)$, which implies that the complexity is $O(|A|^2 \cdot |N|)$. With proper implementation, this complexity estimate can be further reduced.

7.6   Duality in network flow problems

In this section, we examine the structure of the dual of the network flow problem. For simplicity, we restrict ourselves to the uncapacitated case. We provide interpretations of the dual variables, of complementary slackness, and of the duality theorem. Throughout this section, we let Assumption 7.1 be in effect; that is, the network is assumed to be connected and the supplies satisfy $\sum_{i \in N} b_i = 0$.

The dual problem

The dual of the uncapacitated network flow problem is

$$\text{maximize} \quad \mathbf{p}'\mathbf{b}$$
$$\text{subject to} \quad \mathbf{p}'A \leq \mathbf{c}'.$$

Due to the structure of A, the dual constraints are of the form

$$p_i - p_j \leq c_{ij}, \qquad (i,j) \in A.$$

Suppose that $(p_1,\ldots,p_n)$ is a dual feasible solution. Let θ be some scalar and consider the vector $(p_1 + \theta,\ldots,p_n + \theta)$. It is clear that this is also a dual feasible solution. Furthermore, using the equality $\sum_{i \in N} b_i = 0$, we have

$$\sum_{i=1}^n (p_i + \theta)b_i = \sum_{i=1}^n p_i b_i + \theta \sum_{i=1}^n b_i = \sum_{i=1}^n p_i b_i.$$

Thus, adding a constant to all components of a dual vector is of no consequence as far as dual feasibility or the dual objective is concerned. For this reason, we can and will assume throughout this section that p_n has been set to zero. Note that this is equivalent to eliminating the (redundant) flow conservation constraint for node n.

According to the duality theorem in Chapter 4, if the original problem has an optimal solution, so does the dual, and the optimal value of the objective function is the same for both problems. The example that follows provides an interpretation of the duality theorem in the network context.

Example 7.7 Suppose that we are running a business and that we need to transport a quantity $b_i > 0$ of goods from each node $i = 1,\ldots,n-1$, to node n through our private network. The solution to the corresponding network flow problem provides us with the best way of transporting these goods.

Consider now a transportation services company that offers to transport goods from any node i to node n, at a unit price o_i. If (i,j) is an arc in our private network, we can always transport some goods from i to j, at a cost of c_{ij} and then give them to the transportation services company to transport them to node n. This would cost us $z_{ij} + p_j$ per unit of goods. The transportation services company knows $\mathbf{b}$ and $\mathbf{c}$. It wants to take over all of our transportation business, and it sets its prices so that we have no incentive of using arc (i,j). In particular, prices are set so that $p_i \leq c_{ij} + p_j$, and $p_n = 0$. Having ensured that its prices are competitive, it now tries to maximize its total revenue $\sum_{i=1}^{n-1} p_i b_i$. The duality theorem asserts that its optimal revenue is the same as our optimal cost if we were to use our private network. In other words, when the prices are set right, the new options opened up by the transportation services company will not result in any savings on our part.

Sensitivity

We now provide a sensitivity interpretation of the dual variables. In order to establish a connection with the theory of Chapter 5, we assume that we have eliminated the flow conservation constraint associated with node n, and that the remaining equality constraints are linearly independent.

Suppose that $\mathbf{f}$ is a nondegenerate optimal basic feasible solution, associated with a certain tree, and let $\mathbf{p}$ be the optimal solution to the dual.

Consider some node $i \neq n$ and let p_i be the associated dual variable. Let us change the supply b_i to $b_i + \epsilon$, where ϵ is a small positive number, while keeping the supplies $b_2, \ldots, b_{n-1}$ unchanged. The condition $\sum_{i=1}^n b_i = 0$ then requires that b_n be changed to $b_n - \epsilon$, but this only affects the nth equality constraint which has already been omitted. As long as we insist on keeping the same basis, the only available option is to route the supply increment ϵ from node i to the root node n, along the unique path determined by the tree. Because of the way that dual variables are calculated [cf. Eq. (7.10) in Section 7.3], the resulting cost charge is precisely ϵp_i. This is in agreement with the discussion in Chapters 4 and 5, where we saw that a dual variable is the sensitivity of the cost with respect to changes in the right-hand side of the equality constraints.

By following a similar reasoning, we see that if we increase b_i by ϵ, decrease b_j by ϵ, keep all other supplies unchanged, and use the same basis, the resulting cost change is exactly $\epsilon(p_i - p_j)$, in the absence of degeneracy, and for small ϵ. We conclude that, in the absence of degeneracy, $p_i - p_j$ is the marginal cost of shipping an additional unit of flow from node i to node j.

Complementary slackness

The complementary slackness conditions for the minimum cost network flow problem are the following:

(a) If $p_i \neq 0$, then $[Af]_i = b_i$. This condition is automatically satisfied by any feasible flow f.

(b) If $f_{ij} > 0$, then $p_i - p_j = c_{ij}$. This condition is interpreted as follows. We have $p_i - p_j \leq c_{ij}$, by dual feasibility. If $p_i - p_j < c_{ij}$, then there is a way of sending flow from i to j, which is less expensive than using arc (i,j). Hence, that arc should not carry any flow.

From Theorem 4.5 in Section 4.3, we know that f is primal optimal and p is dual optimal if and only if f is primal feasible, p is dual feasible, and complementary slackness holds. Consider now Figure 7.26, which captures the dual feasibility constraint $p_i - p_j \leq c_{ij}$, the nonnegativity constraint $f_{ij} \geq 0$, and the second complementary slackness condition. We then obtain the following result.

Theorem 7.11 For any uncapacitated network flow problem, the following are equivalent.

(a) The vectors f and p are optimal solutions to the primal and the dual problem, respectively.

(b) The vector f satisfies the flow conservation equation $Af = b$, and for every arc (i,j), the pair $(p_i - p_j, f_{ij})$ satisfies the relations indicated by Figure 7.26.
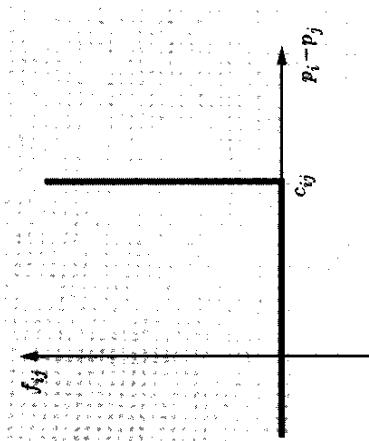
Figure 7.26: Illustration of the complementary slackness conditions. For any arc (i,j), the pair $(p_i - p_j, f_{ij})$ must lie on the heavy line.

A circuit analogy

We now draw an analogy between networks, as defined in this chapter, and electrical circuits. We visualize each node in the network as a place where several "wires" meet, and each arc as a two-terminal circuit element through which current may flow. Let us think of f_{ij} as the current on arc (i,j), and let b_i be the current pumped into the circuit at node i, by means of a current source. Then, the flow conservation equation $Af = b$ amounts to Kirchoff's current law. Let us view p_i as an electric potential. In these terms, Figure 7.26 specifies a relation between the "potential difference" $p_i - p_j$ across arc (i,j) and the current through that same arc. Such a relation is very much in the spirit of Ohm's law (potential difference equals current times resistance) except that here the relation between the potential difference and the current is a bit more complicated.

In circuit terms, Theorem 7.11 can be restated as follows. The vectors f and p are optimal solutions to the primal and dual problem, respectively, if and only if they are equal to an equilibrium state of an electrical circuit, where each circuit element is described by the relation specified by Figure 7.26. If circuit elements with the properties indicated by Figure 7.26 were easy to assemble and calibrate, we could build a circuit, drive it with current sources, and let it come to equilibrium. This would be an analog device that solves the network flow problem. While such devices do not seem promising at present, the conceptual connections with circuit theory are quite deep, and are valid in greater generality (e.g., in network flow problems with a convex nonlinear cost function).

7.7 Dual ascent methods*

In this section, we introduce a second major class of algorithms for the network flow problem, based on dual ascent. These algorithms maintain at all times a dual feasible solution which, at each iteration, is updated in a direction of increase of the dual objective (such a direction is called a *dual ascent* direction), until the algorithm terminates with a dual optimal solution. Algorithms of this type seem to be among the fastest available. In this section, we only consider the special case where all arc capacities are infinite; the reader is referred to the literature for extensions to the general case.

Recall that the dual of the network flow problem takes the form

$$\text{maximize} \quad \sum_{i=1}^{n} p_i b_i$$

$$\text{subject to} \quad p_i \leq c_{ij} + p_j, \qquad (i, j) \in \mathcal{A}.$$

Given a dual feasible vector $\mathbf{p}$, we are interested in changing $\mathbf{p}$ to a new feasible vector $\mathbf{p} + \theta \mathbf{d}$, where θ is a positive scalar, and where $\mathbf{d}$ satisfies $\mathbf{d}'\mathbf{b} > 0$ (which makes $\mathbf{d}$ a dual ascent direction).
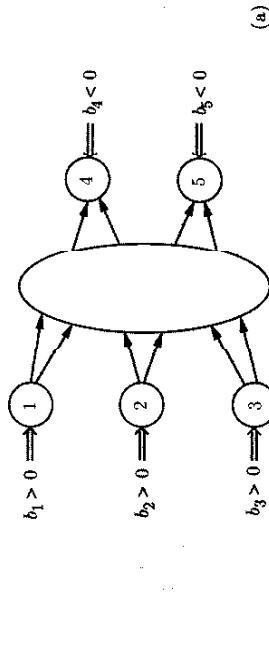
Let S be some subset of the set $\mathcal{N} = \{1, \ldots, n\}$ of nodes. The *elementary direction* $\mathbf{d}^S$ associated with S is defined as the vector with components

$$d_i^S = \begin{cases} 1, & \text{if } i \in S, \\ 0, & \text{if } i \notin S. \end{cases}$$
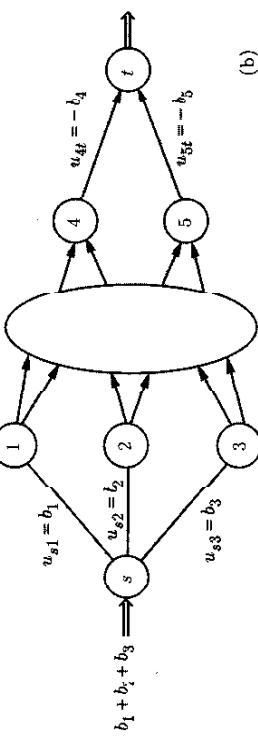
Moving along an elementary direction is the same as picking a set S of nodes and raising the "price" p_i of each one of these nodes by the same amount. A remarkable property of network flow problems is that the search for a feasible ascent direction can be confined to the set of elementary directions, as we now show.

Theorem 7.12 *Let* $\mathbf{p}$ *be a feasible solution to the dual problem. Then, either* $\mathbf{p}$ *is dual optimal or there exists some* $S \subset \mathcal{N}$ *and some* $\theta > 0$, *such that* $\mathbf{p} + \theta \mathbf{d}^S$ *is dual feasible and* $(\mathbf{d}^S)'\mathbf{b} > 0$.

Proof. Let $S \subset \mathcal{N}$ and consider the vector $\mathbf{d}^S$. We start by deriving conditions under which $\mathbf{p} + \theta \mathbf{d}^S$ is feasible for some $\theta > 0$. We only need to check whether any active dual constraints are violated by moving along $\mathbf{d}^S$.

Note that the dual constraint corresponding to an arc $(i, j) \in \mathcal{A}$ is active if and only if $p_i = c_{ij} + p_j$, in which case we say that the arc is *balanced*. Clearly, if (i, j) is a balanced arc and if $i \in S$, raising the value of p_i will violate the constraint $p_i \leq c_{ij} + p_j$, unless the value of p_j is also raised. We conclude that dual feasibility of $\mathbf{p} + \theta \mathbf{d}^S$, for some $\theta > 0$, amounts to the


Figure 7.27: (a) A network with some source nodes and some sink nodes, and in which we have only kept the balanced arcs. (b) A corresponding maximum flow problem; all arcs have infinite capacity with the exception of the arcs (s, i) and (j, t), where i is a source and j is a sink in the original network. There is a feasible solution to the problem in (a) if and only if the optimal value in the maximum flow problem in (b) is equal to $V = b_1 + b_2 + b_3$.

following requirement:

$$\text{if } i \in S \text{ and } (i, j) \text{ is balanced,} \quad \text{then } j \in S. \tag{7.15}$$

Let $Q_+ = \{i \in \mathcal{N} \mid b_i > 0\}$ be the set of source nodes and let $Q_- = \{i \in \mathcal{N} \mid b_i < 0\}$ be the set of sink nodes. Let $V = \sum_{i \in Q_+} b_i$ be the total amount of flow that has to be routed from the sources to the sinks. Our first step is to determine whether the entire supply can be routed to the sinks using *only balanced arcs*. This is accomplished by solving a maximum flow problem of the type shown in Figure 7.27.

Let us run the labeling algorithm, starting from a maximum flow $\mathbf{f}$. Since we already have a maximum flow, no augmenting path is found and node t remains unlabeled. We partition the set $\{1, \ldots, n\}$ of original nodes into sets S and $\overline{S}$ of labeled and unlabeled nodes, respectively. Then, the situation is as shown in Figure 7.28(a).

Let

$$A = \sum_{i \in Q_+ \cap S} f_{si}, \qquad C = \sum_{j \in Q_- \cap S} f_{jt},$$

$$B = \sum_{i \in Q_+ \cap \bar{S}} f_{si}, \qquad D = \sum_{j \in Q_- \cap \bar{S}} f_{jt};$$

see Figure 7.28(b) for an interpretation. The total flow F that leaves node s is equal to $A + B$. On the other hand, all of the flow must at some point traverse an arc that starts in $\{s\} \cup S$ and ends in $\{t\} \cup \bar{S}$. By adding the flow of all such arcs, we obtain $F = B + C$. We conclude that $A = C$, or

$$\sum_{i \in Q_+ \cap S} f_{si} = \sum_{j \in Q_- \cap S} f_{jt}.$$

For every labeled sink node $j \in Q_- \cap S$, we have $f_{jt} = |b_j| = -b_j$, because otherwise node t would have been labeled, which shows that

$$\sum_{i \in Q_+ \cap S} f_{si} = \sum_{j \in Q_- \cap S} |b_j|.$$

We finally note that

$$(\mathbf{d}^S)'\mathbf{b} = \sum_{i \in S} b_i = \sum_{i \in Q_+ \cap S} b_i - \sum_{j \in Q_- \cap S} |b_j| \geq \sum_{i \in Q_+ \cap S} f_{si} - \sum_{j \in Q_- \cap S} |b_j| = 0.$$

We distinguish between two cases. If $(\mathbf{d}^S)'\mathbf{b} > 0$, we have a dual ascent direction, as desired. On the other hand, if $(\mathbf{d}^S)'\mathbf{b} = 0$, we must have $f_{si} = b_i$ for every $i \in Q_+ \cap S$. Since we also have $f_{si} = b_i$ for every $i \in Q_+ \cap \bar{S}$, it follows that the value of the maximum flow is equal to $V = \sum_{i \in Q_+} b_i$, and we have a feasible solution to the original (primal) network flow problem. In addition, since positive flow is only carried by the balanced arcs, complementary slackness holds, and we have an optimal solution to the primal and the dual problem. □

Theorem 7.12 leads to a general class of algorithms for the network flow problem.
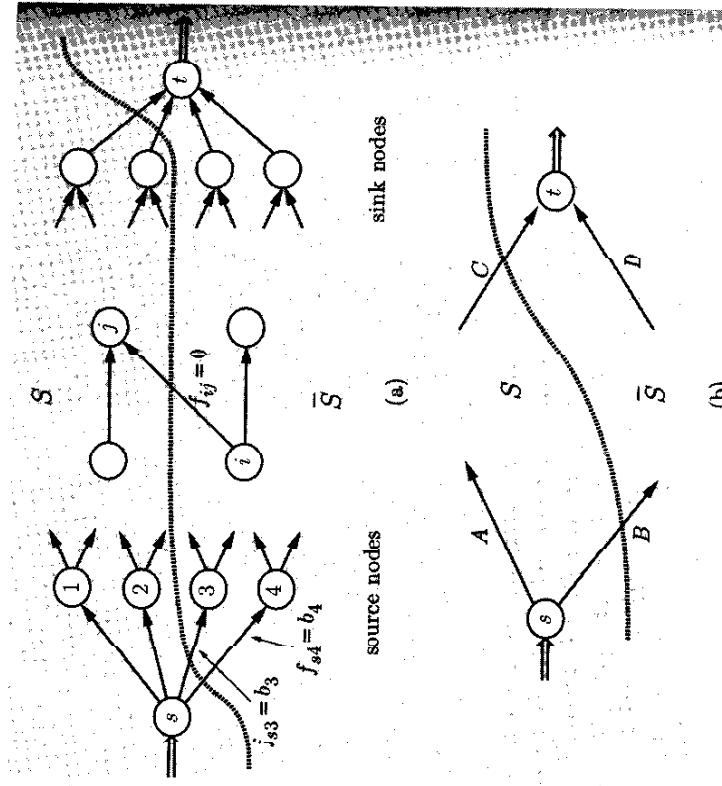
Figure 7.28: The cut obtained at termination of the labeling algorithm, for the network involving only balanced arcs. (a) If a source node i is not labeled, we must have $f_{si} = b_i$ and arc (s,i) is saturated. If a sink node j is labeled, we must have $f_{jt} = |b_j|$ and arc (j,t) is saturated, because otherwise node t would also be labeled. If (i,j) is a balanced arc and if $i \in S$, then we must also have $j \in S$, because otherwise node j would have been labeled (recall that arc capacities are infinite). If (i,j) is a balanced arc and i is not in S, j can be either in S or outside S; if $j \in S$, we must have $f_{ij} = 0$, because otherwise node i would have been labeled. (b) Interpretation of the variables A, B, C, D.

Dual ascent algorithm

1. A typical iteration starts with a dual feasible solution $\mathbf{p}$.

2. Search for a set $S \subset N$ with the property

   if $i \in S$ and (i,j) is balanced, then $j \in S$,

   and such that $\sum_{i \in S} b_i > 0$. If no such set S exists, $\mathbf{p}$ is dual optimal and the algorithm terminates.

3. Update $\mathbf{p}$ to $\mathbf{p} + \theta^* \mathbf{d}^S$, where θ^* is the largest value for which $\mathbf{p} + \theta \mathbf{d}^S$ is dual feasible. If $\theta^* = \infty$, the algorithm terminates; otherwise, go back to Step 2.

The value of θ^* in the dual ascent algorithm is easily determined, as follows. We consider each constraint $p_i \leq c_{ij} + p_j$. The possibility that $\mathbf{p} + \theta \mathbf{d}^S$ may violate this constraint arises only if $i \in S$ and $j \notin S$. For such pairs (i,j), we need $p_i + \theta \leq c_{ij} + p_j$, and we obtain

$$\theta^* = \min_{\{(i,j) \in A \mid i \in S, \ j \notin S\}} (c_{ij} + p_j - p_i). \qquad (7.16)$$

If there is no (i,j) for which $i \in S$ and $j \notin S$, then θ can be taken arbitrarily large, and we let $\theta^* = \infty$. Since we are moving in a direction of dual cost increase, this implies that the optimal dual cost is $+\infty$ and, in particular, the primal problem is infeasible.

Our next results deals with the finite termination of the algorithm.

Theorem 7.13 Suppose that the optimal cost is finite. If the cost coefficients c_{ij} are all integer, and if the dual ascent algorithm is initialized with an integer dual feasible vector, it terminates in a finite number of steps.

Proof. Suppose that the algorithm is initialized with an integer vector $\mathbf{p}$. Then, the value of θ^* is integer. (It cannot be infinite, because the dual optimal cost would also be infinite, which we assumed not to be the case.) Let $v = \min_S (\mathbf{d}^S)' \mathbf{b}$, where the minimum is taken over all S for which $(\mathbf{c}^S)' \mathbf{b} > 0$. Clearly v is positive. Since θ^* is integer, every iteration increases the dual objective by at least v. It follows that the algorithm must terminate after a finite number of steps. $\square$

There are several variations of the dual ascent algorithm which differ primarily in the method that they use to search for an elementary dual ascent direction. If the set S is chosen as in the proof of Theorem 7.12, we have the so-called *primal-dual* method. (When specialized to the assignment problem, it is also known as the *Hungarian* method.) It can be verified

that the primal-dual method uses a "steepest" ascent direction, that is, an elementary ascent direction that maximizes $(\mathbf{d}^S)' \mathbf{b}$ (Exercise 7.30). On the other hand, the so-called *relaxation* method tries to discover an elementary ascent direction $\mathbf{d}^S$ as quickly as possible. In one implementation, a one-element set S is tried first. If it cannot provide a direction of ascent, the set is progressively enlarged until an ascent direction is found. In practice, a greedy search of this type pays off and the relaxation method is one of the fastest available methods for linear network flow problems.

In all of the available dual ascent methods, the search for an elementary ascent direction is streamlined and organized by maintaining a nonnegative vector $\mathbf{f}$ of primal flow variables. Throughout the algorithm, the vectors $\mathbf{f}$ and $\mathbf{p}$ are such that the complementary slackness condition $(c_{ij} + p_j - p_i) f_{ij} = 0$ is enforced. (That is, flow is only carried by balanced arcs.) If such a complementary vector $\mathbf{f}$ is primal feasible, we have an optimal solution to both the primal and the dual. For this reason, dual ascent algorithms can be alternatively described by focusing on the vector $\mathbf{f}$, and by interpreting the different steps as an effort to attain primal feasibility. (This is also the historical reason for the term "primal-dual.")

The primal-dual method

In this subsection, we consider in greater depth the primal-dual method. We do that in order to develop a complexity estimate, and also to illustrate how a network algorithm can be made more efficient by suitable refinements.

The primal-dual method is the special case of the dual ascent algorithm, where the set S is chosen exactly as in the proof of Theorem 7.12. In particular, given a dual feasible vector $\mathbf{p}$, we form a maximum flow problem, in which only the balanced arcs are retained, and we let S be the set of nodes in $\{1, \ldots, n\}$ that are labeled at termination of the maximum flow algorithm. We then update the price vector from $\mathbf{p}$ to $\mathbf{p} + \theta^* \mathbf{d}^S$, form a new maximum flow problem, and continue similarly.

We provide some observations that form the basis of efficient implementations of the algorithm.

(a) *The maximum flow and the current dual vector satisfy the complementary slackness condition* $(c_{ij} + p_j - p_i) f_{ij} = 0$. This is because in the maximum flow problem, we only allow flow on balanced arcs.

(b) *If we determine a maximum flow and then perform a dual update, the complementary slackness condition* $(c_{ij} + p_j - p_i) f_{ij} = 0$ *is preserved.* Suppose that an arc (i,j) carries positive flow in the solution to the maximum flow problem under the old prices. In particular, (i,j) must have been a balanced arc before the dual update. Note that $i \in S$ if and only if $j \in S$. (If $i \in S$, then j gets labeled because the arc capacity is infinite; if $j \in S$, then i gets labeled because $f_{ij} > 0$.) This implies that p_i and p_j are changed by the same amount, the arc (i,j) remains balanced, and complementary slackness is preserved.

(c) An important consequence of observation (b) is that subsequent to a dual update, we do not need to solve a new maximum flow problem from scratch. Instead, we use the maximum flow under the old prices as an initial feasible solution to the maximum flow problem under the new prices. Furthermore, the nodes that were labeled at termination of the maximum flow algorithm under the old prices, will be labeled at the first pass of the labeling algorithm under the new prices. [This is because if node j got its label from a node i through a balanced arc (i,j) or (j,i), then p_i and p_j get raised by the same amount, the arc (i,j) or (j,i) remains balanced, and that arc can be used to label j under the new prices.] Our conclusion is that subsequent to a dual update and given the current flow, we do not need to start the labeling algorithm from scratch, but we can readily assign a label to all nodes that were previously labeled.

(d) A dual update (with $\theta^* < \infty$) results in at least one unlabeled node becoming labeled. Consider an arc (i,j) with $i \in S$, $j \notin S$, and such that $\theta^* = c_{ij} + p_j - p_i$. Such an arc exists by the definition of θ^*. Subsequent to the dual update, this arc becomes balanced. At the first pass of the labeling algorithm, node j will inherit a label from node i.

The preceding observations lead to a new perspective of the primal-dual method. Instead of viewing the algorithm as a sequence of dual updates, with maximum flow problems solved in between, we can view it as a sequence of applications of the labeling algorithm, resulting in flow augmentations, interrupted by dual updates that create new labeled nodes.

At the beginning of a typical iteration, we have a price vector, a flow vector that only uses balanced arcs, and a set of labeled nodes; these are nodes to which additional flow can be sent, using only balanced arcs. We distinguish two cases:

(a) If node t is labeled, we have discovered an augmenting path and we are not yet at an optimal solution to the maximum flow problem. We push as much flow as possible along the augmenting path. At this point, we delete all labels and start another round of the labeling algorithm, to see whether further flow augmentation is possible.

(b) If node t is not labeled, we have a maximum flow and we perform a dual update. Right after the dual update, we resume with the labeling algorithm, but without erasing the old labels. Recall that a dual update results in at least one new balanced arc (i,j), with node i previously labeled and node j previously unlabeled. Node i remains labeled and j will now become labeled. Since every dual update results in an additional node being labeled, node t will become labeled after at most n dual updates, and a flow augmentation will take place

We can now get an upper bound on the running time of the algorithm.

Let, as before, V be the sum of the supplies at the source nodes. Assuming that all supplies are integer, there can be at most V flow augmentations. Since there can be at most n dual updates between any two successive flow augmentations, the algorithm terminates after at most nV dual updates. If at each dual update we determine θ^* using Eq. (7.16), we need $O(m)$ arithmetic operations per dual update, and the running time of the algorithm is $O(mnV) = O(n^4 B)$, where $B = \max_i |b_i|$. With a more clever way of computing θ^*, the running time can be brought down to $O(n^3 B)$ (Exercise 7.28). For the assignment problem, we have $B=1$, and we obtain the so-called Hungarian method, which runs in time $O(n^3)$.
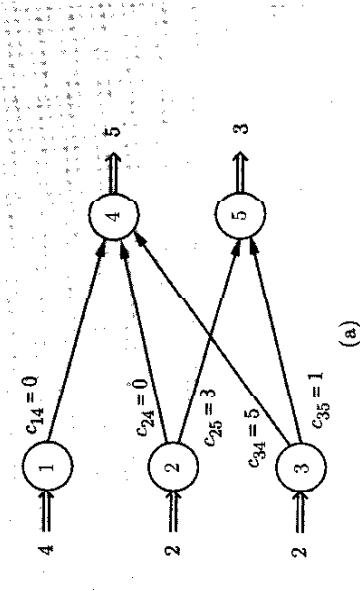
Example 7.8 We go through an example of the primal-dual method. Consider the network shown in Figure 7.29(a), and let us start with the dual vector $\mathbf{p} = (1,1,1,0)$. It is easily checked that we have $p_i \leq c_{ij} + p_j$ for all arcs (i,j), and we therefore have a dual feasible solution. The balanced arcs are $(1,4)$, $(2,4)$, $(3,5)$. In Figure 7.29(b), we form a maximum flow problem involving only the balanced arcs. We solve this problem using the Ford-Fulkerson algorithm. At termination, we obtain the labels and the flows shown in Figure 7.29(c). (Node 2 inherits a label from node 4.) The set of labeled nodes is $S = \{1,2,4\}$ and the corresponding elementary direction is $\mathbf{d}^S = (1,1,0,1,0)$. The only arc (i,j) with $i \in S$, $j \notin S$ is the arc $(2,5)$, and Eq. (7.16) yields $\theta^* = 2$. The new dual vector is $\mathbf{p}+\theta^*\mathbf{d}^S = (3,3,1,3,0)$. The arc $(2,5)$ has now become balanced and all nodes that were labeled remain labeled. Since node 2 is labeled, and arc $(2,5)$ has become balanced, node 5 gets labeled. Finally, because arc $(5,t)$ is unsaturated ($f_{5t} = 2 < 3 = |b_5|$), node t also gets labeled. At this point, we have identified a path through which additional flow can be shipped, namely the path $s,1,4,2,5,t$. By shipping one unit of flow along this path, the value of the flow becomes 8. We now have a feasible solution to the original primal problem, which satisfies complementary slackness, and is therefore optimal. If primal feasibility had not been attained, we would erase all labels and rerun the labeling algorithm, in an attempt to discover a new augmenting path.

Comparison with the dual simplex method
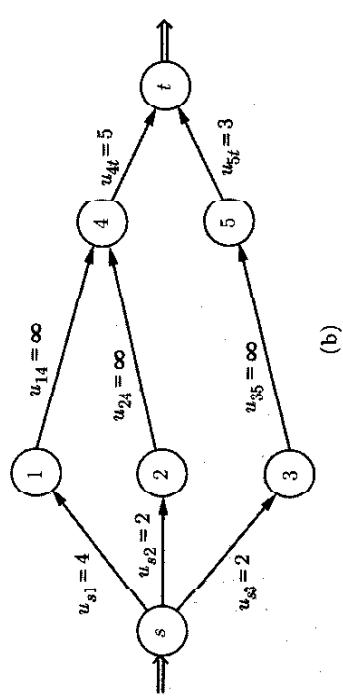
Network flow problems (like all linear programming problems) can be solved by the dual simplex method. This is also a dual ascent method, in the sense that it maintains a dual feasible solution and keeps increasing the dual objective. Furthermore, it can be verified that dual updates in the dual simplex method only take place along elementary directions (Exercise 7.31). On the other hand, the dual simplex method can only visit basic feasible solutions in the dual feasible set. In contrast, the methods considered in this section, have more directions to choose from and do not always move along the edges of the dual feasible set.

A key difference between the dual simplex method and the dual ascent methods of this section is in the nature of the auxiliary flow information

Figure 7.29: Illustration of the primal-dual method in Example 7.8.
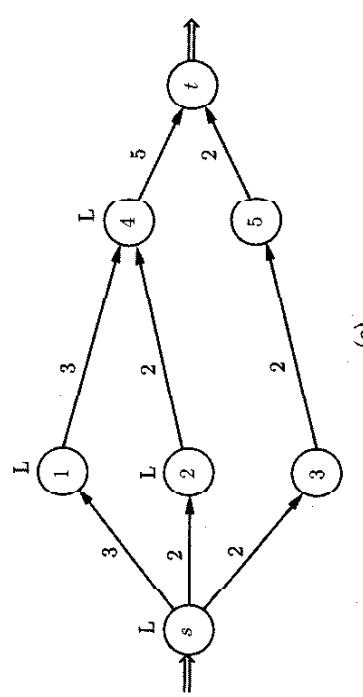
that they employ. In the dual simplex method, we maintain a basic solution to the primal; in particular, the flow conservation constraints are always satisfied. If the basic solution is infeasible, it is only because some of the nonnegativity constraints are violated. In contrast, with the dual ascent methods of this section, auxiliary flow variables are always nonnegative, but we allow the flow conservation equations to be violated.

7.8   The assignment problem and the auction algorithm

The auction algorithm, which is the subject of this section, is a method that can be used to solve general network flow problems. We restrict ourselves to a special case, the assignment problem, because it results in a simpler and more intuitive form of the algorithm. The auction algorithm resembles dual ascent methods, except that it only changes the price of a single node at a time. Given a nonoptimal feasible solution to the dual, it is sometimes impossible to find a dual ascent direction involving a single node. For this reason, a typical iteration may result in a temporary deterioration (i.e., decrease) of the dual objective. As long as this deterioration is kept small, the algorithm is guaranteed to make progress in the long run, and can be viewed as an approximate dual ascent method. Our presentation bypasses this approximate dual ascent interpretation, for which the reader is referred to the literature.

The problem

$$\text{minimize} \quad \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij} f_{ij}$$

$$\text{subject to} \quad \sum_{i=1}^{n} f_{ij} = 1, \qquad j = 1,\ldots,n,$$

$$\sum_{j=1}^{n} f_{ij} = 1, \qquad i = 1,\ldots,n,$$

$$f_{ij} \geq 0, \qquad \forall\, i,j.$$

is known as the *assignment* problem. One interpretation is that there are n persons and n projects and that we wish to assign a different person to each project while minimizing a linear cost function of the form $\sum_{(i,j)} c_{ij} f_{ij}$, where $f_{ij} = 1$ if the ith person is assigned to the jth project, and $f_{ij} = 0$ otherwise. With this interpretation, it would be natural to introduce the additional constraint $f_{ij} \in \{0,1\}$. However, this is unnecessary for the following reasons. First, the constraint $f_{ij} \leq 1$ is implied by the constraints that we already have. Second, Corollary 7.2 implies that the assignment

problem always has an integer optimal solution. In particular, if we solve the assignment problem using the simplex method or the negative cost cycle algorithm, the optimal value obtained for each variable f_{ij} will be zero or one.

Let us now digress to mention an interesting special case of the assignment problem. Suppose that the cost coefficients c_{ij} are either zero or one. The resulting problem is called the *bipartite matching* problem and has the following interpretation. We have $c_{ij} = 0$ if and only if person i is compatible with project j and we are interested in finding as many compatible person-project pairs as possible. If the optimal value turns out to be 0, we say that there exists a *perfect matching*. Besides being an assignment problem, the bipartite matching problem is also a special case of the max-flow problem (send as much flow as possible from persons to projects using only zero cost arcs) and as such it can be also solved using maximum flow algorithms. There are even better special purpose algorithms, which can be found in the literature.

Duality and complementary slackness

We form the dual of the assignment problem. We associate a dual variable r_i with each constraint $\sum_{j=1}^{n} f_{ij} = 1$, and a dual variable p_j to each constraint $\sum_{i=1}^{n} f_{ij} = 1$. Then, the dual problem takes the form

$$\text{maximize} \quad \sum_{i=1}^{n} r_i + \sum_{j=1}^{n} p_j$$

$$\text{subject to} \quad r_i + p_j \leq c_{ij}, \qquad \forall\, i, j.$$

It is clear from the form of the dual constraints that once the values of $p_1, \dots, p_n$ are determined, $\sum_{i=1}^{n} r_i$ is maximized if we set each r_i to the largest value allowed by the constraints $r_i + p_j \leq c_{ij}$, which is

$$r_i = \min_{j=1,\dots,n} \{c_{ij} - p_j\}. \tag{7.17}$$

This leads to the following equivalent dual problem:

$$\text{maximize} \quad \sum_{j=1}^{n} p_j + \sum_{i=1}^{n} \min_j \{c_{ij} - p_j\}. \tag{7.18}$$

Note that this is an unconstrained problem with a piecewise linear concave objective function.

We now consider the complementary slackness conditions for the assignment problem, which are the following:

(a) flow must be conserved;

(b) if $f_{ij} > 0$, then $r_i + p_j = c_{ij}$.

Using Eq. (7.17) to eliminate r_i, the second complementary slackness condition is equivalent to

$$\text{if } f_{ij} > 0, \quad \text{then } p_j - c_{ij} = -r_i = \max_k \{p_k - c_{ik}\}. \tag{7.19}$$

Condition (7.19) admits the following interpretation: each project k carries a reward p_k and if person i is assigned to it, there is a cost c_{ik}. The difference $p_k - c_{ik}$ is viewed as the profit to person i derived from carrying out project k. Condition (7.19) then states that each person should be assigned to a most profitable project.

Auction mechanisms

We recall that a pair of primal and dual solutions is optimal if and only if we have primal and dual feasibility, and complementary slackness. Having defined r_i according to Eq. (7.17), dual feasibility holds automatically. Thus, the problem boils down to finding a set of prices p_j and a feasible assignment, for which the condition (7.19) holds. This motivates a bidding mechanism whereby persons bid for the most profitable projects. It can be visualized by thinking about a set of contractors who compete for the same projects and therefore keep lowering the price (or reward) they are willing to accept for any given project.

Naive auction algorithm

1. *Bidding phase.* Given a set of prices $p_1, \dots, p_n$ for the different projects, and a partial assignment of persons to projects, each unassigned person finds a best project j, that maximizes the profit $p_j - c_{ij}$, and "bids" for it, by accepting a lower price. In particular, the price is lowered by

   (profit of the best project) − (profit of the second best project).

   This is the maximum amount by which the price could be lowered before the best project ceases to be the best one.

2. Following the bidding phase, there is an assignment phase during which every project is assigned to the lowest bidder (if any). The new price of each project is set to the value of the lowest bid. The old holder of the project (if any) now becomes unassigned.

Example 7.9 Consider an assignment problem involving three persons and three objects; see Figure 7.30. Suppose that all p_i are equal to one, that person 1 is assigned to project 1, person 2 is assigned to object 2, and person 3 is unassigned.

Person 3 computes the profits of the different projects; they are $1 - 0 = 1$ for the first and second project, and $1 - 1 = 0$ for the third project. Person 3 bids for the second object. The bid for project 2 cannot be lower than one, because that would make project 2 less profitable than project 1. Hence, the bid is equal to one. Person 3, as the sole bidder, is assigned project 2, and person 2 becomes unassigned. However, there is no price change. In the next iteration, person 2 who is unassigned goes through a similar process, and bids for project 2. The price is again unchanged, and we end up in exactly the same situation as when the algorithm was started.

As Example 7.9 shows, the naive auction algorithm does not always work. The reason is that if there are two equally profitable projects, a bidder cannot lower the price of either, and the algorithm gets deadlocked. However, the algorithm works properly after a simple modification. Let us fix a positive number ϵ. The bid placed for a project is lower by ϵ than what it would have been if we wished that project to remain the best one; as a result, the project comes short, by ϵ, of being the most profitable one. A complete description of the algorithm is given below.

Figure 7.30: An assignment problem. The costs c_{i1} and c_{i2} for the first two projects are zero, for every i. The costs c_{i3} for the third project are equal to one, for every i.

Example 7.10 We apply the auction algorithm to the problem considered in Example 7.9. Once more, we assume that persons 1 and 2 are assigned to projects 1 and 2, respectively, and the initial prices are all equal to 1. Person 3 chooses to bid for project 2 and decreases its price to $1 - \epsilon$. Person 2 becomes unassigned and computes the profits of the different projects; they are: $1 - 0 = 1$, $(1 - \epsilon) - 0 = 1 - \epsilon$, and $1 - 1 = 0$, respectively. Project 1 is the most profitable. Its price is to be brought down so that its profit becomes equal to the profit of the second best project, minus ϵ. Therefore, the bid is equal to $1 - 2\epsilon$.

At the next iteration, person 1, who is unassigned, bids for project 2 and brings its price down to $1 - 3\epsilon$. The same process is then repeated. At each iteration, projects 1 and 2 have prices that are within ϵ of each other. An unassigned person always bids for the one that has the larger price, and brings its price down by 2ϵ. After a certain number of iterations, the prices of projects 1 and 2 become negative. At that point, project 3 finally becomes profitable, receives a bid, becomes assigned, and the algorithm terminates.

Note that a bid pushes the price of a project below the level at which that project would be the most profitable. For this reason, persons will not, in general, be assigned to their most profitable project, and the complementary slackness conditions fail to hold. On the other hand, since persons may underbid only by ϵ, the complementary slackness conditions are close to being satisfied. This motivates our next definition.

Definition 7.3 *Consider a set of prices p_i and a partial assignment where each assigned person $i \in S$ is assigned a project j_i. We say that the ϵ-complementary slackness condition holds if we have*

$$p_{j_i} - c_{ij_i} \geq \max_k \{p_k - c_{ik}\} - \epsilon, \quad \forall \, i \in S.$$

The following result deals with a key property of the auction algorithm.

Theorem 7.14 *Throughout the auction algorithm, the ε-complementary slackness condition is satisfied.*

Proof. The condition is satisfied initially, before any person is assigned. Whenever a person i is assigned a project j_i, the price is chosen so that the profit $p_{j_i} - c_{ij_i}$ cannot be smaller than the profit of any other project by more than ϵ, assuming the other prices do not change. If the prices of some other projects do change, they can only go down, and project j_i is again guaranteed to be within ϵ of being most profitable. As long as a person holds the same project, the price of that project cannot change, and its profit stays constant. In the meantime, the prices of any other projects can only go down, thus reducing their profits, which means that the person still holds a project whose profit is within ϵ of the maximum profit.   □

We also have the following result that ensures the finite termination of the algorithm.

Theorem 7.15 *The auction algorithm terminates after a finite number of stages with a feasible assignment.*

Proof. The proof rests on the following observations:

(a) Once a project receives a bid, it gets assigned to some person. Once a project is assigned, it may be later reassigned to another person, but it will never become unassigned. Thus, if all projects have received at least one bid, then all projects are assigned and, consequently, all persons are also assigned.

(b) If all persons are assigned, no person bids and the algorithm terminates.

(c) If the algorithm does not terminate, then some project never gets assigned. Such a project has never received a bid and its price is fixed at its initial value.

(d) If the algorithm does not terminate, some project receives an infinite number of bids. Since every successive bid lowers its price by at least ϵ, the price of such a project decreases to $-\infty$.

Using observations (c) and (d), a project that has never received a bid must eventually become more profitable than any project that receives an infinite number of bids. On the other hand, for a project to receive an infinite number of bids, it must remain more profitable than any project that has not received any bids. This is a contradiction, which establishes

that every project will eventually receive a bid. Using observations (a) and (b), the algorithm must eventually terminate with all persons assigned to projects.   □

The preceding proof generalizes to the case where some assignments are not allowed, which is the same as setting some of the coefficients c_{ij} to infinity. However, a slightly more involved argument is needed (see Exercise 7.32).

At termination of the auction algorithm, we have:

(a) primal feasibility (all persons are assigned a project);

(b) dual feasibility (if we define $r_i = \max_k\{p_k - c_{ik}\}$, we have a dual feasible solution);

(c) ε-complementary slackness (Theorem 7.14).

If we had complementary slackness instead of ε-complementary slackness, linear programming theory would imply that we have an optimal solution. As it turns out, because of the special structure of the problem, ε-complementary slackness is enough, when ϵ is sufficiently small.

Theorem 7.16 *If the cost coefficients c_{ij} are integer and if*

$$0 < \epsilon < 1/n,$$

the auction algorithm terminates with an optimal solution.

Proof. Let j_i be the project assigned to person i when the algorithm terminates. Using ε-complementary slackness, we have

$$p_{j_i} - c_{ij_i} \geq \max_j\{p_j - c_{ij}\} - \epsilon, \quad \forall\, i.$$

By adding these inequalities over all i, and rearranging, we obtain

$$\sum_{i=1}^n c_{ij_i} \leq \sum_{i=1}^n \left(p_{j_i} - \max_j\{p_j - c_{ij}\} \right) + n\epsilon$$

$$= \sum_{i=1}^n \left(p_{j_i} + \min_j\{c_{ij} - p_j\} \right) + n\epsilon.$$

Let z be the cost of an optimal assignment. The sum in the right-hand side of the above equation is the same as the dual objective function [cf. Eq. (7.18)] and by weak duality, it is bounded above by the optimal cost z. This implies that

$$\sum_{i=1}^n c_{ij_i} \leq z + n\epsilon < z + 1.$$

linear optimization.max

On the other hand

$$\sum_{i=1}^{n} c_{ji} \geq z,$$

by the definition of z. Since z and all c_{ji} are integer, we conclude that

$$\sum_{i=1}^{n} c_{ji} = z,$$

and optimality has been established. □

Discussion

Let us assume, for simplicity, that $c_{ij} \geq 0$ for all i, j, and let $C = \max_{i,j} c_{ij}$. Suppose that the algorithm is initialized with all projects having the same prices. If some project has received C/ϵ or more bids, then its price is lower than the price of any project that has not received any bids, by at least C. (This is because each bid lowers the price by at least ϵ.) At that point, a project that has not received any bids would become more profitable. We conclude that every project receives at most C/ϵ bids. The total number of bids is at most nC/ϵ. Since there is at least one bid at each iteration, this is also a bound on the number of iterations. Finally, the computational effort per iteration is easily seen to be $O(n^2)$. If we let ϵ be slightly smaller than $1/n$, the version of the auction algorithm that we have described here runs in time $O(n^4C)$.

The auction algorithm can be sped up using the idea of ϵ-scaling. One first uses a relatively large value of ϵ, and obtains a solution which is optimal within $n\epsilon$. (The proof is the same as the proof of Theorem 7.16.) Then, the obtained prices are used to start another solution phase, with a smaller value of ϵ, etc. This device leads to better theoretical running time estimates and also to improved performance in practice.

7.9   The shortest path problem

The shortest path problem is an important problem that arises in a multitude of applications in transportation networks, communication networks, optimal control, as well as a subproblem of more complex problems. As will be seen shortly, it can be posed as a network flow problem. However, practical methods for solving the shortest path problem do not rely on the network flow formulation. Instead, they are centered around a set of optimality conditions, known as Bellman's equation, which are intimately related to the subject of dynamic programming (see Section 11.3). We will use duality to derive Bellman's equation, and we will then proceed to develop a suite of algorithms. Some of these algorithms are of a somewhat ad hoc nature, but they are quite efficient in practice.

Throughout this section, the words walk, path, and cycle will always mean *directed* walk, path, and cycle, respectively; that is, all arcs are traversed in the forward direction. This should not lead to any confusion, because in this section we never need to consider walks, paths, or cycles that are not directed.

Formulation

We are given a directed graph $G = (\mathcal{N}, \mathcal{A})$ with n nodes and m arcs. For each arc $(i,j) \in \mathcal{A}$, we are also given a cost or *length* c_{ij}; in general, the numbers c_{ij} are allowed to be negative. The *length* of a walk, path, or cycle is defined as the sum of the lengths of its arcs. A path from a certain node to another is said to be *shortest* if it has minimum length among all possible paths with the same origin and destination. A *shortest walk* from a node to another is defined similarly. A shortest walk and a shortest path from one node to another are not necessarily the same. In particular, if there exists a cycle with negative length, we can construct walks whose length converges to $-\infty$ (we can traverse the cycle several times before reaching the destination). On the other hand, the length of any path is bounded below by $-nC$, where $C = \max_{(i,j)\in\mathcal{A}} |c_{ij}|$. If all cycles have nonnegative length, there is no incentive to go around a cycle and, for this reason, a shortest path is also a shortest walk. Conversely, any cycles contained in a shortest walk must have zero length; by removing such cycles, we obtain a shortest path.

The shortest path problem can be posed in a few different ways; for example, we might be interested in a shortest path from a given origin to a given destination, or we might be interested in shortest paths from each of a number of selected origins to each of several destinations. We will focus on the problem of finding a shortest path from all possible origins to a particular destination node, which is called the *all-to-one shortest path* problem, as well as on the problem of finding shortest paths for all possible origin destination pairs, which is called the *all-pairs shortest path* problem.

Before continuing, we introduce two more concepts that will prove useful. Consider a tree, and suppose that all arcs are assigned directions so that we have a (directed) path from every node $i \neq n$ to node n. Such a directed graph will be called an *intree rooted at node n*; see Figure 7.31. If it happens that for every $i \neq n$, the path from i to n along the tree is a shortest path, we say that we have a *tree of shortest paths*.

Relation to the network flow problem

We consider here the all-to-one shortest path problem. For concreteness, we assume that node n is the destination node. We also assume that there exists at least one path from every node $i \neq n$ to node n, which means that the all-to-one shortest path problem is feasible. Finally, and without loss of

On the other hand

$$\sum_{i=1}^{n} c_{ij_i} \geq z,$$

by the definition of z. Since z and all c_{ij_i} are integer, we conclude that

$$\sum_{i=1}^{n} c_{ij_i} = z,$$

and optimality has been established. □

Discussion

Let us assume, for simplicity, that $c_{ij} \geq 0$ for all i, j, and let $C = \max_{i,j} c_{ij}$. Suppose that the algorithm is initialized with all projects having the same prices. If some project has received C/ϵ or more bids, then its price is lower than the price of any project that has not received any bids, by at least C. (This is because each bid lowers the price by at least ϵ.) At that point, a project that has not received any bids would become more profitable. We conclude that every project receives at most C/ϵ bids. The total number of bids is at most nC/ϵ. Since there is at least one bid at each iteration, this is also a bound on the number of iterations. Finally, the computational effort per iteration is easily seen to be $O(n^2)$. If we let ϵ be slightly smaller than $1/n$, the version of the auction algorithm that we have described here runs in time $O(n^4C)$.

The auction algorithm can be sped up using the idea of ϵ-scaling. One first uses a relatively large value of ϵ, and obtains a solution which is optimal within $n\epsilon$. (The proof is the same as the proof of Theorem 7.16.) Then, the obtained prices are used to start another solution phase, with a smaller value of ϵ, etc. This device leads to better theoretical running time estimates and also to improved performance in practice.

7.9　The shortest path problem

The shortest path problem is an important problem that arises in a multitude of applications in transportation networks, communication networks, optimal control, as well as a subproblem of more complex problems. As will be seen shortly, it can be posed as a network flow problem. However, practical methods for solving the shortest path problem do not rely on the network flow formulation. Instead, they are centered around a set of optimality conditions, known as Bellman's equation, which are intimately related to the subject of dynamic programming (see Section 11.3). We will use duality to derive Bellman's equation, and we will then proceed to develop a suite of algorithms. Some of these algorithms are of a somewhat ad hoc nature, but they are quite efficient in practice.

Throughout this section, the words walk, path, and cycle will always mean directed walk, path, and cycle, respectively; that is, all arcs are traversed in the forward direction. This should not lead to any confusion, because in this section we never need to consider walks, paths, or cycles that are not directed.

Formulation

We are given a directed graph $G = (N, A)$ with n nodes and m arcs. For each arc $(i, j) \in A$, we are also given a cost or length c_{ij}; in general, the numbers c_{ij} are allowed to be negative. The length of a walk, path, or cycle is defined as the sum of the lengths of its arcs. A path from a certain node to another is said to be shortest if it has minimum length among all possible paths with the same origin and destination. A shortest walk from a node to another is defined similarly. A shortest walk and a shortest path from one node to another are not necessarily the same. In particular, if there exists a cycle with negative length, we can construct walks whose length converges to $-\infty$ (we can traverse the cycle several times before reaching the destination). On the other hand, the length of any path is bounded below by $-nC$, where $C = \max_{(i,j)\in A} |c_{ij}|$. If all cycles have nonnegative length, there is no incentive to go around a cycle and, for this reason, a shortest path is also a shortest walk. Conversely, any cycles contained in a shortest walk must have zero length; by removing such cycles, we obtain a shortest path.

The shortest path problem can be posed in a few different ways; for example, we might be interested in a shortest path from a given origin to a given destination, or we might be interested in shortest paths from each of a number of selected origins to each of several destinations. We will focus on the problem of finding a shortest path from all possible origins to a particular destination node, which is called the all-to-one shortest path problem, as well as on the problem of finding shortest paths for all possible origin-destination pairs, which is called the all-pairs shortest path problem.

Before continuing, we introduce two more concepts that will prove useful. Consider a tree, and suppose that all arcs are assigned directions so that we have a (directed) path from every node $i \neq n$ to node n. Such a directed graph will be called an in-tree rooted at node n; see Figure 7.31. If it happens that for every $i \neq n$, the path from i to n along the tree is a shortest path, we say that we have a tree of shortest paths.

Relation to the network flow problem

We consider here the all-to-one shortest path problem. For concreteness, we assume that node n is the destination node. We also assume that there exists at least one path from every node $i \neq n$ to node n, which means that the all-to-one shortest path problem is feasible. Finally, and without loss of

Figure 7.31: An intree rooted at node 6.

generality, we assume that there are no outgoing arcs from node n. These assumptions will remain in effect throughout this section.

We view the graph G as a network of infinite capacity arcs. Suppose that each one of the nodes $1, \ldots, n-1$ is a source node, with unit supply, and that node n is the only sink node, with a demand of $n-1$. If we pose the problem of minimizing $\sum_{(i,j)\in A} c_{ij} f_{ij}$ over all feasible flow vectors, it should be clear that for every node i other than n, one unit of flow should be shipped from node i to node n, at least cost. As long as there are no negative length cycles, this should be done along a shortest path. If on the other hand, there are negative length cycles, the optimal cost in the network flow problem is $-\infty$, because we could "push" an arbitrarily large amount of flow around such a cycle. This discussion is refined in the following theorem.

Theorem 7.17 *Consider the shortest path problem in a directed graph with n nodes and the associated network flow problem. We assume that there is a path to node n from every other node, and that node n has no outgoing arcs.*

(a) *If there exists a negative length cycle, the optimal cost in the network flow problem is $-\infty$.*

(b) *Suppose that all cycles have nonnegative length. If a feasible tree solution is optimal, then the corresponding tree is a tree of shortest paths.*

(c) *Suppose that all cycles have nonnegative length. If we fix p_n to zero, the dual problem has a unique solution p^*, and p_i^* is the shortest path length from node i.*

Proof. Part (a) is trivial. For part (b), we first note that in a feasible tree solution, all arcs in the tree must be oriented from the leaves towards the root and therefore form an intree rooted at node n. This is because if some

arc (i,j) in the tree is pointing away from the root, the flow on that arc must be negative, contradicting feasibility. If for some node i there exists a path from i to n whose length is smaller than that of the path on the tree, the feasible tree solution is not optimal, because some flow could be redirected to that path. Thus an optimal feasible tree solution provides us with a tree of shortest paths and this proves part (b).

Let $p_n^* = 0$ and let $(p_1^*, \ldots, p_{n-1}^*)$ be the vector of dual variables associated with an optimal feasible tree solution. For each arc on the tree, we have $p_i^* = c_{ij} + p_j^*$. Since all arcs are oriented towards the root, we can add the equalities $p_i^* = c_{ij} + p_j^*$ along a path contained in the tree, and conclude that p_i^* is the length of the path from node i to node n. Note that this is a shortest path, since we are dealing with an optimal feasible tree solution. Thus, p_i^* is the shortest path length.

We finally note that every feasible tree solution is nondegenerate. This is because any arc (i,j) on the tree must carry the supply at node i. It follows that the dual problem has a unique solution. $\square$

The connection between shortest paths, network flows, and linear programming duality is illustrated by our next example, which arises in practical context.

Example 7.11 (Project management) A project consists of a set of jobs and a set of *precedence relations* In particular, we are given a set A of job pairs (i,j) indicating that job i cannot start, before job j is completed. Let c_i be the duration of job i. We wish to identify the least possible duration of the project. We will show that this can be accomplished by solving a shortest path problem.

In addition to the original jobs, we introduce two artificial jobs s and t, of zero duration, that signify the beginning and the completion of the project. We augment the set A by introducing the additional precedence relations (s,i) and (i,t) for all jobs i. Let p_i be the time that job i begins. A precedence relation $(i,j) \in A$ leads to a constraint $p_j \geq p_i + c_i$, that is, project j cannot begin before the completion time $p_i + c_i$ of project i. The project duration is $p_t - p_s$ and the minimal project duration is obtained by solving the following problem:

$$\text{minimize} \quad p_t - p_s$$
$$\text{subject to} \quad p_j - p_i \geq c_i, \qquad \forall (i,j) \in A.$$

The dual of this problem is

$$\text{maximize} \quad \sum_{(i,j)\in A} c_i f_{ij}$$
$$\text{subject to} \quad \sum_{\{j|(j,i)\in A\}} f_{ji} - \sum_{\{j|(i,j)\in A\}} f_{ij} = b_i, \qquad \forall i,$$
$$f_{ij} \geq 0, \qquad \forall (i,j) \in A.$$

Here, $b_s = -1$, $b_t = 1$, and $b_i = 0$ for $i \neq s,t$. This is a shortest path problem, where each precedence relation $(i,j) \in A$ corresponds to an arc with cost of $-c_i$. It is natural to assume that the set of arcs A does not contain any cycles,

because otherwise the project cannot be completed. In that case, the network is guaranteed to have no negative cost cycles.

Bellman's equation

Recall that $b_1 = \cdots = b_{n-1} = 1$. Under the convention $p_n = 0$, the dual problem is of the form

$$\text{maximize} \quad \sum_{i=1}^{n-1} p_i$$

$$\text{subject to} \quad p_i \le c_{ij} + p_j, \qquad \forall \, (i,j) \in \mathcal{A}.$$

It is evident that if all components of $\mathbf{p}$, except for p_i, are fixed to some values, the remaining component p_i should be set to the largest value allowed by the constraints, that is, $\min_{k \in O(i)} \{c_{ik} + p_k\}$. [Recall that $O(i)$ is the set of endpoints of arcs that are outgoing from node i.] We conclude that the optimal solution $\mathbf{p}^*$ to the dual problem, which is the same as the vector of shortest path lengths, satisfies

$$p_i^* = \min_{k \in O(i)} \{c_{ik} + p_k^*\}, \qquad i = 1, \ldots, n-1, \tag{7.20}$$

where $p_n^* = 0$. This is a system of $n-1$ nonlinear equations in $n-1$ unknowns, and is known as Bellman's equation. It has a rather intuitive interpretation: suppose that we are interested in paths that start at node i, but that we also impose the additional constraint that the path must start with the arc (i,k). Then, the best we can do is to find a shortest path from node k to n, for a total length of $c_{ik} + p_k^*$. However, since the first node k is of our own choosing, we should make an optimal choice of k, and therefore the length of a shortest path is $\min_{k \in O(i)} \{c_{ik} + p_k^*\}$. The key idea behind Bellman's equation is the so-called *principle of optimality*: if a shortest path from i to n goes through an intermediate node k, then the portion of the path from k to n is also a shortest path.

We have argued that the shortest path lengths satisfy Bellman's equation. Thus, one possible method of computing shortest path distances is by trying to solve Bellman's equation directly. However, some care is needed, because Bellman's equation may have several solutions, and only one of them will give us the correct shortest path lengths; an example is given in Figure 7.32. It turns out that the shortest path lengths are the unique solution to Bellman's equation if all cycles have positive lengths. If all cycles have nonnegative length, we can only assert that the shortest path lengths are the *largest* solution to Bellman's equation (Exercise 7.33).

The Bellman-Ford algorithm

A common method for solving a system of equations of the form $\mathbf{x} = \mathbf{F}(\mathbf{x})$ is to use the iteration $\mathbf{x} := \mathbf{F}(\mathbf{x})$. If we attempt to solve Bellman's equation
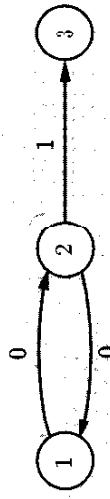

Figure 7.32: Consider a graph with three nodes and let the arc lengths be as indicated. The shortest path lengths to node 3 are $p_1^* = p_2^* = 1$. Bellman's equation is of the form $p_1 = p_2$ and $p_2 = \min\{p_1, 1\}$. It is easily seen that $p_1 = p_2 = \beta$ is a solution to Bellman's equation for every $\beta \le 1$. Note that the shortest path lengths are the largest solution to Bellman's equation.

in this fashion, we obtain the Bellman-Ford algorithm. In the discussion that follows, we again assume that node n has no outgoing arcs.

Let $p_i(t)$ be the length of a shortest walk from node i to node n that uses at most t arcs; we let $p_i(t) = \infty$ if no such walk exists. We use the convention $p_n(t) = 0$ for all t, and $p_i(0) = \infty$ for all $i \ne n$. Note that $p_i(t+1) \le p_i(t)$ for all i and t, because as t increases, there are more walks to choose from. A shortest walk from node i to node n that uses at most $t+1$ arcs, consists of an initial arc (i,k) and a walk from node k to node n that consists of at most t arcs. Of course, the latter walk should be chosen as short as possible and its length is therefore $p_k(t)$. Since node k should also be chosen in the most profitable fashion, we have

$$p_i(t+1) = \min_{k \in O(i)} \{c_{ik} + p_k(t)\}, \qquad i = 1, \ldots, n-1,$$

and this equation defines the Bellman-Ford algorithm. We now discuss the termination properties of the algorithm.

(a) Suppose that there are no negative length cycles. Then, there exists a shortest walk, which is also a shortest path, and has at most $n-1$ arcs. In particular, $p_i(n-1) = p_i^*$. Allowing for a walk with n or more arcs cannot reduce the total length, and we have $p_i(n) = p_i(n-1)$ for all nodes.

(b) Suppose that there exists a negative length cycle. Suppose for a moment, that we also have $\mathbf{p}(n) = \mathbf{p}(n-1)$. This implies that $\mathbf{p}(t) = \mathbf{p}(n)$ for all $t \ge n$ and the length of any walk is bounded below. However, in the presence of negative length cycles, there exist walks whose length tends to $-\infty$. This is a contradiction and proves that $\mathbf{p}(n) \ne \mathbf{p}(n-1)$.

By comparing the two cases just discussed, we see that no more than n iterations are needed. If $\mathbf{r}(n) = \mathbf{p}(n-1)$, then $\mathbf{p}(n)$ is the vector of shortest path lengths. (An example is given in Figure 7.33.) If on the other hand $\mathbf{p}(n) \ne \mathbf{p}(n-1)$, we conclude that there exists a negative length cycle.

delete the arcs on the cycle and we are left with a walk with fewer than n arcs whose length is no greater than $p_i(n)$. This contradicts the inequality $p_i(n) < p_i(n-1)$. We conclude that by tracing the successors of node i, we will discover a negative length cycle.

Label correcting methods

Label correcting methods are a general class of shortest path algorithms, that have proved to be very efficient in practice. They are similar in spirit to the Bellman-Ford algorithm, but they are more flexible, hence the potential for improved performance.

The key idea is to maintain at each node j, a label p_j equal to the length of the shortest walk from j to n discovered thus far. Given a walk from j to n, of length p_j, there exists a walk from i to n of length $c_{ij} + p_j$. Thus, each time that p_j is revised downwards ("corrected"), we also have an opportunity to revise downwards the labels of all nodes i that have an outgoing arc to node j (the *predecessors* of j). The algorithm maintains a *list* S of all nodes whose labels have been revised downwards, and such that the revision has not yet been propagated to their predecessors. (The list S plays a role similar to the list of labeled but not yet scanned nodes in the labeling algorithm of Section 7.5.)

Label correcting algorithm

The algorithm is initialized with $S = \{n\}$, $p_n = 0$, and $p_i = \infty$ for every $i \neq n$. A typical iteration is as follows.

1. Remove a node j from S.

2. For every node $i \neq n$ such that (i,j) is an arc, do the following. Let $p_i := \min\{p_i, c_{ij} + p_j\}$. If the new value of p_i is smaller, add node i to the set S.

3. If S is empty, the algorithm terminates. Otherwise, go back to Step 1.

The label of a node is always equal to the length of some walk to node n. (Except when the label is infinite, indicating that a path has not yet been discovered.) This is easily shown by induction. Indeed, assuming this to be true before an update, the new label $\min\{p_i, c_{ij} + p_j\}$ is either equal to the length p_i of a previously identified walk, or is equal to the length $c_{ij} + p_j$ of a walk that starts with arc (i,j) and follows a previously identified walk from j to n.

We now establish the finite termination of the algorithm. We assume that all cycles have nonnegative length. Let p_i^0 be the first finite label assigned to node i. Any walk from i to n whose length is less than p_i^0 consists of a path from i to n, an arbitrary number of zero length cycles,

Figure 7.33: We apply the Bellman-Ford algorithm to the graph shown. Node 4 is the destination node. We have $p_4(t) = 0$ for all t, and

$$p_1(0) = \infty, \quad p_1(1) = \infty, \quad p_1(2) = 9, \quad p_1(3) = 9,$$
$$p_2(0) = \infty, \quad p_2(1) = 7, \quad p_2(2) = 7, \quad p_2(3) = 7,$$
$$p_3(0) = \infty, \quad p_3(1) = 1, \quad p_3(2) = 1, \quad p_3(3) = 1.$$

We observe that $\mathbf{p}(3) = \mathbf{p}(2)$ and, therefore, $\mathbf{p}(2)$ is equal to the shortest path length vector $\mathbf{p}^*$.

The computational complexity of the algorithm is $O(mn)$ because there are at most n iterations and at each iteration, each arc is only examined once.

We have focused so far on the computation of the shortest path lengths rather than the shortest paths. The reason is that once the shortest path lengths are available, shortest paths can be determined fairly easily (Exercise 7.34). The task of finding shortest paths is made even easier if in the course of the algorithm, we maintain some information that allows us to backtrack and recover a shortest path. This is done as follows. For every node i, we keep a record of a *successor* node $s(i)$, chosen as the first node in a path whose total length is equal to the current estimate $p_i(t)$ available at node i. Determining a successor node with such a property is simple: whenever we have $p_i(t+1) < p_i(t)$, we delete the old successor of i, if any, and let $s(i)$ be such that $p_i(t+1) = c_{is(i)} + p_{s(i)}(t)$.

As noted earlier, the Bellman-Ford algorithm provides us with a method for checking whether there are any negative length cycles. Besides detecting the existence of a negative length cycle, some applications such as the negative cost cycle algorithm of Section 7.4, require the construction of a negative length cycle. This can be accomplished as follows. Consider a node i for which $p_i(n) < p_i(n-1)$. By starting at node i and going from each node to its successor, we obtain a walk with n arcs whose length is $p_i(n)$. Since there are only n nodes in the graph, this walk must contain a cycle. Suppose that this cycle has nonnegative length. Let us

and a bounded number of positive length cycles. Since zero length cycles have no effect on the length of the walk, the possible values of p_i that are smaller than p_i^0, are finitely many. This implies that there can only be finitely many downward revisions of each label. After some point, there will be no more revisions, and each iteration will only result in the removal of some node from S. It follows that S eventually becomes empty and the algorithm terminates.

We conclude our analysis, by analyzing the correctness of the algorithm.

Theorem 7.18 *Suppose that there exists a path from every node to node n, and that all cycles have nonnegative length. Then, the label correcting algorithm eventually terminates with the label p_i of each node equal to the shortest path length p_i^*.*

Proof. Consider a shortest path $i_1, i_2, \ldots, i_t = n$ from some node i_1 to n. By the definition of the algorithm, we have $p_n = 0 = p_n^*$, at all times. At the first iteration of the algorithm, we have $S = \{n\}$, the predecessors of n are examined, and we set $p_{i_{t-1}} = c_{i_{t-1}n}$, which is equal to $p_{i_{t-1}}^*$. (This is because the last arc of a shortest path is itself a shortest path.)

Consider now an intermediate node i_k in the path, and suppose that the final label p_{i_k} is equal to $p_{i_k}^*$. Since p_{i_k} was initially infinite, its label has changed at least once. The last time that p_{i_k} was changed, and was set to $p_{i_k}^*$, node i_k entered the set S. When at some later iteration, i_k exited S, $p_{i_{k-1}}$ was set to $\min\{p_{i_{k-1}}, c_{i_{k-1}i_k} + p_{i_k}^*\}$. This is less than or equal to $c_{i_{k-1}i_k} + p_{i_k}^* = p_{i_{k-1}}^*$. On the other hand, $p_{i_{k-1}}$ is the length of some walk, and can be no smaller than $p_{i_{k-1}}^*$. We have therefore completed an inductive proof that $p_{i_k} = p_{i_k}^*$ for all k.

The practical efficiency of label correcting methods is highly dependent on the rule used to select a node from the list S. It is interesting to note that for certain rules, including some that have been very successful in practice, the worst-case complexity is exponential in n. The reader is referred to the literature for a more detailed discussion.

Dijkstra's algorithm

Dijkstra's algorithm is an alternative to the Bellman-Ford algorithm and label correcting methods. We will see shortly that Dijkstra's algorithm is more efficient, but can only be applied if all arc lengths are nonnegative, which will be assumed throughout this section. The key idea in Dijkstra's algorithm is to identify the nodes in the order of the corresponding shortest path lengths, starting with a node for which the shortest path length is smallest. In order to simplify the presentation we assume that c_{ij} is defined

for every pair (i,j) of distinct nodes (with $i \neq n$), but may be equal to infinity for some pairs.

Our first step is to show that a node ℓ with a smallest shortest path length is easy to find. Nonnegativity of the arc lengths is crucial here.

Theorem 7.19 *Suppose that $c_{ij} \geq 0$ for all i, j. Let $\ell \neq n$ be such that*

$$c_{\ell n} = \min_{i \neq n} c_{in}.$$

Then, $p_\ell^* = c_{\ell n}$ and $p_\ell^* \leq p_k^*$ for all $k \neq n$.

Proof. Any path to node n has a last arc (i,n) whose length c_{in} is at least $c_{\ell n}$. Thus, $p_k^* \geq c_{\ell n}$ for all $k \neq n$. For node ℓ, we also have $p_\ell^* \leq c_{\ell n}$. We conclude that $p_\ell^* = c_{\ell n} \leq p_k^*$ for all $k \neq n$.   □

Suppose that ℓ and p_ℓ^* have been determined as in Theorem 7.19, and consider an arbitrary node i. One of the options available at that node is to traverse the arc (i, ℓ) and visit node ℓ. Once at node ℓ, we should traverse arc (ℓ, n), because this is a shortest path from ℓ to n. Thus, once an arc (i, ℓ) is traversed, the reversal of arc (ℓ, n) can be assumed to be automatic. We can therefore replace the two arcs (i, ℓ) and (ℓ, n) by a single arc $(i, n)'$ of length $c_{i\ell} + c_{\ell n}$; once we do that for every $i \neq \ell, n$, node ℓ can be taken out of the picture. Note that a node i may now have two direct arcs to node n, the original arc (i, n) as well as the new artificial arc $(i, n)'$. Naturally, any shortest path would only use the least expensive of the two. We therefore remove $(i, n)'$ and replace c_{in} by

$$\min\{c_{in}, c_{i\ell} + c_{\ell n}\}.$$

We are left with a new shortest path problem with one node less. We apply the same process to the new shortest path problem. Each iteration evaluates the shortest path length for one more node and, therefore, after $n-1$ iterations, the algorithm terminates.

The resulting algorithm is summarized next.

Dijkstra's algorithm

1. Find a node $\ell \neq n$ such that $c_{\ell n} \leq c_{in}$ for all $i \neq n$. Set $p_\ell^* = c_{\ell n}$.
2. For every node $i \neq \ell, n$, set

$$c_{in} := \min\{c_{in}, c_{i\ell} + c_{\ell n}\}.$$

3. Remove node ℓ from the graph and apply the same steps to the new graph.
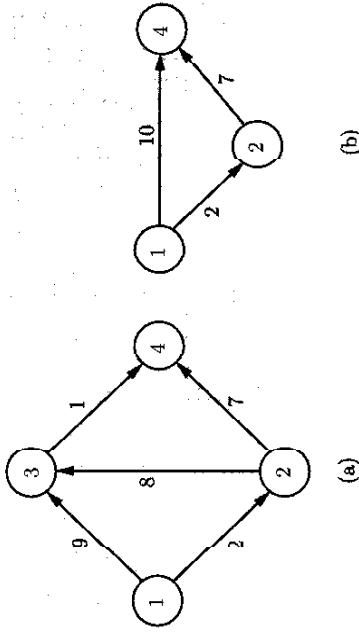
Figure 7.34: (a) A graph with arc lengths. The arcs that are not shown have infinite length. (b) The graph obtained after one iteration of Dijkstra's algorithm.

Example 7.12 We apply Dijkstra's algorithm to graph shown in Figure 7.34(a), with node $n = 4$ being the destination node. We have $\ell = 3$ and $p_3^* = 1$. The following arc lengths are modified: $c_{14} := \min\{\infty, 9+1\} = 10$ and $c_{24} := \min\{7, 8+1\} = 7$. We now eliminate node 3 and obtain the graph shown in Figure 7.34(b). We obtain $\ell = 2$ and $p_2^* = 7$. The arc length c_{14} is modified by $c_{14} = \min\{10, 2+7\} = 9$. Node 2 is eliminated. Since node 1 is the only nonterminal node left, p_1^* is equal to the current value of c_{14}, which is 9.

We now estimate the computational complexity of the Dijkstra algorithm. A typical iteration starts by comparing the coefficients c_{in} and this takes $O(n)$ time. Having determined ℓ, we need to update c_{in} for each node i. We conclude that there are only $O(n)$ arithmetic operations per iteration. The overall complexity is $O(n^2)$, which is one order of magnitude better than the Bellman-Ford algorithm. For a dense graph with $\Omega(n^2)$ arcs, any shortest path algorithm needs $\Omega(n^2)$ arithmetic operations because, in general, every arc has to be examined at least once. Thus, for dense graphs, Dijkstra's algorithm is the best possible.

For sparse graphs, that is, when m is much smaller than n, the computational complexity of Dijkstra's algorithm can be brought down to $O(m \log n)$. Doing so requires keeping the coefficients c_{in} in a suitable data structure that allows us to obtain the smallest such coefficient with minimal work.

Reduction to the case of nonnegative arc lengths and the all-pairs problem

Suppose that some of the arc lengths are negative, but that all cycles have nonnegative length. Let p_i^* be the shortest path length from node i to node

n. From Bellman's equation, we have

$$p_i^* \le c_{ij} + p_j^*, \qquad (7.21)$$

for all arcs (i,j). Let us now construct a new shortest path problem in which the arc lengths c_{ij} are replaced by new arc lengths $\bar{c}_{ij}$, defined by

$$\bar{c}_{ij} = c_{ij} + p_j^* - p_i^*.$$

Using Eq. (7.21), we have $\bar{c}_{ij} \ge 0$ for all $(i,j) \in \mathcal{A}$. Under the new arc lengths, the length of any path $i_1, \ldots, i_t$ from some node i_1 to some other node i_t is given by

$$\sum_{\tau=1}^{t-1} \bar{c}_{i_\tau i_{\tau+1}} = \sum_{\tau=1}^{t-1} (c_{i_\tau i_{\tau+1}} + p_{i_{\tau+1}}^* - p_{i_\tau}^*) = p_{i_t}^* - p_{i_1}^* + \sum_{\tau=1}^{t-1} c_{i_\tau i_{\tau+1}}.$$

In particular, for any given pair of nodes, a shortest path under the new arc lengths is a shortest path under the old arc lengths, and conversely. Since the new arc lengths are nonnegative, we are in a position to apply Dijkstra's algorithm.

If we are only interested in a single destination, the transformation that we have just described is of no particular use. On the other hand, if we are interested in the all-pairs problem, we can solve a single all-to-one problem, using the Bellman-Ford algorithm, transform the arc lengths, and finally solve $n-1$ additional all-to-one problems (ore problem for every possible destination) using Dijkstra's algorithm. The overall complexity s $O(n^3) + (n-1) \cdot O(n^2) = O(n^3)$. This is much better than applying the Bellman-Ford algorithm n times, which would require $O(n^4)$ time. For sparse graphs, the running time can be brought down to $O(nm \log n)$ by using an efficient implementation of Dijkstra's algorithm. An alternative $O(n^3)$ algorithm for the all-pairs problem is developed in Exercise 7.38.

7.10 The minimum spanning tree problem

We are given a connected *undirected* graph $G = (\mathcal{N}, \mathcal{E})$, with n nodes. For each edge $e \in \mathcal{E}$, we are also given a cost coefficient c_e. (Recall that an edge in an undirected graph is an unordered pair $e = \{i,j\}$ of distinct nodes in $\mathcal{N}$.) A *minimum spanning tree* (MST) is defined as a spanning tree such that the sum of the costs of its edges is as small as possible.

The minimum spanning tree problem arises naturally in many applications. For example, if edges correspond to communication links, a spanning tree is a set of links that allows every node to communicate (possibly, indirectly) to every other node. Then, a minimum spanning tree is a communication network that provides this type of connectivity, and whose cost is the smallest possible. The minimum spanning tree problem

also arises as a subproblem of more complex, seemingly unrelated, problems. An example will be seen in Section 11.5, where it forms a basis for heuristic for the traveling salesman problem.

Even though the MST problem is not a network flow problem, we include it in this chapter, because of its graph-theoretic structure. We will see that it can be solved by means of a simple greedy algorithm. A greedy algorithm is one consisting of a sequence of choices that appear to be best in the short run. For certain problems, like the MST, short run optimal decisions turn out to be optimal in the long run as well. The algorithm that we describe builds an MST by progressively adding edges to a current tree. At any stage, we have a tree and we add a least expensive edge that connects a node in the tree with a node outside the tree.

Greedy algorithm for the minimum spanning tree problem

1. The input to the algorithm is a connected undirected graph $G = (\mathcal{N}, \mathcal{E})$ and a coefficient c_e for each edge $e \in \mathcal{E}$. The algorithm is initialized with a tree $(\mathcal{N}_1, \mathcal{E}_1)$ that has a single node and no edges ($\mathcal{E}_1$ is empty).

2. Once $(\mathcal{N}_k, \mathcal{E}_k)$ is available, and if $k < n$, we consider all edges $\{i, j\} \in \mathcal{E}$ such that $i \in \mathcal{N}_k$ and $j \notin \mathcal{N}_k$. Choose an edge $e^* = \{i, j\}$ of this type whose cost is smallest. Let
$$\mathcal{N}_{k+1} = \mathcal{N}_k \cup \{j\}, \qquad \mathcal{E}_{k+1} = \mathcal{E}_k \cup \{e^*\}.$$

Since at each stage we connect a node in the current tree with a node outside the tree, no cycles are ever formed, and we always have a tree. The set $\mathcal{N}_n$ has n elements and, therefore, $(\mathcal{N}_n, \mathcal{E}_n)$ is a spanning tree. It remains to show that it is a minimum spanning tree. This is accomplished by showing a somewhat stronger property.

Theorem 7.20 For $k = 1, \ldots, n$, the tree $(\mathcal{N}_k, \mathcal{E}_k)$ is part of some MST. That is, there exists an MST $(\mathcal{N}, \overline{\mathcal{E}})$ such that $\mathcal{E}_k \subset \overline{\mathcal{E}}_k$.

Proof. The proof uses induction on k. The result is trivially true for $k = 1$, because the empty set $\mathcal{E}_1$ is a subset of the edge set of any spanning tree.

Suppose now that $k < n$, and that $\mathcal{E}_k$ is a subset of some MST $\overline{\mathcal{E}}_k$. [We are slightly abusing terminology by referring to $\overline{\mathcal{E}}_k$ instead of $(\mathcal{N}, \overline{\mathcal{E}}_k)$ as a spanning tree.] Let $e^* = \{i, j\}$ be the edge added to $\mathcal{E}_k$; that is, $i \in \mathcal{N}_k$, $j \notin \mathcal{N}_k$, and $\mathcal{E}_{k+1} = \mathcal{E}_k \cup \{e^*\}$. If $e^* \in \overline{\mathcal{E}}_k$, then $\mathcal{E}_{k+1}$ is also a subset of $\overline{\mathcal{E}}_k$, and the induction hypothesis is verified for $k+1$, with $\overline{\mathcal{E}}_{k+1} = \overline{\mathcal{E}}_k$.

Suppose now that $e^* \notin \overline{\mathcal{E}}_k$. Then, e^*, together with $\overline{\mathcal{E}}_k$, forms a unique cycle [Theorem 7.1(d)]. This cycle must contain a second edge (call it $\overline{e}$) with one endpoint in $\mathcal{N}_k$ and another outside $\mathcal{N}_k$; see Figure 7.35. Since
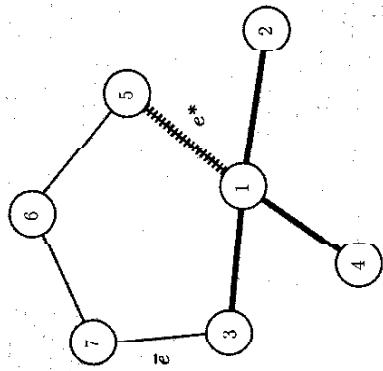
Figure 7.35: The thicker edges correspond to a tree $(\mathcal{N}_4, \mathcal{E}_4)$ involving 4 nodes. This is assumed to be part of an MST $\overline{\mathcal{E}}_4$, which consists of all edges shown, with the exception of e^*. If the algorithm selects e^*, its cost can be no greater than the cost of $\overline{e}$, and $\mathcal{E}_4 \cup \{e^*\}$ is part of an alternative MST, in which $\overline{e}$ is replaced by e^*.

the algorithm selected e^* rather than $\overline{e}$ to be added to $\mathcal{E}_k$, we must have $c_{e^*} \leq c_{\overline{e}}$. Let us now take the MST $\overline{\mathcal{E}}_k$, delete edge $\overline{e}$, and replace it by e^*. We obtain a new spanning tree, call it $\overline{\mathcal{E}}_{k+1}$, and the cost change is $c_{e^*} - c_{\overline{e}} \leq 0$. By the optimality of $\overline{\mathcal{E}}_k$, we must have $c_{e^*} = c_{\overline{e}}$, and both spanning trees are optimal. We now note that $\mathcal{E}_{k-1}$ is a subset of the MST $\overline{\mathcal{E}}_{k+1}$, and the induction is complete. $\square$

Having proved the correctness of the algorithm, we now discuss its computational complexity. We have $n-1$ iterations. At each iteration, we need to examine each edge to see whether it is eligible for becoming part of the tree, and we then need to find the least expensive one, which can all be done in time $O(n^2)$. Thus, the overall complexity is $O(n^3)$. With a more clever implementation, it can be brought down to $O(n^2)$; see Exercise 7.39.

7.11 Summary

In this chapter, we provided an overview of a broad range of topics related to network flow problems, and we have covered most of the major available methodologies.

Network flow problems are special cases of linear programming problems, and can be solved by applying general purpose methods, suitably

tuned to exploit the network structure. For example, the primal or the dual simplex method can be used. As we have pointed out, the underlying network structure allows for simple and efficient rules for updating the basic variables and the reduced costs. In addition, when the problem data are integer, integer arithmetic can also be employed.

An important property of network flow problems that we discovered in the course of our development, relates to integrality of basic solutions. Assuming that problem data are integer, we have shown that basic solutions to the primal and the dual have integer coordinates. The key reason behind this property is that the determinant of any basis matrix $\mathbf{B}$ has unit magnitude. Unfortunately, there are only precious few classes of linear programming problems that have such remarkable properties.

Besides fine tuning the simplex method, we also developed some algorithms that are specially tailored to network flow problems. These include the negative cost cycle algorithm of Section 7.4 and the dual ascent methods of Section 7.7. These two methods are dual to each other in many ways that can be made mathematically precise, but which are beyond our scope. Nevertheless, it is important to point out a common feature. In both methods, a direction of improvement is identified by examining only a finite number of possible directions, which are independent of the numerical values of the input data. (In the negative cost cycle algorithm, the directions considered correspond to simple circulations. In dual ascent methods, the directions considered correspond to subsets of the set of nodes.)

Both the negative cost cycle algorithm and the dual ascent methods can be described at a high level of generality, while leaving a lot of freedom on how to choose a cycle or a dual ascent direction. By making some more specific choices, the worst-case number of iterations can be reduced. Furthermore, the search for a direction of cost improvement, carried out in the course of each iteration, usually has a lot of room for increased efficiency. (An example of this is our development of the primal-dual method, where the search for an ascent direction is implemented by means of an auxiliary maximum flow problem and the labeling algorithm.) Such refinements lead to improved worst-case complexity bounds. It should be kept in mind, however, that worst-case complexity bounds may not accurately reflect the performance of an algorithm in practice.

The network flow problem contains some important special cases that can be solved by suitable special purpose algorithms. We saw the Ford-Fulkerson algorithm for the maximum flow problem, the auction algorithm for the assignment problem, and a number of (somewhat ad hoc) methods for the shortest path problem. Auction algorithms can also be developed for the general network flow problem, but this is a direction that we did not pursue.

The minimum spanning tree problem is somewhat disjoint from the rest of the chapter. It was included because of its importance, and also because it shares an underlying graph-theoretic structure.

7.12 Exercises

Exercise 7.1 (The caterer problem) A catering company must provide to a client r_i tablecloths on each of N consecutive days. The catering company can buy new tablecloths at a price of p dollars each, or launder the used ones. Laundering can be done at a fast service facility that makes the tablecloths unavailable for the next n days and costs f dollars per tablecloth, or at a slower facility that makes tablecloths unavailable for the next m days (with $m > n$) at a cost of g dollars per tablecloth ($g < f$). The caterer's problem is to decide how to meet the client's demand at minimum cost, starting with no tablecloths and under the assumption that any leftover tablecloths have no value.

(a) Show that the problem can be formulated as a network flow problem. *Hint:* Use a node corresponding to clean tablecloths and a node corresponding to dirty tablecloths for each day; more nodes may also be needed.

(b) Show explicitly the form of the network if $N = 5$, $n = 1$, $m = 3$.

Exercise 7.2 Consider a wood product company that owns M forest units and wants to find an optimal cutting schedule over a period of K years. Forest unit i is predicted to have a_{ij} tons of wood available for harvesting during period j. The company wants to meet a demand of d_j tons during year j. However, due to capacity limitations, it can only harvest up to u_j tons during that year. Wood harvested in past years can be stored and used to meet demand in subsequent years, but there is a cost of c_j for storing one ton of wood between year $j - 1$ and j. We also assume that wood that is available but not harvested during a year remains available for harvesting in later years. Formulate the problem of determining a minimum cost harvesting schedule that meets the demand as a network flow problem.

Exercise 7.3 (The tournament problem) Each of n teams plays against every other team a total of k games. Assume that every game ends in a win or a loss (no draws) and let x_i be the number of wins of team i. Let X be the set of all possible outcome vectors $(x_1, \ldots, x_n)$. Given an arbitrary vector $(x_1, \ldots, x_n)$, we would like to determine whether it belongs to X, that is, whether it is a possible tournament outcome vector. Provide a network flow formulation of this problem.

Exercise 7.4 (Piecewise linear convex costs)

(a) Consider the capacitated network flow problem except that the cost at each arc is a piecewise linear convex function of the flow on that arc. Show that the problem can be reduced to one with linear costs but in which we allow multiple arcs with the same start node and end node.

(b) Show that a capacitated problem in which we have multiple arcs with the same start node and end node can be reduced to a problem without any such multiple arcs.

Exercise 7.5 (Equivalence of uncapacitated network flow and transportation problems) Consider an uncapacitated network flow problem and assume that $c_{ij} \geq 0$ for all arcs. Let S_+ and S_- be the sets of source and sink nodes, respectively. Let d_{ij} be the length of a shortest directed path from node $i \in S_+$ to node $j \in S_-$. We construct a transportation problem with the same

source and sink nodes, and the same values for the supplies and the demands. For every source node i and every sink node j, we introduce a direct link with cost d_{ij}. Show that the two problems have the same optimal cost.

Exercise 7.6 (Equivalence of capacitated network flow and transportation problems) Consider a capacitated network flow problem defined by a graph $G = (N, A)$ and the data u_{ij}, c_{ij}, b_i. Assume that the capacity u_{ij} of every arc is finite. We construct a related transportation problem as follows. For every arc $(i,j) \in A$, we form a source node in the transportation problem with supply u_{ij}. For every node $i \in N$, we construct a sink node with demand $\sum_{\{k|(i,k)\in A\}} u_{ik} - b_i$.

At every supply node (i,j) there are two outgoing infinite capacity arcs: one goes to demand node i, and its cost coefficient is 0; the other goes to demand node ... and its cost coefficient is c_{ij}. See Figure 7.36 for an illustration.
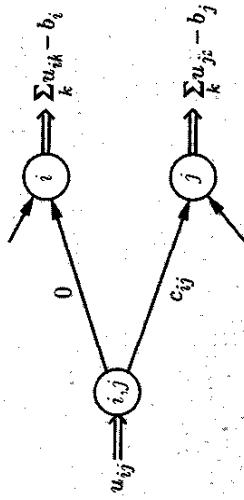
Figure 7.36: The transportation problem in Exercise 7.6.

Show that there is a one-to-one correspondence between feasible flows... in the two problems and that the cost of the two corresponding flows is the same...

Exercise 7.7 (Lower bounds on arc flows) Consider a network flow problem in which we impose an additional constraint $f_{ij} \geq d_{ij}$ for every arc (i,j)... Construct an equivalent network flow problem in which there are no nonzero lower bounds on the arc costs. *Hint:* Let $\bar{f}_{ij} = f_{ij} - d_{ij}$ and construct a new network for the arc flows $\bar{f}_{ij}$. How should b_i be changed?

Exercise 7.8 Consider a transportation problem in which all cost coefficient c_{ij} are positive. Suppose that we increase the supply at some source nodes and the demand at some sink nodes. (In order to maintain feasibility, we assume that the increases are such that total demand is equal to total supply.) Is it true that the value of the optimal cost will also increase? Prove or provide a counterexample.

Exercise 7.9 Consider the uncapacitated network flow problem shown in Figure 7.37. The label next to each arc is its cost.

(a) What is the matrix A corresponding to this problem?

(b) Solve the problem using the network simplex algorithm. Start with the tree indicated by the dashed arcs in the figure.
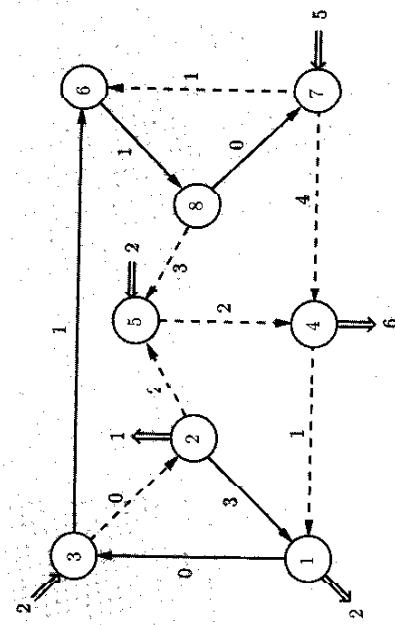
Figure 7.37: The network flow problem in Exercise 7.9.

Exercise 7.10 Consider the uncapacitated network flow problem shown in Figure 7.38. The label next to each arc is its cost. Consider the spanning tree indicated by the dashed arcs in the figure and the associated basic solution.

(a) What are the values of the arc flows corresponding to this basic solution? Is this a basic feasible solution?

(b) For this basic solution, find the reduced cost of each arc in the network.

(c) Is this basic solution optimal?

(d) Does there exist a nondegenerate basic feasible solution?

(e) Find an optimal dual solution.

(f) By how much can we increase c_{56} [the cost of arc (5,6)] and still have the same optimal basic feasible solution?

(g) If we increase the supply at node 1 and the demand at node 9 by a small positive amount δ, what is the change in the value of the optimal cost?

(h) Does this problem have a special structure that makes it simpler than the general uncapacitated network flow problem?

Exercise 7.11 (Degeneracy in a transportation problem) Consider a transportation problem with two source nodes s_1, s_2, and n demand nodes $1, \ldots, n$. All arcs (s_i, j) are assumed to be present and to have infinite capacity. Let $D = \sum_{i=1}^{n} d_i$ be the total demand. Let the supply at each source node be equal to $D/2$.

(a) How many basic variables are there in a basic feasible solution?

(b) Show that there exists a degenerate basic feasible solution if and only if there exists some set $S \subset \{1, \ldots, n\}$ such that $\sum_{i \in S} d_i = D/2$.
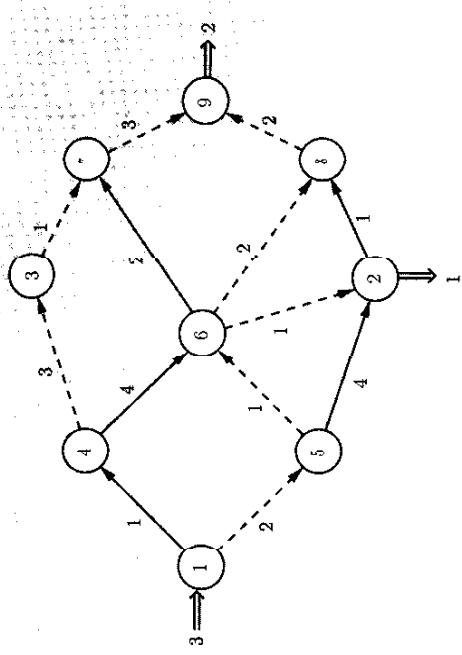
Figure 7.38: The network flow problem in Exercise 7.10.

Exercise 7.12* (**Degeneracy in the assignment problem**) Consider the polyhedron $P \subset \Re^2$ defined by the constraints

$$\sum_{i=1}^k f_{ij} = 1, \qquad j = 1,\ldots,k,$$

$$\sum_{j=1}^k f_{ij} = 1, \qquad i = 1,\ldots,k,$$

$$f_{ij} \geq 0, \qquad i,j = 1,\ldots,k.$$

(a) Show that F has $k!$ basic feasible solutions and that if $k > 1$, every basic feasible solution is degenerate.

(b) Show that there are $2^{k-1}k^{k-2}$ different bases that lead to any given basic feasible solution.

Exercise 7.13 Suppose that we are given a noninteger optimal solution to an uncapacitated network flow problem with integer data.

(a) Show that there exists a cycle with every arc on the cycle carrying a positive flow. What can you say about the cost of such a cycle?

(b) Suggest a method for constructing an integer optimal solution, without solving the problem from scratch. Hint: Remove cycles.

Exercise 7.14 (**Decomposition of circulations**) Let A be the node-arc incidence matrix associated with a directed graph with m arcs. Suppose that a

vector f satisfies $Af = 0$. Show that there exists a nonnegative integer k (with $k \leq m$), cycles $C_1,\ldots,C_k$, and nonnegative scalars $a_1,\ldots,a_k$, such that:

ii) $f = \sum_{i=1}^k a_i h^{C_i}$,

(i) for every arc (k,ℓ) on a cycle C_i, $h^{C_i}_{k\ell}$ and $f_{k\ell}$ have the same sign.

Furthermore, show that if f is an integer vector, then the coefficients $a_1,\ldots,a_k$ can be chosen to be integer. Hint: Reverse the arcs that carry negative flow and apply Lemma 7.1.

Exercise 7.15 (**Flow decomposition theorem**) State and prove a result analogous to the flow decomposition theorem in Exercise 7.14, for the case of a flow vector f that satisfies $Af = b$. Hint: Besides cycles, use paths as well.

Exercise 7.16* (**Negative cost cycle algorithm under the largest improvement rule**) Consider the variant of the negative cost cycle algorithm in which we always choose a cycle C with the largest value of $\delta(C)|c'h^C|$. Let f be the current flow and let f^* be an optimal flow.

(a) Show that $f^* - f$ is equal to a nonnegative linear combination of at most m simple circulations, where m is the number of arcs. Furthermore, each such simple circulation is associated with an unsaturated cycle. Hint: Use the result in Exercise 7.14

(b) Show that under the largest improvement rule, the cost improvement at each iteration is at least $(c'f - c'f^*)/m$.

(c) Assuming that all problem data are integer, show that the algorithm terminates after $O(m\log(mCU))$ iterations, where C and U are upper bounds for $|c_{ij}|$ and u_{ij}, respectively.

Exercise 7.17 Consider a network flow problem and assume that there exists at least one feasible solution. We wish to show that the optimal cost is $-\infty$ if and only if there exists a negative cost directed cycle such that every arc on the cycle has infinite capacity.

(a) Provide a proof based on the flow decomposition theorem.

(b) For uncapacitated problems, provide a proof based on the network simplex method.

Exercise 7.18 Show that there is a one-to-one correspondence between augmenting paths in the maximum flow algorithm and negative cost unsaturated cycles in the network flow formulation of the maximum flow problem.

Exercise 7.19 Consider the maximum flow problem. Describe an algorithm with $O(|A|)$ running time that determines whether the value of the maximum flow is infinite.

Exercise 7.20 (**Duality and the max-flow min-cut theorem**) Consider the maximum flow problem.

(a) Let p_i be a price variable associated with the flow conservation constraint at node i. Let q_{ij} be a price variable associated with the capacity constraint at arc (i,j). Write down a minimization problem, with variables p_i and q_{ij}, whose dual is the maximum flow problem.

(b) Show that the optimal value in the minimization problem is equal to the minimum cut capacity, and prove the max-flow min-cut theorem.

Exercise 7.21 (Finding a feasible solution) Show that a feasible solution to a capacitated network problem (if one exists) can be found by solving a maximum flow problem.

Exercise 7.22 (Connectivity and vulnerability) Consider a directed graph and let us fix an origin node s and a destination node t. We define the *connectivity* of the graph as the maximum number of directed paths from s to t that do not share any nodes. We define the *vulnerability* of the graph as the minimum number of nodes (besides s and t) that need to be removed so that there exists no directed path from s to t. Prove that connectivity is equal to vulnerability. *Hint:* Convert the connectivity problem to a maximum flow problem.

Exercise 7.23 (The marriage problem) A small village has n unmarried men, n unmarried women, and m marriage brokers. Each broker knows a subset of the men and women and can arrange up to b_i marriages between any pair of men and women that she knows. Assuming that marriages are heterosexual and that each person can get married at most once, we are interested in determining the maximum number of marriages that are possible. Show that the answer can be found by solving a maximum flow problem.

Exercise 7.24 * (König-Egerváry theorem) Consider an $m \times n$ matrix whose entries are zero or one. We refer to a row or a column as a *line*. We say that a set of lines is a *cover* if every unit entry lies or one of the lines in the set. A set of unit entries are called *independent* if no two of them lie on the same line. Prove that the maximum cardinality of an independent set is equal to the smallest cardinality of a cover. *Hint:* Formulate an appropriate maximum flow problem.

Exercise 7.25 (The scaling method for the maximum flow problem) This exercise illustrates the *scaling method*, a common technique for reducing the complexity of network flow algorithms.

Consider a maximum flow problem II. Let n be the number of nodes, let u_{ij} be the capacity of arc (i, j), assumed integer, and let v be the value of a maximum flow. We construct a scaled problem II_s in which the capacity of each arc (i, j) is $\lfloor u_{ij}/2 \rfloor$, and we let v_s be the corresponding optimal value. (The notation $\lfloor a \rfloor$ stands for the largest integer k that satisfies $k \le a$.)

(a) Consider an optimal flow for the problem II_s, and multiply it by 2. Show that the result is a feasible flow for the original problem II.

(b) Show that $2v_s \le v \le 2v_s + n^2$.

(c) Consider running the Ford-Fulkerson algorithm on problem II, starting with the feasible flow described in (a). How many flow augmentations will be needed, and what is the total computational effort?

(d) Show how to solve the maximum flow problem with a total of $O(n^4 \log U)$ arithmetic operations, where U is an upper bound on the capacities u_{ij}.

Exercise 7.26 * (Birkhoff-von Neumann theorem) A square matrix **A** is called *doubly stochastic* if $\sum_{i=1}^n a_{ij} = 1$ for all j, $\sum_{j=1}^n a_{ij} = 1$ for all i, and all entries are nonnegative. A matrix **P** is called a *permutation matrix* if each row and each column has exactly one nonzero entry, which is equal to 1.

(a) Let $\mathbf{P}_1, \ldots, \mathbf{P}_k$ be permutation matrices, and let $\lambda_1, \ldots, \lambda_k$ be nonnegative scalars that sum to 1. Show that $\sum_{i=1}^k \lambda_i \mathbf{P}_i$ is doubly stochastic.

(b) Let **A** be a doubly stochastic matrix. Show that there exist permutation matrices $\mathbf{P}_1, \ldots, \mathbf{P}_k$, and nonnegative scalars $\lambda_1, \ldots, \lambda_k$ that sum to 1, such that $\mathbf{A} = \sum_{i=1}^k \lambda_i \mathbf{P}_i$. *Hint:* Consider the assignment problem.

Exercise 7.27 Consider the transportation problem shown in Figure 7.39, and solve it using the primal-dual method. Use $\mathbf{p} = (1, 1, 0, 0)$ to start the algorithm.
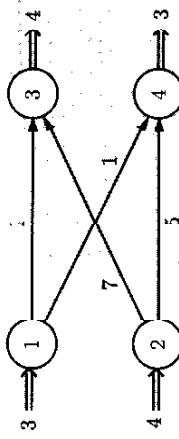
Figure 7.39: The transportation problem in Exercise 7.27. Arc costs are shown next to each arc.

Exercise 7.28 This exercise develops a more efficient method for computing θ^* in the primal-dual method. Let S be the set of nodes whose prices are to increase, as in the description of the general dual ascent algorithm. For every $j \notin S$, let

$$\theta_j^* = \min_{\{i \in S \mid (i,j) \in A\}} (c_{ij} + p_i - p_i).$$

(a) Show that $\theta^* = \min_{j \notin S} \theta_j^*$.

(b) Suppose that some node $k \notin S$ satisfies $\theta_k^* = \theta^*$, so that node k enters the set S subsequent to the price increase. Let

$$\bar{\theta}_j = \min_{\{i \in S \cup \{k\} \mid (i,j) \in A\}} (c_{ij} + p_j - p_i), \qquad j \notin S \cup \{k\}.$$

Show that $\bar{\theta}_j = \min\{\theta_j^*, c_{kj} + p_j - p_k\}$.

(c) Explain how to carry out each dual update in time proportional to n times the number of previously unlabeled nodes that become labeled.

(d) Show that the primal-dual method can be implemented so that it runs in time $O(n^3 B)$, where $B = \max_i |b_i|$.

Exercise 7.29 Consider a bipartite matching problem and suppose that every node has the same degree d. Show that there exists a perfect matching. *Hint:* Convert to a maximum flow problem and use the max-flow min-cut theorem.

Exercise 7.30* (**The primal-dual method as steepest dual ascent**) Consider the dual ascent algorithm. Show that the choice of the set S in the primal-dual method maximizes $(\mathbf{d}^S)'\mathbf{b}$ over all sets S for which $\mathbf{d}^S$ is a feasible direction.

Exercise 7.31 (**Dual simplex method for network flow problems**) Consider the uncapacitated network flow problem.

(a) Show that every spanning tree determines a basic solution to the dual problem.

(b) Given a basic feasible solution to the dual problem, associated with a certain tree, show that it is optimal if and only if the corresponding tree solution to the primal is feasible.

(c) If the tree solution in part (b) is infeasible, remove an arc that carries negative flow. Given that we wish to maintain dual feasibility, how should an arc be chosen to enter the tree?

(d) Note that the entering arc divides the tree into two parts. Consider the dual variables following a dual simplex update. Show that the dual variables in one part of the tree remain unchanged and in the other part of the tree they are all changed by the same amount.

Exercise 7.32 (**Termination of the auction algorithm**) Consider a variation of the assignment problem in which we are given a subset A of the set of person-project pairs, and we allow f_{ij} to be nonzero only if $(i,j) \in A$. We modify the bidding phase of the auction algorithm as follows. A person i takes into consideration only the profits $p_k - c_{ik}$ of those projects k for which $(i,k) \in A$. Suppose that this form of the auction algorithm fails to terminate. Let I be the set of persons that bid an infinite number of times. Let J be the set of projects that receive an infinite number of bids.

(a) Show that if $i \in I$ and $(i,j) \in A$, then $j \in J$.

(b) Show that the cardinality of I is strictly larger than the cardinality of J.

(c) Show the problem must be infeasible.

Exercise 7.33 (**Shortest path lengths and Bellman's equation**) Consider the all-to-one shortest path problem, and let $\mathbf{p}^*$ be the vector of shortest path lengths.

(a) Show that if every (directed) cycle has positive length, then Bellman's equation has a unique solution, equal to the shortest path lengths.

(b) Show that if every (directed) cycle has nonnegative length, and if $\mathbf{p}$ is a solution to Bellman's equation, then $\mathbf{p} \le \mathbf{p}^*$. Hint: Consider $\max\{p_i, p_i^*\}$.

Exercise 7.34 (**From shortest path lengths to shortest paths**) Suppose that all directed cycles in a directed graph have nonnegative costs. Furthermore, suppose that the shortest path length p_i^* from any node to node n is known. Provide an algorithm that uses this information to determine a shortest path from node 1 to node n.

Exercise 7.35 (**Convergence of the Bellman-Ford algorithm**) This exercise develops an alternative proof of the convergence of the Bellman-Ford algorithm. Assume that the length of every cycle is nonnegative.

(a) Prove that $\mathbf{p}(t+1) \le \mathbf{p}(t)$ for all t

(b) Prove that $\mathbf{p}(t) \ge \mathbf{p}^*$ for all t, and conclude that $\mathbf{p}(t)$ has a limit.

(c) Prove that $\mathbf{p}(t)$ can take only a finite number of values and therefore converges.

(d) Prove that the limit satisfies Bellman's equation.

(e) Prove that the algorithm converges to $\mathbf{p}^*$.

Exercise 7.36 (**Minimization of the mean cost of a cycle using linear programming**) Consider a directed graph in which each arc is associated with a cost c_{ij}. For any directed cycle, we define its mean cost as the sum of the costs of its arcs, divided by the number of arcs. We are interested in a directed cycle whose mean cost is minimal. We assume that there exists at least one directed cycle.

Consider the linear programming problem

maximize λ

subject to $p_i + \lambda \le p_j + c_{ij}$, for all arcs (i,j).

(a) Show that this maximization problem is feasible.

(b) Show that if $(\lambda, \mathbf{p})$ is a feasible solution to the maximization problem, then the mean cost of every directed cycle is at least λ.

(c) Show that the maximization problem has an optimal solution.

(d) Show how an optimal solution to the maximization problem can be used to construct a directed cycle with minimal mean cost.

Exercise 7.37 (**Minimization of the mean cost of a cycle using the Bellman-Ford algorithm**) Consider a directed graph in which each arc is associated with a cost c_{ij}. For any directed cycle, we define its mean cost as the sum of the costs of its arcs, divided by the number of arcs. We are interested in a directed cycle whose mean cost is minimal. We assume that there exists at least one directed cycle.

(a) Consider the algorithm

$$p_i(t+1) = \min_{j \in O(i)} \{c_{ij} + p_j(t)\}, \qquad \text{for all } i,$$

initialized with $p_i(0) = 0$ for all i. Show that $p_i(t)$ is equal to the length of a shortest walk that starts at i and and traverses t arcs.

(b) Prove that the optimal mean cycle cost λ satisfies

$$\lambda = \min_{i=1,\dots,n} \max_{0 \le k \le n-1} \left(\frac{p_i(n) - p_i(k)}{n - k} \right),$$

where n is the number of nodes.

Exercise 7.38 (**Floyd-Warshall all-pairs shortest path algorithm**) Consider the all-pairs shortest path problem and assume that there are no negative

linear optimization.max

Chapter 10

Integer programming formulations

Contents

Chapter 10

Integer programming formulations

Contents

In discrete optimization problems, we seek to find a solution x^* in a discrete set F that optimizes (minimizes or maximizes) an objective function $c(x)$ defined for all $x \in F$. Discrete optimization problems arise in a great variety of contexts in science and engineering. A natural and systematic way to study a broad class of discrete optimization problems is to express them as integer programming problems.

The (linear) integer programming problem is the same as the linear programming problem except that some of the variables are restricted to take integer values. In general, given matrices A, B, and vectors b, c, d, the problem

$$\text{minimize} \quad c'x + d'y$$
$$\text{subject to} \quad Ax + By = b$$
$$x, y \geq 0$$
$$x \text{ integer,}$$

is the *mixed integer programming* problem. Notice that even if there are inequality constraints, we can still write the problem in the above form by adding slack or surplus variables. If there are no continuous variables y, the problem is called the *integer programming* problem. If furthermore there are no continuous variables and the components of the vector x are restricted to be either 0 or 1, the problem is called the *zero-one* (or *binary*) *integer programming* problem (ZOIP). Finally, it is customary to assume that the entries of A, B, b, c, d are integers.

Integer programming is a rather powerful modeling framework that provides great flexibility for expressing discrete optimization problems. On the other hand, the price for this flexibility is that integer programming seems to be a much more difficult problem than linear programming. In this chapter, we introduce general guidelines for obtaining "strong" integer programming formulations for discrete optimization problems. We introduce modeling techniques, discuss what constitutes a strong formulation, and compare alternative formulations of the same problem.

10.1 Modeling techniques

In this section, we outline some modeling techniques that facilitate the formulation of discrete optimization problems as integer programming problems. In comparison to linear programming, integer programming is significantly richer in modeling power. Unfortunately, there is no systematic way to formulate discrete optimization problems, and devising a good model is often an art, which we plan to explore through examples.

Binary choice

An important use of a binary variable x is to encode a choice between two alternatives: we may set x to zero or one, depending on the chosen alternative.

Example 10.1 (The zero-one knapsack problem) The knapsack problem was introduced in Chapter 6. We discuss here another variant of the problem, in which the decision variables are constrained to be binary. We are given n items. The jth item has weight w_j and its value is c_j. Given a bound K on the weight that can be carried in a knapsack, we would like to select items to maximize the total value. In order to model this problem, we define a binary variable x_j which is 1 if item j is chosen, and 0 otherwise. The problem can then be formulated as follows:

$$\text{maximize} \quad \sum_{j=1}^{n} c_j x_j$$
$$\text{subject to} \quad \sum_{j=1}^{n} w_j x_j \leq K$$
$$x_j \in \{0,1\}, \qquad j = 1,\ldots,n.$$

Forcing constraints

A very common feature in discrete optimization problems is that certain decisions are dependent. In particular, suppose decision A can be made only if decision B has also been made. In order to model such a situation, we can introduce binary variables x (respectively, y) equal to 1 if decision A (respectively, B) is chosen, and 0 otherwise. The dependence of the two decisions can be modeled using the constraint

$$x \leq y,$$

i.e, if $y = 0$ (decision B is not made), then $x = 0$ (decision A cannot be made). Next, we present an example where forcing constraints are used.

Example 10.2 (Facility location problems) Suppose we are given n potential facility locations and a list of m clients who need to be serviced from these locations. There is a fixed cost c_j of opening a facility at location j, while there is a cost d_{ij} of serving client i from facility j. The goal is to select a set of facility locations and assign each client to one facility, while minimizing the total cost.

In order to model this problem, we define a binary decision variable y_j for each location j, which is equal to 1 if facility j is selected, and 0 otherwise. In addition, we define a binary variable x_{ij}, which is equal to 1 if client i is served by facility j, and 0 otherwise. The facility location problem is then formulated as follows:

$$\text{minimize} \quad \sum_{j=1}^{r} c_j y_j + \sum_{i=1}^{m} \sum_{j=1}^{n} d_{ij} x_{ij}$$
$$\text{subject to} \quad \sum_{j=1}^{r} x_{ij} = 1, \qquad \forall\, i,$$
$$x_{ij} \leq y_j, \qquad \forall\, i, j,$$
$$x_{ij}, y_j \in \{0,1\}, \qquad \forall\, i, j, \tag{10.1}$$
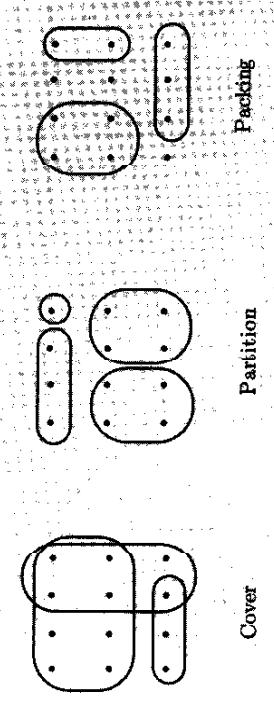
Figure 10.2: A cover, a partition, and a packing.

$$\text{minimize} \quad \sum_{i=1}^{k} \lambda_i f(a_i)$$

$$\text{subject to} \quad \sum_{i=1}^{k} \lambda_i = 1,$$

$$\lambda_1 \leq y_1,$$

$$\lambda_i \leq y_{i-1} + y_i, \qquad i = 2, \ldots, k-1,$$

$$\lambda_k \leq y_{k-1},$$

$$\sum_{i=1}^{k-1} y_i = 1,$$

$$\lambda_i \geq 0,$$

$$y_i \in \{0, 1\}.$$

Notice that if $y_j = 1$, then $\lambda_i = 0$ for i different than j or $j + 1$.

The previous collection of examples is by no means an exhaustive list of possible modeling devices. They only serve to illustrate the power of modeling with binary variables. In order to acquire more confidence, we introduce some more examples.

Example 10.3 (The set covering, set packing, and set partitioning problems) Let $M = \{1, \ldots, m\}$ and $N = \{1, \ldots, n\}$. Let $M_1, M_2, \ldots, M_n$ be a given collection of subsets of M. For example, the collection might consist of all subsets of size at least k. We are also given a weight c_j for each set M_j in the collection. We say that a subset F of N is a *cover* of M if $\cup_{j \in F} M_j = M$. We say that F is a *packing* of M if $M_j \cap M_k$ is empty for all $j, k \in F$, $j \neq k$. We say that F is a *partition* of M if it is both a cover and a packing of M (see Figure 10.2). The *weight* of a subset F of N is defined as $\sum_{j \in F} c_j$.

In the *set covering* problem we would like to find a cover F of *minimum weight*, in the *set packing* problem we would like to find a packing F of *maximum*

weight, while in the *set partitioning* problem both minimization and maximization versions are possible. In order to formulate these problems as integer programming problems, we introduce the $m \times n$ *incidence matrix* A of the family $\{M_j \mid j \in N\}$, whose entries are given by

$$a_{ij} = \begin{cases} 1, & \text{if } i \in M_j, \\ 0, & \text{otherwise} \end{cases}$$

We also define a decision variable x_j, $j = 1, \ldots, n$, which is equal to 1 if $j \in F$, and 0 otherwise. Let $\mathbf{x} = (x_1, \ldots, x_n)$. Then F is a cover, packing, partition if and only if

$$A\mathbf{x} \geq \mathbf{e}, \qquad A\mathbf{x} \leq \mathbf{e}, \qquad A\mathbf{x} = \mathbf{e},$$

respectively, where $\mathbf{e}$ is an m-dimensional vector with all components equal to 1.

The previous formulation types encompass a variety of important problems such as the assignment problem, crew scheduling problems, vehicle routing problems, etc.

A sequencing problem with setup times

A flexible machine can perform m operations, indexed from 1 to m. Each operation j requires a unique tool j. The machine can simultaneously hold B tools in its tool magazine, where $B < m$. Loading or unloading tool j into the machine magazine requires s_j units of setup time. Only one tool at a time can be loaded or unloaded by the machine. At the start of the day, n jobs are waiting to be processed by the machine. Each job i requires multiple operations. Let J_i denote the set of operations required by job i, and assume for simplicity that for all i, $|J_i|$ is no larger than the magazine capacity B of the machine. Before the machine can start processing job i, all the required tools belonging to the set J_i must be setup on the machine. If a tool $j \in J_i$, is already loaded on the machine, we avoid the setup time for tool j. If tool $j \in J_i$ is not already loaded, we must set it up, possibly (if the tool magazine is currently full) after unloading an existing tool that job i does not require. Once the tools are setup, all $|J_i|$ operations of job i are processed. Notice that, because of commonality in tool requirements for different jobs and the limited magazine capacity, the setup time required prior to each job is sequence dependent. We want to formulate an integer programming problem to determine the optimal job sequence that minimizes the total setup time to complete all the jobs. We assume that at the start of the day, the tool magazine is completely empty. We define decision variables that capture the job sequence:

$$x_{ir} = \begin{cases} 1, & \text{if job } i \text{ is the } r\text{th job processed}, \\ 0, & \text{otherwise.} \end{cases}$$

Chap. 10   Integer programming formulations

time algorithm for the matching problem and showed that the convex hull of the integer feasible solutions to the matching problem is given by P_{matching}. For a textbook exposition of matching algorithms see Papadimitriou and Steiglitz (1982), and Nemhauser and Wolsey (1988). Much more information or the traveling salesman problem can be found in Lawler et al. (1985).

Chapter 11

Integer programming methods

Contents

linear optimization.max

Unlike linear programming problems, integer programming problems are very difficult to solve. In fact, no efficient general algorithm is known for their solution. In this chapter, we review algorithms for integer programming problems, we develop a duality theory that facilitates algorithmic development, and discuss evidence suggesting that that these problems are inherently hard.

There are three main categories of algorithms:

(a) **Exact algorithms** that are guaranteed to find an optimal solution, but may take an exponential number of iterations. They include cutting plane (Section 11.1), branch and bound and branch and cut (Section 11.2), and dynamic programming methods (Section 11.3).

(b) **Approximation algorithms** that provide in polynomial time a suboptimal solution together with a bound on the degree of suboptimality (Section 11.5).

(c) **Heuristic algorithms** that provide a suboptimal solution, but without a guarantee on its quality. Although the running time is not guaranteed to be polynomial, empirical evidence suggests that some of these algorithms find a good solution fast. As examples we introduce local search methods (Section 11.6), and simulated annealing (Section 11.7).

Duality theory is central to linear programming. Integer programming also has a duality theory, presented in Section 11.4, which provides bounds on the optimal cost. These bounds are very useful in exact algorithms, as they can be used to avoid enumerating too many feasible solutions, and in approximation algorithms, as they provide performance guarantees.

Given our inability to solve integer programming problems efficiently, it is natural to ask whether such problems are inherently hard. Complexity theory, reviewed in Section 11.8, offers some insights on this question. It provides us with a class of problems with the following property: if a polynomial time algorithm exists for any problem in this class, then all integer programming problems can be solved by an efficient algorithm, but this is considered unlikely.

11.1 Cutting plane methods

We consider the integer programming problem

$$\begin{aligned}
\text{minimize} \quad & c'x \\
\text{subject to} \quad & Ax = b \\
& x \geq 0 \\
& x \text{ integer,}
\end{aligned}$$

(11.1)

and its linear programming relaxation

$$\begin{aligned}
\text{minimize} \quad & c'x \\
\text{subject to} \quad & Ax = b \\
& x \geq 0.
\end{aligned}$$

(11.2)

The main idea in cutting plane methods is to solve the integer programming problem (11.1) by solving a sequence of linear programming problems, as follows. We first solve the linear programming relaxation (11.2) and find an optimal solution x^*. If x^* is integer, then it is an optimal solution to the integer programming problem (11.1). If not, we find an inequality that all integer solutions to (11.1) satisfy, but x^* does not. We add this inequality to the linear programming problem to obtain a tighter relaxation, and we iterate this step.

A generic cutting plane algorithm

1. Solve the linear programming relaxation (11.2). **Let x^* be an optimal solution.**

2. If x^* is integer stop; x^* is an optimal solution to (11.1).

3. If not, add a linear inequality constraint to (11.2) that all integer solutions to (11.1) satisfy, but x^* does not; go to Step 1.

Note that this method is just a variation of the cutting plane algorithm introduced in Section 6.3. As in that section, the main idea is to generate a violated constraint, whenever the relaxed problem gives rise to an infeasible solution. The performance of a cutting plane method depends critically on the choice of the inequality used to "cut" x^*. We review next ways to introduce cuts that give rise to particular cutting plane algorithms.

Example 11.1 (An example of a cut) Let x^* be an optimal basic feasible solution to (11.2) with at least one fractional basic variable. Let N be the set of indices of the nonbasic variables. Consider any solution to the integer programming problem such that $x_i = 0$ for all $i \in N$. Then, it is a solution to the linear programming problem as well, and it must be the same as the basic feasible solution x^*. Since x^* is not feasible for the integer programming problem, then all feasible integer solutions satisfy

$$\sum_{j \in N} x_j \geq 1.$$

This is the inequality that we add to the relaxation (11.2). Note that all integer solutions to (11.1) satisfy it, while the optimal solution x^* to the relaxation violates it.

The Gomory cutting plane algorithm

The first finitely terminating algorithm for integer programming was a cutting plane algorithm proposed by Gomory in 1958, which uses some detailed information from the optimal simplex tableau.

We solve the standard form linear programming problem (11.2) with the simplex method. Let $\mathbf{x}^*$ be an optimal basic feasible solution and let $\mathbf{B}$ be an associated optimal basis. We partition $\mathbf{x}$ into a subvector $\mathbf{x}_B$ of basic variables and a subvector $\mathbf{x}_N$ of nonbasic variables. Recall from Chapter 3 that a tableau provides us with the coefficients of the equation $\mathbf{B}^{-1}\mathbf{A}\mathbf{x} = \mathbf{B}^{-1}\mathbf{b}$. Let N be the set of indices of nonbasic variables. Let $\mathbf{A}_N$ be the submatrix of $\mathbf{A}$ with columns $\mathbf{A}_i$, $i \in N$. From the optimal tableau, we obtain the coefficients of the constraints

$$\mathbf{x}_B + \mathbf{B}^{-1}\mathbf{A}_N\mathbf{x}_N = \mathbf{B}^{-1}\mathbf{b}.$$

Let $\bar{a}_{ij} = (\mathbf{B}^{-1}\mathbf{A}_j)_i$, and $\bar{a}_{i0} = (\mathbf{B}^{-1}\mathbf{b})_i$. We consider one equality from the optimal tableau, in which $\bar{a}_{i0}$ is fractional:

$$x_i + \sum_{j \in N} \bar{a}_{ij} x_j = \bar{a}_{i0}.$$

Since $x_j \geq 0$ for all j, we have

$$x_i + \sum_{j \in N} \lfloor \bar{a}_{ij} \rfloor x_j \leq x_i + \sum_{j \in N} \bar{a}_{ij} x_j = \bar{a}_{i0}.$$

Since x_j should be integer, we obtain

$$x_i + \sum_{j \in N} \lfloor \bar{a}_{ij} \rfloor x_j \leq \lfloor \bar{a}_{i0} \rfloor.$$

This inequality is valid for all integer solutions, but it is not satisfied by $\mathbf{x}^*$. The reason is that $x_i^* = \bar{a}_{i0}$, $x_j^* = 0$ for all nonbasic $j \in N$, and $\lfloor \bar{a}_{i0} \rfloor < \bar{a}_{i0}$ (since $\bar{a}_{i0}$ was assumed fractional)

It has been shown that by systematically adding these cuts, and using the dual simplex method with appropriate anticycling rules, we obtain a finitely terminating algorithm for solving general integer programming problems. See Section 5.1 on how to apply the dual simplex method, when new inequality constraints are added. In practice, however, this method has not been particularly successful.

Example 11.2 (Illustration of the Gomory cutting plane algorithm) We consider the integer programming problem

minimize $\quad x_1 - 2x_2$
subject to $\quad -4x_1 + 5x_2 \leq 9$
$\qquad\qquad x_1 + x_2 \leq 4$
$\qquad\qquad x_1, x_2 \geq 0$
$\qquad\qquad x_1, x_2$ integer.
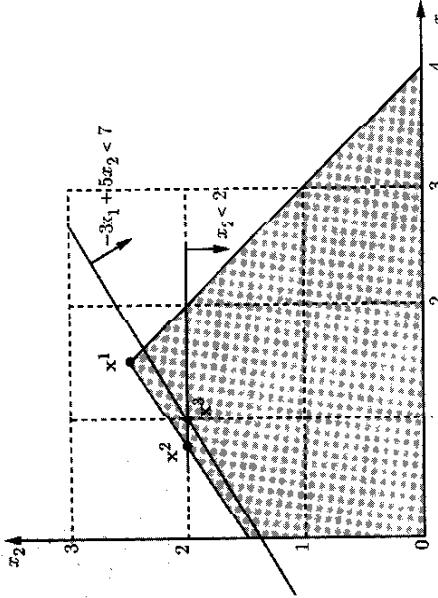
Figure 11.1: The Gomory cutting plane algorithm for Example 11.2. The shaded region is the feasible set of the linear programming relaxation.

We transform the problem in standard form

minimize $\quad x_1 - 2x_2$
subject to $\quad -4x_1 + 6x_2 + x_3 \qquad = 9$
$\qquad\qquad x_1 + x_2 \qquad + x_4 = 4$
$\qquad\qquad x_1, \ldots, x_4 \geq 0$
$\qquad\qquad x_1, \ldots, x_4$ integer.

We solve the linear programming relaxation, and the optimal solution (in terms of the original variables) is $\mathbf{x}^1 = (15/10, 25/10)$. One of the equations in the optimal tableau is

$$x_2 + \frac{1}{10}x_3 + \frac{1}{10}x_4 = \frac{25}{10}.$$

We apply the Gomory cutting plane algorithm, and we find the cut

$$x_2 \leq 2.$$

We augment the linear programming relaxation by adding the constraints $x_2 + x_5 = 2$, $x_5 \geq 0$, and we find that the new optimal solution is $\mathbf{x}^2 = (3/4, 2)$. One of the equations in the optimal tableau is

$$x_1 - \frac{1}{4}x_3 + \frac{6}{4}x_5 = \frac{3}{4}.$$

We add a new Gomory cut

$$x_1 - x_3 + x_5 \leq 0,$$

which, in terms of the original variables x_1, x_2, is

$$-3x_1 + 5x_2 \leq 7.$$

We add this constraint, together with the previously added constraint $x_2 \leq 2$, and find that the new optimal solution is $\mathbf{x}^3 = (1,2)$. Since the solution $\mathbf{x}^3$ is integer, it is an optimal solution to the original problem; see Figure 11.1.

A difficulty with general purpose cutting plane algorithms is that the added inequalities cut only a very small piece of the feasible set of the linear programming relaxation. As a result, the practical performance of such algorithms has not been impressive. For this reason, cutting plane algorithms with deeper cuts have been designed. These cuts utilize the structure of the particular integer programming problem. We illustrate such methods with an example.

Example 11.3 (The weighted independent set problem) Given an undirected graph $G = (\mathcal{N}, \mathcal{E})$ and weights w_i for each $i \in \mathcal{N}$, the weighted independent set problem asks for a collection of nodes S of maximum weight, so that no two nodes in S are adjacent. We let $x_i = 1$ if node i is selected in the independent set, and $x_i = 0$, otherwise. The problem can then be formulated as follows:

$$\text{maximize} \quad \sum_{i=1}^{n} w_i x_i$$

$$\text{subject to} \quad x_i + x_j \leq 1, \qquad (i,j) \in \mathcal{E},$$
$$x_i \in \{0,1\}, \qquad i \in \mathcal{N}.$$

A collection of nodes U such that for any $i, j \in U$ we have $(i,j) \in \mathcal{E}$, is called a *clique*. Clearly the following inequality is valid for all feasible solutions to the independent set problem:

$$\sum_{i \in U} x_i \leq 1, \quad \text{for any clique } U.$$

A set of nodes $U = \{i_1, \ldots, i_k\}$ is called a *cycle* if the only edges joining nodes in U are $\{i_1 \, i_2\}, \{i_2, i_3\}, \ldots, \{i_k, i_1\}$. For any cycle U of odd cardinality, there can be no more that $(|U| - 1)/2$ nodes in an independent set; otherwise, two of these nodes will be adjacent. Therefore, the inequality

$$\sum_{i \in U} x_i \leq \frac{|U| - 1}{2}, \quad \text{for any cycle } U \text{ such that } |U| \text{ is odd,}$$

must hold.

The inequalities we derived above utilize the particular combinatorial structure of the maximum independent set problem. If we use these inequalities in the generic cutting plane method we described, the performance of the algorithm is greatly enhanced. However, given an $\mathbf{x}^*$, we must search for a violated inequality of either type, which can be difficult.

11.2 Branch and bound

Branch and bound uses a "divide and conquer" approach to explore the set of feasible integer solutions. However, instead of exploring the entire feasible set, it uses bounds on the optimal cost to avoid exploring certain parts of the set of feasible integer solutions.

Let F be the set of feasible solutions to the problem

$$\text{minimize} \quad \mathbf{c}'\mathbf{x}$$
$$\text{subject to} \quad \mathbf{x} \in F.$$

[For example, F could be the set of integer feasible solutions to the problem (11.1).] We partition the set F into a finite collection of subsets $F_1, \ldots, F_k$, and solve separately each one of the subproblems

$$\text{minimize} \quad \mathbf{c}'\mathbf{x}$$
$$\text{subject to} \quad \mathbf{x} \in F_i, \quad i = 1, \ldots, k.$$

We then compare the optimal solutions to the subproblems, and choose the best one. Each subproblem may be almost as difficult as the original problem and this suggests trying to solve each subproblem by means of the same method; that is, by splitting it into further subproblems, etc. This is the branching part of the method and leads to a tree of subproblems; see Figure 11.2.
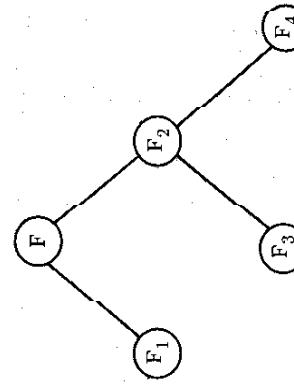

Figure 11.2: A tree of subproblems: the feasible set F is partitioned into F_1 and F_2; also, F_2 is partitioned into F_3 and F_4.

We also assume that there is a fairly efficient algorithm, which for every F_i of interest, computes a *lower bound* $b(F_i)$ to the optimal cost of the corresponding subproblem; that is,

$$b(F_i) \leq \min_{\mathbf{x} \in F_i} \mathbf{c}'\mathbf{x}.$$

The basic idea is that while the optimal cost in a subproblem may be difficult to compute exactly, a lower bound might be a lot easier to obtain.

A popular method to obtain such a bound is to use the optimal cost of the linear programming relaxation.

In the course of the algorithm, we will also occasionally solve certain subproblems to optimality, or simply evaluate the cost of certain feasible solutions. This allows us to maintain an upper bound U on the optimal cost, which could be the cost of the best feasible solution encountered thus far.

The essence of the method lies in the following observation. If the lower bound $b(F_i)$ corresponding to a particular subproblem satisfies $b(F_i) \geq U$, then this subproblem need not be considered further, since the optimal solution to the subproblem is no better than the best feasible solution encountered thus far.

The following is a high-level description of the resulting algorithm. At any point, the algorithm keeps in memory a set of outstanding (active) subproblems and the cost U of the best feasible solution so far. Initially, U is set either to ∞ or to the cost of some feasible solution, if one happens to be available. A typical stage of the algorithm proceeds as follows.

A generic branch and bound algorithm

1. Select an active subproblem F_i.
2. If the subproblem is infeasible, delete it; otherwise, compute $b(F_i)$ for the corresponding subproblem.
3. If $b(F_i) \geq U$, delete the subproblem.
4. If $b(F_i) < U$, either obtain an optimal solution to the subproblem, or break the corresponding subproblem into further subproblems, which are added to the list of active subproblems.

There are several "free parameters" in this algorithm. The best choices are usually dictated by experience.

(a) There are different ways of choosing an active subproblem. Two extreme choices are "breadth-first search" and "depth-first search."

(b) There may be several ways of obtaining a lower bound $b(F_i)$ on the optimal cost of a subproblem. One possibility that we have already mentioned is to consider the linear programming relaxation. We consider other possibilities in Section 11.4.

(c) There are usually several ways of breaking a problem into subproblems.

As an illustration, we use as a lower bound $b(F_i)$ the optimal cost of the linear programming relaxation whereby the integrality constraints are ignored. If an integer optimal solution to the relaxation is obtained, then it is automatically an optimal solution to the corresponding integer programming problem as well, and there is no need for expanding into
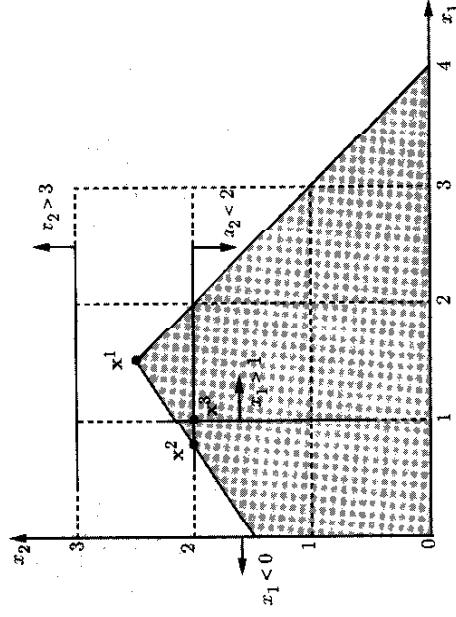
Figure 11.3: Branch and bound in Example 11.4.

further subproblems. We only need to update U (if the cost of this optimal solution is better than the previous value of U), and we can delete the current subproblem. If the optimal solution x^* to the linear programming relaxation is not integer we choose a component x_i for which x_i^* is not integer and create two subproblems, by adding either of the constraints

$$x_i \leq \lfloor x_i^* \rfloor, \qquad \text{or} \qquad x_i \geq \lceil x_i^* \rceil.$$

(Note that both constraints are violated by x^*. If x^* is the unique optimal solution to the linear programming relaxation, then the optimal cost in the relaxation of either of the new subproblems will be strictly larger.) Given that a subproblem differs from its parent only in the fact that a single new constraint has been added, we can solve the linear programming relaxation of a subproblem by means of the dual simplex method, starting from x^*. We may then expect than an optimal solution to the new linear programming problem will be obtained after only a small number of iterations.

Example 11.4 (Illustration of branch and bound) We solve the problem of Example 11.2 by branch and bound; see Figure 11.3. Initially, $U = \infty$. We solve the linear programming relaxation and the optimal solution is $x^1 = (1.5, 2.5)$. Then, $b(F)$ is the optimal cost of the relaxation, i.e., $b(F) = -3.5$. We create two subproblems, by adding the constraints $x_2 \geq 3$ (subproblem F_1), or $x_2 \leq 2$ (subproblem F_2). The active list of subproblems is $\{F_1, F_2\}$. The linear programming relaxation of subproblem F_1 is infeasible and, therefore, we can delete this subproblem from the active list. The optimal solution to the linear programming
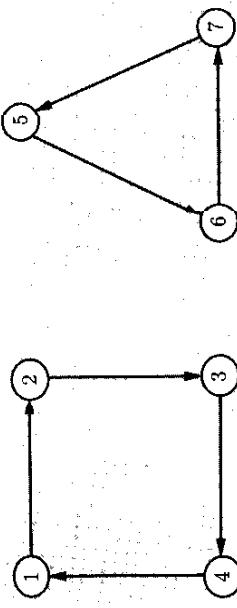
relaxation of subproblem F_2 is $\mathbf{x}^2 = (3/4, 2)$, and thus $b(F_2) = -3.25$. We further decompose subproblem F_2 into two subproblems, since either $x_1 \geq 1$ (subproblem F_3), or $x_1 \leq 0$ (subproblem F_4). The active list of subproblems is now $\{F_3, F_4\}$. The optimal solution to the linear programming relaxation of Subproblem F_3 is $\mathbf{x}^3 = (1, 2)$, which is integer and therefore, $U = -3$. We delete subproblem F_3 from the active list. The optimal solution to the linear programming relaxation of subproblem F_4 is $\mathbf{x}^4 = (0, 3/2)$, and thus $b(F_4) = -3$. Since $b(F_4) \geq U$, we do not need to further explore subproblem F_4. Since the active list of subproblems is empty, we terminate. The optimal integer solution is $\mathbf{x}^3 = (1, 2)$.

Example 11.5 (A branch and bound method for the directed traveling salesman problem) Given a directed graph $G = (N, A)$ with n nodes, and a cost c_{ij} for every arc, we want to solve the traveling salesman problem on G using branch and bound. The objective is to find a tour (a directed cycle that visits all nodes) of minimum cost. We let x_{ij} equal to 1, if i and j are consecutive nodes in a tour, and 0, otherwise. The optimal cost in the problem

$$\text{minimize} \quad \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij}$$

$$\text{subject to} \quad \sum_{i=1}^{n} x_{ij} = 1, \qquad j = 1,\ldots,n,$$

$$\sum_{j=1}^{n} x_{ij} = 1, \qquad i = 1,\ldots,n,$$

$$x_{ij} \in \{0,1\},$$

provides a lower bound on the cost of an optimal tour, because every tour must satisfy the above constraints. We recognize this as an assignment problem. However, not every feasible solution to the assignment problem corresponds to a tour, and for this reason the optimal costs for the two problems are not the same. In particular, an optimal solution to the assignment problem may correspond to a collection of "subtours"; see Figure 11.4.

Suppose now that we use the assignment problem to obtain a lower bound on the cost of the traveling salesman problem. If the optimal solution to the assignment problem corresponds to a tour, such a tour is optimal for the traveling salesman problem. If not, we split the problem into subproblems. Each additional subproblem involves a single additional constraint of the form $x_{ij} = 0$. This is equivalent to prohibiting (i,j) from being consecutive nodes in a tour, and can be also accomplished by setting c_{ij} to a prohibitively high value. Note that adding such a constraint to the traveling salesman or to the assignment problem, still leaves us with a traveling salesman or assignment problem, respectively. Therefore, all subproblems constructed in the course of the branch and bound algorithm will also correspond to instances of the traveling salesman problem and lower bounds can be obtained by solving the related assignment problems. The only remaining issue is how to decide which constraints $x_{ij} = 0$ to add. A natural alternative is to choose one or more subtours and let each subproblem

Figure 11.4: Consider a directed traveling salesman problem with seven nodes. The vector $\mathbf{x}$ corresponding to these two subtours is a feasible solution to the assignment problem.

prohibit one of their arcs. For example, if the optimal solution to the assignment problem is as in Figure 11.4, we can create subproblems by adding one of the constraints $x_{12} = 0, x_{23} = 0, x_{34} = 0, x_{41} = 0, x_{56} = 0, x_{67} = 0, x_{75} = 0$. If the current assignment problem has a unique optimal solution, this solution is made infeasible by the constraints that are added during branching. For this reason, the optimal cost in each subproblem is strictly larger, and improved lower bounds are obtained.

It should be clear that the success of branch and bound methods depends critically on the availability of tight lower bounds. (In Section 11.4 we introduce a duality theory for integer programming that leads to such bounds.) While the branch and bound algorithm may take exponential time in the worst case (see Exercise 11.4), it often produces acceptable solutions in a reasonably short amount of time, especially when tight lower bounds are available.

Branch and cut

Another variant of the method, often called *branch and cut*, utilizes cuts when solving the subproblems. In particular, we augment the formulation of the subproblems with additional cuts, in order to improve the bounds obtained from the linear programming relaxations. We illustrate the method with an example.

Example 11.6 (Illustration of branch and cut) We solve the problem of Example 11.2 by branch and cut. We first solve the linear programming relaxation and find the optimal solution $\mathbf{x}^1 = (15, 2.5)$. As before, we create subproblems F_1 (corresponding to $x_2 \geq 3$) and F_2 (corresponding to $x_2 \leq 2$). We delete subproblem F_1, because its linear programming relaxation is infeasible. In order to solve subproblem F_2, we add the constraint $-x_1 + x_2 \leq 1$ which is satisfied by all integer solutions to subproblem F_2. The optimal solution to the linear programming relaxation is now $\mathbf{x} = (1,2)$, which is integer, and thus we terminate

with the optimal solution. Note that by adding the cut $-x_1 + x_2 \leq 1$, we avoid further enumeration. This is typical in branch and cut. If we can add "deep" cuts, we can accelerate branch and bound considerably. However, finding such cuts is nontrivial.

11.3 Dynamic programming

In the previous section, we introduced branch and bound, which is an exact intelligent enumerative technique that attempts to avoid enumerating a large portion of the feasible integer solutions. In this section, we introduce another exact technique, called dynamic programming, that solves integer programming problems sequentially.

We illustrate the method by deriving a dynamic programming algorithm for the traveling salesman problem. We will then discuss general principles on how to develop dynamic programming algorithms for other integer programming problems.

Example 11.7 (A dynamic programming algorithm for the traveling salesman problem) Let $G = (N, A)$ be a directed graph with n nodes and let c_{ij} be the cost of arc (i, j). We view the choice of a tour as a sequence of choices; we start at node 1; then, at each stage, we choose which node to visit next. After a number of stages, we have visited a subset S of N and we are at a current node $k \in S$. Let $C(S, k)$ be the minimum cost over all paths that start at node 1, visit all nodes in the set S exactly once, and end up at node k. If we call (S, k) a state, this state can be reached from any state of the form $(S \setminus \{k\}, m)$, with $m \in S \setminus \{k\}$, at a transition cost of c_{mk}. Thus, $C(S, k)$ can be interpreted as the least possible sum of transition costs, over all sequences of transitions that take us from state $(\{1\}, 1)$ to state (S, k). Therefore, we have the recursion

$$C(S, k) = \min_{m \in S \setminus \{k\}} \left(C(S \setminus \{k\}, m) + c_{mk} \right), \qquad k \in S, \qquad (11.3)$$

and $C(\{1\}, 1) = 0$. There are 2^n choices for S, $O(n)$ choices for k, and a total of $O(n2^n)$ states (S, k). Each time that $C(S, k)$ is evaluated for some new state according to Eq. (11.3), $O(n)$ arithmetic operations are needed. Therefore, with $O(n^2 2^n)$ operations, we can obtain $C(\{1, \ldots, n\}, k)$ for all k. The length of an optimal tour is then given by

$$\min_k \left(C(\{1, \ldots, n\}, k) + c_{k1} \right).$$

This algorithm, although exponential, is much better than exhaustive enumeration of all $n!$ tours. Realistically, it can only be used to solve instances of the traveling salesman problem involving up to 20 nodes.

More generally, devising a dynamic programming algorithm for an integer programming problem involves the following steps.

Guidelines for constructing dynamic programming algorithms

1. View the choice of a feasible solution as a sequence of decisions occurring in stages, and so that the total cost is the sum of the costs of individual decisions.

2. Define the state as a summary of all relevant past decisions.

3. Determine which state transitions are possible. Let the cost of each state transition be the cost of the corresponding decision.

4. Write a recursion on the optimal cost from the origin state to a destination state.

The most crucial step is usually the definition of a suitable state. Let us apply the method to another problem.

A dynamic programming algorithm for the zero-one knapsack problem

Let us consider the version of the zero-one knapsack problem we introduced in Example 10.1:

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

$$\text{subject to} \quad \sum_{j=1}^n w_j x_j \leq K$$

$$x_j \in \{0, 1\}.$$

We assume that K and all c_j, w_j are positive integers. We derive a dynamic programming algorithm for the zero-one knapsack problem by decomposing it into stages. Instead of picking a vector $(x_1, \ldots, x_n)$ all at once, we visualize the problem as one in which decisions are made for one item at a time. After i decisions, we have decided which ones out of the first i items are to be included in the knapsack, and have therefore determined values for the variables $x_1, \ldots, x_i$. At that point, the value accumulated is $\sum_{j=1}^i c_j x_j$ and the weight accumulated is $\sum_{j=1}^i w_j x_j$.

Let $W_i(u)$ be the least possible weight that has to be accumulated in order to attain a total value of u using only items in the set $\{1, \ldots, i\}$. Let $W_i(u) = \infty$, if it is impossible to accumulate a total value of u using only the first i items. We use the convention $W_0(0) = 0$, and $W_0(u) = \infty$, if $u \neq 0$, which reflects the fact that the value accumulated using no items is zero. We then have the following recursion:

$$W_{i+1}(u) = \min \{W_i(u), \; W_i(u - c_{i+1}) + w_{i+1}\}. \qquad (11.4)$$

In words, this recursion means the following. If we wish to accumulate a total value of u, using some of the first $i + 1$ items. while accumulating as

little weight as possible, there are two alternatives depending on whether item $i+1$ is used or not. If item $i+1$ is not used, then the best we can do is to accumulate a total value of u, while using only some of the first i items and do that with the least possible accumulated weight, which is $W_i(u)$. Alternatively, if item $i+1$ is used, since it has a value of c_{i+1}, we must have accumulated a total value of $u - c_{i+1}$ using the first i items. Of course, the first i decisions should be made so that the value $u - c_{i+1}$ is accumulated with the least possible weight, which is $W_i(u-c_{i+1})$, and to which we must then add the weight of item $i+1$. We can now interpret recursion (11.4) as stating that $W_{i+1}(u)$ is given by the best of the two alternatives that we have just described.

We continue with a slightly different interpretation of recursion (11.4). Let us say that we are at *state* (i,u) if we have considered the first i items, have picked some of them, and have accumulated a total value of u. We then build a state transition diagram indicating which states can be reached from which other state. Notice that when in state (i,u) we can either decide to pick item $i+1$ and move to state $(i+1, u+c_{i+1})$ or we can decide to skip item $i+1$ and move to state $(i+1, u)$. We represent states as nodes and possible transitions by directed arcs; see Figure 11.5.
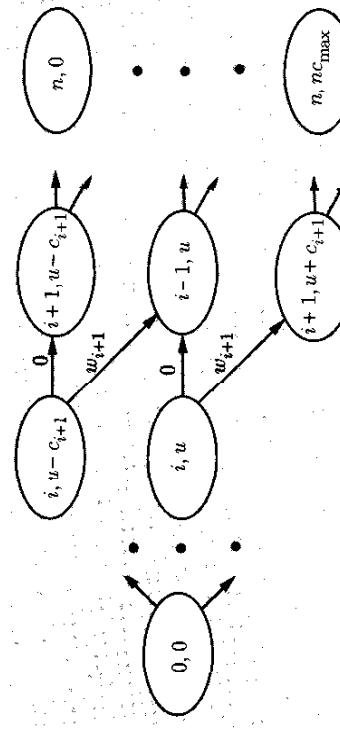
Figure 11.5: The state transition diagram for the dynamic programming approach to the zero-one knapsack problem.

In addition, we associate a weight to each arc (or transition) which is the additional weight added in the course of this transition. Thus, the transition from (i,u) to $(i+1, u)$ carries zero weight, while the transition from (i,u) to $(i+1, u+c_{i+1})$ carries weight w_{i+1}.

Initially, we are at state $(0,0)$; no item has been considered and no value has been accumulated. A sequence of decisions, involving items $1, \ldots, i$ corresponds to a directed path from node $(0,0)$ to some node of the form (i,u). Furthermore, the sum of the weights along the path corresponds to the accumulated weight. We conclude that $W_i(u)$ is equal to the least weight of all paths from node $(0,0)$ to node (i,u), and is equal

to infinity if no such path exists. We may then recognize Eq. (11.4) as the Bellman equation associated with this shortest path problem (see Section 7.9).

Let
$$c_{max} = \max_{i=1,\ldots,n} c_i.$$

If $u > nc_{max}$, then no state of the form (i,u) is reachable. By restricting to states of the form (i,u) with $u \leq nc_{max}$, we see that the total number of states of interest is of the order of $n^2 c_{max}$. Using recursion (11.4), the value of $W_i(u)$ for all states of interest, can be computed in time $O(n^2 c_{max})$. Once this is done, the optimal value u^* is given by

$$u^* = \max\{u \mid W_n(u) \leq K\},$$

which can be determined with only an additional $O(nc_{max})$ effort. Optimal values for the variables $x_1, \ldots, x_n$ are then determined by an optimal path from node $(0,0)$ to node (n, u^*). We have thus proved the following result.

Theorem 11.1 *The zero-one knapsack problem can be solved in time* $O(n^2 c_{max})$.

An alternative, and somewhat more natural, dynamic programming algorithm for the same problem could be obtained by defining $C_i(w)$ as the maximum value that can be accumulated using some of the first i items subject to the constraint that the total accumulated weight is equal to w. We would then obtain the recursion

$$C_{i+1}(w) = \max\{C_i(w), C_i(w - w_{i+1}) + c_{i+1}\}.$$

By considering all states of the form (i,w) with $w \leq K$, an algorithm with complexity $O(nK)$ would be obtained. However, our previous algorithm is better suited to the purposes of developing an approximation algorithm, which will be done in Section 11.5.

The algorithm of Theorem 11.1 is an exponential time algorithm. This is because the size of an instance of the zero-one knapsack problem is

$$O\big(n(\lg c_{max} + \log w_{max}) + \log K\big),$$

where $w_{max} = \max_i w_i$. However, it becomes polynomial if c_{max} is bounded by some polynomial in n. More formally, for any integer d, we can define the problem KNAPSACK(d), as the problem consisting of all instances of the zero-one knapsack problem with $c_i \leq n^d$ for all i. According to Theorem 11.1, KNAPSACK(d) can be solved in time $O(n^{d+2})$, which is polynomial for every fixed d.

11.4 Integer programming duality

In this section, we develop the duality theory of integer programming. This in turn leads to a method for obtaining tight bounds, that are particularly useful for branch and bound. The methodology is closely related to the subject of Section 4.10, but our discussion here is self-contained.

We consider the integer programming problem

$$
\begin{aligned}
\text{minimize} \quad & \mathbf{c'x} \\
\text{subject to} \quad & \mathbf{Ax} \geq \mathbf{b} \\
& \mathbf{Dx} \geq \mathbf{d} \\
& \mathbf{x} \text{ integer,}
\end{aligned}
\tag{11.5}
$$

and assume that $\mathbf{A}, \mathbf{D}, \mathbf{b}, \mathbf{c}, \mathbf{d}$ have integer entries. Let Z_{IP} the optimal cost and let

$$X = \{\mathbf{x} \text{ integer} \mid \mathbf{Dx} \geq \mathbf{d}\}.$$

In order to motivate the method, we assume that optimizing over the set X can be done efficiently; for example X may represent the set of feasible solutions to an assignment problem. However, adding the constraints $\mathbf{Ax} \geq \mathbf{b}$ to the problem, makes the problem difficult to solve. We next consider the idea of introducing a dual variable for every constraint in $\mathbf{Ax} \geq \mathbf{b}$. Let $\mathbf{p} \geq \mathbf{0}$ be a vector of dual variables (also called *Lagrange multipliers*) that has the same dimension as the vector $\mathbf{b}$. For a fixed vector $\mathbf{p}$, we introduce the problem

$$
\begin{aligned}
\text{minimize} \quad & \mathbf{c'x} + \mathbf{p'}(\mathbf{b} - \mathbf{Ax}) \\
\text{subject to} \quad & \mathbf{x} \in X,
\end{aligned}
\tag{11.6}
$$

and denote its optimal cost by $Z(\mathbf{p})$. We will say that we *relax* or *dualize* the constraints $\mathbf{Ax} \geq \mathbf{b}$. For a fixed $\mathbf{p}$, the above problem can be solved efficiently, as we are optimizing a linear objective over the set X. We next observe that $Z(\mathbf{p})$ provides a bound on Z_{IP}.

Lemma 11.1 *If the problem (11.5) has an optimal solution and if $\mathbf{p} \geq \mathbf{0}$, then $Z(\mathbf{p}) \leq Z_{IP}$.*

Proof. Let $\mathbf{x}^*$ denote an optimal solution to (11.5). Then, $\mathbf{b} - \mathbf{Ax}^* \leq \mathbf{0}$ and, therefore,

$$\mathbf{c'x}^* + \mathbf{p'}(\mathbf{b} - \mathbf{Ax}^*) \leq \mathbf{c'x}^* = Z_{IP}.$$

Since $\mathbf{x}^* \in X$,

$$Z(\mathbf{p}) \leq \mathbf{c'x}^* + \mathbf{p'}(\mathbf{b} - \mathbf{Ax}^*),$$

and therefore, $Z(\mathbf{p}) \leq Z_{IP}$. $\qquad\Box$

Since problem (11.6) provides a lower bound to the integer programming problem (11.5) for all $\mathbf{p} \geq \mathbf{0}$, it is natural to consider the tightest such

bound. For this reason, we introduce the problem

$$
\begin{aligned}
\text{maximize} \quad & Z(\mathbf{p}) \\
\text{subject to} \quad & \mathbf{p} \geq \mathbf{0}.
\end{aligned}
\tag{11.7}
$$

We will refer to problem (11.7) as the *Lagrangean dual.* Let

$$Z_D = \max_{\mathbf{p} \geq 0} Z(\mathbf{p}).$$

Suppose for instance, that $X = \{\mathbf{x}^1, \ldots, \mathbf{x}^m\}$. Then $Z(\mathbf{p})$ can be also written as

$$Z(\mathbf{p}) = \min_{i=1,\ldots,m} \left(\mathbf{c'x}^i + \mathbf{p'}(\mathbf{b} - \mathbf{Ax}^i) \right). \tag{11.8}$$

The function $Z(\mathbf{p})$ is concave and piecewise linear, since it is the minimum of a finite collection of linear functions of $\mathbf{p}$ (see Theorem 1.1 in Section 1.3 and Figure 11.6). As a consequence, the problem of computing Z_D [namely, problem (11.7)] can be recast as a linear programming problem, but with a very large number of constraints.

It is clear from Lemma 11.1 that weak duality holds:

Theorem 11.2 *We have $Z_D \leq Z_{IP}$.*

The previous theorem represents the weak duality theory of integer programming. Unlike linear programming, integer programming does not have a strong duality theory. (Compare with Theorem 4.18 in Section 4.10.)

Figure 11.6: The function $Z(\mathbf{p})$ is concave and piecewise linear.

Indeed in Example 11.8, we show that it is possible to have $Z_D < Z_I$. The procedure of obtaining bounds for integer programming problems b calculating Z_D is called *Lagrangean relaxation*. We next investigate t quality of the bound Z_D, in comparison to the one provided by the line programming relaxation of problem (11.5).

On the strength of the Lagrangean dual

The characterization (11.8) of the Lagrangean dual objective does not pr vide particular insight into the quality of the bound. A more revealir characterization is developed in this subsection. Let $CH(X)$ be the conve hull of the set X. We need the following result, whose proof is outline in Exercise 11.8. Since we already know that the convex hull of a fini set is a polyhedron, this result is of interest when the set $\{x \mid Dx \geq d\}$ unbounded and the set X is infinite.

Theorem 11.3 *We assume that the system of linear inequalities $Dx \geq$ d has a feasible solution, and that the matrix D and the vector d ha integer entries. Let*

$$X = \{x \text{ integer} \mid Dx \geq d\}.$$

Then $CH(X)$ is a polyhedron.

The next theorem, which is the central result of this section, charac- terizes the Lagrangean dual as a linear programming problem.

Theorem 11.4 *The optimal value Z_D of the Lagrangean dual is equal to the optimal cost of the following linear programming problem:*

minimize $c'x$

subject to $Ax \geq b$

$\qquad\qquad x \in CH(X)$. (11.9)

Proof. By definition,

$$Z(p) = \min_{x \in X}(c'x + p'(b - Ax)).$$

Since the objective function is linear in x, the optimal cost remains the same if we allow convex combinations of the elements of X. Therefore,

$$Z(p) = \min_{x \in CH(X)}(c'x + p'(b - Ax)),$$

and hence, we have

$$Z_D = \max_{p \geq 0} \min_{x \in CH(X)} (c'x + p'(b - Ax)).$$

Let x^k, $k \in K$, and w^j, $j \in J$, be the extreme points and a complete set of extreme rays of $CH(X)$, respectively. Then, for any fixed p, we have

$$Z(p) = \begin{cases} -\infty, & \text{if } (c - p'A)w^j < 0, \\ & \text{for some } j \in J, \\ \min_{k \in K}(c'x^k + p'(b - Ax^k)), & \text{otherwise.} \end{cases}$$

Therefore, the Lagrangean dual is equivalent to and has the same optimal value as the problem

maximize $\min_{k \in K}(c'x^k + p'(b - Ax^k))$

subject to $(c' - p'A)w^j \geq 0$, $\quad j \in J$,

$\qquad\qquad p \geq 0$,

or equivalently,

maximize y

subject to $y + p'(Ax^k - b) \leq c'x^k$, $\qquad k \in K$,

$\qquad\qquad p'Aw^j \leq c'w^j$, $\qquad j \in J$,

$\qquad\qquad p \geq 0$.

Taking the linear programming dual of the above problem, and using strong duality, we obtain that Z_D is equal to the optimal cost of the problem

minimize $c'\left(\displaystyle\sum_{k \in K} \alpha_k x^k + \sum_{j \in J} \beta_j w^j\right)$

subject to $\displaystyle\sum_{k \in K} \alpha_k = 1$

$\qquad\qquad A\left(\displaystyle\sum_{k \in K} \alpha_k x^k + \sum_{j \in J} \beta_j w^j\right) \geq b$

$\qquad\qquad \alpha_k, \beta_j \geq 0$, $\quad k \in K, j \in J$.

Since,

$$CH(X) = \left\{ \sum_{k \in K} \alpha_k x^k + \sum_{j \in J} \beta_j w^j \ \Big| \ \sum_{k \in K} \alpha_k = 1, \ \alpha_k, \beta_j \geq 0, \ k \in K, j \in J \right\},$$

the result follows. □

Chapter 12

The art in linear optimization

Contents

linear optimization.max

In previous chapters, we developed the theory of linear optimization. In this chapter, we turn our attention to the art in linear optimization, i.e., to the process of modeling, exploiting problem structure, and fine tuning of optimization algorithms.

In recent years, the availability of workstations and optimization libraries has advanced optimization capabilities significantly. Large scale linear optimization problems arising in practice involve thousands (sometimes millions) of variables and constraints. Therefore, these constraints must be described efficiently. Towards this goal, special modeling languages have been developed. We briefly discuss modeling languages in Section 12.1. In Section 12.2, we mention some powerful linear optimization libraries, with particular emphasis on the size of the problems they can solve. We also make some general observations on the relative merits of different methods.

Especially for large scale problems, it is important to utilize their special structure in order to solve them efficiently. Although there are some general approaches (cf. Chapter 6), solving large scale problems involves imaginative modeling as well as creative use of optimization libraries. In Section 12.3, we illustrate the art in using optimization algorithms, in the context of the fleet assignment problem, a large scale integer programming problem in air transportation. In Section 12.4, we illustrate the art in modeling linear optimization problems, in the context of controlling air traffic in a network of airports. In Section 12.5, we illustrate the art of combining formulations, optimization algorithms, and heuristics, in the context of job shop scheduling.

12.1 Modeling languages for linear optimization

Most realistic problems involve a large number of variables and constraints. As a result, it would be cumbersome to form a linear optimization problem such as

$$\text{minimize} \quad c'x$$
$$\text{subject to} \quad Ax = b$$
$$x \geq 0$$
$$x \text{ integer},$$

by entering the entries of A, b, and c one at a time. Practical linear optimization problems, however, involve classes of constraints that follow a particular pattern. For example, in the assignment problem, one set of constraints can be described compactly as

$$\sum_{i=1}^{n} x_{ij} = 1, \qquad j = 1, \dots, n.$$

Modeling languages are software packages that recognize compact descriptions of formulations and output the matrix A and the vectors b, c.

Although conceptually simple, these programs are very useful as they significantly cut the time from model conception to actual solution. Modeling languages are then connected with particular solvers that on input A, b, c, output an optimal solution to the problem. Two particular modeling languages are GAMS and AMPLE.

Example 12.1 If $n = 100$, the previous family of constraints for the assignment problem can be written in GAMS as:

```
SET
    I /1 * 100/
    J /1 * 100/
VARIABLES
    X(I,J);
EQUATIONS
    Constraint(J);
    Constraint(J)...SUM(I, X(I,J)) = E = 1
```

12.2 Linear optimization libraries and general observations

There are over 200 different commercial linear optimization libraries in the marketplace. These libraries vary by their degree of flexibility, user-friendliness, ability to handle large problems, platforms, support, etc. We mention three of them that have been used to solve large scale linear optimization problems. Our choice of libraries is only for illustration and does not imply that they are better than others.

(a) OSL, which is an optimization subroutine library from IBM; its linear programming code is an implementation of the simplex method. It also includes a branch and bound algorithm for integer programming.

(b) OB1, which is an implementation of interior point methods (variations of the path following methods discussed in Sections 9.4 and 9.5).

(c) CPLEX, which contains implementations of the simplex method, interior point algorithms, and a branch and bound algorithm for integer programming.

These software packages run on a wide variety of computational environments (PCs, workstations, mainframes, supercomputers). In order to give some insight into the size of problems these codes can solve, we report in Table 12.1 examples of the performance of these software packages that have been reported in the literature (all examples are linear programming problems). The reason for presenting these examples, which involve very different problems, is to only offer a rough indication of the size of problems that have been successfully solved in practice. However, the reader should be warned that the running times of the same package on different problems of the same size could be vastly different.

Code	Constraints	Variables	Time	Computer
OSL	105,000	155,000	240 mins	IBM 3090
OSL	750	12,000,000	27 mins	IBM 3090
OB1	10,000	233,000	12 mins	IBM 3090
CPLEX	145	1,000,000	6 mins	Cray Y-MP
CPLEX	41,000	79,000	3 mins	Cray 2

Table 12.1: Sample performance of various linear programming codes.

Some key problem parameters that affect performance are the number of constraints and the sparsity of the constraint matrix A. The running times are also significantly affected by implementation issues like the numerical linear algebra techniques used, the data structures employed, how sparsity is exploited, etc.

We now present some general, mostly empirical, guidelines regarding the performance of various algorithms:

(a) The a priori availability of a basis greatly affects the performance of the simplex method. If we can start with a basis, significantly fewer iterations may be required, particularly if a problem is similar to a previously solved one. Even if the problem is significantly different, using an advanced basis might help. For example, if we are combining several smaller models into a larger one, using an optimal basis from one of the smaller problems can help the solution to the larger problem greatly.

(b) Many linear programming problems solve faster using the dual simplex method rather than primal simplex. In particular, highly degenerate problems with little variability in the right-hand side coefficients, but significant variability in the cost coefficients, solve much faster using the dual simplex method.

(c) If a linear programming problem has some portion with a network structure, the network simplex method can be often used to obtain quickly a good initial basis for the larger problem.

(d) Certain large, sparse problems solve faster using interior point methods. Examples include problems with over 1000 rows or columns, containing a relatively small number of nonzeros per column, and problems with staircase or banded structures in the constraint matrix. In general, the performance of interior point methods is highly

dependent on the number of nonzeros in the Cholesky factors, and is also affected by the presence of columns with a relatively high number of nonzero entries.

Simplex and interior point methods have different numerical properties, sensitivities, and behavior. We next discuss some of their major differences

(a) Solutions provided by the simplex method are always basic, whereas optimal solutions found by an interior point algorithm are not necessarily basic. In the case of multiple optimal solutions, interior point algorithms find a solution in the interior of the set of optimal solutions. Therefore, when an interior point algorithm is used alone, we do not obtain a basis that can be used for reoptimization. In practice, it is rare that one wants to solve only one instance of an optimization problem. Typically, one would like to solve a series of instances of a problem that are small variations of each other. In that case, the simplex method has an advantage compared with an interior point algorithm. Also, since an interior point solution is not basic, there is no information available for sensitivity analysis. A common solution technique for large scale, sparse problems is to use an interior point algorithm to find an optimal solution, convert it to an optimal basic feasible solution, and then use the simplex method for sensitivity analysis and reoptimization.

(b) Interior point algorithms are sensitive to the presence of an unbounded set of optimal solutions, whereas the simplex method is not. On the other hand, the simplex method is sensitive to the presence of degeneracy, whereas interior point algorithms are less so.

(c) Simplex and interior point methods have different memory requirements. Interior point algorithms can require significantly more memory than the simplex method, depending on the sparsity of the Cholesky factors.

We revisit most of these issues in the next sections.

12.3 The fleet assignment problem

In this section, we illustrate the art in using optimization algorithms, in the context of the fleet assignment problem, a large scale integer programming problem in air transportation. Given a flight schedule and a set of aircraft of different types, the fleet assignment problem faced by an airline is to determine which type of aircraft should fly each flight segment on the airline's daily (or weekly) schedule. These strategic decisions have a major impact on revenue. For this reason, many airlines around the world have devoted a lot of resources to solving this problem. We describe some re-

lated research that offers general insights into many of the issues we have introduced throughout the book.

We use the following notation. There is a set of available fleets, i.e., aircraft types, denoted by $\mathcal{F}$. The number of aircraft available in fleet $f \in \mathcal{F}$ is denoted by $S(f)$. There is also a given schedule. The set of cities served by the schedule is denoted by C. The set of flights in the schedule is denoted by $\mathcal{L}$, with elements (o, d, t), where $o, d \in C$ represent the origin and the destination of the flight, respectively, and t represents time of scheduled departure. There are costs c_{fodt} for assigning an aircraft from fleet f to the flight (o, d, t). We pick a particular reference time t_0 (for example 3 a.m. eastern standard time). Time is partitioned into intervals of equal size. We have a sequence of times $t_1, \ldots, t_n$, and we assume that arrivals and departures only happen at these discrete instances. We use t^- to denote the time preceding t, and t^+ to denote the following time. We let $t(f, o, d)$ be the time fleet f takes to travel from the origin o to the destination d. We let $O(t_0)$ be the set of all flights $(o, d, t) \in \mathcal{L}$ that are flying during the interval $[t_0, t_0^+]$, which we assume to be fixed ahead of time. There is also a set $\mathcal{H}$ of pairs of flights that must be performed by an aircraft of the same fleet. These flights are called "required through." In the following discussion, we will ignore issues related to maintenance and crew planning, so that we can focus on the most important issues.

The objective is to assign an aircraft from some fleet f to each flight (o, d, t) so as to minimize the total cost. For every $f \in \mathcal{F}$ and $(o, d, t) \in \mathcal{L}$, we introduce the following decision variables:

$$x_{fodt} = \begin{cases} 1, & \text{if fleet } f \text{ is used for the flight from } o \text{ to } d \\ & \quad \text{departing at time } t, \\ 0, & \text{otherwise,} \end{cases}$$

$y_{fot} =$ number of aircraft on the ground from fleet f that stay at city o during the interval $[t, t^+]$,

$z_{fot} =$ number of aircraft from fleet f that arrive at city o at time t.

The variables z_{fot} and x_{fodt} are related as follows:

$$z_{fot} = \sum_{\{(d,o,\tau) \in \mathcal{L} | \tau + t(f,d,o) = t\}} x_{fdo\tau}$$

The model can be formulated as an integer programming problem:

$$\text{minimize} \quad \sum_{f \in \mathcal{F}} \sum_{(o,d,t) \in \mathcal{L}} c_{fodt} x_{fodt}$$

subject to the constraints

$$\sum_{f \in \mathcal{F}} x_{fodt} = 1, \qquad \forall (o, d, t) \in \mathcal{L},$$

$$z_{fot} + y_{fot^-} - \sum_{d \in C} x_{fodt} - y_{fot} = 0, \qquad \forall f, o, t,$$

$$x_{fodt} - x_{fdd't'} = 0, \qquad \forall f \in \mathcal{F}, \\ \forall ((o, d, t), (d, d', t')) \in \mathcal{H},$$

$$\sum_{(o,d,t) \in O(t_0)} x_{fodt} + \sum_{o \in C} y_{fot_0} \leq S(f), \qquad \forall f \in \mathcal{F},$$

$$x_{fodt} \in \{0, 1\},$$
$$y_{fo} \geq 0,$$
$$y_{fot} \text{ integer.}$$

The first set of constraints requires that each flight should be flown by exactly one fleet. The second set of constraints represents conservation of flow of aircraft. The third set of constraints enforces that an aircraft of the same type flies both legs of "required through" flights. The fourth set of constraints requires that the total number of aircraft in fleet f that are either flying or are on the ground at the reference time t_0 is at most $S(f)$. Because of flow conservation, if this set of constraints is satisfied for one time period, it will be satisfied for all time periods.

We consider three problem instances. The first is a hypothetical smaller instance that is used to test various algorithmic ideas, while the other two are real instances faced by Delta airlines in their daily schedules. We first discuss the solution to the linear programming relaxation. The sizes are reported in Table 12.2.

Inst.	Fleets	Flights	Var.	Rows	Col.	Nonzeros
A	4	1709	6636	13689	17148	42371
B	11	2589	22679	47994	65254	159064
C	11	2589	22746	48209	65164	163472

Table 12.2: Problem sizes for fleet assignment instances. The last column indicates the number of nonzero elements of the constraint matrix **A**.

linear optimization.max

The effect of preprocessing

Several optimization codes have an algebraic preprocessing option that reduces the size of the problem by eliminating variables whose values are fixed by other variables. For example, a constraint $y = \sum_i a_i x_i$, allows y to be removed from the problem, along with this constraint. Preprocessing can also identify empty rows or columns, and can eliminate redundant rows. The effect of preprocessing in instance A is shown in Table 12.3. The table also reports solution times for the linear programming relaxation of the problem on an IBM RS/6000 Model 320, using OSL Release 2 with primal simplex. It can be seen that preprocessing can dramatically decrease the size of the problem and the computation time, which decreases by a factor of four.

Instance A	Rows	Columns	Iterations	CPU seconds
no prepr.	13689	17148	39429	10094
prepr.	5579	9508	18975	2381

Table 12.3: Preprocessing can dramatically decrease the size and the computation times.

Which simplex algorithm?

As we have seen in earlier chapters, one can use either the primal or the dual simplex algorithm, as well as different pivoting rules to solve linear programming problems. We next illustrate that the use of different simplex variants can have a significant impact on the computation times. In Table 12.4, we report computational results for instances E and C, using three simplex variants, known as:

(a) The primal devex simplex.

(b) The primal steepest edge simplex.

(c) The dual steepest edge simplex.

The computations were performed on an IBM RS/6000 Model 550, using OSL. While the two primal simplex variants are very close, the dual steepest edge simplex algorithm takes approximately half as many iterations and half the time.

Simplex versus interior point methods

The results of Chapter 9 suggest that interior point algorithms take a significantly smaller number of iterations to find an optimal solution. In Table

Simplex variant	Iterations Inst. B	Time Inst. B	Iterations Inst. C	Time Inst. C
Primal Devex	33101	3257.5	29463	2779.9
Primal SE	33097	3194.5	32811	3199.1
Dual SE	15408	1431.8	14954	1461.5

Table 12.4: The effect of different pivoting rules for different simplex variants. The time is in CPU seconds.

12.5, we report computational results for all instances using OSL's interior point code, which is a variant of the path following method covered in Sections 9.4 and 9.5.

The number of iterations of the interior point algorithm is significantly smaller than the number of iterations of any simplex variant. While the computation time is smaller than that of the primal simplex variants, the dual steepest edge simplex is still faster than the interior point method. However, this conclusion might depend on the structure of the particular instances.

Instance	Iterations	Time
A	32	213.3
B	38	2141.8
C	39	2205.2

Table 12.5: The performance of interior point methods.

Avoiding degeneracy through perturbation

The fleet assignment problem is vastly primal and dual degenerate. In order to improve performance, we can randomly perturb the cost vector. This reduces the degree of dual degeneracy and improves the performance of the dual simplex algorithm. The performance of the interior point algorithm is generally not affected.

The crossover problem

Due to dual degeneracy, and when the cost vector was not perturbed, the problem had many optimal solutions. The interior point algorithm did not

Rule	Nodes in B&B Inst. E	Time Inst. B	Nodes in B&B Inst. C	Time Inst. C
D	2000+	6743.3	499	809.3
R_1	46	258.9	141	703.2
R_2	96	591.2	60	249.3

Table 12.6: The effect of various branching rules in the branch and bound code. Rule D refers to the default option, while R_1 and R_2 refer to two branching rules that are particular to the fleet assignment problem. The time reported here is the time spent for branch and bound, after obtaining an optimal basic feasible solution to the linear programming relaxation.

(7) Use optimizer's preprocessing to further reduce the size.
(8) Run the dual steepest edge simplex method.
(9) Branch and bound

Notice that in both cases, we use the simplex method to reoptimize. As a general rule, the simplex method has a clear advantage whenever we start with near optimal solutions.

Tables 12.7 and 12.8 illustrate the performance of the overall algorithm using the dual simplex method and the interior point algorithm, respectively. Notice that the degree of suboptimality, defined as $(Z - Z_{LP})/Z_{LP}$, where Z is the cost of the integer solution found, and Z_{LP} is the cost of the linear programming relaxation, is extremely small regardless of the method of solution, and the computational times are quite reasonable. In Tables 12.7 and 12.8, the column % IP-LP represents $100 \times (Z - Z_{LP})/Z_{LP}$.

Inst.	Iterations Dual SE	Time Dual SE	Time in B&B	% IP-LP	Total Time
B	15351	1501.8	3360.8	0.020	5027.9
C	14177	1376.0	636.5	0.012	2176.3

Table 12.7: The performance of the overall algorithm, using the dual steepest edge simplex method.

This example offers several insights that seem to have wide applicability. We summarize them next:

converge to a basic feasible solution, but rather to an interior point of the set of optimal solutions to the linear programming relaxation. For problems of this type, it has been observed that an optimal basic feasible solution has many variables equal to either zero or one. Thus, if we could extract a basic feasible solution, we would have made progress towards an integer solution. For this reason, it is desirable to extract an optimal basic feasible solution from the optimal interior solution, which is known as the *crossover problem*.

Because OSL allows its simplex algorithm to begin with a nonbasic solution, the code does solve the crossover problem. The interior point algorithm needed 32 iterations and 213.3 CPU seconds to solve instance A. The crossover problem was solved by OSL's primal simplex method in 5873 iterations and 284 seconds. In this example, the time to solve the crossover problem exceeded the time to solve the original problem by an interior point method, but this may not hold in general.

Given that we are interested in an integer solution, the following rounding scheme was used. If a variable is larger than 0.99, it is fixed to 1. If we first fix these variables, then use the preprocessing routines to fix other variables and remove redundant rows, and finally use the crossover routines, the crossover time goes down from 284 seconds to an astonishingly fast 2.4 seconds.

The effect of heuristics in branch and bound

After finding an optimal basic feasible solution to the linear programming relaxation and fixing to 1 the variables that were at least 0.99, we still need to find an integer solution for the remaining variables. For this purpose, the branch and bound routine of OSL was used. One could use the default branching strategies provided by the package or develop specialized branching rules. Table 12.6 compares the default option and two heuristic branching rules. It can be seen that different branching rules that exploit the structure of the problem can significantly affect performance.

Overall performance

The overall algorithms, using the dual steepest edge (respectively, the path following) algorithm, are as follows:

(1) Use optimizer's preprocessing.
(2) Perturb all costs.
(3) Run the dual steepest edge simplex (respectively, the path following) method.
(4) Remove perturbation.
(5) Reoptimize with original cost.
(6) Fix variables with value at least 0.99, to 1.

Inst.	Time of Interior	Time of Simplex	Time in B&B	% IP-LP	Total Time
B	2141.8	15.33	258.9	0.013	2551.6
C	2205.2	27.11	703.2	0.012	3069.4

Table 12.8: The performance of the overall algorithm based on the interior point algorithm.

(a) Preprocessing can significantly decrease the size of a model and, therefore, drastically improve performance.

(b) Different simplex variants have quite different behavior. Dual steepest edge simplex seems to perform better. In several codes, it is the default option.

(c) Interior point algorithms take few iterations to converge. As the size of the problem increases, they seem to perform better than many, but not all, simplex variants.

(d) As degeneracy can affect performance of the simplex method adversely, perturbation of the cost vector decreases (and often eradicates) dual degeneracy, which improves the performance of the dual simplex method.

(e) Especially for solving integer programming problems, it is important to obtain an optimal basic feasible solution to the linear programming relaxation. Fixing some variables first in an optimal interior solution, using preprocessing again, and then calling the simplex algorithm to find an optimal basic feasible solution seems advantageous.

(f) Fixing some variables in an optimal solution to the linear programming relaxation to zero or one, and using brancing rules tailored to the particular problem at hand, leads to significant computational advantages.

12.4 The air traffic flow management problem

While the previous section illustrates the art in developing practical optimization algorithms, this section illustrates the art in developing effective linear optimization models, in the context of the problem of controlling air traffic.

Throughout the United States and Europe, demand for airport use has been increasing rapidly in recent years, while airport capacity has been

stagnating. Acute congestion in many major airports has been the unfortunate result. For example, each of the thirty-three major airports in the US was expected to exceed 20,000 hours of annual delays by 1997. The ground and airborne delays caused by congestion create direct costs to the airlines and indirect (opportunity) costs to the passengers. Direct costs from ground delays include crew, maintenance, and depreciation costs, while direct costs from airborne delays include, in addition, fuel and depreciation costs. Although estimates of congestion costs are difficult to measure, there seems to be agreement[1] that they amount to billions of dollars. Given that several European and US airlines have been suffering yearly losses that also amount to billions of dollars[2], congestion is a problem of undeniable practical significance.

Faced with the realities of congestion, the Federal Aviation Administration (FAA) has been using *ground-holding* policies to reduce delay costs. These short-term policies consider airport capacities and flight schedules as fixed for a given time period, and adjust the flow of aircraft on a real-time basis by imposing "ground holds" on certain flights. The FAA uses a computerized procedure based on a first-come, first-served rule, in order to select appropriate ground holds. These selections are further enhanced through the experience of its air traffic controllers. The motivation for ground-holding is as follows. Suppose it has been determined that if an aircraft departs on time, it will encounter congestion, incurring an airborne delay as it awaits landing clearance at its destination airport. However, by delaying its departure, the aircraft could arrive at its destination at a later time when minimal congestion is expected, thus, incurring no airborne delay. As airborne delays are more costly than ground-holding delays, the objective of ground-holding policies is to "translate" anticipated airborne delays to the ground by delaying departures.

In this section, we consider the effect of using a linear optimization approach to control air traffic in a network of airports that could represent either the national US network or the European network. The airspace is divided into sectors. Each flight passes through contiguous sectors while it is en route to its destination. There is a restriction on the number of airplanes that may fly within a sector at any given time. This number is dependent on the number of aircraft that an air traffic controller can manage at any one time, the geographic location, and the weather conditions. We will refer to the restrictions on the number of aircraft in a given sector at a given time as the en route sector capacities. We formulate the problem of minimizing the effects of congestion as a 0-1 integer programming problem.

Consider a set of flights, $\mathcal{F} = \{1, \ldots, F\}$, a set of airports, $\mathcal{K} = \{1, \ldots, K\}$, a set of time periods of unit duration, $\mathcal{T} = \{1, \ldots, T\}$ (note

---

[1]For example, The Stanford Research Institute estimated in 1990 annual costs of $5 billion in Europe due to congestion. The Federal Aviation Administration estimated in 1988 the annual costs of delays in the US at $1.4 billion.

[2]During the years 1991-1994, US airlines sustained losses totaling about $9 billion.

that we have discretized time), a set of sectors $\mathcal{J} = \{1, \ldots, J\}$, and a set of pairs of flights that are continuations of each other, $C = \{(f', f) \mid f'$ is continued by flight $f\}$. We assume that $\mathcal{K} \subset \mathcal{J}$. In particular, the first and last sector in every flight's path is an airport. Associated with every continued flight f is a turnaround time s_f, which is the minimum time that an airplane needs to stay on the ground in order to be prepared for the next flight. We shall refer to any particular time period t as the "time t." Note that by "flight," we mean a "flight leg" between two airports. Also, flights referred to as "continued" are those flights whose aircraft is scheduled to perform a later flight within some time interval of its scheduled arrival. The problem input data are as follows:

Data:

N_f = number of sectors in the path of flight f

$$P(f, i) = \begin{cases} \text{the departure airport,} & \text{if } i = 1, \\ \text{the arrival airport,} & \text{if } i = N_f, \\ \text{the } i\text{th sector in flight } f\text{'s path,} & \text{if } i \neq \{1, N_f\} \end{cases}$$

$P_f = (P(f,1), \ldots, P(f, N_f))$

$D_k(t)$ = departure capacity of airport k at time t

$A_k(t)$ = arrival capacity of airport k at time t

$S_j(t)$ = capacity of sector j at time t

d_f = scheduled departure time of flight f

r_f = scheduled arrival time of flight f

s_f = turnaround time for flight f

c_f^g = cost of holding flight f on the ground for one time period

c_f^a = cost of holding flight f in the air for one time period

l_{fj} = minimum number of time periods that flight f must spend in sector j

T_f^j = set of allowed times for flight f to arrive at sector j

$\underline{T}_f^j$ = first time period in the set T_f^j

$\overline{T}_f^j$ = last time period in the set T_f^j

Objective:

The objective is to decide how long each flight is to be held on the ground and in the air, in order to minimize the total delay cost.

Decision variables:

A very important aspect of the modeling process is the choice of the decision variables. For every flight f, sector j, and time t, we introduce the following

---
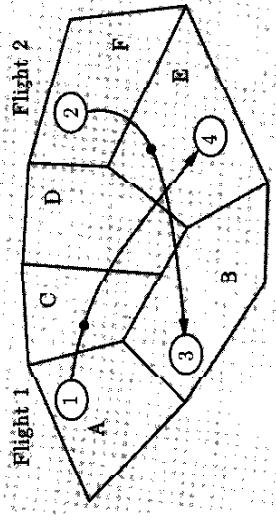
Figure 12.1: Two possible flight routes.

decision variables:

$$w_{ft}^j = \begin{cases} 1, & \text{if flight } f \text{ arrives at sector } j \text{ by time } t, \\ 0, & \text{otherwise.} \end{cases}$$

Recall that the first and last sectors on each flight path are airports. So, if $j = P(f, 1)$, then w_{ft}^j equals 1, if flight f takes off from airport $P(f, 1)$ by time t, while if $j = P(f, N_f)$, then w_{ft}^j equals 1, if flight f lands at airport $P(f, N_f)$ by time t. The above definition, using *by time* t and not *at time* t, is critical to the understanding of the formulation and the practical success of the model. Also, recall that we have defined for each flight a list P_f including the departure airport, the pertinent sectors, and the arrival airport, so that a variable w_{ft}^j will only be needed for those elements j in the list P_f. Moreover, we have defined T_f^j as the set of feasible times for flight f to arrive at sector j, so that a variable w_{ft}^j will only be needed for those times within T_f^j. Thus, whenever the variable w_{ft}^j is used in the formulation, it is assumed that (f, j, t) is a feasible combination.

Example 12.2 To ensure the clarity of the model, consider two flights traversing a set of sectors; see Figure 12.1. In this example, there are two flights, 1 and 2, each with the following associated data:

$$P_1 = (1, A, C, D, E, 4) \quad \text{and} \quad P_2 = (2, F, E, D, B, 3).$$

If the position of the aircraft at time t is indicated by the dots in the figure, then the variables for these flights at that time will be:

$$w_{1,t}^1 = 1, \quad w_{1,t}^A = 1, \quad w_{1,t}^C = 1, \quad w_{1,t}^D = 0, \quad w_{1,t}^E = 0, \quad w_{1,t}^4 = 0,$$
$$w_{2,t}^2 = 1, \quad w_{2,t}^F = 1, \quad w_{2,t}^E = 1, \quad w_{2,t}^D = 0, \quad w_{2,t}^B = 0, \quad w_{2,t}^3 = 0.$$

Having defined the variables w_{ft}^j, we can express several quantities of interest as linear functions of these variables:

(a) The variable u_{ft}^j defined to be 1 if flight f arrives at sector j at time t, and 0 otherwise, can be expressed as follows:

$$u_{ft}^j = w_{ft}^j - w_{f,t-1}^j,$$

and vice versa,

$$w_{ft}^j = \sum_{t' \le t} u_{ft'}^j.$$

As discussed earlier, the variables w_{ft}^j are only defined in the time range $T_f^j = [\underline{T}_f^j, \overline{T}_f^j]$, so that $w_{f,\underline{T}_f^j-1}^j = 0$. Furthermore, one variable per flight-sector pair can be eliminated from the formulation by setting $w_{f,\overline{T}_f^j}^j = 1$. Since flight f has to arrive at sector j by the last possible time in its time window, we can simply set it equal to 1 before solving the problem.

(b) Noticing that $P(f,1)$ represents the departure airport for flight f, the total number g_f of time units that flight f is held on the ground is the actual departure time minus the scheduled departure time, ie.,

$$g_f = \sum_{\{t \in T_f^k \mid k=P(f,1)\}} t u_{ft}^k - d_f$$

$$= \sum_{\{t \in T_f^k \mid k=P(f,1)\}} t(w_{ft}^k - w_{f,t-1}^k) - d_f.$$

(c) Noticing that $P(f,N_f)$ represents the destination airport for flight f, the total number a_f of time units that flight f is held in the air can be expressed as the actual arrival time minus the scheduled arrival time minus the amount of time that the flight has been held on the ground, ie.,

$$a_f = \sum_{\{t \in T_f^k \mid k=P(f,N_f)\}} t u_{ft}^k - r_f - g_f$$

$$= \sum_{\{t \in T_f^k \mid k=P(f,N_f)\}} t(w_{ft}^k - w_{f,t-1}^k) - r_f - g_f.$$

The objective function:

The objective is to minimize total delay cost. Using the above defined variables g_f and a_f (the ground and air delay, respectively), the objective function can be expressed simply as follows:

$$\text{minimize} \sum_{f \in \mathcal{F}} (c_f^g g_f + c_f^a a_f).$$

A 0-1 integer programming formulation

Substituting the expressions we derived above for the variables g_f and a_f in terms of w_{ft}^j, omitting those terms that do not depend on the decision variables, and rearranging, we obtain the following formulation:

$$\text{minimize} \sum_{f \in \mathcal{F}} \left[(c_f^g - c_f^a) \sum_{\{t \in T_f^k \mid k=P(f,1)\}} t(w_{ft}^k - w_{f,t-1}^k) \right.$$
$$\left. + c_f^a \sum_{\{t \in T_f^k \mid k=P(f,N_f)\}} t(w_{ft}^k - w_{f,t-1}^k) \right]$$

subject to the constraints

$$\sum_{\{f \mid P(f,1)=k\}} (w_{ft}^k - w_{f,t-1}^k) \le D_k(t), \qquad \forall k \in \mathcal{K},\ t \in \mathcal{T},$$

$$\sum_{\{f \mid P(f,N_f)=k\}} (w_{ft}^k - w_{f,t-1}^k) \le A_k(t), \qquad \forall k \in \mathcal{K},\ t \in \mathcal{T},$$

$$\sum_{\{(f,j') \mid P(f,i)=j, P(f,i+1)=j', i<N_f\}} (w_{ft}^j - w_{ft}^{j'}) \le S_j(t), \qquad \forall j \in \mathcal{J},\ t \in \mathcal{T}$$

$$w_{f,t+l_{f_j}}^j - w_{ft}^j \le 0, \qquad \begin{cases} \forall f \in \mathcal{F},\ t \in T_f^j,\ j=P(f,i),\ \text{such} \\ \text{that } i < N_f,\ \text{and } j'=P(f,i+1), \end{cases}$$

$$w_{ft}^k - w_{f,t-s_{f'}}^k \le 0, \qquad \begin{cases} \forall (f',f) \in \mathcal{C},\ t \in T_f^k,\ \text{such that} \\ k=P(f,1)=P(f',N_f), \end{cases}$$

$$w_{f,t}^j - w_{f,t-1}^j \ge 0, \qquad \forall f \in \mathcal{F},\ i \in P_f,\ t \in T_f^j,$$
$$w_{ft}^j \in \{0,1\}, \qquad \forall f \in \mathcal{F},\ i \in P_f,\ t \in T_f^j.$$

The first three sets of constraints take into account the capacities of various aspects of the system. The first set of constraints ensures that the number of flights which may take off from airport k at time t, will not exceed the departure capacity of airport k at time t. Likewise, the second set of constraints ensures that the number of flights which may arrive at airport k at time t, will not exceed the arrival capacity of airport k at time t. In each case, the difference $w_{ft}^k - w_{f,t-1}^k$ will be equal to 1, only when the first term is 1 and the second term is 0. Thus, the differences $w_{ft}^k - w_{f,t-1}^k$ capture the time at which a flight uses a given airport. The third set of constraints ensures that the sum of all flights which may be in sector j at time t will not exceed the capacity of sector j at time t. The difference

$$\sum_{f,j} (w_{ft}^j - w_{ft}^{j'})$$

is the number of flights that are in sector j at time t, since the first term will be 1 if flight f has arrived at sector j by time t, and the second term

will be 1 if flight f has arrived at the next sector by time t. So the only flights that will contribute a value of 1 to this sum are the flights that have arrived at j and have not yet departed by time t.

The fourth, fifth, and sixth sets of constraints represent the three types of connectivity in the problem: connectivity between sectors, connectivity within airports, and connectivity in time. The fourth set of constraints represents connectivity between sectors. It stipulates that if a flight arrives at sector j' by time $t + l_{fj}$, then it must have arrived at sector j by time t, where j and j' are consecutive sectors in the path of flight f. In other words, a flight cannot enter the next sector on its path until it has spent at least l_{fj} time units (the minimum possible) traveling through sector j, the current sector in its path.

The fifth set of constraints represents connectivity within airports. It handles the cases in which a flight is continued, i.e., the flight's aircraft is scheduled to perform a later flight within some time interval. We call the first flight f' and the following flight f. This set of constraints states that if flight f departs from airport k by time t, then flight f' must have arrived at airport k by time $t - s_{f'}$, where $s_{f'}$ is the turnaround time.

Finally, the sixth set of constraints represents connectivity in time. Thus, if a flight has arrived at sector j by time t, then w^j_{ft} has to have a value of 1 for all later time periods, $t' \geq t$.

The major reason we used the variables w^j_{ft}, as opposed to the variables w^j_{ft}, is that the variables w^j_{ft} nicely capture the three types of connectivity in the air traffic control problem: connectivity between sectors, connectivity within airports, and connectivity in time.

Computational results

The preceding formulation has been extensively tested using real data from both the US and the European networks. As an example, two realistic size data sets obtained directly from the Official Airline Guide (OAG) were provided by the FAA. The first one consisted of of 278 flights, 10 airports, and 178 sectors, tested over a 7 hour time frame with 5 minute intervals. The second of these data sets consisted of 1002 flights, 18 airports, and 305 sectors, tested over an 8 hour time frame with 5 minute intervals. The sector crossing times, sector and airport capacities, and required turnaround times were all provided by the FAA. These data sets are comparable to those in the problem being solved daily by the FAA.

For the first problem, consisting of 43226 constraints and 18733 variables, an optimal solution to the linear programming relaxation was found in approximately 30 minutes on a SUN SPARC 20 workstation using CPLEX 3.0 as the optimization solver and GAMS 2.25 as the modeling language. Furthermore, the solution obtained was completely integer. In other words, there is no need to use any integer programming methods. The second and larger data set consisting of 151662 constraints and 69497 variables, was

solved to optimality in approximately 2 hours, again achieving completely integer solutions.

Similar results were obtained for the European network. For a data set provided by EUROCONTROL, a problem involving 2293 flights, 25 sectors, and with all costs equal to 1 (i.e., the objective is to minimize the total delay), an optimal completely integer solution was found in approximately one hour on a SUN 10 workstation. The total delay in the optimal solution was 60% lower than the delay under the first-come-first-serve heuristic that is exercised by EUROCONTROL. This illustrates the significant impact that linear optimization can have in practice.

Another observation that is important for achieving short computation times, is that in the absence of capacity constraints, the remaining inequalities define the dual of a network flow problem, for which we know that an integer optimal solution exists and can be found by the network simplex method. As a result, the model is first ran as a network flow problem, ignoring the capacity constraints, and a basis is found. Then, the capacity constraints are introduced and the problem is solved using the dual simplex method.

While the linear programming relaxation does not always have integer optimal solutions, this turned out to be the case for these and other test problems. Compared with other formulations that have been proposed in the literature, the preceding formulation performs significantly better. One naturally wonders why this has been the case. A partial explanation is that the three sets of constraints that express the three types of connectivity in the problem are "facets" of the convex hull of the set of feasible solutions.

We summarize the principal insights from this example, which have wide applicability:

(a) Defining the "right" set of variables for a linear optimization problem can have an important impact on the size and quality of the formulation, as well as on the solution time.

(b) The key to solving large scale integer programming problems is to obtain *strong formulations*, i.e., formulations that closely approximate the convex hull of the set of integer solutions. This can be achieved by introducing constraints that are "facets" of the convex hull of the set of feasible solutions.

(c) Extracting a network subproblem is important in achieving short computational times. The network simplex algorithm, for example, is significantly faster than the general simplex method, and provides a useful initial basis for the solution to the complete problem.

12.5   The job shop scheduling problem

Our objective in this section is to show that strong formulations, even involving an exponential number of constraints, coupled with heuristics

and branch and bound methods, can lead to efficient algorithms for large scale linear optimization problems. We start with a simple one-machine scheduling problem, and continue with the job shop scheduling problem, in which several jobs need to be processed by several machines.

A single machine scheduling problem

A set $N = \{1, \ldots, n\}$ of jobs needs to be scheduled on a single machine. The processing time of job $i \in N$ is some positive number p_i. A schedule is called *nonpreemptive* if, once the machine begins processing a job, it must complete processing before starting another job. A schedule is called *nonidling* if the machine can only stay idle after all jobs have been processed. A schedule is called *feasible* if it is nonpreemptive and the machine works on at most one job at a time.

Let C_i be the completion time of job i, $i \in N$. The objective is to find a feasible schedule that minimizes

$$\sum_{i \in N} w_i C_i,$$

where $w_i \geq 0$, $i \in N$, are given weights. A natural question is to characterize the set of vectors $(C_1, \ldots, C_n)$ that correspond to feasible schedules. Clearly, the following conditions need to be satisfied

$$C_j \geq p_j, \quad j \in N,$$

$$C_j \geq C_i + p_j \quad \text{or} \quad C_k \geq C_j + p_k, \quad j, k \in N, \ j \neq k,$$

The last condition states that in every schedule, either job k is processed before job j, or job j is processed before job k. Figure 12.2 depicts the set of all feasible completion times vectors for $N = \{1, 2\}$, and shows that this set is not convex.

Next, we find constraints on the convex hull of the set of feasible completion time vectors. Towards this goal, we need the following characterization of the optimal schedule.

Theorem 12.1 Assume that

$$\frac{w_1}{p_1} \geq \frac{w_2}{p_2} \geq \ldots \geq \frac{w_n}{p_n}.$$

Then, the sequence of jobs $1, 2, \ldots, n$ is optimal.

Proof. Suppose that there is another optimal schedule. Then, there are two jobs i, j, such that job j is processed just before job i, $j > i$, and $w_i/p_i \geq w_j/p_j$. Hence,

$$C_j = C + p_j, \qquad C_i = C + p_i + p_j$$
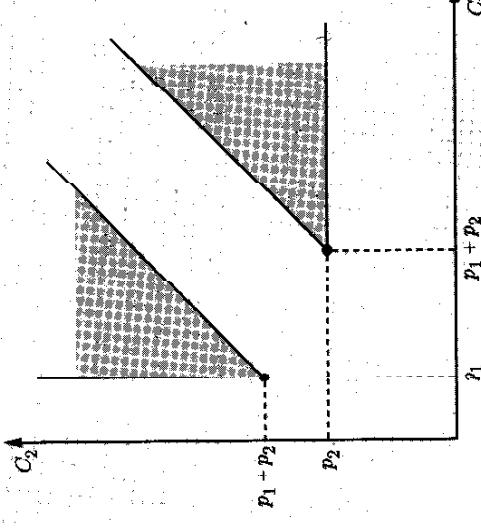
Figure 12.2: The set of all feasible completion times vectors (C_1, C_2) for $N = \{1, 2\}$. Unfortunately, the feasible set is the union of two disjoint polyhedra. It is not a convex set.

where C is the completion time of the job that precedes job j in the schedule. Let $Z = \sum_{k \in N} w_k C_k$. Consider now a schedule in which we process job i before job j, and which is otherwise the same. Under the new schedule, all completion times except for those of jobs i and j are the same. The new completion times of jobs i and j become:

$$C_i' = C + p_i, \qquad C_j' = C + p_i + p_j.$$

Let $Z' = \sum_{k \in N} w_k C_k'$. The difference in cost between the two schedules is

$$\begin{aligned}
Z' - Z &= w_i C_i' + w_j C_j' - w_i C_i - w_j C_j \\
&= w_i (C + p_i) + w_j (C + p_i + p_j) \\
&\quad - w_j (C + p_j) - w_i (C + p_i + p_j) \\
&= w_j p_i - w_i p_j \\
&= p_i p_j \left(\frac{w_j}{p_j} - \frac{w_i}{p_i} \right) \\
&\leq 0,
\end{aligned}$$

and therefore, the new schedule is also optimal. Performing more pairwise interchanges, we conclude that the schedule in which jobs are processed in the sequence $1, 2, \ldots, n$ is optimal. $\square$

Suppose that $w_i = p_i$ for all i. Then, from Theorem 12.1, it follows that all nonidling schedules are optimal. In particular, the schedule in

which the ith job processed is job i is optimal. The completion time of job i in this schedule is $C_i^* = \sum_{k=1}^{i} p_k$. Therefore, for all schedules,

$$\sum_{i=1}^{n} p_i C_i \geq \sum_{i=1}^{n} p_i C_i^*$$
$$= \sum_{i=1}^{n} p_i \sum_{k=1}^{i} p_k$$
$$= \frac{1}{2}\sum_{i=1}^{n} p_i^2 + \frac{1}{2}\left(\sum_{i=1}^{n} p_i\right)^2.$$

Consider a set $S \subset N$ of jobs. Applying the previous inequality to that set, we obtain

$$\sum_{i\in S} p_i C_i \geq \frac{1}{2}\sum_{i\in S} p_i^2 + \frac{1}{2}\left(\sum_{i\in S} p_i\right)^2, \quad \forall S \subset N. \quad (12.1)$$

Note that we have exponentially many inequalities.

We have shown that the completion times $(C_1,\ldots,C_n)$ associated with any feasible schedule must satisfy Eq. (12.1). There is also a converse result whose proof we omit: every vector $(C_1,\ldots,C_n)$ that satisfies Eq. (12.1) belongs to the convex hull of feasible completion time vectors.

For $N = \{1,2\}$, the convex hull of the set of feasible completion time vectors (see also Figure 12.2) is

$$C_1 \qquad \geq p_1$$
$$C_2 \geq p_2$$
$$p_1 C_1 + p_2 C_2 \geq p_1^2 + p_2^2 - p_1 p_2.$$

Based on the above discussion, the single machine scheduling problem is equivalent to

minimize $\quad \sum_{i=1}^{n} v_i C_i$

subject to $\quad \sum_{i\in S} p_i C_i \geq \frac{1}{2}\sum_{i\in S} p_i^2 + \frac{1}{2}\left(\sum_{i\in S} p_i\right)^2, \quad \forall S \subset N. \quad (12.2)$

The function

$$p(S) = \frac{1}{2}\sum_{i\in S} p_i^2 + \frac{1}{2}\left(\sum_{i\in S} p_i\right)^2$$

turns out to be *supermodular*, i.e., for all $S, T \subset N$,

$$p(S) + p(T) \leq p(S \cap T) + p(S \cup T).$$

Note that a function is supermodular if $-p(S)$ is submodular[3].

As discussed in Chapters 6 and 1', when solving problems with a large number of constraints, it is important to be able to check whether a given vector is feasible, and if not, to be able to generate a violated inequality. (This has been referred to as the *separation* problem; see also Section 8.5.) Since we can efficiently solve the optimization problem (Theorem 12.1), it should not be surprising that the separation problem can also be efficiently solved. This is accomplished by our next result, which exploits the special structure of the function $p(S)$.

Theorem 12.2 *Given a vector $(C_1,\ldots,C_n)$, we can decide whether $\sum_{i\in S} p_i C_i \geq p(S)$ for all $S \subset N$, or find a violated inequality by the following algorithm:*

1. *Sort the jobs in order of increasing C_i, and let S_i be the set containing the first i jobs in the sorted sequence.*

2. *Among the n sets $S_1,\ldots,S_n$, select a set S_k that maximizes*

$$f(S) = p(S) - \sum_{i\in S} p_i C_i.$$

3. *If $f(S_k) \leq 0$, then $\sum_{i\in S} p_i C_i \geq p(S)$ for all $S \subset N$. Otherwise, a violated inequality, involving the set S_k, has been found.*

Proof. Let S be a set that maximizes the function

$$f(S) = p(S) - \sum_{i\in S} p_i C_i,$$

over all $S \subset N$. For $j \in S$, we can use the definition of $f(S)$ and $p(S)$ to obtain

$$f(S) = f(S\setminus\{j\}) + p_j\sum_{i\in S} p_i - p_j C_j.$$

Since $f(S\setminus\{j\}) \leq f(S)$, we obtain that for all $j \in S$,

$$C_j \leq \sum_{i\in S} p_i.$$

For $j \notin S$, we have

$$f(S \cup \{j\}) = f(S) + p_j\left(p_j + \sum_{i\in S} p_i - C_j\right).$$

---

[3]Theorem 12.1 can also be proved using the results of Exercise 8.10 on submodular function minimization.

Since $f(\bar{S} \cup \{j\}) \le f(S)$, we obtain that for all $j \notin S$,

$$C_j - p_j \ge \sum_{i \in S} p_i.$$

Therefore, if $j \in S$, then $C_j \le \sum_{i \in S} p_i$. If $j \notin S$ then $C_j > \sum_{i \in S} p_i$. Hence, if S maximizes $f(S)$, then $j \in S$ if and only if $C_j \le \sum_{i \in S} p_i$. This implies that if $j \in S$, then for every m such that $C_m \le C_j$, we have $m \in S$. Therefore, we may sort the jobs in order of increasing C_k and construct the nested family of sets $S_1, \ldots, S_n$, where S_i contains the first i jobs in the sorted sequence. The optimum subset S may then be found among $S_1, \ldots, S_n$, and the correctness of the algorithm follows.

Note that the most time consuming operation is the sorting in Step 1, which can be accomplished in $O(n \log n)$ time; see, e.g Cormen, Leiserson, and Rivest (1990).

Single machine scheduling problems with release times, deadlines, and precedence constraints

We next show that the separation algorithm provided by Theorem 12.2 leads to an efficient computation of lower bounds for several difficult (NP-hard) variations of the single machine scheduling problem. Suppose that job i must be completed between time r_i and d_i. In addition, there are precedence constraints among jobs that are described by a directed graph $G = (\mathcal{N}, \mathcal{A})$, where

$$\mathcal{A} = \{(i,j) \mid \text{job } i \text{ must be processed before job } j\}.$$

The problem of finding a schedule that minimizes $\sum_{i \in \mathcal{N}} w_i C_i$, in the presence of release times, deadlines, and precedence constraints, is known to be NP-hard [Garey and Johnson (1979)]. However, the following linear programming problem provides a lower bound on the optimal objective function value:

minimize $\displaystyle\sum_{i=1}^{n} w_i C_i$

subject to $\displaystyle\sum_{i \in S} p_i C_i \ge \frac{1}{2} \sum_{i \in S} p_i^2 + \frac{1}{2} \left(\sum_{i \in S} p_i \right)^2,$ $\quad \forall S \subset \mathcal{N},$ (12.3)

$\qquad\qquad r_i \le C_i \le d_i,$ $\qquad\qquad\qquad\qquad \forall i \in \mathcal{N},$

$\qquad\qquad C_j \ge C_i + p_j,$ $\qquad\qquad\qquad\qquad \forall (i,j) \in \mathcal{A}.$

Because of Theorem 12.2, we can solve the separation problem, i.e., identify a violated constraint if one exists, in polynomial time. This allows us to use the cutting plane algorithm of Section 6.3 to solve problem (12.3). From a theoretical point of view, this also leads to a polynomial time (but impractical) method, based on the ellipsoid algorithm (cf. Section 8.5).

Job	1	2	3	4
p_i	2	6	8	11
d_i	10	28	29	30
w_i	0.5	2	3	4

Table 12.9: Data for the single machine scheduling problem with deadlines.

Example 12.3 We consider the 4-job, single machine problem with deadlines d_i, but without release times (all r_i are equal to p_i) or precedence constraints. The problem data are shown in Table 12.9. In order to solve the linear programming problem (12.3), we start with a small number of constraints. The initial linear programming problem is

minimize $\quad 0.5C_1 + 2C_2 + 3C_3 + 4C_4$

subject to $\quad 2C_1 + 6C_2 + 8C_3 + 11C_4 \ge 477$

$\qquad\qquad 2 \le C_1 \le 10$

$\qquad\qquad 6 \le C_2 \le 28$

$\qquad\qquad 8 \le C_3 \le 29$

$\qquad\qquad 11 \le C_4 \le 30.$

An optimal solution is $\mathbf{C}^* = (C_1^*, C_2^*, C_3^*, C_4^*) = (10, 28, 8, 20\frac{5}{11})$, with objective function value $166\frac{9}{11}$. However, the vector $\mathbf{C}^*$ does not correspond to a feasible schedule, since $C_1^* < C_4^* < C_4^* < C_2^*$. We now invoke the separation algorithm. Note that $C_3^* < C_1^* < C_4^* < C_2^*$. Since

$f(\{3\}) = 0,$ $\quad f(\{1,3\}) = 0,$ $\quad f(\{1,3,4\}) = 6,$ $\quad f(\{1,2,3,4\}) = 0,$

the subset maximizing $f(S)$ s $S = \{1,3,4\}$. We therefore add the constraint

$$2C_1 + 8C_3 + 11C_4 \ge 315.$$

The new optimal solution is $\mathbf{C}^* = (10, 27, 8, 21)$ with value 167. Since this corresponds to a feasible schedule, it is an optimal solution to the original scheduling problem.

It is possible, however, that the optimal solution to the linear programming problem (12.3) corresponds to an infeasible schedule. In particular, there might be two jobs i and j, such that $C_i \le C_j < C_i + p_j$. We can then propose a branch and bound algorithm, in which we branch by considering the cases that either job i is processed before job j, and therefore, $C_j \ge C_i + p_j$, or job j is processed before job i, and therefore, $C_i \ge C_j + p_i$. We add the corresponding constraint to the linear programming problem, find an improved lower bound, and continue until a feasible solution is obtained.

Even when the optimal solution to the linear programming problem (12.3) corresponds to an infeasible schedule, a feasible schedule can always be constructed as long as there are no deadlines ($d_i = \infty$). Let C_i^* be an optimal solution to the linear programming problem (12.3). We sort the C_i^*, and we create a feasible schedule by processing the jobs in the same order, inserting idle periods whenever needed to satisfy the constraints $C_i \geq r_i$. Note that the precedence constraints will automatically be satisfied.

Next, we provide evidence that the heuristic provides reasonable solutions by showing that it constitutes a 1-approximation algorithm when both release times and deadlines are absent ($r_i = p_i$, $d_i = \infty$). We assume, without loss of generality, that $C_1^* \leq C_2^* \leq \ldots \leq C_n^*$. In the heuristic solution, the completion time of job j is $\overline{C}_j = \sum_{k=1}^{j} p_k$. Since C_j^* is a feasible solution to the linear programming problem, we obtain

$$C_j^* \sum_{k=1}^{j} p_k \geq \sum_{k=1}^{j} p_k C_k^*$$

$$\geq \frac{1}{2} \sum_{k=1}^{j} p_k^2 + \frac{1}{2} \left(\sum_{k=1}^{j} p_k \right)^2$$

$$\geq \frac{1}{2} \left(\sum_{k=1}^{j} p_k \right)^2,$$

and thus;

$$C_j^* \geq \frac{1}{2} \sum_{k=1}^{j} p_k = \frac{1}{2} \overline{C}_j.$$

Let $Z_{LP} = \sum_{k \in N} w_k C_k^*$. Let $Z_H = \sum_{k \in N} w_k \overline{C}_k$. We have thus shown that

$$Z_{LP} \leq Z_H \leq 2 Z_{LP},$$

i.e., the heuristic produces a solution within a factor of two from the optimal in the worst case, which is the best known bound for this NP-hard problem.

Job shop scheduling

We consider the problem of scheduling a set N of n jobs on m machines. Job $i \in N$ consists of k stages, each of which must be completed on a particular machine (we assume for simplicity that all jobs have the same number k of stages). The pair (i,j), called task (i,j), represents the jth stage of the ith job. The processing time of task (i,j) is p_{ij}. The completion time of job i is the completion time of the last task of job i, i.e., task (i,k). The objective is to find a schedule that minimizes the weighted sum of job completion times, subject to the following restrictions:

(a) The schedule must be nonpreemptive. That is, once a machine begins processing a stage of a job, it must complete that stage before doing anything else.

(b) Each machine may work on at most one task at any given time.
(c) The stages of each job must be completed in order.

In order to formulate the problem, let C_{ij} be the completion time of task (i,j). For each machine r, $r = 1, \ldots, m$, let M_r represent the set of tasks that must be completed on it.

The objective function:
The objective is

$$\text{minimize} \quad \sum_{i=1}^{n} w_i C_{ik}.$$

Constraints:
In order to ensure that the stages of each job are completed in order, we add the constraints

$$C_{ij} \geq C_{i,j-1} + p_{ij}, \quad \forall i, j.$$

These constraints ensure that a stage of a job cannot begin before the previous stage is completed.

In addition, all tasks in the set M_r need not to be scheduled on machine r. Therefore, the completion times C_{ij}, for $(i,j) \in M_r$, need to satisfy the constraints (12.1) of the single machine scheduling problem:

$$\sum_{(i,j)\in S} C_{ij} p_{ij} \geq \frac{1}{2} \sum_{(i,j)\in S} p_{ij}^2 + \frac{1}{2} \left(\sum_{(i,j)\in S} p_{ij} \right)^2, \quad \forall S \subset M_r.$$

Therefore, the following linear programming problem provides a lower bound on the optimal objective function value over all feasible schedules:

$$\text{minimize} \quad \sum_{i=1}^{n} w_i C_{ik}$$

subject to the constraints

$$\sum_{(i,j)\in S} C_{ij} p_{ij} \geq \frac{1}{2} \sum_{(i,j)\in S} p_{ij}^2 + \frac{1}{2} \left(\sum_{(i,j)\in S} p_{ij} \right)^2, \quad \forall S \subset M_r, \forall r,$$

$$C_{ij} \geq C_{i,j-1} + p_{ij}, \quad \forall i, j. \tag{12.4}$$

Note that by using the separation algorithm described in Theorem 12.2 for each r, $r = 1, \ldots, m$, we can solve the separation problem (identify violated constraints) in $O(mn \log n)$ time.

As in the single machine case with release times, deadlines, and precedence constraints, an optimal solution to the linear programming problem

	stage 1	stage 2	stage 3	w_i
job 1	2	3	2	1
job 2	3	4	1	1
job 3	2	2	2	1

Table 12.10: Processing times for a job shop with 3 machines, 3 jobs, and 3 stages per job.

We solve the linear programming problem (12.4). The solution is given in Table 12.11. The optimal objective function value is 23.67. The ordering of the

	stage 1	stage 2	stage 3
job 1	2.00	5.00	7.00
job 2	5.67	9.67	10.67
job 3	2.00	4.00	6.00

Table 12.11: The job completion times C^*_{ij} provided by the solution to the linear programming problem (12.4).

tasks for each machine is as follows. Machine 1 processes tasks in the order

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

Machine 2 processes tasks in the order

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

Machine 3 processes tasks in the order

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

However, the schedule is not feasible, because for the tasks $(2,1)$ and $(3,3)$ that are processed by machine 1, we have

$$C^*_{21} < C^*_{33} < C^*_{21} + p_{33}.$$

The job shop heuristic then gives the schedule shown in Table 12.12, with objective function value of 26.

If we apply the branch and bound algorithm to the solution to the linear programming problem (12.4), we add the constraint $C_{33} - C_{21} \ge 3$ and find that the objective function value of the new linear programming problem improves

---

(12.4) does not necessarily correspond to a feasible schedule for the job shop scheduling problem. However, a feasible solution can be easily generated by the following heuristic:

Job shop scheduling heuristic

1. Solve the linear programming problem (12.4) by using the cutting plane method and the separation algorithm described in Theorem 12.2.

2. Create a feasible schedule based on the order of the optimal C^*_{ij} as follows. For each machine r, sort the set $\{C^*_{ij} \mid (i,j) \in M_r\}$ from lowest to highest. That will give us an ordering of tasks for each machine.

3. Each machine processes its tasks in order, as soon as the jobs become available.

For example, suppose machine 1 must process tasks $(1,1)$, $(2,1)$, $(3,4)$, and $(4,2)$. Suppose that $C^*_{11} = 13$, $C^*_{21} = 8$, $C^*_{34} = 18$, and $C^*_{42} = 12$. Then, machine 1 will process the tasks in order of increasing completion times, that is, in the order $(2,1)$, $(4,2)$, $(1,1)$, $(3,4)$. Task $(2,1)$ is immediately available, so it will be processed immediately, and the completion time of task $(2,1)$ will be equal to its processing time. Machine 1 will then wait until stage 1 of job 4 has been completed, so that it can begin work on task $(4,2)$. When the machine finishes task $(4,2)$, task $(1,1)$ is immediately available, so it will be processed immediately.

Moreover, a branch and bound algorithm can be devised as follows. After solving the linear programming problem (12.4), we can detect that a schedule is infeasible if there are tasks (i,j) and (k,l) processed on the same machine r, such that

$$C^*_{ij} \le C^*_{kl} < C^*_{ij} + p_{kl}.$$

As in the single machine scheduling problem, we branch by considering the cases that either task (i,j) is processed before task (k,l) on machine r, and therefore, $C_{kl} \ge C_{ij} + p_{kl}$, or task (k,l) is processed before task (i,j) on machine r, and therefore, $C_{ij} \ge C_{kl} + p_{ij}$. We add the corresponding constraint to the linear programming problem, find an improved lower bound, and continue until a feasible solution is obtained.

Example 12.4 We consider a job shop with 3 machines, 3 jobs, and 3 stages per job. The processing times p_{ij} and the weights w_i are given in Table 12.10. The sets M_i, $i = 1, 2, 3$ are as follows.

$$M_1 = \{(1,1), (2,1), (3,3)\},$$
$$M_2 = \{(1,2), (2,3), (3,1)\},$$
$$M_3 = \{(1,3), (2,2), (3,2)\}.$$

	stage 1	stage 2	stage 3
job 1	2	5	7
job 2	5	11	12
job 3	2	4	7

Table 12.12: The job completion times provided by the job shop heuristic.

to 24.5, but the corresponding schedule is still infeasible. Further branching finds that the optimal objective function value becomes 25 and finds a feasible schedule with objective function value equal to 25. Alternatively, to explore the other half of the branch and bound tree, we add the constraint $C_{21} - C_{33} \geq 2$ to the original linear programming problem (12.4), and we find that the optimal objective function value increases to 26. As we have mentioned, the job scheduling heuristic produces a schedule whose cost is equal to 26. Since this is equal to the lower bound provided by the linear programming problem, the heuristic has produced an optimal schedule.

Computational results suggest that the linear programming problem (12.4) for problems involving up to 20 machines, 20 jobs, and 20 stages per job can be solved in minutes in a SUN SPARC 20 using a cutting plane algorithm and the CPLEX optimization library. The job shop heuristic quickly produces feasible schedules for problems involving up to 20 machines, 20 jobs, and 20 stages per job, that are approximately within a factor of two of the lower bound provided by the linear programming problem (12.4). The branch and bound algorithm routinely solves such problems to within 5% – 10% from the lower bound in less than an hour. The key to success in this application is:

(a) Although the formulation (12.4) has an exponential number of constraints, it can be solved efficiently, because the separation problem can be solved fast.

(b) It is easy to check whether the solution to the linear programming relaxation is infeasible and to identify violated constraints. This gives a natural and effective way to branch.

12.6 Summary

The availability of workstations, modeling languages, and optimization libraries has advanced optimization capabilities significantly. However, such

---

advances do not imply that users can naively apply this technology to construct effective decision support systems. At the hands of an imaginative analyst the process of modeling, exploiting structure, and fine tuning of optimization algorithms will lead to better solution methods and, therefore, to more informed, insightful, and better decisions.

12.7 Exercises

Exercise 12.1* (The stable matching problem) This problem models how hospitals and residents are matched. There is a set of n hospitals and n medical residents. Each hospital has a strictly ordered list of the n residents and each resident has a strictly ordered list of the n hospitals. A perfect matching M of the hospitals and residents is called unstable if there exist a hospital i and a resident j, who are not matched under M, but prefer each other over their assigned mates under M. A perfect matching is stable if no such pair exists in the matching.

(a) Formulate the problem of deciding whether a stable matching exists as an integer programming problem.

(b) Using linear programming theory prove that a stable matching always exists

(c) Use linear programming to compute such a stable matching for $n = 9$. The preference order of the hospitals is given in the following table. For example, intern 1 is the second choice of hospital 1.

	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9
H_1	2	8	3	9	6	5	7	1	4
H_2	7	3	2	1	8	5	4	6	9
H_3	7	8	3	9	6	5	2	1	4
H_4	1	3	5	7	4	6	8	9	2
H_5	6	3	1	8	2	5	9	4	7
H_6	2	6	1	4	8	5	7	9	3
H_7	2	7	9	8	3	4	1	5	6
H_8	3	8	4	9	5	6	1	7	2
H_9	4	8	3	9	6	5	7	1	2

Similarly, the preference order of the interns is given in the following table. For example, hospital 1 is the third choice of intern 1.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8	H_9
I_1	3	7	9	8	2	4	1	5	6
I_2	6	2	3	8	1	4	5	7	9
I_3	6	3	1	8	2	5	9	4	7
I_4	9	8	3	2	5	6	7	1	4
I_5	3	8	2	9	5	6	1	7	4
I_6	2	6	1	4	8	5	7	9	3
I_7	8	2	9	3	6	7	5	1	4
I_8	9	1	5	7	4	6	8	2	3
I_9	9	8	7	6	1	2	4	5	3

Job/Stage	1	2	3	4	5	6	7	8	9	10
1	29	78	9	36	49	11	62	56	44	21
2	43	90	75	11	69	28	46	46	72	30
3	91	85	39	74	90	10	12	89	45	33
4	81	95	71	99	9	52	85	98	22	43
5	14	6	22	61	26	69	21	49	72	53
6	84	2	52	95	48	72	47	65	6	25
7	46	37	61	13	32	21	32	89	30	55
8	31	86	46	74	32	88	19	68	36	79
9	76	69	76	51	85	11	40	89	26	74
10	85	13	61	7	64	76	47	52	90	45

Table 12.13: Processing times for a job shop with 10 machines, 10 jobs, and 10 stages per job. For example $p_{11} = 29$ and $p_{12} = 78$.

Exercise 12.2 (The electric power capacity expansion problem revisited) Solve Example 6.5 in two ways: as a single linear programming problem and using Benders decomposition. Use the following data.

The generator capacity costs in hundreds of thousands of dollars per megawatt (MW) are

$$c_1 = 4.0, \quad c_2 = 2.5,$$

the operating costs are

$$f_{11} = 4.3, \; f_{21} = 2.0, \; f_{31} = 0.5, \; f_{12} = 8.7, \; f_{22} = 4.0, \; f_{32} = 1.0,$$

and the unserved demand penalties are

$$g_1 = g_2 = g_3 = 10.0.$$

The minimum generator capacities in MW are

$$b_1 = b_2 = 1000.$$

For every load level i, the demands in MW are

$$d_{i,1} = 900, \; d_{i,2} = 1000, \; d_{i,3} = 1100, \; d_{i,4} = 1200,$$

with probabilities

$$p_{i,1} = 0.15, \; p_{i,2} = 0.45, \; p_{i,3} = 025, \; p_{i,4} = 0.15.$$

The availability of generator 1 is

$$a_{1,1} = 1.0, \; a_{1,2} = 0.9, \; a_{1,3} = 0.5, \; a_{1,4} = 0.1,$$

with probabilities

$$q_{,1} = 0.2, \; q_{1,2} = 0.3, \; q_{1,3} = 0.4, \; q_{1,4} = 0.1.$$

Machine/Job	1	2	3	4	5	6	7	8	9	10
M_1	1	-	2	3	2	7	2	2	1	2
M_2	2	6	1	1	3	2	1	3	2	1
M_3	3	2	4	2	1	1	4	1	5	3
M_4	4	5	3	8	5	4	3	10	3	8
M_5	5	3	10	4	6	9	10	5	9	9
M_6	6	3	6	10	4	3	6	4	4	7
M_7	7	7	8	5	10	8	5	6	7	4
M_8	8	9	7	7	8	10	9	9	8	10
M_9	9	10	5	6	7	5	8	7	10	5
M_{10}	10	4	9	9	9	6	7	8	6	6

Table 12.14: Each row represents the stages from each job that a machine needs to process. For example, machine 1 needs to process tasks $(1,1), (2,1), (3,2), \ldots, (10,2)$.

The availability of generator 2 is

$$a_{2,1} = 1.0, \; a_{2,2} = 0.9, \; a_{2,3} = 0.7, \; a_{2,4} = 0.1, \; a_{2,5} = 0,$$

with probabilities

$$q_{2,1} = 0.1, \; q_{2,2} = 0.2, \; q_{2,3} = 0.5, \; q_{2,4} = 0.1, \; q_{2,5} = 0.1.$$

Exercise 12.3 Using a modeling language, formulate Exercise 10.4, and solve it using an optimization library.

Exercise 12.4 * (A large scale job shop scheduling problem) Solve the job shop scheduling problem with 10 machines, 10 jobs, and 10 stages per job. Table 12.13 depicts the processing times p_{ij}, and Table 12.14 specifies the sets M_r. The weights are $w_i = 1$, so our objective is to minimize the average completion time.

Exercise 12.5 * (A large scale traveling salesman problem) Generate 1000 random points in the unit square $[0,1]^2$. The distance c_{ij} is the Euclidean distance between the points i and j. Solve the linear programming relaxation of the cutset formulation of the traveling salesman problem on these 1000 points. Based on the solution to the linear programming relaxation, develop a branch and cut algorithm to generate near optimal traveling salesman tours. *Hint:* Use Lagrangean relaxation for bounding.

Exercise 12.6 * (A large scale facility location problem) Generate 200 random points in the unit square $[0,1]^2$. The distance c_{ij} is the Euclidean distance between the points i and j. Solve the facility location problem for $K = 10$ facilities, where each facility must be one of the 200 random points.

Player	r_i	a_i	h_i	s_i	d_i
p_1	1	7	5.9	10	10
p_2	2	14	6.0	14	9
p_3	3	12	6.3	19	8
p_4	4	4	6.0	18	6
p_5	5	9	6.2	20	8
p_6	7	6	6.4	21	10
p_7	7	8	6.6	23	10
p_8	4	2	6.4	13	5
p_9	8	2	6.8	17	8
p_{10}	5	5	6.3	25	8
p_{11}	10	6	6.8	20	9
p_{12}	8	8	6.7	30	10
p_{13}	10	2	7.2	24	9
p_{14}	9	5	6.8	15	7
p_{15}	6	3	6.8	17	6
p_{16}	16	2	6.7	3	6
p_{17}	11	1	7.3	27	9
p_{18}	12	5	7.1	26	10
p_{19}	11	1	7.2	21	9
p_{20}	9	1	7.0	14	8

Table 12.15: The rebounding average r_i, assists average a_i, height h_i, scoring average s_i, and overall defense ability d_i for player $i = 1, \ldots, 20$.

Exercise 12.7 Solve Exercise 10.2 using the data of Table 12.15. The desired targets are $r = 7$, $a = 6$, $h = 6.6$, $s = 18$, $d = 8.5$.

Exercise 12.8 * (A large scale fixed charge network design problem) Generate 100 random points in the unit square $[0,1]^2$. Let G be the complete directed graph on these 100 points. Let b_i be the demand or supply of point i. The demands for the first 99 points are independent and uniformly distributed in the interval $[-100, 100]$. The demand for the 100th point is equal to $-\sum_{i=1}^{99} b_i$. Let the transportation cost c_{ij} be equal to the Euclidean distance between points i and j, and let the construction cost d_{ij} be ten times the Euclidean distance. Solve the fixed charge network design problem on these 100 points (see Exercise 10.9 for the definition of the fixed charge network design problem).

Exercise 12.9* (The graph coloring problem) Given an undirected graph $G = (\mathcal{N}, \mathcal{E})$, we want to assign a color to every node in $\mathcal{N}$, so that adjacent nodes are assigned different colors, and the total number of colors is minimized. Generate a graph on 100 nodes, so that if you draw it on the plane no two edges intersect. Such graphs are called *planar*. Solve the problem by using a branch

---

and bound approach. *Hint:* We can always color planar graphs using 4 colors. Use this information to assist the bounding process.

12.8   Notes and sources

12.1 For further information on the modeling languages GAMS and AMPLE, see the corresponding manuals. The *OR/MS Today* journal has frequent surveys of optimization solvers. The journal *Interfaces* contains many successful applications of linear optimization methods to problems arising in telecommunications, finance, transportation, manufacturing, services, etc.

12.2 Table 12.1 has been compiled by Weber (1995), based on experiments by several researchers. The general guidelines for the relative performance of various algorithms are from our experimentation with large scale linear programming problems, and also from the manual of the CPLEX optimization library.

12.3 The model, the solution methodology, and the computational results for the fleet assignment problem are taken from Hane et al. (1995). For further advances and an application of the fleet assignment problem in an industrial context, see Rushmeier and Kontogiorgis (1997).

12.4 The model and the computational results for the flow management problem in the US network are taken from Bertsimas and Stock (1997). The computational results for the European network are from Vranas (1996).

12.5 The formulation of the single machine scheduling problem is from Queyranne (1993). The indexing rule (Theorem 12.1) is from Smith (1956). The job shop scheduling heuristic is motivated by the work of Schultz (1996). A comprehensive review of known formulations and approximation algorithms for machine scheduling problems is given in Hall et al. (1996). The branch and bound algorithm, and the computational results for the job shop scheduling problem are from Bertsimas and Hsu (1997). An alternative cutting plane approach for job shop scheduling problems, in which we are interested in minimizing the maximum completion time, is proposed in Applegate and Cook (1991). For the same problem, a different enumerative but effective approach is proposed in Martin and Shmoys (1996).

12.7 The stable matching problem addressed in Exercise 12.1 was first defined and solved in Gale and Shapley (1962). A linear programming approach to the problem is given in Teo (1996). Exercise 12.2 is almost identical to a problem formulated and solved in Infanger (1993). The data in Exercise 12.4 are from Fisher and Thompson (1963). This is a famous instance of the job shop scheduling problem, and it took decades until a provably optimal solution was obtained.