2.50. Show how to transform a minimum cost flow problem stated in (2.1) into a circulation problem. Establish a one-to-one correspondence between the feasible solutions of these two problems. (Hint: Introduce two new nodes and some arcs.)

2.51. Show that by adding an extra node and appropriate arcs, we can formulate any minimum cost flow problem with one or more inequalities for supplies and demands (i.e., the mass balance constraints are stated as \( \leq b(i) \) for a supply node \( i \), and/or \( \geq b(j) \) for a demand node \( j \)) into an equivalent problem with all equality constraints (i.e., \( = b(k) \) for all nodes \( k \)).

3

ALGORITHM DESIGN AND ANALYSIS

Numerical precision is the very soul of science.
—Sir D’Arcy Wentworth Thompson

Chapter Outline

3.1 Introduction
3.2 Complexity Analysis
3.3 Developing Polynomial-Time Algorithms
3.4 Search Algorithms
3.5 Flow Decomposition Algorithms
3.6 Summary

3.1 INTRODUCTION

Scientific computation is a unifying theme that cuts across many disciplines, including computer science, operations research, and many fields within applied mathematics and engineering. Within the realm of computational problem solving, we almost always combine three essential building blocks: (1) a recipe, or algorithm, for solving a particular class of problems; (2) a means for encoding this procedure in a computational device (e.g., a calculator, a computer, or even our own minds); and (3) the application of the method to the data of a specific problem. For example, to divide one number by another, we might use the iterative algorithm of long division, which is a systematic procedure for dividing any two numbers. To solve a specific problem, we could use a calculator that has this algorithm already built into its circuitry. As a first step, we would enter the data into storage locations on the calculator; then we would instruct the calculator to apply the algorithm to our data.

Although dividing two numbers is an easy task, the essential steps required to solve this very simple problem—designing, encoding, and applying an algorithm—are similar to those that we need to address when solving complex network flow problems. We need to develop an algorithm, or a mathematical prescription, for solving a class of network flow problems that contains our problem—for example, to solve a particular shortest path problem, we might use an algorithm that is known to solve any shortest path problem with nonnegative arc lengths. Since solving a network flow problem typically requires the solution of an optimization model with hundreds or thousands of variables, equations, and inequalities, we will invariably solve the problem on a computer. Doing so requires that we not only express the mathematical steps of the algorithm as a computer program, but that we also develop data structures for manipulating the large amounts of information required to rep-
resent the problem. We also need a method for entering the data into the computer and for performing the necessary operations on it during the course of the solution procedure.

In Chapter 2 we considered the lower-level steps of the computational problem-solving hierarchy; that is, we saw how to represent network data and therefore how to encode and manipulate the data within a computer. In this chapter we consider the highest level of the solution hierarchy: How do we design algorithms, and how do we measure their effectiveness? Although the idea of an algorithm is an old one—Chinese mathematicians in the third century B.C. had already devised algorithms for solving small systems of simultaneous equations—researchers did not begin to explore the notion of algorithmic efficiency as discussed in this book in any systematic and theoretical sense until the early 1970s. This particular subject matter, known as computational complexity theory, provides a framework and a set of analysis tools for gauging the work performed by an algorithm as measured by the elementary operations (e.g., addition, multiplication) it performs. One major stream of research in computational complexity theory has focused on developing performance guarantees or worst-case analyses that address the following basic question: When we apply an algorithm to a class of problems, can we specify an upper bound on the amount of computations that the algorithm will require? Typically, the performance guarantee is measured with respect to the size of the underlying problem: for example, for network flow problems, the number \( n \) of nodes and the number \( m \) of arcs in the underlying graph. For example, we might state that the complexity of an algorithm for solving shortest path problems with nonnegative arc lengths is \( 2^n \), meaning that the number of computations grow no faster than twice the square of the number of nodes. In this case we say that the algorithm is “good” because its computations are bounded by a polynomial in the problem size (as measured by the number of nodes). In contrast, the computational time for a “bad” algorithm would grow exponentially when applied to a certain class of problems. With the theoretical worst-case bound in hand, we can now assess the amount of work required to solve (nonnegative length) shortest path problems as a function of the input size. We also have a tool for comparing any two algorithms: the one with the smaller complexity bound is preferred from the viewpoint of a worst-case analysis.

Network optimization problems have been the core and influential subject matter in the evolution of computational complexity theory. Researchers and analysts have developed many creative ideas for designing efficient network flow algorithms based on the concepts and results emerging in the study of complexity theory; at the same time, many ideas originating in the study of network flow problems have proven to be useful in developing and analyzing a wide variety of algorithms in many other problem domains. Although network optimization has been a constant subject of study throughout the years, researchers have developed many new results concerning complexity bounds for network flow algorithms at a remarkable pace in recent years. Many of these recent innovations draw on a small set of common ideas, which are simultaneously simple and powerful.

Our intention in this chapter is to bring together some of the most important of these ideas. We begin by reviewing the essential ingredients of computational complexity theory, including the definition and computational implications of good algorithms. We then describe several key ideas that appear to be mainstays in the development and analysis of good network flow algorithms. One idea is an approximation strategy, known as scaling, that solves a sequence of “simple” approximate versions of a given problem (determined by scaling the problem data) in such a way that the problems gradually become better approximations of the original problem. A second idea is a geometric improvement argument that is quite useful in analyzing algorithms; it shows that whenever we make sufficient (i.e., fixed percentage) improvements in the objective function at every iteration, an algorithm is good. We also describe some important tools that can be used in analyzing or streamlining algorithms: (1) a potential function method that provides us with a scalar integer-valued function that summarizes the progress of an algorithm in such a way that we can use it to bound the number of steps that the algorithm takes, and (2) a parameter balancing technique that permits us to devise an algorithm based on some underlying parameter and then to set the parameter so that we minimize the number of steps required by the algorithm. Next, we introduce the idea of dynamic programming, which is a useful algorithmic strategy for developing good algorithms. The dynamic programming technique decomposes the problem into stages and uses a recursive relationship to go from one stage to another. Finally, we introduce the binary search technique, another well-known technique for obtaining efficient algorithms. Binary search performs a search over the feasible values of the objective function and solves an easier problem at each search point.

In this chapter we also describe important and efficient (i.e., good) algorithms that we use often within the context of network optimization: search algorithms that permit us to find all the nodes in a network that satisfy a particular property. Often in the middle of a network flow algorithm, we need to discover all nodes that share a particular attribute; for example, in solving a maximum flow problem, we might want to find all nodes that are reachable from the designated source node along a directed path in the residual network. Search algorithms provide us with a mechanism to perform these important computations efficiently. As such, they are essential, core algorithms used to design other more complex algorithms.

Finally, we study network decomposition algorithms that permit us to decompose a solution to a network flow problem, formulated in terms of arc flows, into a set of flows on paths and cycles. In our treatment of network flow problems, we have chosen to use a model with flows defined on arcs. An alternative modeling approach is to view all flows as being carried along paths and cycles in the network. In this model, the variables are the amount of flow that we send on any path or cycle. Although the arc flow formulation suffices for most of the topics that we consider in this book, on a few occasions such as our discussion of multicommodity flows in Chapter 17, we will find it more convenient to work with a path and cycle flow model. Moreover, even if we do not use the path and cycle flow formulation per se, understanding this model provides additional insight about the nature of network flow problems. The network decomposition algorithms show that the arc flow model and the path and cycle flow model are equivalent, so we could use any of these models for formulating network flow problems; in addition, these algorithms provide us with an efficient computational procedure for finding a set of path and cycle flows that is equivalent to any given set of arc flows.

Sec. 3.1 Introduction
3.2 Complexity Analysis

An algorithm is a step-by-step procedure for solving a problem. By a problem we mean a general model such as the shortest path problem or the minimum cost flow problem. Problems can be subsets of one another. For example, not only does the set of all shortest path problems define a problem, but so does the class of all shortest path problems with nonnegative arc costs. An instance is a special case of a problem with data specified for all the problem parameters. For example, to define an instance of the shortest path problem we would need to specify the network topology $G = (N, A)$, the source and destination nodes, and the values of the arc costs. An algorithm is said to solve a problem $P$ if when applied to any instance of $P$, the algorithm is guaranteed to produce a solution. Generally, we are interested in finding the most efficient algorithm for solving a problem. In the broadest sense, the notion of efficiency involves all the various computing resources needed for executing an algorithm. However, in this book since time is often a dominant computing resource, we use the time taken by an algorithm as our metric for measuring the "most efficient" algorithm.

Different Complexity Measures

As already stated, an algorithm is a step-by-step procedure for solving a problem. The different steps an algorithm typically performs are (1) assignment steps (such as assigning some value to a variable), (2) arithmetic steps (such as addition, subtraction, multiplication, and division), and (3) logical steps (such as comparison of two numbers). The number of steps performed (or taken) by the algorithm is said to be the sum total of all steps it performs. The number of steps taken by an algorithm, which to a large extent determines the time it requires, will differ from one instance of the problem to another. Although an algorithm might solve some "good" instances of the problem quickly, it might take a long time to solve some "bad" instances. This range of possible outcomes raises the question of how we should measure the performance of an algorithm so that we can select the "best" algorithm from among several competing algorithms for solving a problem. The literature has widely adopted three basic approaches for measuring the performance of an algorithm:

1. Empirical analysis. The objective of empirical analysis is to estimate how algorithms behave in practice. In this analysis we write a computer program for the algorithm and test the performance of the program on some classes of problem instances.
2. Average-case analysis. The objective of average-case analysis is to estimate the expected number of steps an algorithm takes. In this analysis we choose a probability distribution for the problem instances and using statistical analysis derive asymptotic expected running times for the algorithm.
3. Worst-case analysis. Worst-case analysis provides upper bounds on the number of steps that a given algorithm can take on any problem instance. In this analysis we count the largest possible number of steps; consequently, this analysis provides a "guarantee" on the number of steps an algorithm will take to solve any problem instance.

Each of these three performance measures has its relative merits and drawbacks. Empirical analysis has several major drawbacks: (1) an algorithm's performance depends on the programming language, compiler, and computer used for the computational experiments, as well as the skills of the programmer who wrote the program; (2) often this analysis is too time consuming and expensive to perform; and (3) the comparison of algorithms is often inconclusive in the sense that different algorithms perform better on different classes of problem instances and different empirical studies report contradictory results.

Average-case analysis has major drawbacks as well: (1) the analysis depends crucially on the probability distribution chosen to represent the problem instances, and different choices might lead to different assessments as to the relative merits of the algorithms under consideration; (2) it is often difficult to determine appropriate probability distributions for problems met in practice; and (3) the analysis often requires quite intricate mathematics even for assessing the simplest type of algorithm—the analysis typically is extremely difficult to carry out for more complex algorithms. Furthermore, the prediction of an algorithm's performance, based on its average-case analysis, is tailored for situations in which the analyst needs to solve a large number of problem instances; it does not provide information about the distribution of outcomes. In particular, although the average-case performance of an algorithm might be good, we might encounter exceptions with little statistical significance on which the algorithm performs very badly.

Worst-case analysis avoids many of these drawbacks. The analysis is independent of the computing environment, is relatively easy to perform, provides a guarantee on the steps (and time) taken by an algorithm, and is definitive in the sense that it provides conclusive proof that an algorithm is superior to another for the worst possible problem instances that an analyst might encounter. Worst-case analysis is not perfect, though: One major drawback of worst-case analysis is that it permits "pathological" instances to determine the performance of an algorithm, even though they might be exceedingly rare in practice. However, the advantages of the worst-case analysis have traditionally outweighed its shortcomings, and this analysis has become the most popular method for measuring algorithmic performance in the scientific literature. The emergence of the worst-case analysis as a tool for assessing algorithms has also had a great impact on the field of network flows, stimulating considerable research and fostering many algorithmic innovations. In this book, too, we focus primarily on worst-case analysis. We also try to provide insight about the empirical performance, particularly in Chapter 18, since we believe that the empirical behavior of algorithms provides important information for guiding the use of algorithms in practice.

Problem Size

To express the time requirement of an algorithm, we would like to define some measure of the "complexity" of the problem instances we encounter. Having a single performance measure for all problem instances rarely makes sense since as the problem instances become larger, they typically become more difficult to solve (i.e., take more time); often the effort required to solve problem instances varies roughly

Algorithm Design and Analysis  Chap. 3  56

Sec. 3.2  Complexity Analysis  57
with their size. Hence to measure the complexity of problem instances, we must consider the "size" of the problem instance. But what is the size of a problem?

Before we address this question, let us discuss what is the size of a data item whose value is \( x \). We can make one of the two plausible assumptions: (1) assume that the size of the data item is \( x \), or (2) assume that the size of the data item is \( \log x \). Of these, for several reasons the second assumption is more common. The primary reason is that \( \log x \) reflects the way that computers work. Most modern computers represent numbers in binary form (i.e., in bits) and store them in memory locations of fixed bit size. The binary representation of item \( x \) requires \( \log x \) bits, and hence the space required to store \( x \) is proportional to \( \log x \).

The size of a network problem is a function of how the problem is stated. For a network problem, the input might be in the form of one of the representations discussed in Section 2.3. Suppose that we specify the network in the adjacency list representation, which is the most space-efficient representation we could use. Then the size of the problem is the number of bits needed to store its adjacency list representation. Since the adjacency list representation stores one pointer for each node and arc, and one data element for each arc cost coefficient and each arc capacity, it requires approximately \( n \log n + m \log m + m \log C + m \log U \) bits to store all of the problem data for a minimum cost network flow problem (recall that \( C \) represents the largest arc cost and \( U \) represents the largest arc capacity). Since \( m \leq n^2 \), \( \log m \leq \log n^2 = 2 \log n \). For this reason, when citing the size of problems using a "big \( O \)" complexity notation that ignores constants (see the subsection entitled "big \( O \)" to follow), we can (and usually do) replace each occurrence of \( \log m \) by the simpler \( \log n \).

In principle, we could express the running time of an algorithm as a function of the problem size; however, that would be unnecessarily awkward. Typically, we will express the running time more simply and more directly as a function of the network parameters \( n \), \( m \), \( \log C \), and \( \log U \).

**Worst-Case Complexity**

The time taken by an algorithm, which is also called the *running time* of the algorithm, depends on both the nature and size of the input. Larger problems require more solution time, and different problems of the same size typically require different solution times due to differences in the data. A *time complexity function* for an algorithm is a function of the problem size and specifies the largest amount of time needed by the algorithm to solve any problem instance of a given size. In other words, the time complexity function measures the rate of growth in solution time as the problem size increases. For example, if the time complexity function of a network algorithm is \( cmn \) for some constant \( c \geq 0 \), the running time needed to solve any network problem with \( n \) nodes and \( m \) arcs is at most \( cnm \). Notice that the time complexity function accounts for the dependence of the running time on the problem size by measuring the largest time needed to solve any problem instance of a given size; at this level of detail in measuring algorithmic performance, the complexity function provides a performance guarantee that depends on the appropriate measure of the problem's input data. Accordingly, we also refer to the time complexity function as the *worst-case complexity* (or, simply, the *complexity*) of the algorithm. We also refer to the worst-case complexity of an algorithm as its *worst-case bound*, for it states an upper bound on the time taken by the algorithm.

**Big \( O \) Notation**

To define the complexity of an algorithm completely, we need to specify the values for one or more constants. In most cases the determination of these constants is a nontrivial task; moreover, the determination might depend heavily on the computer, and other factors. Consider, for example, the following segment of an algorithm, which adds two \( p \times q \) arrays:

```plaintext
for i = 1 to p do
  for j = 1 to q do
    c_{ij} = a_{ij} + b_{ij}.
```

At first glance, this program segment seems to perform exactly \( pq \) additions and the same number of assignments of values to the computer locations storing the values of the variables \( c_{ij} \). This accounting, however, ignores many computations that the computer would actually perform. A computer generally stores a two-dimensional array of size \( p \times q \) as a single array of length \( pq \) and so would typically store the element \( a_{ij} \) at the location \( (i - 1)q + j \) of the array \( a \). Thus each time we retrieve the value of \( a_{ij} \) and \( b_{ij} \) we would need to perform one subtraction, one multiplication, and one addition. Further, whenever the computer would increment the index \( i \) (or \( j \)), it would perform a comparison to determine whether \( i > p \) (or \( j > q \)). Needless to say, such a detailed analysis of an algorithm is very time consuming and not particularly illuminating.

The dependence of the complexity function on the constants poses yet another problem: How do we compare an algorithm that performs \( 5n \) additions and \( 3n \) comparisons with an algorithm that performs \( n \) multiplications and \( 2n \) subtractions? Different computers perform mathematical and logical operations at different speeds, so neither of these algorithms might be universally better.

We overcome these difficulties by ignoring the constants in the complexity analysis. We do so by using "big \( O \)" notation, which has become commonplace in computational mathematics, and replace the lengthy and somewhat awkward expression "the algorithm required \( cnm \) time for some constant \( c \)" by the equivalent expression "the algorithm requires \( O(nm) \) time." We formalize this definition as follows:

An algorithm is said to run in \( O(f(n)) \) time if for some numbers \( c \) and \( n_0 \), the time taken by the algorithm is at most \( cf(n) \) for all \( n \geq n_0 \).

Although we have stated this definition in terms of a single measure \( n \) of a problem-size parameter, we can easily incorporate other size parameters \( m \), \( C \), and \( U \) in the definition.

The big \( O \) notation has several implications. The complexity of an algorithm is an upper bound on the running time of the algorithm for sufficiently large values of \( n \). Therefore, this complexity measure states the asymptotic growth rate of the running time. We can justify this feature of the complexity measure from practical
considerations since we are more interested about the behavior of the algorithm on very large inputs, as these inputs determine the limits of applicability of the algorithm. Furthermore, the big O notation indicates only the most dominant term in the running time, because for sufficiently large \( n \), terms with a smaller growth rate become insignificant as compared to terms with a higher growth rate. For example, if the running time of an algorithm is \( 100n + n^2 + 0.0001n^3 \), then for all \( n \geq 100 \), the second term dominates the first term, and for all \( n \geq 10,000 \), the third term dominates the second term. Therefore, the complexity of the algorithm is \( O(n^3) \).

Another important implication of ignoring constants in the complexity analysis is that we can assume that each elementary mathematical operation, such as addition, subtraction, multiplication, division, assignment, and logical operations, requires an equal amount of time. A computer typically performs these operations at different speeds, but the variation in speeds can typically be bounded by a constant (provided the numbers are not too large), which is insignificant in big O notation. For example, a computer typically multiplies two numbers by repeated additions and the number of such additions are equal to number of bits in the smaller number. Assuming that the largest number can have 32 bits, the multiplication can be at most 32 times more expensive than addition. These observations imply that we can summarize the running time of an algorithm by recording the number of elementary mathematical operations it performs, viewing every operation as requiring an equivalent amount of time.

**Similarity Assumption**

The assumption that each arithmetic operation takes one step might lead us to underestimate the asymptotic running time of arithmetic operations involving very large numbers on real computers since, in practice, a computer must store such numbers in several words of its memory. Therefore, to perform each operation on very large numbers, a computer must access a number of words of data and thus take more than a constant number of steps. Thus the reader should be forewarned that the running times are misleading if the numbers are exponentially large. To avoid this systematic underestimation of the running time, in comparing two running times, we will sometimes assume that both \( C \) (i.e., the largest arc cost) and \( U \) (i.e., the largest arc capacity) are polynomially bounded in \( n \) (i.e., \( C = O(n^k) \) and \( U = O(n^k) \), for some constant \( k \)). We refer to this assumption as the *similarity assumption*.

**Polynomial- and Exponential-Time Algorithms**

We now consider the question of whether or not an algorithm is "good." Ideally, we would like to say that an algorithm is good if it is sufficiently efficient to be usable in practice, but this definition is imprecise and has no theoretical grounding. An idea that has gained wide acceptance in recent years is to consider a network algorithm "good" if its worst-case complexity is bounded by a polynomial function of the problem's parameters (i.e., it is a polynomial function of \( n, m, \log C, \) and \( \log U \)). Any such algorithm is said to be a *polynomial-time algorithm*. Some examples of polynomial-time bounds are \( O(n^2) \), \( O(nm) \), \( O(m + n \log C) \), \( O(nm \log(n^2 m)) \), and \( O(nm + n^2 \log U) \). (Note that \( n \log n \) is polynomially bounded because its growth rate is slower than \( n \).) A polynomial-time algorithm is said to be a *strongly polynomial-time algorithm* if its running time is bounded by a polynomial function in only \( n \) and \( m \), and does not involve \( \log C \) or \( \log U \), and is a *weakly polynomial-time algorithm* otherwise. Some strongly polynomial time bounds are \( O(n^2 m) \) and \( O(n \log n) \). In principle, strongly polynomial-time algorithms are preferred to weakly polynomial-time algorithms because they can solve problems with arbitrary large values for the cost and capacity data.

Note that in this discussion we have said that an algorithm is polynomial time if its running time is bounded by a polynomial in the network parameters \( n, m, \log C, \) and \( \log U \). Typically, in computational complexity we say that an algorithm is polynomial time if its running time is bounded by a polynomial in the problem size, in this case \( n \log n + m \log m + n \log C + m \log U \); however, it is easy to see that the running time of a network problem is bounded by a polynomial in its problem size if and only if it is also bounded by a polynomial in the problem parameters. For example, if the running time is bounded by \( n^{100} \), it is strictly less than the problem size to the 100th power. Similarly, if the running time is bounded by the problem size to the 100th power, it is less than \( n \log n + m \log m + n \log C + m \log U \), which in turn is bounded by \( n^2 + m^2 + n \log C + m \log U \), which is a polynomial in \( n, m, \log C, \) and \( \log U \).

An algorithm is said to be an *exponential-time algorithm* if its worst-case running time grows as a function that cannot be polynomially bounded by the input length. Some examples of exponential time bounds are \( O(nc) \), \( O(2^n) \), \( O(n^2) \), and \( O(n^{n^{cn}}) \). (Observe that \( n^c \) cannot be bounded by a polynomial function of \( n \) and \( \log C \).) We say that an algorithm is a *pseudopolynomial-time algorithm* if its running time is polynomially bounded in \( n, m, C, \) and \( U \). The class of pseudopolynomial-time algorithms is an important subclass of exponential-time algorithms. Some examples of pseudopolynomial-time bounds are \( O(m + nC) \) and \( O(mC) \). For problems that satisfy the similarity assumption, pseudopolynomial-time algorithms become polynomial-time algorithms, but the algorithms will not be attractive if \( C \) and \( U \) are high-degree polynomials in \( n \).

There are several reasons for preferring polynomial-time algorithms to exponential-time algorithms. Any polynomial-time algorithm is asymptotically superior to any exponential-time algorithm, even in extreme cases. For example, \( n^{4000} \) is smaller than \( n^{1.1} \log n \) if \( n \) is sufficiently large (i.e., \( n \geq 2^{100,000} \)). Figure 3.1 illustrates the growth rates of several typical complexity functions. The exponential complexity functions have an explosive growth rate and, in general, they are able to solve only small problems. Further, much practical experience has shown that the polynomials encountered in practice typically have a small degree, and generally, polynomial-time algorithms perform better than exponential-time algorithms.

<table>
<thead>
<tr>
<th>( n )</th>
<th>( \log n )</th>
<th>( n^{0.5} )</th>
<th>( n^2 )</th>
<th>( n^3 )</th>
<th>( 2^n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3.32</td>
<td>3.16</td>
<td>10^2</td>
<td>10^3</td>
<td>10^3</td>
</tr>
<tr>
<td>100</td>
<td>6.64</td>
<td>10.00</td>
<td>10^4</td>
<td>10^6</td>
<td>1.27 \times 10^{10}</td>
</tr>
<tr>
<td>1000</td>
<td>9.97</td>
<td>31.62</td>
<td>10^6</td>
<td>10^8</td>
<td>1.07 \times 10^{18}</td>
</tr>
<tr>
<td>10,000</td>
<td>13.29</td>
<td>100.00</td>
<td>10^9</td>
<td>10^{10}</td>
<td>0.99 \times 10^{23}</td>
</tr>
</tbody>
</table>

**Figure 3.1 Growth rates of some polynomial and exponential functions.**

**Sec. 3.2 Complexity Analysis**
A brief examination of the effects of improved computer technology on algorithms is even more revealing in understanding the impact of various complexity functions. Consider a polynomial-time algorithm whose complexity is \(O(n^p)\). Suppose that the algorithm is able to solve a problem of size \(n_1\) in 1 hour on a computer with speed of \(s_1\) instructions per second. If we increase the speed of the computer to \(s_2\), then \(n_2 = n_1 s_2/s_1\) specifies the size \(n_2\) of the problem that the algorithm can solve in the same time. Consequently, a 100-fold increase in computer speed would allow us to solve problems that are 100 times larger. Now consider an exponential-time algorithm with a complexity of \(O(2^n)\). As before, let \(n_1\) and \(n_2\) denote the problem sizes solved on a computer with speeds \(s_1\) and \(s_2\) in 1 hour of computation time. Then \(s_2/s_1 = 2^n/n^2\). Alternatively, \(n_2 = n_1 + \log(s_2/s_1)\). In this case, a 100-fold increase in computer speed would allow us to solve problems that are only about 7 units larger. This discussion shows that a substantial increase in computer speed allows us to solve problems by polynomial-time algorithms that are larger by a multiplicative factor; for exponential-time algorithms we obtain only additive improvements. Consequently, improved hardware capabilities of computers have only a marginal impact on the problem-solving ability of exponential-time algorithms.

Let us pause to summarize our discussion of polynomial and exponential-time algorithms. In the realm of complexity theory, our objective is to obtain an algorithm with the smallest possible growth rate, because an algorithm with smaller growth rate is likely to permit us to solve larger problems in the same amount of computer time (depending on the associated constants). For example, we prefer \(O(n^p)\) to \(O(n^q)\), for any \(p < q\), and \(O(n^p)\) if \(n^p\) is a polynomial. However, running times involving more than one parameter, such as \(O(n \log n)\) and \(O(n^2)\), might not be comparable. If \(m < n^{\log n}\), then \(O(m \log n)\) is superior; otherwise, \(O(n^2)\) is superior.

Can we say that a polynomial-time algorithm with a smaller growth rate would run faster in practice, or even that a polynomial-time algorithm would empirically outperform an exponential-time algorithm? Although this statement is generally true, there are many exceptions to the rule. A classical exception is provided by the simplex method and Khachiyan's "ellipsoid" algorithm for solving linear programming problems. The simplex algorithm is known to be an exponential-time algorithm, but in practice it runs much faster than Khachiyan's polynomial-time algorithm. Many of these exceptions can be explained by the fact that the worst-case complexity is greatly inferior to the average complexity of some algorithms, while for other algorithms the worst-case and average complexity might be comparable. As a consequence, considering worst-case complexity as synonymous with average complexity can lead to incorrect conclusions.

Sometimes, we might not succeed in developing a polynomial-time algorithm for a problem. Indeed, despite their best efforts spanning several decades, researchers have been unable to develop polynomial-time algorithms for a huge collection of important combinatorial problems; all known algorithms for these problems are exponential-time algorithms. However, the research community has been able to show that most of these problems belong to a class of problems, called \(NP\)-complete problems, that are equivalent in the sense that if there exists a polynomial-time algorithm for one problem, there exists a polynomial-time algorithm for every other \(NP\)-complete problem. Needless to say, developing a polynomial-time algorithm for some \(NP\)-complete problem is one of the most challenging and intriguing issues facing the research community; the available evidence suggests that no such algorithm exists. We discuss the theory of \(NP\)-completeness in greater detail in Appendix B.

**Big \( \Omega \) and Big \( \Theta \) Notation**

The big \( O \) notation that we introduced earlier in this section is but one of several convenient notational devices that researchers use in the analysis of algorithms. In this subsection we introduce two related notational constructs: the big \( \Omega \) (big omega) notation and the big \( \Theta \) (big theta) notation.

Just as the big \( O \) notation specifies an upper bound on an algorithm's performance, the big \( \Omega \) notation specifies a lower bound on the running time.

An algorithm is said to be \( \Omega(f(n)) \) if for some numbers \( c' \) and \( n_0 \) and all \( n \geq n_0 \), the algorithm takes at least \( c'f(n) \) time on some problem instance.

The reader should carefully note that the big \( O \) notation and the big \( \Omega \) notation are defined in somewhat different ways. If an algorithm runs in \( O(f(n)) \) time, every instance of the problem of size \( n \) takes at most \( c f(n) \) time for a constant \( c \). On the other hand, if an algorithm runs in \( \Omega(f(n)) \) time, some instance of size \( n \) takes at least \( c' f(n) \) time for a constant \( c' \).

The big \( \Theta \) (big theta) notation provides both a lower and an upper bound on an algorithm’s performance.

An algorithm is said to be \( \Theta(f(n)) \) if the algorithm is both \( O(f(n)) \) and \( \Omega(f(n)) \).

We generally prove an algorithm to be an \( O(f(n)) \) algorithm and then try to see whether it is also an \( \Omega(f(n)) \) algorithm. Notice that the proof that the algorithm requires \( O(f(n)) \) time does not imply that it would actually take \( c f(n) \) time to solve all classes of problems of the type we are studying. The upper bound of \( c f(n) \) could be "too loose" and might never be achieved. There is always a distinct possibility that by conducting a more clever analysis of the algorithm we might be able to improve the upper bound of \( c f(n) \), replacing it by a "tighter" bound. However, if we prove that the algorithm is also \( \Omega(f(n)) \), we know that the upper bound of \( c f(n) \) is "tight" and cannot be improved by more than a constant factor. This result would imply that the algorithm can actually achieve its upper bound and no tighter bound on the algorithm's running time is possible.

**Potential Functions and Amortized Complexity**

An algorithm typically performs some basic operations repetitively with each operation performing a sequence of steps. To bound the running time of the algorithm we must bound the running time of each of its basic operations. We typically bound the total number of steps associated with an operation using the following approach: We obtain a bound on the number of steps per operation, obtain a bound on the number of operations, and then take a product of the two bounds. In some of the
algorithms that we study in this book, the time required for a certain operation might vary depending on the problem data and/or the stage the algorithm is in while solving a problem. Although the operation might be easy to perform most of the time, occasionally it might be quite expensive. When this happens and we consider the time for the operation corresponding to the worst-case situation, we could greatly overestimate the running time of the algorithm. In this situation, a more global analysis is required to obtain a “tighter” bound on the running time of the operation. Rather than bounding the number of steps per operation and the number of operations executed in the algorithm, we should try to bound the total number of steps over all executions of these operations. We often carry out this type of worst-case analysis using a potential function technique.

We illustrate this concept on a problem of inserting and removing data from a data structure known as a stack (see Appendix A for a discussion of this data structure). On a stack $S$, we perform two operations:

- $push(x, S)$. Add element $x$ to the top of the stack $S$.
- $popall(S)$. Pop (i.e., take out) every element of $S$.

The operation $push(x, S)$ requires $O(1)$ time and the operation $popall(S)$ requires $O(|S|)$ time. Now assume that starting with an empty stack, we perform a sequence of $n$ operations in which push and popall operations occur in a random order. What is the worst-case complexity of performing this sequence of $n$ operations?

A naive worst-case analysis of this problem might proceed as follows. Since we require at most $n$ push operations, and each push takes $O(1)$ time, the push operations require a total of $O(n)$ time. A popall requires $O(|S|)$ time and since $|S| \geq n$, the complexity of this operation is $O(n)$. Since our algorithm can invoke at most $n$ popall operations, these operations take a total of $O(n^2)$ time. Consequently, a random sequence of $n$ push and popall operations has a worst-case complexity of $O(n^2)$.

However, if we look closely at the arguments we will find that the bound of $O(n^2)$ is a substantial overestimate of the algorithm’s computational requirements. A popall operation pops $|S|$ items from the stack, one by one until the stack becomes empty. Now notice that any element that is popped from the stack must have been pushed into the stack at some point, and since the number of push operations is at most $n$, the total number of elements popped out of the stack must be at most $n$. Consequently, the total time taken by all popall operations is $O(n)$. We can therefore conclude that a random sequence of $n$ push and popall operations has a worst-case complexity of $O(n)$.

Let us provide a formal framework, using potential functions, for conducting the preceding arguments. Potential function techniques are general-purpose techniques for establishing the complexity of an algorithm by analyzing the effects of different operations on an appropriately defined function. The use of potential functions enables us to define an “accounting” relationship between the occurrences of various operations of an algorithm so that we can obtain a bound on the operations that might be difficult to obtain using other arguments.

Let $\phi(k) = |S|$ denote the number of items in the stack at the end of the $k$th step; for the purpose of this argument we define a step as either a push or a popall operation. We assume that we perform the popall step on a nonempty stack; for otherwise, it requires $O(1)$ time. Initially, $\phi(0) = 0$. Each push operation increases $\phi(k)$ by 1 unit and takes 1 unit of time. Each popall step decreases $\phi(k)$ by at least 1 unit and requires time proportional to $\phi(k)$. Since the total increase in $\phi$ is at most $n$ (because we invoke at most $n$ popall steps), the total decrease in $\phi$ is also at most $n$. Consequently, the total time taken by all push and popall steps is $O(n)$.

This argument is fairly representative of the potential function arguments. Our objective was to bound the time for the popalls. We did so by defining a potential function that decreases whenever we perform a popall. The potential increases only when we perform a push. Thus we can bound the total decrease by the total increase in $\phi$. In general, we bound the number of steps of one type by using known bounds on the number of steps of other types.

The analysis we have just discussed is related to the concept known as amortized complexity. An operation is said to be of amortized complexity $O(f(n))$ if the time to perform a sequence of $k$ operations is $O(kf(n))$ for sufficiently large $k$.

In our preceding example, the worst-case complexity of performing $k$ popalls for $k \geq n$ is $O(k)$; hence the amortized complexity of the popall operation is $O(1)$.

Roughly speaking, the amortized complexity of an operation is the “average” worst-case complexity of the operation so that the total obtained using this average will indeed be an upper bound on the number of steps performed by the algorithm.

**Parameter Balancing**

We frequently use the parameter balancing technique in situations when the running time of an algorithm is a function of a parameter $k$ and we wish to determine the value of $k$ that gives the smallest running time. To be more specific, suppose that the running time of an algorithm is $O(f(n, m, k)) + g(n, m, k)$ and we wish to determine an optimal value of $k$. We shall assume that $f(n, m, k) \geq 0$ and $g(n, m, k) \geq 0$ for all feasible values of $k$. The optimization problem is easy to solve if the functions $f(n, m, k)$ and $g(n, m, k)$ are both either monotonically increasing or monotonically decreasing in $k$. In the former case, we set $k$ to the smallest possible value, and in the latter case, we set $k$ to the largest possible value. Finding the optimal value of $k$ is more complex if one function is monotonically decreasing and the other function is monotonically increasing. So let us assume that $f(n, m, k)$ is monotonically decreasing in $k$ and $g(n, m, k)$ is monotonically increasing in $k$.

One method for selecting the optimal value of $k$ is to use differential calculus. That is, we differentiate $f(n, m, k) + g(n, m, k)$ with respect to $k$, set the resulting expression equal to zero, and solve for $k$. A major drawback of this approach is that finding a value of $k$ that will set the expression to zero, and so determine the optimal value of $k$, is often a difficult task. Consider, for example, a shortest path algorithm (which we discuss in Section 4.7) that runs in time $O(m \log n + n \log m)$. In this case, choosing the optimal value of $k$ is not trivial. We can restate the algorithm’s time bound as $O(m \log n + n \log n)(\log k)$. The derivative of this expression with respect to $k$ is

$$
(nk \log n \log k - m \log n - nk \log n)(k \log k)^2.
$$

Sec. 3.2 Complexity Analysis
cycle. Recall from the illustration shown in Figure 5.4 how the predecessor graph might contain a directed cycle. This algorithm works as follows. We first designate the source node as labeled and all other nodes are unlabeled. Then, one by one, we examine each unlabeled node \( k \) and perform the following operations: We assign a label \( d(k) \) to node \( k \), trace the predecessor indices starting at node \( k \), and assign the label \( k \) to all the nodes encountered until we reach the first already labeled node, say node \( i \). If nodes \( k \) and \( i \) have the same labels, then the predecessor graph contains a cycle, which must be a negative cycle (why?). The reader can verify that this algorithm requires \( O(n) \) time to check the presence of a directed cycle in the predecessor graph. Consequently, if we apply this algorithm after every \( n \) distance updates for some constant \( n \), the computations it performs will not add to the worst-case complexity of any label-correcting algorithm.

In general, at the time that the algorithm relabels node \( j, d(j) = d(i) + c_{ij} \) for some node \( i \) which is the predecessor of \( j \). We refer to the arc \( (i, j) \) as a predecessor arc. Subsequently, \( d(i) \) might decrease, and the labels will satisfy the condition \( d(j) \geq d(i) + c_{ij} \) as long as \( \text{pred}(j) = i \). Suppose that \( P \) is a path of predecessor arcs from node \( i \) to node \( j \). The inequalities \( d(j) \geq d(k) + c_{kj} \) for all arcs \( (k, j) \) on this path imply that \( d(j) \) is at least the length of this path. Consequently, no node \( j \) with \( d(j) \leq -nC \) is connected to node \( s \) on a path consisting only of predecessor arcs. We conclude that tracing back predecessor arcs from node \( j \) must lead to a cycle, and by Exercise 5.56, any such cycle must be negative.

The FIFO label-correcting algorithm is also capable of easily detecting the presence of a negative cycle. Recall that we can partition the node examinations in the FIFO algorithm into several phases and that the algorithm examines any node at most once within each pass. To implement this algorithm, we record the number of times that the algorithm examines each node. If the network contains a negative cycle, it examines any node at most \( (n - 1) \) times [because it makes at most \( (n - 1) \) passes]. Therefore, if it examines a node more than \( (n - 1) \) times, the network must contain a negative cycle. We can also use the technique described in the preceding paragraph to identify negative cycles.

The FIFO label-correcting algorithm detects the presence of negative cycles or obtains shortest path distances in \( O(nm) \) time, which is the fastest available strongly polynomial-time algorithm for networks with arbitrary arc lengths. However, for problems that satisfy the similarity assumption, other weaker polynomial-time algorithms run faster than the FIFO algorithm. These approaches formulate the shortest path problem as an assignment problem (as described in Section 12.7) and then use an \( O(n^{1/2}m \log(nC)) \) time assignment algorithm to solve the problem (i.e., either finds a shortest path or detects a negative cycle).

### 5.6 ALL-PAIRS SHORTEST PATH PROBLEM

The all-pairs shortest path problem requires that we determine shortest path distances between every pair of nodes in a network. In this section we suggest two approaches for solving this problem. The first approach, called the repeated shortest path algorithm, is well suited for sparse networks. The second approach is a generalization of the label-correcting algorithm discussed in previous sections; we refer to this procedure as the all-pairs label-correcting algorithm. It is especially well suited for dense networks. In this section we describe the generic all-pairs label-correcting algorithm and then develop a special implementation of this generic algorithm, known as the Floyd–Warshall algorithm, that runs in \( O(n^3) \) time.

In this section we assume that the underlying network is strongly connected (i.e., it contains a directed path from any node to every other node). We can easily satisfy this assumption by selecting an arbitrary node, say \( s \), and adding arcs \( (s, i) \) and \( (i, s) \) of sufficiently large cost for all \( i \in N - \{s\} \), if these arcs do not already exist. For reasons explained earlier, we also assume that the network does not contain a negative cycle. All the algorithms we discuss, however, are capable of detecting the presence of a negative cycle. We discuss situations with negative cycles at the end of this section.

#### Repeated Shortest Path Algorithm

If the network has nonnegative arc lengths, we can solve the all-pairs shortest path problem by applying any single-source shortest path algorithm \( n \) times, considering each node as the source node once. If \( S(n, m, C) \) denotes the time needed to solve a shortest path problem with nonnegative arc lengths, this approach solves the all-pairs shortest path problem in \( O(n(n, m, C)) \) time.

If the network contains some negative arcs, we first transform the network to one with nonnegative arc lengths. We select a node \( s \) and use the FIFO label-correcting algorithm, described in Section 5.4, to compute the shortest distances from node \( s \) to all other nodes. The algorithm either detects the presence of a negative cycle or terminates with the shortest path distances \( d(j) \). In the first case, the all-pairs shortest path problem has no solution, and in the second case, we consider the shortest path problem with arc lengths equal to their reduced arc lengths with respect to the distance labels \( d(j) \). Recall from Section 5.2 that the reduced arc length of an arc \( (i, j) \) with respect to the distance labels \( d(j) \) is \( c'_{ij} = c_{ij} + d(i) - d(j) \), and if the distance labels are shortest path distances, then \( c'_{ij} \geq 0 \) for all arcs \( (i, j) \) in \( A \) [see Property 5.2(c)]. Since this transformation produces nonnegative reduced arc lengths, we can then apply the single-source shortest path algorithm for problems with nonnegative arc lengths \( n \) times (by considering each node as a source once) to determine shortest path distances between all pairs of nodes in the transformed network. We obtain the shortest path distance between nodes \( k \) and \( l \) in the original network by adding \( d(l) - d(k) \) to the corresponding shortest path distance in the transformed network [see Property 5.2(b)]. This approach requires \( O(nm) \) time to solve the first shortest path problem, and if the network contains no negative cycles, it requires an extra \( O(nS(n, m, C)) \) time to compute the remaining shortest path distances. Therefore, this approach determines all pairs shortest path distances in \( O(nm + nS(n, m, C)) = O(nS(n, m, C)) \) time. We have established the following result.

**Theorem 5.4.** The repeated shortest path algorithm solves the all-pairs shortest path problem in \( O(nS(n, m, C)) \) time.

In the remainder of this section we study the generic all-pairs label-correcting algorithm. Just as the generic label-correcting algorithm relies on shortest path optimality conditions, the all-pairs label-correcting algorithm relies on all-pairs shortest path optimality conditions, which we study next.

**Sec. 5.6 All-Pairs Shortest Path Problem**
All-Pairs Shortest Path Optimality Conditions

Let \( [i, j] \) denote a pair of nodes \( i \) and \( j \) in the network. The all-pairs label-correcting algorithm maintains a distance label \( d[i, j] \) for every pair of nodes; this distance label, if finite, represents the length of some directed walk from node \( i \) to node \( j \) and hence will be upper bound on the shortest path length from node \( i \) to node \( j \). The algorithm updates the matrix of distance labels until they represent shortest path distances. It uses the following generalization of Theorem 5.1:

**Theorem 5.5 (All-Pairs Shortest Path Optimality Conditions).** For every pair of nodes \( [i, j] \in N \times N \), let \( d[i, j] \) represent the length of some directed path from node \( i \) to node \( j \) satisfying \( d[i, j] = 0 \) for all \( i \in N \), and \( d[i, j] \leq c_{ij} \) for all \( (i, j) \in E \). These distances represent all-pairs shortest path distances if and only if they satisfy the following all-pairs shortest path optimality conditions:

\[
d[i, j] \leq d[i, k] + d[k, j] \quad \text{for all nodes } i, j, \text{ and } k. \tag{5.3}
\]

**Proof.** We use a contradiction argument to establish that the shortest path distances \( d[i, j] \) must satisfy the conditions (5.3). Suppose that \( d[i, k] + d[k, j] < d[i, j] \) for nodes \( i, j, \) and \( k \). The union of the shortest paths from node \( i \) to node \( k \) and node \( k \) to node \( j \) is a directed walk of length \( d[i, k] + d[k, j] \) from node \( i \) to node \( j \). This directed walk decomposes into a directed path, say \( P \), from node \( i \) to node \( j \) and some directed cycles (see Exercise 3.51). Since each directed cycle in the network has nonnegative length, the length of the path \( P \) is at most \( d[i, k] + d[k, j] \), contradicting the optimality of \( d[i, j] \).

We now show that if the distance labels \( d[i, j] \) satisfy the conditions in (5.3), they represent shortest path distances. We use an argument similar to the one we used in proving Theorem 5.1. Let \( P \) be a directed path of length \( d[i, j] \) consisting of the sequence of nodes \( i = i_1 = i_2 = \cdots = i_k = j \). The condition (5.3) implies that

\[
d[i, j] = d[i_1, i_2] \leq d[i_1, i_2] + d[i_2, i_3] \leq c_{i_1i_2} + d[i_1, i_2],
\]

\[
d[i_2, i_3] \leq c_{i_2i_3} + d[i_2, i_3],
\]

\[
\vdots
\]

\[
d[i_{k-1}, i_k] \leq c_{i_{k-1}i_k}.
\]

These inequalities, in turn, imply that

\[
d[i, j] \leq c_{i_1i_2} + c_{i_2i_3} + \cdots + c_{i_{k-1}i_k} = \sum_{(i, j) \in P} c_{ij}.
\]

Therefore, \( d[i, j] \) is a lower bound on the length of any directed path from node \( i \) to node \( j \). By assumption, \( d[i, j] \) is also an upper bound on the shortest path length from node \( i \) to node \( j \). Consequently, \( d[i, j] \) must be the shortest path length between these nodes which is the derived conclusion of the theorem.

**All-Pairs Generic Label-Correcting Algorithm**

The all-pairs shortest path optimality conditions (throughout the remainder of this section we refer to these conditions simply as the optimality conditions) immediately yield the following generic all-pairs label-correcting algorithm: Start with some distance labels \( d[i, j] \) and successively update these until they satisfy the optimality conditions. Figure 5.6 gives a formal statement of the algorithm. In the algorithm we refer to the operation of checking whether \( d[i, j] > d[i, k] + k[j, j] \), and if so, then setting \( d[i, j] = d[i, k] + d[k, j] \) as a triple operation.

**Algorithm all-pairs label-correcting:**

```
begin
set \( d[i, j] = \infty \) for all \( (i, j) \in N \times N \);
set \( d[i, i] = 0 \) for all \( i \in N \);
for each \( (i, j) \in A \) do \( d[i, j] = c_{ij} \);
while the network contains three nodes \( i, j, \) and \( k \) satisfying \( d[i, j] > d[i, k] + d[k, j] \) do \( d[i, j] = d[i, k] + d[k, j] \);
end;
```

**Figure 5.6** Generic all-pairs label-correcting algorithm.

To establish the finiteness and correctness of the generic all-pairs label-correcting algorithm, we assume that the data are integral and that the network contains no negative cycle. We first consider the correctness of the algorithm. At every step the algorithm maintains the invariant property that whenever \( d[i, j] < \infty \), the network contains a directed walk of length \( d[i, j] \) from node \( i \) to node \( j \). We can use induction on the number of iterations to show that this property holds at every step. Now consider the directed walk of length \( d[i, j] \) from node \( i \) to node \( j \) at the point when the algorithm terminates. This directed walk decomposes into a directed path, say \( P \), from node \( i \) to node \( j \), and possibly some directed cycles. None of these cycles could have a positive length, for otherwise we would contradict the optimality of \( d[i, j] \).

Therefore, all of these cycles must have length zero. Consequently, the path \( P \) must have length \( d[i, j] \). The distance labels \( d[i, j] \) also satisfy the optimality conditions (5.3), for these conditions are the termination criteria of the algorithm. This conclusion establishes the fact that when the algorithm terminates, the distance labels represent shortest path distances.

Now consider the finiteness of the algorithm. Since all arc lengths are integer and \( C \) is the largest magnitude of any arc length, the maximum (finite) distance label is bounded from above by \( nC \) and the minimum distance label is bounded from below by \( -nC \). Each iteration of the generic all-pairs label-correcting algorithm decreases some \( d[i, j] \). Consequently, the algorithm terminates within \( O(n^3) \) iterations. This bound on the algorithm's running time is pseudopolynomial and is not attractive from the viewpoint of worst-case complexity. We next describe a specific implementation of the generic algorithm, known as the Floyd–Warshall algorithm, that solves the all-pairs shortest path problem in \( O(n^3) \) time.

**Floyd–Warshall Algorithm**

Notice that given a matrix of distances \( d[i, j] \), we need to perform \( O(n^3) \) triple operations in order to test the optimality of this solution. It is therefore surprising that the Floyd–Warshall algorithm obtains a matrix of shortest path distances within \( O(n^3) \) computations. The algorithm achieves this bound by applying the triple op-
erations cleverly. The algorithm is based on inductive arguments developed by an application of a dynamic programming technique.

Let \( d^k[i,j] \) represent the length of a shortest path from node \( i \) to node \( j \) subject to the condition that this path uses only the nodes \( 1, 2, \ldots, k - 1 \) as internal nodes. Clearly, \( d^{k+1}[i,j] \) represents the actual shortest path distance from node \( i \) to node \( j \). The Floyd–Warshall algorithm first computes \( d^1[i,j] \) for all node pairs \( i \) and \( j \). Using \( d^1[i,j] \), it then computes \( d^2[i,j] \) for all node pairs \( i \) and \( j \). It repeats this process until it obtains \( d^{k+1}[i,j] \) for all node pairs \( i \) and \( j \), when it terminates. Given \( d^k[i,j] \), the algorithm computes \( d^{k+1}[i,j] \) using the following property.

**Property 5.6.** \( d^{k+1}[i,j] = \min(d^k[i,j], d^k[i,k] + d^k[k,j]) \).

This property is valid for the following reason. A shortest path that uses only the nodes \( 1, 2, \ldots, k \) as internal nodes either (1) does not pass through node \( k \), in which case \( d^{k+1}[i,j] = d^k[i,j] \), or (2) does pass through node \( k \), in which case \( d^{k+1}[i,j] = d^k[i,k] + d^k[k,j] \). Therefore, \( d^{k+1}[i,j] = \min(d^k[i,j], d^k[i,k] + d^k[k,j]) \).

Figure 5.7 gives a formal description of the Floyd–Warshall algorithm.

**Algorithm Floyd–Warshall:**

begin
  for all node pairs \( i, j \in N \times N \) do
    \( d[i,j] := \infty \) and \( \text{pred}[i,j] := 0 \);
  for all nodes \( i \in N \) do \( d[i,i] := 0 \);
  for each arc \( (i,j) \in A \) do \( d[i,j] := c_{ij} \) and \( \text{pred}[i,j] := i \);
  for each \( k \in 1 \) to \( n \) do
    for each \( i,j \in N \times N \) do
      if \( d[i,j] > d[i,k] + d[k,j] \) then
        begin
          \( d[i,j] := d[i,k] + d[k,j] \);
          \( \text{pred}[i,j] := \text{pred}[k,j] \);
        end;
end;

The Floyd–Warshall algorithm uses predecessor indices, \( \text{pred}[i,j] \), for each node pair \( [i,j] \). The index \( \text{pred}[i,j] \) denotes the last node prior to node \( j \) in the tentative shortest path from node \( i \) to node \( j \). The algorithm maintains the invariant property that when \( d[i,j] \) is finite, the network contains a path from node \( i \) to node \( j \) of length \( d[i,j] \). Using the predecessor indices, we can obtain this path, say \( P \), from node \( k \) to node \( i \) as follows. We backtrack along the path \( P \) starting at node \( i \). Let \( g = \text{pred}[i,k] \). Then \( g \) is the node prior to node \( i \) in \( P \). Similarly, \( h = \text{pred}[k,g] \) is the node prior to node \( g \) in \( P \), and so on. We repeat this process until we reach node \( k \).

The Floyd–Warshall algorithm clearly performs \( n \) major iterations, one for each \( k \), and within each major iteration, it performs \( O(1) \) computations for each node pair. Consequently, it runs in \( O(n^3) \) time. We thus have established the following result.

**Theorem 5.7.** The Floyd–Warshall algorithm computes shortest path distances between all pairs of nodes in \( O(n^3) \) time.

---

**Detection of Negative Cycles**

We now address the issue of detecting a negative cycle in the network if one exists. In the generic all-pairs label-correcting algorithm, we incorporate the following two tests whenever the algorithm updates a distance label \( d[i,j] \) during a triple iteration:

1. If \( i = j \), check whether \( d[i,j] < 0 \).
2. If \( i \neq j \), check whether \( d[i,j] < -nC \).

If either of these two tests is true, the network contains a negative cycle. To verify this claim, consider the first time during a triple iteration when \( d[i,i] = 0 \) for some node \( i \). At this time \( d[i,i] = d[i,k] + d[k,i] \) for some node \( k \neq i \). This condition implies that the network contains a directed walk from node \( i \) to node \( k \) and a directed walk from node \( k \) to node \( i \), and that the sum of the lengths of these two walks is \( d[i,i] \), which is negative. The union of these two walks is a closed walk, which can be decomposed into a set of directed cycles (see Exercise 3.51).

Since \( d[i,i] < 0 \), at least one of these directed cycles must be negative.

We next consider the situation in which \( d[i,j] < -nC \) for some node pair \( i \) and \( j \). Consider the first time during a triple iteration when \( d[i,j] < -nC \). At this time the network contains a directed walk from node \( i \) to node \( j \) of length less than \(-nC\).

As we observed previously, we can decompose this walk into a directed path \( P \) from node \( i \) to node \( j \) and some directed cycles. Since the path \( P \) must have a length of at least \((-n - 1)C \), at least one of these cycles must be a negative cycle.

Finally, we observe that if the network contains a negative cycle, then eventually \( d[i,i] < 0 \) for some node \( i \), or \( d[i,j] < -nC \) for some node pair \( [i,j] \), because the distance labels continue to decrease by an integer amount at every iteration. Therefore, the generic label-correcting algorithm will always determine a negative cycle if one exists.

In the Floyd–Warshall algorithm, we detect the presence of a negative cycle simply by checking the condition \( d[i,i] < 0 \) whenever we update \( d[i,i] \) for some node \( i \). It is easy to see that whenever \( d[i,i] < 0 \), we have detected the presence of a negative cycle. In Exercise 5.37 we show that whenever the network contains a negative cycle, then during the computations we will eventually satisfy the condition \( d[i,i] < 0 \) for some \( i \).

We can also use an extension of the method described in Section 5.5, using the predecessor graph, to identify a negative cycle in the Floyd–Warshall algorithm. The Floyd–Warshall algorithm maintains a predecessor graph for each node \( k \) in the network, which in the absence of a negative cycle is a directed out-tree rooted at node \( k \) (see Section 5.3). If the network contains a negative cycle, eventually the predecessor graph contains a cycle. For any node \( k \), the predecessor graph consists of the arcs \( \{ \text{pred}[k,i], i \in N - \{k\} \} \). Using the method described in Section 5.5, we can determine whether or not any predecessor graph contains a cycle. Checking this condition for every node requires \( O(n^2) \) time. Consequently, if we use this method after every \( an^2 \) triple iterations for some constants \( a \), the computations will not add to the worst-case complexity of the Floyd–Warshall algorithm.

---

**Sec. 5.6 All-Pairs Shortest Path Problem**
5.7 MINIMUM COST-TO-TIME RATIO CYCLE PROBLEM

The minimum cost-to-time ratio cycle problem is defined on a directed graph $G$ with both a cost and a travel time associated with each arc: we wish to find a directed cycle in the graph with the smallest ratio of its cost to its travel time. The minimum cost-to-time ratio problem arises in an application known as the tramp steamer problem, which we defined in Application 4.4. A tramp steamer travels from port to port, carrying cargo and passengers. A voyage of the steamer from port $i$ to port $j$ earns $p_{ij}$ units of profit and requires time $t_{ij}$. The captain of the steamer wants to know what ports the steamer should visit, and in which order, in order to maximize its mean daily profit. We can solve this problem by identifying a directed cycle with the largest possible ratio of total profit to total travel time. The tramp steamer then continues to sail indefinitely around this cycle.

In the tramp steamer problem, we wish to identify a directed cycle $W$ of $G$ with the maximum ratio \[ \frac{\sum_{(i,j) \in W} c_{ij}}{\sum_{(i,j) \in W} t_{ij}}. \] We can convert this problem into a minimization problem by defining the cost $\mu$ of each arc $(i,j)$ as $c_{ij} = -p_{ij}$. We then seek a directed cycle $W$ with the minimum value for the ratio \[ \mu(W) = \frac{\sum_{(i,j) \in W} c_{ij}}{\sum_{(i,j) \in W} t_{ij}}. \]

We assume in this section that all data are integral, that $t_{ij} \geq 0$ for every arc $(i,j) \in A$, and that $\sum_{(i,j) \in W} t_{ij} > 0$ for every directed cycle $W$ in $G$.

We can solve the minimum cost-to-time ratio cycle problem (or, simply, the minimum ratio problem) by repeated applications of the negative cycle detection algorithm. Let $\mu^*$ denote the optimal objective function value of the minimum cost-to-time ratio cycle problem. For any arbitrary value of $\mu$, let us define the length of each arc as $l_{ij} = c_{ij} - \mu t_{ij}$. With respect to these arc lengths, we could encounter three situations:

\[ \text{Case 1. } G \text{ contains a negative (length) cycle } W. \]

In this case, $\sum_{(i,j) \in W} (c_{ij} - \mu t_{ij}) < 0$. Alternatively,

\[ \mu > \frac{\sum_{(i,j) \in W} c_{ij}}{\sum_{(i,j) \in W} t_{ij}} \geq \mu^*. \]  

(5.4)

Therefore, $\mu$ is a strict upper bound on $\mu^*$.

\[ \text{Case 2. } G \text{ contains no negative cycle, but does contain a zero-length cycle } W^+. \]

The fact that $G$ contains no negative cycle implies that $\sum_{(i,j) \in W} (c_{ij} - \mu t_{ij}) \geq 0$ for every directed cycle $W$. Alternatively,

\[ \mu = \frac{\sum_{(i,j) \in W^+} c_{ij}}{\sum_{(i,j) \in W^+} t_{ij}} \text{ for every directed cycle } W. \]  

(5.5)

Similarly, the fact that $G$ contains a zero-length cycle $W^+$ implies that

\[ \mu = \frac{\sum_{(i,j) \in W^+} c_{ij}}{\sum_{(i,j) \in W^+} t_{ij}} \text{ for every directed cycle } W^+. \]  

(5.6)

The conditions (5.5) and (5.6) imply that $\mu = \mu^*$, so $W^+$ is a minimum cost-to-time ratio cycle.

\[ \text{Case 3. } \text{Every directed cycle } W \text{ in } G \text{ has a positive length.} \]

In this case $\sum_{(i,j) \in W} (c_{ij} - \mu t_{ij}) > 0$ for every directed cycle $W$. Alternatively,

\[ \mu < \frac{\sum_{(i,j) \in W} c_{ij}}{\sum_{(i,j) \in W} t_{ij}} \text{ for every directed cycle } W. \]  

(5.7)

Consequently, $\mu$ is a strict lower bound on $\mu^*$.

The preceding case analysis suggests the following search procedure for solving the minimum cost-to-time ratio problem. We guess a value $\mu$ for $\mu^*$, define arc lengths as $(c_{ij} - \mu t_{ij})$, and apply any shortest path algorithm. If the algorithm identifies a negative cycle, $\mu$ exceeds $\mu^*$ and our next guess should be smaller. If the algorithm terminates with shortest path distances, we look for a zero-length cycle (as described in Exercise 5.19). If we do find a zero-length cycle $W^+$, then we stop; otherwise, $\mu$ is smaller than $\mu^*$, so our next guess should be larger. To implement this general solution approach, we need to define what we mean by "smaller" and "larger." The following two search algorithms provide us with two methods for implementing this approach.

Sequential search algorithm. Let $\mu^*$ be a known upper bound on $\mu^*$. If we solve the shortest path problem with $(c_{ij} - \mu^* t_{ij})$ as arc lengths, we either find a zero-length cycle $W$ or find a negative cycle $W$. In the former case, $W$ is a minimum