The McGraw-Hill Companies

Introduction to the Design and Analysis of Algorithms

A Strategic Approach

R. C. T. Lee
National Chi Nan University, Taiwan

S. S. Tseng
National Chiao Tung University, Taiwan

R. C. Chang
National Chiao Tung University, Taiwan

Y. T. Tsai
Providence University, Taiwan

Singapore • Boston • Burr Ridge, IL • Dubuque, IA • Madison, WI • New York
San Francisco • St. Louis • Bangkok • Bogotá • Caracas • Kuala Lumpur
Lisbon • London • Madrid • Mexico City • Milan • Montreal • New Delhi
Santiago • Seoul • Sydney • Taipei • Toronto

The McGraw-Hil Companies

Introduction to the Design and Analysis of Algorithms
A Strategic Approach

1 2 3 4 5 6 7 8 9 10 TBC 09 08 07 06 05

When ordering this title, use ISBN 007-124346-1

Printed in Singapore

About the Authors

R. C. T. LEE received his B. Sc. degree from the Department of Electrical Engineering of National Taiwan University and Ph. D. degree from the Department of Electrical Engineering and Computer Science from University of California, Berkeley. He is a professor of both the Computer Science and Information Engineering Department and Information Management Department of National Chi Nan University. Professor Lee is an IEEE fellow. He co-authored *Symbolic Logic and Mechanical Theorem Proving*, which has been translated into Japanese, Russian and Italian.

S. S. TSENG and **R. C. CHANG** both received their B. Sc. and Ph. D. degrees from the Department of Computer Engineering, National Chiao Tung University. Both of them are professors in the Department of Computer and Information Science, National Chiao Tung University, Taiwan.

Y. T. TSAI received his B. Sc. degree from the Department of Computer and Information Science of National Chiao Tung University and Ph. D. degree from the Department of Computer Science of the National Tsing-Hua University. He is presently an associate professor in the Department of Information Science and Management of the Providence University.

Contents

List of Figures

Chapter 1

Chapter 2

Chapter 3

Chapter 6

Chapter 7

Chapter 9

Chapter 11

Chapter 12

Preface

There are probably many reasons for studying algorithms. The primary one is to enable readers to use computers efficiently. A novice programmer, without good knowledge of algorithms, may prove to be a disaster to his boss and the organization where he works. Consider someone who is going to find a minimal spanning tree. If he does so by examining all possible spanning trees, no computer which exists now, or will exist in the future, is good enough for him. Yet, if he knows Prim's algorithm, an IBM personal computer is sufficient. In another example where someone wants to solve speech recognition, it would be very difficult for him to get started. Yet, if he knows the longest common subsequence problem can be solved by the dynamic programming approach, he will find the problems to be surprisingly easy. Algorithm study is not only important to computer scientists. Communication engineers also use dynamic programming or A^* algorithms in coding. The largest group of non-computer scientists who benefit very much from studying algorithms are those who work on molecular biology. When one wants to compare two DNA sequences, two protein, or RNA, 3-dimensional structures, one needs to know sophisticated algorithms.

Besides, studying algorithms is great fun. Having studied algorithms for so long, the authors are still quite excited whenever they see a new and well-designed algorithm or some new and brilliant idea about the design and analysis of algorithms. They feel that they have a moral responsibility to let others share their fun and excitement. Many seemingly difficult problems can actually be solved by polynomial algorithms while some seemingly trivial problems are proved to be NP-complete. The minimal spanning tree problem appears to be quite difficult to many novice readers, yet it has polynomial algorithms. If we twist the problem a little bit so that it becomes the traveling salesperson problem, it suddenly becomes an NP-hard problem. Another case is the 3-satisfiability problem. It is an NP-complete problem. By lowering the dimensionality, the 2-satisfiability problem becomes a P problem. It is always fascinating to find out these facts.

In this book, we adopt a rather different approach to introduce algorithms. We actually are not introducing algorithms; instead, we are introducing the strategies which can be used to design algorithms. The reason is simple: It is obviously more important to know the basic principles, namely the strategies, used in designing algorithms, than algorithms themselves. Although not every algorithm is based on one of the strategies which we introduce in this book, most of them are.

Prune-and-search, amortized analysis, on-line algorithms, and polynomial-time approximation schemes are all relatively new ideas. Yet they are quite important ideas. Numerous newly developed algorithms are based on amortized analysis, as you will see at the end of the chapter on this topic.

We start by introducing strategies used to design polynomial algorithms. When we have to cope with problems which appear to be difficult ones and do not have polynomial algorithms at present, we will introduce the concept of NP-completeness. It is easy to apply the concept of NP-completeness, yet it is often difficult to grasp the physical meaning of it. The critical idea is actually why every NP problem instance is related to a set of Boolean formulas and the answer of this problem instance is "yes" if and only if the formulas are satisfiable. Once the reader understands this, he can easily appreciate the importance of NP-completeness. We are confident that the examples presented to explain this idea will help most students to easily understand NP-completeness.

This book is intended to be used as a textbook for senior undergraduate students and junior graduate students. It is our experience that we cannot cover all the materials if this is used in one semester (roughly 50 hours). Therefore, we recommend that all chapters be touched evenly, but not completely if only one semester is available. Do not ignore any chapter! The chapter on NP-completeness is very important and should be made clear to the students. The most difficult chapter is Chapter 10 (Amortized Analysis) where the mathematics is very much involved. The instructor should pay close attention to the basic ideas of amortized analysis, instead of being bogged down in the proofs. In other words, the students should be able to understand why a certain data structure, coupled with a good algorithm, can perform very well in the amortized sense. Another rather difficult chapter may be Chapter 12 (On-Line Algorithms).

Most algorithms are by no means easy to understand. We have made a tremendous effort to present the algorithms clearly. Every algorithm that is introduced is accompanied by examples. Every example is presented with figures. We have provided more than 400 figures in our book which will be very helpful to novice readers.

We have also cited many books and papers on algorithm design and analysis. Latest results are specifically mentioned so that the readers can easily find directions for further research. The Bibliography lists 825 books and papers, which correspond to 1,095 authors.

We present experimental results when it is appropriate. Still, the instructor should encourage students to test the algorithms by implementing them. At the

end of each chapter, is a list of papers for further reading. It is important to encourage the students to read some of them in order to advance their understanding of algorithms. Perhaps they will appreciate how hard the authors worked to decipher many of the difficult-to-read papers! We have also included some programs, written in Java, for the students to practice.

It is simply impossible to name all the persons who have helped the authors tremendously in preparing this book; there are just too many of them. However, they all belong to one class. They are either the authors' students or colleagues (many students later became colleagues). In the weekly Friday evening seminars, we always had lively discussions. These discussions pointed out new directions of algorithm research and helped us decide which material should be included in the book. Our graduate students have been monitoring roughly 20 academic journals to ensure that every important paper on algorithms is stored in a database with keywords attached to it. This database is very valuable for writing this book. Finally, they read the manuscript of this book, offered criticisms and performed experiments for us. We cannot imagine completing this book without the help from our colleagues and students. We are immensely grateful to all of them.

chapter

1

Introduction

This book introduces the design and analysis of algorithms. Perhaps it is meaningful to discuss a very important question first: Why should we study algorithms? It is commonly believed that in order to obtain high speed computation, it suffices to have a very high speed computer. This, however, is not entirely true. We illustrate this with an experiment whose results make it clear that a good algorithm implemented on a slow computer may perform much better than a bad algorithm implemented on a fast computer. Consider the two sorting algorithms, insertion sort and quick sort, roughly described below.

Insertion sort examines a sequence of data elements from the left to the right one by one. After each data element is examined, it will be inserted into an appropriate place in an already sorted sequence. For instance, suppose that the sequence of data elements to be sorted is

11, 7, 14, 1, 5, 9, 10.

Insertion sort works on the above sequence of data elements as follows:

Sorted sequence	**Unsorted sequence**
11	7, 14, 1, 5, 9, 10
7, 11	14, 1, 5, 9, 10
7, 11, 14	1, 5, 9, 10
1, 7, 11, 14	5, 9, 10
1, 5, 7, 11, 14	9, 10
1, 5, 7, 9, 11, 14	10
1, 5, 7, 9, 10, 11, 14	

In the above process, consider the case where the sorted sequence is 1, 5, 7, 11, 14 and the next data element to be sorted is 9. We compare 9 with 14 first. Since 9 is smaller than 14, we then compare 9 with 11. Again we must continue. Comparing 9 with 7, we find out that 9 is larger than 7. So we insert 9 between 7 and 11.

Our second sorting algorithm is called quick sort, which will be presented in detail in Chapter 2. Meanwhile, let us just give a very high level description of this algorithm. Imagine that we have the following data elements to be sorted:

10, 5, 1, 17, 14, 8, 7, 26, 21, 3.

Quick sort will use the first data element, namely 10, to divide all data elements into three subsets: those smaller than 10, those larger than 10 and those equal to 10. That is, we shall have data represented as follows:

(5, 1, 8, 7, 3) (10) (17, 14, 26, 21).

We now must sort two sequences:

(5, 1, 8, 7, 3)
and (17, 14, 26, 21).

Note that they can be sorted independently and the dividing approach can be used recursively on the two subsets. For instance, consider

17, 14, 26, 21.

Using 17 to divide the above set of data, we have

(14) (17) (26, 21).

After we sort the above sequence 26, 21 into 21, 26, we shall have

14, 17, 21, 26

which is a sorted sequence.

Similarly,

(5, 1, 8, 7, 3)

can be sorted into

(1, 3, 5, 7, 8).

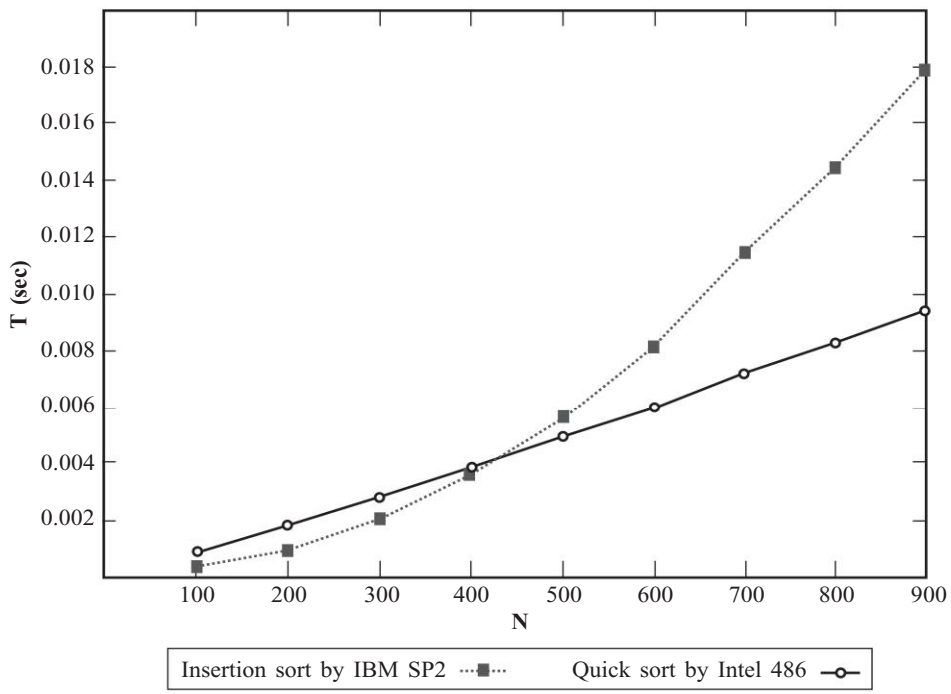
Combining all the sorted sequences into one, we have

1, 3, 5, 7, 8, 10, 14, 17, 21, 26

which is a sorted sequence.

It will be shown later that quick sort is much better than the insertion sort. The question is: how good is it? To compare quick sort and insertion sort, we implemented quick sort on an Intel 486 and insertion sort on an IBM SP2. The IBM SP2 is a super computer which defeated the chess master in 1997 while Intel 486 is only a personal computer. For each number of points, 10 sets of data were randomly generated and the average time was obtained for both algorithms. Figure 1–1 shows the experimental results. We can see that when the number of

FIGURE 1–1 Comparison of the performance of insertion sort and quick sort.



data elements is less than 400, Intel 486 implemented with quick sort is inferior to IBM SP2 with insertion sort. For the number of data elements larger than 400, Intel 486 performs much better than IBM SP2.

What does the above experiment mean? It certainly indicates one important fact: a fast computer with an inferior algorithm may perform worse than a slow computer with a superior algorithm. In other words, *if you are rich, but without a good knowledge of algorithms, you may not be able to compete with a poor fellow who understands algorithms very well*.

If we accept that it is important to study algorithms, then we must be able to analyze algorithms to determine their performance. In Chapter 2, we present a brief introduction of some basic concepts related to the analysis of algorithms. In this chapter, our algorithm analysis is introductory in every sense. We shall introduce some excellent algorithm analysis books at the end of Chapter 2.

After introducing the concept of the analysis of algorithms, we turn our attention to the complexity of problems. We note that there are easy problems and difficult problems. A problem is easy if it can be solved by an efficient algorithm, perhaps an algorithm with polynomial-time complexity. Conversely, if a problem cannot be solved by any polynomial-time algorithm, it must be a difficult problem. Usually, given a problem, if we already know that there exists an algorithm which solves the problem in polynomial time, we are sure that it is an easy problem. However, if we have not found any polynomial algorithm to solve the problem, we can hardly conclude that we can never find any polynomial-time algorithm to solve this problem in the future. Luckily, there is a theory of NP-completeness which can be used to measure the complexity of a problem. If a problem is proved to be NP-complete, then it will be viewed as a difficult problem and the probability that a polynomial-time algorithm can be found to solve it is very small. Usually, the concept of NP-completeness is introduced at the end of a textbook.

It is interesting to note here that many problems which do not appear to be difficult are actually NP-complete problems. Let us give some examples.

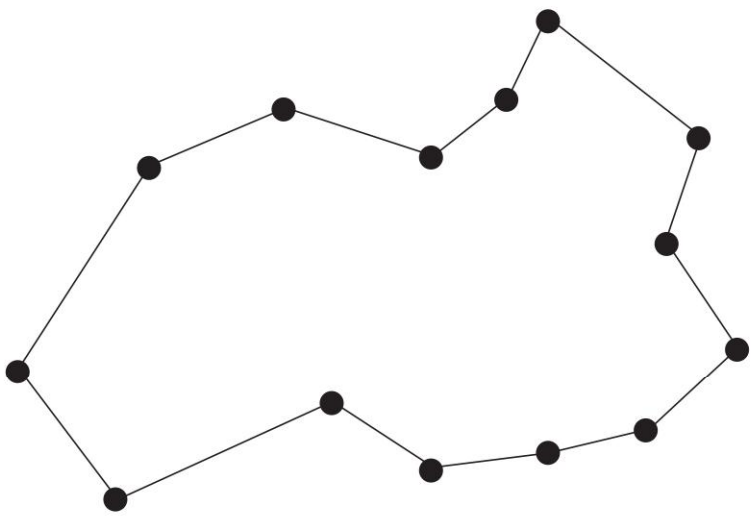
First of all, consider the 0/1 knapsack problem. An informal description of this problem is as follows: Imagine that we are fleeing from an invading army. We must leave our beloved home. We want to carry some valuable items with us. But, the total weight of the goods which we are going to carry with us cannot exceed a certain limit. How can we maximize the value of goods which we carry without exceeding the weight limit? For instance, suppose that we have the following items:

	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8
Value	10	5	1	9	3	4	11	17
Weight	7	3	3	10	1	9	22	15

If the weight limit is 14, then the best solution is to select P_1, P_2, P_3 and P_5. This problem turns out to be an NP-complete problem. As the number of items becomes large, it will be hard for us to find an optimal solution.

Another seemingly easy problem is also an NP-complete problem. This is the traveling salesperson problem. To intuitively explain it, let us consider that there are many cities. Our diligent salesperson must travel to every city, but we require that no city can be visited twice and the tour should also be the shortest. For example, an optimal solution of a traveling salesperson problem instance is shown in Figure 1–2, where the distance between two cities is their Euclidean distance.

FIGURE 1–2 An optimal solution of a traveling salesperson problem instance.



Finally, let us consider the partition problem. In this problem, we are given a set of integers. Our question is: Can we partition these integers into two subsets

S_1 and S_2 such that the sum of S_1 is equal to the sum of S_2? For instance, for the following set

$$\{1, 7, 10, 9, 5, 8, 3, 13\},$$

we can partition the above set into

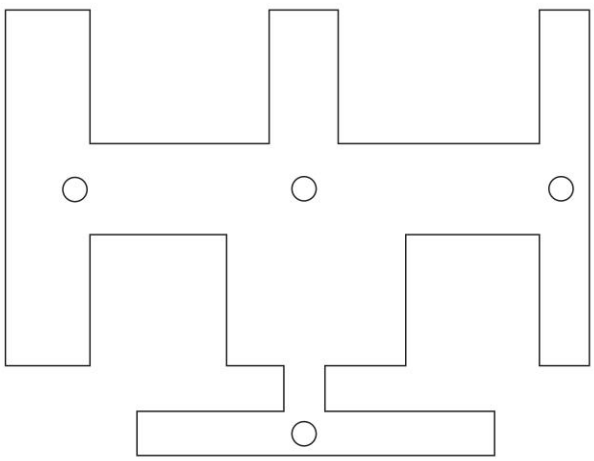
$$S_1 = \{1, 10, 9, 8\}$$
$$\text{and} \quad S_2 = \{7, 5, 3, 13\}.$$

We can prove that the sum of S_1 is equal to the sum of S_2.

We need to solve this partition problem very often. For instance, one public key encryption scheme involves this problem. Yet, it is an NP-complete problem.

Here is another problem which will be interesting to many readers. We all know that there are art galleries where priceless treasures are being stored. It is very important that those treasures are not stolen or damaged. So guards must be placed in the galleries. Consider Figure 1–3, which shows an art gallery.

FIGURE 1–3 An art gallery and its guards.



If we place four guards in the art gallery as shown by the circles in Figure 1–3, every wall of their art gallery will be monitored by at least one guard. This means that the entire art gallery will be sufficiently monitored by these four guards. The art gallery problem is: Given an art gallery in the form of a polygon,

determine the minimum number of guards and their placements such that the entire art gallery can be monitored by these guards. It may be a big surprise to many readers that this is also an NP-complete problem.

Even if one appreciates the fact that good algorithms are essential, one may still wonder whether it is important to study the design of algorithms because it is possible that good algorithms can be easily obtained. In other words, if a good algorithm can be obtained simply by intuition, or by common sense, then it is not worth the effort to study the design of algorithms.

When we introduce the minimum spanning tree problem below, we shall show that this seemingly combinational explosive problem actually has a very simple algorithm that can be used to solve the problem efficiently. An informal description of the minimum spanning tree problem is given below.

Imagine that we have many cities, shown in Figure 1–4. Suppose we want to connect all the cities into a spanning tree (a spanning tree is a graph connecting all of the cities without cycles) in such a way that the total length of the spanning tree is minimized. For example, for the set of cities shown in Figure 1–4, a minimum spanning tree is shown in Figure 1–5.

FIGURE 1–4 A set of cities to illustrate the minimum spanning tree problem.

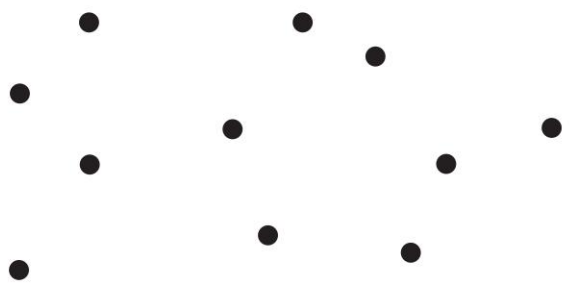
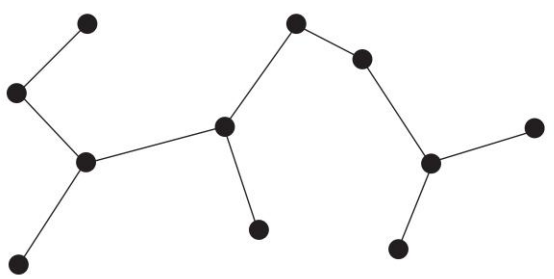


FIGURE 1–5 A minimum spanning tree for the set of cities shown in Figure 1–4.



An intuitive algorithm to find such a minimum spanning tree is to find all possible spanning trees. For each spanning tree, determine its total length and a minimum spanning tree can be found after all possible spanning trees have been exhaustively enumerated. This turns out to be very expensive because the number of possible spanning trees is very large indeed. It can be proved that the total number of possible spanning trees for n cities is n^{n-2}. Suppose that $n = 10$. We already have 10^8 trees to enumerate. When n is equal to 100, this number is increased to 100^{98} (10^{196}), which is so large that no computer can handle this kind of data. In the United States, a telephone company may have to deal with the case where n is equal to 5,000. So, we must have a better algorithm.

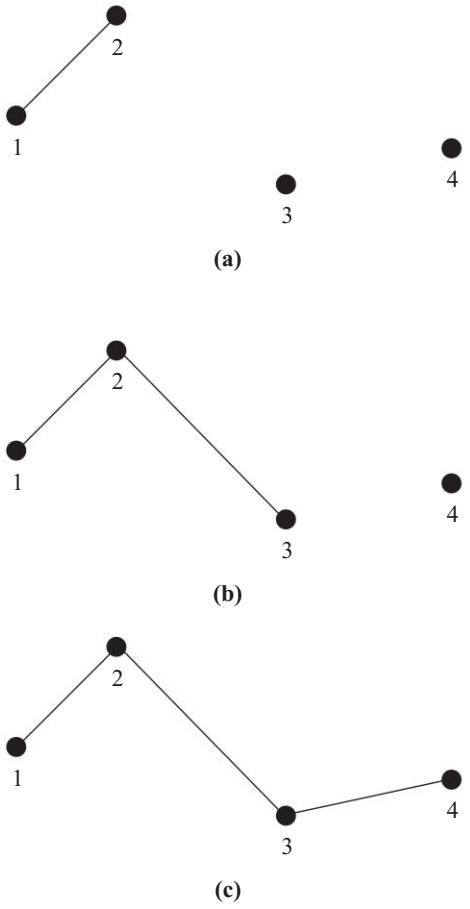
Actually, there is an excellent algorithm to solve this minimum spanning tree problem. We illustrate this algorithm through an example. Consider Figure 1–6. Our algorithm will first find out that d_{12} is the smallest among all intercity distances. We may therefore connect cities 1 and 2, as shown in Figure 1–7(a). After doing this, we consider $\{1, 2\}$ as a set and the rest of the cities as another set. We then find out that the shortest distance among these two sets of cities is d_{23}. We connect cities 2 and 3. Finally, we connect 3 and 4. The entire process is shown in Figure 1–7.

FIGURE 1–6 An example to illustrate an efficient minimum spanning tree algorithm.

It can be proved that this simple, efficient algorithm always produces an optimal solution. That is, the final spanning tree it produces is always a minimum spanning tree. The proof of the correctness of this algorithm is by no means easy. The strategy behind this algorithm is called the greedy method. Usually, an algorithm based on the greedy method is quite efficient.

Unfortunately, many problems which are similar to the minimum spanning tree problem cannot be solved by the greedy method. For instance, the traveling salesperson problem is such a problem.

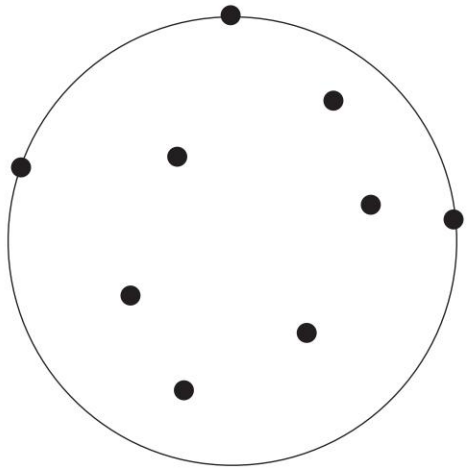
FIGURE 1–7 The minimum spanning tree algorithm illustrated.



One example which cannot be easily solved by exhaustive searching is the 1-center problem. In the 1-center problem, we are given a set of points and we must find a circle covering all these points such that the radius of the circle is minimized. For instance, Figure 1–8 shows an optimal solution of a 1-center problem. How are we to start working on this problem? We will show later in this book that the 1-center problem can be solved by the prune-and-search strategy.

The study of algorithm design is almost a study of strategies. Thanks to many researchers, excellent strategies have been discovered and these strategies can be used to design algorithms. We cannot claim that every excellent algorithm must be based on one of the general strategies. Yet, we can definitely say that a complete knowledge of strategies is absolutely invaluable to the study of algorithms.

FIGURE 1–8 1-center problem solution.



We now recommend a list of books on algorithms which are suitable references for further reading.

Aho, A. V., Hopcroft, J. E. and Ullman, J. D. (1974): *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass.

Basse, S. and Van Gelder, A. (2000): *Computer Algorithms: Introduction to Design and Analysis*, Addison-Wesley, Reading, Mass.

Brassard, G. and Bratley, P. (1988): *Algorithmics: Theory and Practice*, Prentice-Hall, Englewood Cliffs, New Jersey.

Coffman, E. G. and Lueker, G. S. (1991): *Probabilistic Analysis of Packaging & Partitioning Algorithms*, John Wiley & Sons, New York.

Cormen, T. H. (2001): *Introduction to Algorithms*, McGraw-Hill, New York.

Cormen, T. H., Leiserson, C. E. and Rivest, R. L. (1990): *Introduction to Algorithms*, McGraw-Hill, New York.

Dolan A. and Aldous J. (1993): *Networks and Algorithms: An Introductory Approach*, John Wiley & Sons, New York.

Evans, J. R. and Minieka, E. (1992): *Optimization Algorithms for Networks and Graphs*, 2nd ed., Marcel Dekker, New York.

Garey, M. R. and Johnson, D. S. (1979): *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, California.

Gonnet, G. H. (1983): *Handbook of Algorithms and Data Structures*, Addison-Wesley, Reading, Mass.

Goodman, S. and Hedetniemi, S. (1980): *Introduction to the Design and Analysis of Algorithms*, McGraw-Hill, New York.

Gould, R. (1988): *Graph Theory*, Benjamin Cummings, Redwood City, California.

Greene, D. H. and Knuth, D. E. (1981): *Mathematics for the Analysis of Algorithms*, Birkhäuser, Boston, Mass.

Hofri, M. (1987): *Probabilistic Analysis of Algorithms*, Springer-Verlag, New York.

Horowitz, E. and Sahni, S. (1976): *Fundamentals of Data Structures*, Computer Science Press, Rockville, Maryland.

Horowitz, E., Sahni, S. and Rajasekaran, S. (1998): *Computer Algorithms*, W. H. Freeman, New York.

King, T. (1992): *Dynamic Data Structures: Theory and Applications*, Academic Press, London.

Knuth, D. E. (1969): *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, Addison-Wesley, Reading, Mass.

Knuth, D. E. (1973): *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, Mass.

Kozen, D. C. (1992): *The Design and Analysis of Algorithms*, Springer-Verlag, New York.

Kronsjö, L. I. (1987): *Algorithms: Their Complexity and Efficiency*, John Wiley & Sons, New York.

Kucera, L. (1991): *Combinatorial Algorithms*, IOP Publishing, Philadelphia.

Lewis, H. R. and Denenberg, L. (1991): *Data Structures and Their Algorithms*, Harper Collins, New York.

Manber, U. (1989): *Introduction to Algorithms: A Creative Approach*, Addison-Wesley, Reading, Mass.

Mehlhorn, K. (1987): *Data Structures and Algorithms: Sorting and Searching*, Springer-Verlag, New York.

Moret, B. M. E. and Shapiro, H. D. (1991): *Algorithms from P to NP*, Benjamin Cummings, Redwood City, California.

Motwani, R. and Raghavan P. (1995): *Randomized Algorithms*, Cambridge University Press, Cambridge, England.

Mulmuley, K. (1998): *Computational Geometry: An Introduction through Randomized Algorithms*, Prentice-Hall, Englewood Cliffs, New Jersey.

Neapolitan, R. E. and Naimipour, K. (1996): *Foundations of Algorithms*, D.C. Heath and Company, Lexington, Mass.

Papadimitriou, C. H. (1994): *Computational Complexity*, Addison-Wesley, Reading, Mass.

Purdom, P. W. Jr. and Brown, C. A. (1985): *The Analysis of Algorithms*, Holt, Rinehart and Winston, New York.

Reingold, E., Nievergelt, J. and Deo, N. (1977): *Combinatorial Algorithms, Theory and Practice*, Prentice-Hall, Englewood Cliffs, New Jersey.

Sedgewick, R. and Flajolet, D. (1996): *An Introduction to the Analysis of Algorithms*, Addison-Wesley, Reading, Mass.

Shaffer, C. A. (2001): *A Practical Introduction to Data Structures and Algorithm Analysis*, Prentice-Hall, Englewood Cliffs, New Jersey.

Smith, J. D. (1989): *Design and Analysis of Algorithms*, PWS Publishing, Boston, Mass.

Thulasiraman, K. and Swamy, M. N. S. (1992): *Graphs: Theory and Algorithms*, John Wiley & Sons, New York.

Uspensky, V. and Semenov, A. (1993): *Algorithms: Main Ideas and Applications*, Kluwer Press, Norwell, Mass.

Van Leeuwen, J. (1990): *Handbook of Theoretical Commputer Science: Volume A: Algorithms and Complexity*, Elsevier, Amsterdam.

Weiss, M. A. (1992): *Data Structures and Algorithm Analysis*, Benjamin Cummings, Redwood City, California.

Wilf, H. S. (1986): *Algorithms and Complexity*, Prentice-Hall, Engelwood Cliffs, New York.

Wood, D. (1993): *Data Structures, Algorithms, and Performance*, Addison-Wesley, Reading, Mass.

For advanced studies, we recommend the following books:

For Computational Geometry

Edelsbrunner, H. (1987): *Algorithms in Combinatorial Geometry*, Springer-Verlag, Berlin.

Mehlhorn, K. (1984): *Data Structures and Algorithms 3: Multi-Dimensional Searching and Computational Geometry*, Springer-Verlag, Berlin.

Mulmuley, K. (1998): *Computational Geometry: An Introduction through Randomized Algorithms*, Prentice-Hall, Englewood Cliffs, New Jersey.

O'Rourke, J. (1998): *Computational Geometry in C*, Cambridge University Press, Cambridge, England.

Pach, J. (1993): *New Trends in Discrete and Computational Geometry*, Springer-Verlag, New York.

Preparata, F. P. and Shamos, M. I. (1985): *Computational Geometry: An Introduction*, Springer-Verlag, New York.

Teillaud, M. (1993): *Towards Dynamic Randomized Algorithms in Computational Geometry*, Springer-Verlag, New York.

For Graph Theory

Even, S. (1987): *Graph Algorithms*, Computer Science Press, Rockville, Maryland.

Golumbic, M. C. (1980): *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York.

Lau, H. T. (1991): *Algorithms on Graphs*, TAB Books, Blue Ridge Summit, PA.

McHugh, J. A. (1990): *Algorithmic Graph Theory*, Prentice-Hall, London.

Mehlhorn, K. (1984): *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*, Springer-Verlag, Berlin.

Nishizeki, T. and Chiba, N. (1988): *Planar Graphs: Theory and Algorithms*, Elsevier, Amsterdam.

Thulasiraman, K. and Swamy, M. N. S. (1992): *Graphs: Theory and Algorithms*, John Wiley & Sons, New York.

For Combinatorics

Lawler, E. L. (1976): *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York.

Lawler, E. L., Lenstra, J. K., Rinnooy Kan, A. H. G. and Shamoys, D. B. (1985): *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, John Wiley & Sons, New York.

Martello, S. and Toth, P. (1990): *Knapsack Problem Algorithms & Computer Implementations*, John Wiley & Sons, New York.

Papadimitriou, C. H. and Steiglitz, K. (1982): *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, New Jersey.

For Advanced Data Structures

Tarjan, R. E. (1983): *Data Structures and Network Algorithms*, Society of Industrial and Applied Mathematics, Vol. 29.

For Computational Biology

Gusfield, D. (1997): *Algorithms on Strings, Trees and Sequences: Computer*

Science and Computational Biology, Cambridge University Press, Cambridge, England.

Pevzner, P. A. (2000): *Computational Molecular Biology: An Algorithmic Approach*, The MIT Press, Boston.

Setubal, J. and Meidanis, J. (1997): *Introduction to Computational Biology*, PWS Publishing Company, Boston, Mass.

Szpankowski, W. (2001): *Average Case Analysis of Algorithms on Sequences*, John Wiley & Sons, New York.

Waterman, M. S. (1995): *Introduction to Computational Biology*: *Maps, Sequences and Genomes*, Chapman & Hall/CRC, New York.

For Approximation Algorithms

Hochbaum, D. S. (1996): *Approximation Algorithms for NP-Hard Problems*, PWS Publisher, Boston.

For Randomized Algorithms

Motwani, R. and Raghavan, P. (1995): *Randomized Algorithms*, Cambridge University Press, Cambridge, England.

For On-Line Algorithms

Borodin, A. and El-Yaniv, R. (1998): *Online Computation and Competitive Analysis*, Cambridge University Press, Cambridge, England.

Fiat, A. and Woeginger, G. J. (editors) (1998): Online Algorithms: The State of the Arts, *Lecture Notes in Computer Science*, Vol. 1442. Springer-Verlag, New York.

There are many academic journals which regularly publish papers on algorithms. In the following, we recommend the most popular ones:

- *Acta Informatica*
- *Algorithmica*
- *BIT*
- *Combinatorica*
- *Discrete and Computational Geometry*
- *Discrete Applied Mathematics*
- *IEEE Transactions on Computers*
- *Information and Computations*

- *Information Processing Letters*
- *International Journal of Computational Geometry and Applications*
- *International Journal of Foundations on Computer Science*
- *Journal of Algorithms*
- *Journal of Computer and System Sciences*
- *Journal of the ACM*
- *Networks*
- *Proceedings of the ACM Symposium on Theory of Computing*
- *Proceedings of the IEEE Symposium on Foundations of Computing Science*
- *SIAM Journal on Algebraic and Discrete Methods*
- *SIAM Journal on Computing*
- *Theoretical Computer Science*

chapter

2

The Complexity of Algorithms and the Lower Bounds of Problems

In this chapter, we shall discuss some basic issues related to the analysis of algorithms. Essentially, we shall try to clarify the following issues:

(1) Some algorithms are efficient and some are not. How do we measure the goodness of an algorithm?
(2) Some problems are easy to solve and some are not. How do we measure the difficulty of a problem?
(3) How do we know that an algorithm is optimal for a problem? That is, how can we know that there does not exist any other better algorithm to solve the same problem? We shall show that all of these problems are related to each other.

2-1 The Time Complexity of an Algorithm

We usually say that an algorithm is good if it takes a short time to run and requires a small amount of memory space. However, traditionally, a more important factor in determining the goodness of an algorithm is the time needed to execute it. Throughout this book, unless stated otherwise, we shall be concerned with the time criterion.

To measure the time complexity of an algorithm, one is tempted to write a program for this algorithm and see how fast it runs. This is not appropriate because there are so many factors unrelated to the algorithm which affect the performance of the program. For example, the capability of the programmer, the language used, the operating system and even the compiler for the particular language will all have effects on the time needed to run the program.

In algorithm analysis, we shall always choose a particular step which occurs in the algorithm, and perform a mathematical analysis to determine the number of steps needed to complete the algorithm. For instance, the comparison of data items cannot be avoided in any sorting algorithm and therefore it is often used to measure the time complexity of sorting algorithms.

Of course, one may legitimately protest that in some sorting algorithms, the comparison of data is not a dominating factor. In fact, we can easily show examples that in some sorting algorithms, the movement of data is the most time-consuming action. In such a situation, it appears that we should use the movement of data, not the comparison of data, to measure the time complexity of this particular sorting algorithm.

We usually say that the cost of executing an algorithm is dependent on the size of the problem, n. For instance, the number of points in the Euclidean traveling salesperson problem defined in Section 9–2 is the problem size. As expected, most algorithms need more time to complete as n increases.

Suppose that it takes $(n^3 + n)$ steps to execute an algorithm. We would often say that the time complexity of this algorithm is in the order of n^3. Since the term n^3 dominates n and as n becomes very large, the term n is not so significant as compared with n^3. We shall now give this casual and commonly used statement a formal and precise meaning.

Definition

$f(n) = O(g(n))$ if and only if there exist two positive constants c and n_0 such that $|f(n)| \le c|g(n)|$ for all $n \ge n_0$.

From the above definition, we understand that, if $f(n) = O(g(n))$, then $f(n)$ is bounded, in certain sense, by $g(n)$ as n is very large. If we say that the time complexity of an algorithm is $O(g(n))$, we mean that it always takes less than c times $|g(n)|$ to run this algorithm as n is large enough for some c.

Let us consider the case where it takes $(n^3 + n)$ steps to complete an algorithm. Then

$$\begin{aligned} f(n) &= n^3 + n \\ &= \left(1 + \frac{1}{n^2}\right) n^3 \\ &\le 2n^3 \quad \text{for } n \ge 1. \end{aligned}$$

Therefore, we may say that the time complexity is $O(n^3)$ because we may take c and n_0 to be 2 and 1 respectively.

Next, we shall clarify a very important point, a common misunderstanding about the order of magnitude of the time complexity of algorithms.

Suppose that we have two algorithms A_1 and A_2 that solve the same problem. Let the time complexities of A_1 and A_2 be $O(n^3)$ and $O(n)$ respectively. If we ask the same person to write two programs for A_1 and A_2 and run these two programs under the same programming environment, would the program for A_2 run faster than that for A_1? It is a common mistake to think that the program for A_2 will always run faster than that for A_1. Actually, this is not necessarily true for one simple reason: It may take more time to execute a step in A_2 than in A_1. In other words, although the number of steps required by A_2 is smaller than that required by A_1, in some cases, A_1 still runs faster than A_2. Suppose the time needed for each step of A_1 is 1/100 of that for A_2. Then the actual computing time for A_1 and A_2 are n^3 and $100n$ respectively. For $n < 10$, A_1 runs faster than A_2. For $n > 10$, A_2 runs faster than A_1.

The reader may now understand the significance of the constant appearing in the definition of the function $O(g(n))$. This constant cannot be ignored. However, no matter how large the constant, its significance diminishes as n increases. If the complexities of A_1 and A_2 are $O(g_1(n))$ and $O(g_2(n))$ respectively and $g_1(n) < g_2(n)$ for all n, we understand that as n is large enough, A_1 runs faster than A_2.

Another point that we should bear in mind is that we can always, at least theoretically, hardwire any algorithm. That is, we can always design a circuit to implement an algorithm. If two algorithms are hardwired, the time needed to execute a step in one algorithm can be made to be equal to that in the other algorithm. In such a situation, the order of magnitude is even more significant. If the time complexities of A_1 and A_2 are $O(n^3)$ and $O(n)$ respectively, then we know that A_2 is better than A_1 if both are thoroughly hardwired. Of course, the above discussion is meaningful only if we can master the skill of hardwiring exceedingly well.

The significance of the order of magnitude can be seen by examining Table 2–1. From Table 2–1, we can observe the following:

(1) It is very meaningful if we can find an algorithm with lower order time complexity. A typical case is for searching. A sequential search through a list of n numbers requires $O(n)$ operations in the worst case. If we have a sorted list of n numbers, binary search can be used and the time

TABLE 2–1 Time-complexity functions.

Time complexity function	Problem size: n			
	10	**10^2**	**10^3**	**10^4**
$\log_2 n$	3.3	6.6	10	13.3
n	10	10^2	10^3	10^4
$n \log_2 n$	0.33×10^2	0.7×10^3	10^4	1.3×10^5
n^2	10^2	10^4	10^6	10^8
2^n	1024	1.3×10^{30}	$>10^{100}$	$>10^{100}$
$n!$	3×10^6	$>10^{100}$	$>10^{100}$	$>10^{100}$

complexity is reduced to $O(\log_2 n)$ in the worst case. For $n = 10^4$, the sequential searching may need 10^4 operations while the binary search requires only 14 operations.

(2) While we may dislike the time-complexity functions, such as n^2, n^3, etc., they are still tolerable compared with 2^n. For instance, when $n = 10^4$, $n^2 = 10^8$. But $2^n > 10^{100}$. The number 10^{100} is so large that no matter how fast a computer runs, it cannot solve this problem. Any algorithm with time complexity $O(p(n))$ where $p(n)$ is a polynomial function is a polynomial algorithm. On the other hand, algorithms whose time complexities cannot be bounded by a polynomial function are exponential algorithms.

There is a vast difference between polynomial and exponential algorithms. Sadly, there is a large class of algorithms which is exponential and there does not seem to be any hope that they can be replaced by polynomial algorithms. Every algorithm for solving the Euclidean traveling salesperson problem, for example, is an exponential algorithm up to now. Similarly, every algorithm to solve the satisfiability problem, as defined in Section 8–3, is presently an exponential algorithm. But as we shall see, the minimal spanning tree problem, as defined in Section 3–1, can be solved by polynomial algorithms.

In the above discussion, we were vague about the data. Certainly, for some data, an algorithm may terminate very quickly and for other data, it may behave entirely differently. We shall discuss these topics in the next section.

2-2 THE BEST-, AVERAGE- AND WORST-CASE ANALYSIS OF ALGORITHMS

For any algorithm, we are interested in its behavior under three situations, the best case, the average case and the worst case. Usually, the best-case analysis is the easiest, the worst case the second easiest and the average-case analysis the hardest. In fact, there are still many open problems concerning the average case analysis.

Example 2–1 The Straight Insertion Sort

One of the simplest sorting methods is the straight insertion sort. We are given a sequence of numbers $x_1, x_2, \ldots, x_n$. We scan these numbers from left to right and put x_i to the left of x_{i-1} if x_i is smaller than x_{i-1}. In other words, we continuously move x_i to the left until the numbers to its left are all smaller than or equal to it.

Algorithm 2–1 □ Straight insertion sort

Input: $x_1, x_2, \ldots, x_n$.
Output: The sorted sequence of $x_1, x_2, \ldots, x_n$.

```
For j := 2 to n do
Begin
    i := j − 1
    x := x_j
    While x < x_i and i > 0 do
    Begin
        x_{i+1} := x_i
        i := i − 1
    End
    x_{i+1} := x
End
```

Consider the input sequence: 7, 5, 1, 4, 3, 2, 6. The straight insertion sort will produce the sorted sequence as follows:

7
5, 7
1, 5, 7
1, 4, 5, 7
1, 3, 4, 5, 7
1, 2, 3, 4, 5, 7
1, 2, 3, 4, 5, 6, 7.

In our analysis, we shall use the number of data movements $x := x_j$, $x_{i+1} := x_i$ and $x_{i+1} := x$ as the time-complexity measurement of the algorithm. In the above algorithm, there are two do loops, one outer and one inner. For the outer do loop, two data movement operations, namely $x := x_j$ and $x_{i+1} := x$, will always be executed. Since the inner do loop may or may not be executed, we shall denote the number of data movements executed for x_i in the inner do loop by d_i. Then, obviously, the total number of data movements for straight insertion sort is

$$X = \sum_{i=2}^{n} (2 + d_i)$$
$$= 2(n-1) + \sum_{i=2}^{n} d_i$$

Best Case: $\sum_{i=2}^{n} d_i = 0, X = 2(n-1) = O(n)$

This occurs when the input data are already sorted.

Worst Case: The worst case occurs when the input data are reversely sorted. In this case,

$$d_2 = 1$$
$$d_3 = 2$$
$$\vdots$$
$$d_n = n - 1.$$

Thus,

$$\sum_{i=2}^{n} d_i = \frac{n}{2}(n-1)$$

$$X = 2(n-1) + \frac{n}{2}(n-1) = \frac{1}{2}(n-1)(n+4) = O(n^2).$$

Average Case: When x_i is being considered, $(i-1)$ data elements have already been sorted. If x_i is the largest among the i numbers, the inner do loop will not be executed and there will not be any data movement at all inside this inner do loop. If x_i is the second largest among these i numbers, there will be one data movement and so on. The probability that x_i is the largest is $1/i$. This is also the probabilty that x_i is the jth largest for $1 \le j \le i$. Therefore, the average $(2 + d_i)$ is

$$\frac{2}{i} + \frac{3}{i} + \cdots + \frac{i+1}{i} = \sum_{j=1}^{i} \frac{(j+1)}{i}$$

$$= \frac{i+3}{2}.$$

The average time complexity for straight insertion sort is

$$\sum_{i=2}^{n} \frac{i+3}{2} = \frac{1}{2}\left(\sum_{i=2}^{n} i + \sum_{i=2}^{n} 3\right)$$

$$= \frac{1}{4}(n-1)(n+8) = O(n^2).$$

In summary, the time complexities of the straight insertion sort are as follows:

Best Case: $2(n-1) = O(n)$.

Average Case: $\frac{1}{4}(n+8)(n-1) = O(n^2)$.

Worst Case: $\frac{1}{2}(n-1)(n+4) = O(n^2)$.

Example 2–2 The Binary Search Algorithm

The binary search is a famous searching algorithm. After sorting a set of numbers into an ascending, or descending, sequence, the binary search algorithm starts

from the middle of the sequence. If the testing point is equal to the middle element of the sequence, terminate; otherwise, depending on the result of comparing the testing element and the middle of the sequence, we recursively search the left, or the right, of the sequence.

Algorithm 2–2 □ Binary search

Input: A sorted array $a_1, a_2, \ldots, a_n$, $n > 0$ and X, where $a_1 \le a_2 \le a_3 \le \cdots \le a_n$.
Output: j if $a_j = X$ and 0 if no j exists such that $a_j = X$.

$i := 1$ (* first entry *)
$m := n$ (* last entry *)
While $i \le m$ do
Begin
$j := \left\lfloor \frac{i+m}{2} \right\rfloor$
If $X = a_j$ then output j and stop
If $X < a_j$ then $m := j - 1$
else $i := j + 1$
End
$j := 0$
Output j

The best-case analysis for binary search is rather simple. In the best case, binary search terminates with one step.

The worst-case analysis is also quite simple. It is easy to see that it takes at most $(\lfloor \log_2 n \rfloor + 1)$ steps to complete the binary search. Throughout the rest of this book, unless stated otherwise, $\log n$ means $\log_2 n$.

To simplify our discussion, let us assume that $n = 2^k - 1$.

For the average-case analysis, let us note that if there are n elements, then there is one element for which the algorithm terminates successfully with only one step. This element is located at the $\left\lfloor \frac{1+n}{2} \right\rfloor$-th position of the sorted sequence. There are two elements which will cause the binary search to terminate successfully after two steps. In general, there are 2^{t-1} elements which will cause

the binary search to terminate after t steps, for $t = 1, 2, \ldots, \lfloor \log n \rfloor + 1$. If X is not on the list, then the algorithm terminates unsuccessfully after $\lfloor \log n \rfloor + 1$ steps. Totally, we may say that there are $(2n + 1)$ distinct cases: n cases which will cause the searching to terminate successfully and $(n + 1)$ cases which will cause the binary search to terminate unsuccessfully.

Let $A(n)$ denote the average number of comparisons done in the binary search and $k = \lfloor \log n \rfloor + 1$. Then

$$A(n) = \frac{1}{2n+1}\left(\sum_{i=1}^{k} i2^{i-1} + k(n+1)\right).$$

We shall now prove that

$$\sum_{i=1}^{k} i2^{i-1} = 2^k(k-1)+1. \tag{2–1}$$

The above formula can be proved by induction on k. Equation (2–1) is obviously true for $k = 1$. Assume that Equation (2–1) is valid for $k = m$, $m > 1$. We shall then prove that it is valid for $k = m + 1$. That is, assuming Equation (2–1) is valid, we shall prove

$$\sum_{i=1}^{m+1} i2^{i-1} = 2^{m+1}(m+1-1)+1 = 2^{m+1} \cdot m + 1.$$

Note that

$$\sum_{i=1}^{m+1} i2^{i-1} = \sum_{i=1}^{m} i2^{i-1} + (m+1)2^{m+1-1}.$$

Substituting Equation (2–1) into the above formula, we have

$$\begin{aligned}\sum_{i=1}^{m+1} i2^{i-1} &= 2^m(m-1) + 1 + (m+1)2^m \\ &= 2^m \cdot 2m + 1 \\ &= 2^{m+1} \cdot m + 1.\end{aligned}$$

We have thus proved the validity of Equation (2–1). Using Equation (2–1)

$$A(n) = \frac{1}{2n+1}\,((k-1)2^k + 1 + k2^k).$$

As n is very large, we have

$$\begin{aligned} A(n) &\approx \frac{1}{2^{k+1}}(2^k(k-1) + k2^k) \\ &= \frac{(k-1)}{2} + \frac{k}{2} \\ &= k - \frac{1}{2}. \end{aligned}$$

Therefore $A(n) < k = O(\log n)$.

Now, the reader may wonder whether the result obtained by assuming that $n = 2^k$ is valid for n in general. Let us assume that $t(n)$ is a non-decreasing function, and $t(n) = O(f(n))$ be the time complexity of our algorithm obtained by assuming that $n = 2^k$, and $f(bn) \le c'f(n)$ for some $b \ge 1$ and c' is a constant (this means that f is a smooth function and every polynomial function is such a function). Then

$t(2^k) \le cf(2^k)$ where c is a constant.

Let $n' = 2^{k+x}$ for $0 \le x \le 1$

$$\begin{aligned} t(n') &= t(2^{k+x}) \\ &\le t(2^{k+1}) \le cf(2^{k+1}) \\ &= cf(2^{k+x} \cdot 2^{1-x}) \\ &\le cc'f(2^{k+x}) = c''f(n'). \end{aligned}$$

Therefore, $t(n') = O(f(n'))$.

The above discussion shows that we may assume that $n = 2^k$ to obtain the time complexity function of an algorithm. In the rest of this book, whenever it is necessary, we shall assume that $n = 2^k$ without explaining why we can do so.

In summary, for binary search, we have

Best Case: $O(1)$.
Average Case: $O(\log n)$.
Worst Case: $O(\log n)$.

Example 2–3 Straight Selection Sort

The straight selection sort is perhaps the simplest kind of sorting. Yet the analysis of this algorithm is quite interesting. The straight selection sort can be easily described as follows:

(1) Find the smallest number. Let this smallest number occupy a_1 by exchanging a_1 with this smallest number.
(2) Repeat the above step on the remaining numbers. That is, find the second smallest number and let it occupy a_2.
(3) Continue the process until the largest number is found.

Algorithm 2–3 □ Straight selection sort

Input: $a_1, a_2, \ldots, a_n$.
Output: The sorted sequence of $a_1, a_2, \ldots, a_n$.

```
For j := 1 to n − 1 do
Begin
    f := j
    For k := j + 1 to n do
        If a_k < a_f then f := k
    a_j ↔ a_f
End
```

In the above algorithm, to find the smallest number of $a_1, a_2, \ldots, a_n$, we essentially set a flag f and initially $f = 1$. We then compare a_f with a_2. If $a_f < a_2$, we do nothing; otherwise, we set $f = 2$, compare a_f with a_3 and so on.

It is evident that there are two operations in the straight selection sort: the comparison of two elements and the change of the flag. The number of comparisons of two elements is a fixed number, namely $n(n-1)/2$. That is, no matter what the input data are, we always have to perform $n(n-1)/2$ comparisons. Therefore, we shall choose the number of changing of flags to measure the time complexity of the straight selection sort.

The change of flag depends upon the data. Consider $n = 2$. There are only two permutations:

(1, 2)
and (2, 1).

For the first permutation, no change of flag is necessary while for the second permutation, one change of flag is necessary.

Let $f(a_1, a_2, \ldots, a_n)$ denote the number of changing of flags needed to find the smallest number for the permutation $a_1, a_2, \ldots, a_n$. The following table illustrates the case for $n = 3$.

a_1,	a_2,	a_3	$f(a_1, a_2, a_3)$
1,	2,	3	0
1,	3,	2	0
2,	1,	3	1
2,	3,	1	1
3,	1,	2	1
3,	2,	1	2

To determine $f(a_1, a_2, \ldots, a_n)$, we note the following:

(1) If $a_n = 1$, then $f(a_1, a_2, \ldots, a_n) = 1 + f(a_1, a_2, \ldots, a_{n-1})$ because there must be a change of flag at the last step.
(2) If $a_n \neq 1$, then $f(a_1, a_2, \ldots, a_n) = f(a_1, a_2, \ldots, a_{n-1})$ because there must not be a change of flag at the last step.

Let $P_n(k)$ denote the probability that a permutation $a_1, a_2, \ldots, a_n$ of $\{1, 2, \ldots, n\}$ needs k changes of flags to find the smallest number. For instance, $P_3(0) = \frac{2}{6}$, $P_3(1) = \frac{3}{6}$ and $P_3(2) = \frac{1}{6}$. Then the average number of changes of flags to find the smallest number is

$$X_n = \sum_{k=0}^{n-1} kP_n(k).$$

The average number of changes of flag for the straight selection sort is

$$A(n) = X_n + A(n - 1).$$

To find X_n, we shall use the following equations which we discussed before:

$$\begin{aligned} f(a_1, a_2, \ldots, a_n) &= 1 + f(a_1, a_2, \ldots, a_{n-1}) && \text{if } a_n = 1 \\ &= f(a_1, a_2, \ldots, a_{n-1}) && \text{if } a_n \neq 1. \end{aligned}$$

Based on the above formulas, we have

$$P_n(\mathrm{k}) = P(a_n = 1)P_{n-1}(k-1) + P(a_n \neq 1)P_{n-1}(k).$$

But $P(a_n = 1) = 1/n$ and $P(a_n \neq 1) = (n-1)/n$. Therefore, we have

$$P_n(k) = \frac{1}{n} P_{n-1}(k-1) + \frac{n-1}{n} P_{n-1}(k). \tag{2–2}$$

Furthermore, we have the following formula concerned with the initial conditions:

$$\begin{aligned} P_1(k) &= 1 \qquad \text{if } k = 0 \\ &= 0 \qquad \text{if } k \neq 0 \\ P_n(k) &= 0 \qquad \text{if } k < 0, \text{ and if } k = n. \end{aligned} \tag{2–3}$$

To give the reader some feeling about the formulas, let us note that

$$P_2(0) = \frac{1}{2}$$

and $$P_2(1) = \frac{1}{2};$$

$$P_3(0) = \frac{1}{3} P_2(-1) + \frac{2}{3} P_2(0) = \frac{1}{3} \times 0 + \frac{2}{3} \times \frac{1}{2} = \frac{1}{3}$$

and $$P_3(2) = \frac{1}{3} P_2(1) + \frac{2}{3} P_2(2) = \frac{1}{3} \times \frac{1}{2} + \frac{2}{3} \times 0 = \frac{1}{6}.$$

In the following, we shall prove:

$$X_n = \sum_{k=1}^{n-1} kP_n(k) = \frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{2} = H_n - 1, \tag{2–4}$$

where H_n is the n-th Harmonic number.

We may prove Equation (2–4) by induction. That is, we want to prove that

$$X_n = \frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{2} = H_n - 1$$

is trivially true for $n = 2$, and assuming that Equation (2–4) is true for $n = m$ for $m > 2$, we shall prove that Equation (2–4) is true for $n = m + 1$. Thus, we want to prove

$$X_{m+1} = H_{m+1} - 1$$

$$X_{m+1} = \sum_{k=1}^{m} kP_{m+1}(k)$$

$$= P_{m+1}(1) + 2P_{m+1}(2) + \cdots + mP_{m+1}(m).$$

Using Equation (2–2), we have

$$X_{m+1} = \frac{1}{m+1} P_m(0) + \frac{m}{m+1} P_m(1) + \frac{2}{m+1} P_m(1) + \frac{2m}{m+1} P_m(2)$$

$$+ \cdots + \frac{m}{m+1} P_m(m-1) + \frac{m^2}{m+1} P_m(m)$$

$$= \frac{1}{m+1} P_m(0) + \frac{m}{m+1} P_m(1) + \frac{1}{m+1} P_m(1) + \frac{1}{m+1} P_m(1)$$

$$+ \frac{2m}{m+1} P_m(2) + \frac{1}{m+1} P_m(2) + \frac{2}{m+1} P_m(2)$$

$$+ \frac{3m}{m+1} P_m(3) + \cdots + \frac{1}{m+1} P_m(m-1) + \frac{m-1}{m+1} P_m(m-1)$$

$$= \frac{1}{m+1} (P_m(0) + P_m(1) + \cdots + P_m(m-1)) + \frac{m+1}{m+1} P_m(1)$$

$$+ \frac{2m+2}{m+1} P_m(2) + \cdots + \frac{(m-1)(m+1)}{m+1} P_m(m-1)$$

$$= \frac{1}{m+1} + (P_m(1) + 2P_m(2) + \cdots + (m-1)P_m(m-1))$$

$$= \frac{1}{m+1} + H_m - 1 \qquad \text{(by induction hypothesis)}$$

$$= H_{m+1} - 1.$$

Since the average time complexibility of the straight selection sort is:

$$A(n) = X_n + A(n-1),$$

we have

$$\begin{aligned} A(n) &= H_n - 1 + A(n-1) \\ &= (H_n - 1) + (H_{n-1} - 1) + \cdots + (H_2 - 1) \\ &= \sum_{i=2}^{n} H_i - (n-1) \end{aligned} \tag{2–5}$$

$$\begin{aligned} \sum_{i=1}^{n} H_i &= H_n + H_{n-1} + \cdots + H_1 \\ &= H_n + \left(H_n - \frac{1}{n}\right) + \cdots + \left(H_n - \frac{1}{n} - \frac{1}{n-1} - \cdots - \frac{1}{2}\right) \\ &= nH_n - \left(\frac{n-1}{n} + \frac{n-2}{n-1} + \cdots + \frac{1}{2}\right) \\ &= nH_n - \left(1 - \frac{1}{n} + 1 - \frac{1}{n-1} + \cdots + 1 - \frac{1}{2}\right) \\ &= nH_n - \left(n - 1 - \frac{1}{n} - \frac{1}{n-1} - \cdots - \frac{1}{2}\right) \\ &= nH_n - n + H_n \\ &= (n+1)H_n - n. \end{aligned}$$

Therefore,

$$\sum_{i=2}^{n} H_i = (n+1)H_n - H_1 - n. \tag{2–6}$$

Substituting Equation (2–6) into Equation (2–5), we have

$$\begin{aligned} A(n) &= (n+1)H_n - H_1 - (n-1) - n \\ &= (n+1)H_n - 2n. \end{aligned}$$

As n is large enough,

$$A(n) \cong n \log_e n = O(n \log n).$$

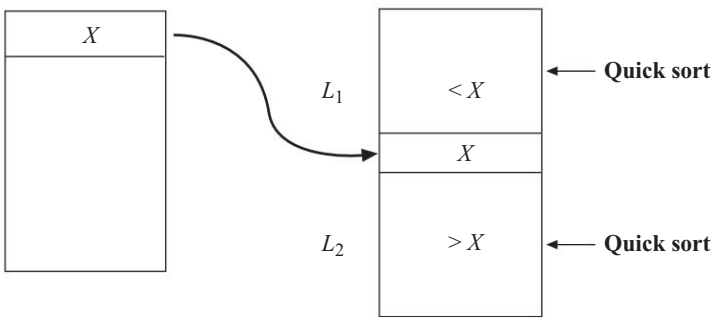
We can summarize the time complexities of the straight selection sort as follows:

Best Case: $O(1)$.
Average Case: $O(n \log n)$.
Worst Case: $O(n^2)$.

Example 2–4 Quick sort

Quick sort is based on the divide-and-conquer strategy. We shall illustrate the divide-and-conquer strategy in detail later. Meanwhile, we may say that the divide-and-conquer strategy divides a problem into two subproblems and solves these two subproblems individually and independently. We later merge the results. Applying this divide-and-conquer strategy to sort, we have a sorting method, called quick sort. Given a set of numbers $a_1, a_2, \ldots, a_n$, we choose an element X to divide $a_1, a_2, \ldots, a_n$ into two lists, as shown in Figure 2–1.

FIGURE 2–1 Quick sort.



After the division, we may apply quick sort to both L_1 and L_2 recursively and the resulting list is a sorted list, because L_1 contains all a_i's smaller than or equal to X and L_2 contains all a_i's larger than X.

The problem is: How do we split the list? We certainly should not use X to scan the whole list and decide whether an a_i is smaller than or equal to X. This will cause a large amount of data movements. As shown in our algorithm, we use two pointers and move them to the middle and perform exchange of data when needed.

Algorithm 2–4 □ Quick sort (f, l)

Input: $a_f, a_{f+1}, \ldots, a_l$.
Output: The sorted sequence of $a_f, a_{f+1}, \ldots, a_l$.

```
If f ≥ l then Return
X := a_f
i := f + 1
j := l
While i < j do
Begin
    While a_j ≥ X and j ≥ f + 1 do
        j := j − 1
    While a_i ≤ X and i ≤ l do
        i := i + 1
    if i < j then a_i ↔ a_j
End
a_f ↔ a_j
Quicksort(f, j − 1)
Quicksort(j + 1, l)
```

The following example illustrates the main spirit of quick sort:

	a_1	a_2	a_3	a_4	a_5	a_6
$X = 3$						
$a_j = a_6 < X$	3	6	1	4	5	2
	$\uparrow i$					$\uparrow j$
	2	6	1	4	5	3
	$\uparrow i$					$\uparrow j$
$a_i = a_2 > X$	2	6	1	4	5	3
		$\uparrow i$				$\uparrow j$
	2	3	1	4	5	6
		$\uparrow i$				$\uparrow j$
	2	3	1	4	5	6
		$\uparrow i$			$\uparrow j$	
	2	3	1	4	5	6
		$\uparrow i$		$\uparrow j$		

$$
\begin{array}{lcccccc}
a_j = a_3 < X & 2 & 3 & 1 & 4 & 5 & 6 \\
 & & \uparrow i & \uparrow j & & & \\
 & 2 & 1 & 3 & 4 & 5 & 6 \\
 & & \uparrow i & \uparrow j & & & \\
 & 2 & 1 & 3 & 4 & 5 & 6 \\
 & & & i\uparrow\uparrow j & & & \\
 & \multicolumn{2}{c}{|\leftarrow \leq 3 \rightarrow|} & =3 & \multicolumn{2}{c}{|\leftarrow \geq 3 \rightarrow|}
\end{array}
$$

The best case of quick sort occurs when X splits the list right in the middle. That is, X produces two sublists which contain the same number of elements. In this case, the first round needs n steps to split the list into two lists. For the next round, for each sublist, we again need $n/2$ (ignoring the element used to split) steps. Therefore, for the second round, we again need $2 \cdot (n/2) = n$ steps. Assuming $n = 2^p$, then we need totally pn steps. But $p = \log_2 n$. Thus, for the best case, the time complexity of quick sort is $O(n \log n)$.

The worst case of quick sort occurs when the input data are sorted or reversely sorted. In these cases, we are merely selecting the extreme (the largest, or the smallest) all the time. Therefore, the total number of steps needed by quick sort in the worst case is

$$n + (n - 1) + \cdots + 1 = \frac{n}{2}(n + 1) = O(n^2).$$

To analyze the average case, let $T(n)$ denote the number of steps needed to perform the quick sort in the average case for n elements. We shall assume that after the splitting operation, the list is divided into two sublists. The first list contains s elements and the second list contains $(n - s)$ elements. The value s ranges from 1 to n and we have to take all possible cases into consideration in order to obtain the average case performance. The following formula can be used to obtain $T(n)$:

$$T(n) = \underset{1 \leq s \leq n}{\text{Ave}} (T(s) + T(n - s)) + cn$$

where cn denotes the number of operations needed for the first splitting operation. (Note that every element is scanned before the list is split into two sublists.)

$$\begin{aligned}
&\underset{1\le s\le n}{\text{Ave}}\,(T(s)+T(n-s))\\
&=\frac{1}{n}\sum_{s=1}^{n}(T(s)+T(n-s))\\
&=\frac{1}{n}(T(1)+T(n-1)+T(2)+T(n-2)+\cdots+T(n)+T(0)).
\end{aligned}$$

Because $T(0)=0$,

$$T(n)=\frac{1}{n}(2T(1)+2T(2)+\cdots+2T(n-1)+T(n))+cn$$

or, $(n-1)T(n)=2T(1)+2T(2)+\cdots+2T(n-1)+cn^2$.

Substituting $n=n-1$ into the above formula, we have

$$(n-2)T(n-1)=2T(1)+2T(2)+\cdots+2T(n-2)+c(n-1)^2.$$

Therefore,

$$\begin{aligned}
&(n-1)T(n)-(n-2)T(n-1)=2T(n-1)+c(2n-1)\\
&(n-1)T(n)-nT(n-1)=c(2n-1)\\
&\frac{T(n)}{n}=\frac{T(n-1)}{n-1}+c\left(\frac{1}{n}+\frac{1}{n-1}\right).
\end{aligned}$$

Recursively,

$$\begin{aligned}
&\frac{T(n-1)}{n-1}=\frac{T(n-2)}{n-2}+c\left(\frac{1}{n-1}+\frac{1}{n-2}\right)\\
&\quad\vdots\\
&\frac{T(2)}{2}=\frac{T(1)}{1}+c\left(\frac{1}{2}+\frac{1}{1}\right).
\end{aligned}$$

We have

$$\frac{T(n)}{n} = c\left(\frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{2}\right) + c\left(\frac{1}{n-1} + \frac{1}{n-2} + \cdots + 1\right)$$
$$= c(H_n - 1) + cH_{n-1}$$
$$= c(H_n + H_{n-1} - 1)$$
$$= c\left(2H_n - \frac{1}{n} - 1\right)$$
$$= c\left(2H_n - \frac{n+1}{n}\right).$$

Finally, we have

$$T(n) = 2cnH_n - c(n+1)$$
$$\cong 2cn \log_e n - c(n+1)$$
$$= O(n \log n).$$

In conclusion, the time complexities for the quick sort are

Best Case: $O(n \log n)$.
Average Case: $O(n \log n)$.
Worst Case: $O(n^2)$.

Example 2–5 The Problem of Finding Ranks

Let us imagine that we have a set of numbers $a_1, a_2, \ldots, a_n$. We shall say that a_i dominates a_j if $a_i > a_j$. If we want to find the number of a_j's which a number a_i dominates, then we may simply sort these numbers into a sequence and the problem will be solved immediately.

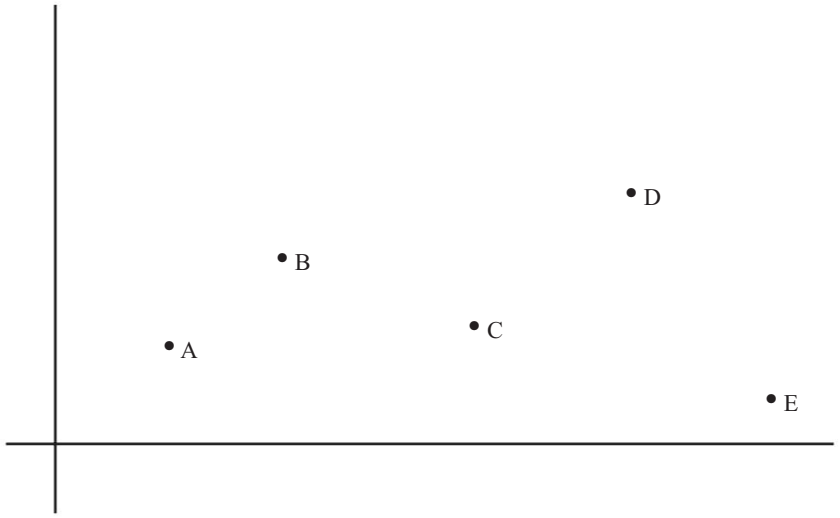
We now extend the problem into the 2-dimension case. That is, each data element is a point in the plane. In this case, we shall first define what we mean by dominance for points in the 2-space.

Definition

Given two points $A = (a_1, a_2)$ and $B = (b_1, b_2)$, A dominates B if and only if $a_i > b_i$ for $i = 1, 2$. If neither A dominates B nor B dominates A, then A and B are incomparable.

Consider Figure 2–2.

FIGURE 2–2 A case of showing the dominance relation.



For the points in Figure 2–2, we have the following relation:

(1) *B*, *C* and *D* dominate *A*.
(2) *D* dominates *A*, *B* and *C*.
(3) All other pairs of points are incomparable.

Having defined the dominance relation, we can further define the rank of a point.

Definition

Given a set *S* of *n* points, the rank of a point *X* is the number of points dominated by *X*.

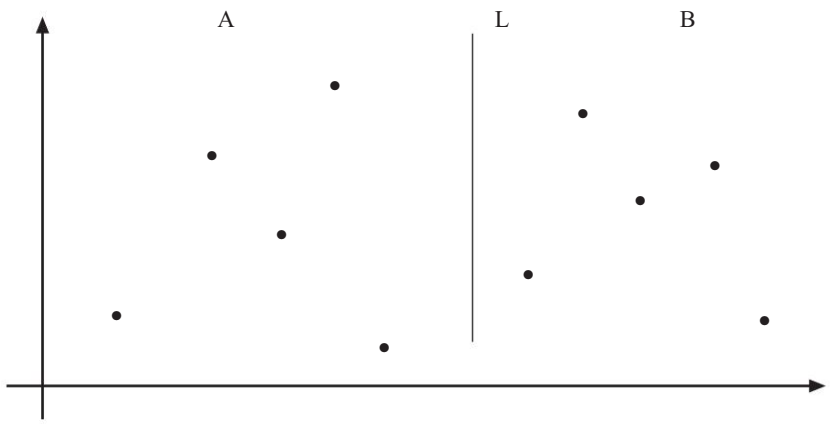
For the points in Figure 2–2, the ranks of *A* and *E* are zero because they do not dominate any point. The ranks of *B* and *C* are one because *A*, and only *A*, is dominated by them. The rank of *D* is three because *A*, *B* and *C* are all dominated by *D*.

Our problem is to find the rank of every point.

A straightforward way to solve this problem is to conduct an exhaustive comparison of all pairs of points. The time complexity of this approach is $O(n^2)$. In the following, we shall show that we can solve this problem in $O(n \log_2 n)$ in the worst case.

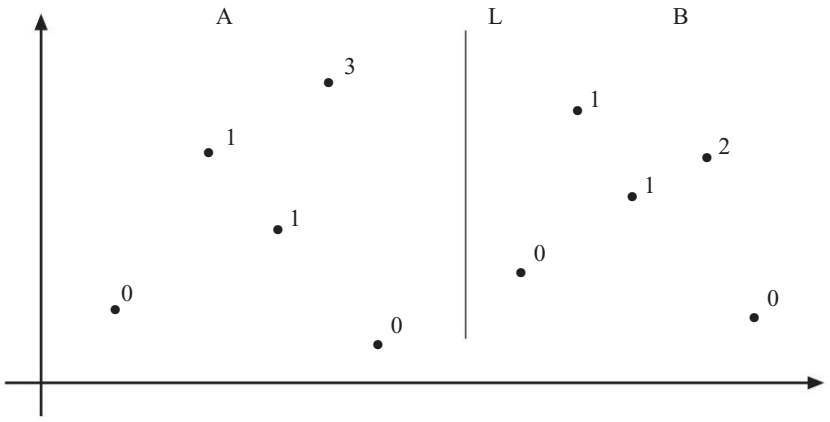
Consider Figure 2–3. Our first step is to find a straight line L perpendicular to the x-axis which separates the set of points into two subsets and these two subsets are of equal size. Let A denote the subset at the left side of L and B denote the subset at the right side of L. It is obvious that the rank of any point in A will not be affected by the presence of B. But, the rank of a point in B may be affected by the presence of A.

FIGURE 2–3 The first step in solving the rank finding problem.



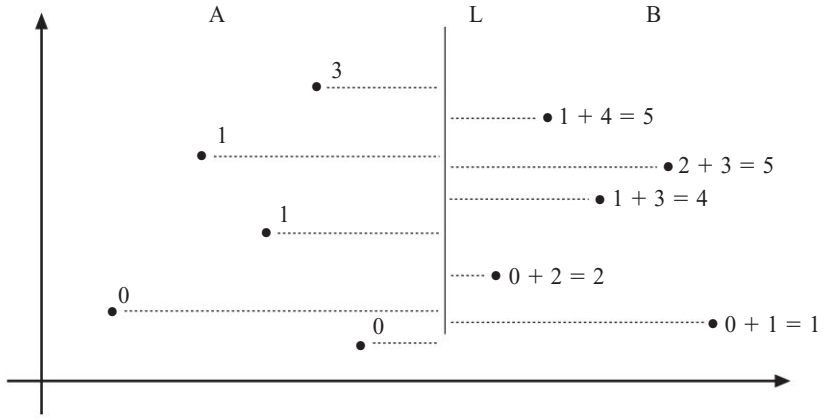
Suppose we find the local ranks of points of A and B separately. That is, we find the ranks of points in A (without considering B) and the ranks of points in B (without considering A). We now depict the local ranks of points in Figure 2–3 in Figure 2–4:

FIGURE 2–4 The local ranks of points in A and B.



We now project all of the points on L. Note that a point P_1 in B dominates a point P_2 in A if and only if the y-value of P_1 is higher than that of P_2. Let P be a point in B. The rank of P is the rank of P, among points in B, plus the number of points in A whose y-values are smaller than the y-value of P. This modification is illustrated in Figure 2–5.

FIGURE 2–5 Modification of ranks.



The following algorithm determines the rank of every point in a plane.

Algorithm 2–5 □ A rank finding algorithm

Input: A set S of planar points $P_1, P_2, \ldots, P_n$.
Output: The rank of every point in S.
Step 1. If S contains only one point, return its rank as 0. Otherwise, choose a cut line L perpendicular to the x-axis such that $n/2$ points of S have X-values less than L (call this set of points A) and the remainder points have X-values greater than L (call this set B). Note that L is a median X-value of this set.
Step 2. Recursively, use this rank finding algorithm to find the ranks of points in A and ranks of points in B.
Step 3. Sort points in A and B according to their y-values. Scan these points sequentially and determine, for each point in B, the number of points in A whose y-values are less than its y-value. The rank of this point is equal to the rank of this point among points in B (found in Step 2), plus the number of points in A whose y-values are less than its y-value.

Algorithm 2–5 is a recursive one. It is based on the divide-and-conquer strategy which divides a problem into two subproblems, solves these two subproblems independently and later merges the subsolutions into the final solution. The time complexity of this algorithm depends on the time complexities of the following steps:

(1) In Step 1, there is an operation which finds the median of a set of numbers. Later in Chapter 7, we shall show that the lowest time complexity of any median finding algorithm is $O(n)$.
(2) In Step 3, there is a sorting operation. In Section 2–3 of this chapter, we shall show that the lowest time complexity of any sorting algorithm is $O(n \log n)$. The scanning takes $O(n)$ steps.

Let $T(n)$ denote the total time to complete the rank finding algorithm. Then

$$T(n) \le 2T(n/2) + c_1 n \log n + c_2 n$$
$$\le 2T(n/2) + cn \log n$$

where c_1, c_2 and c are constants for n large enough.

Let $n = 2^p$. Then

$$
\begin{aligned}
T(n) &\le 2T(n/2) + cn \log n \\
&\le 2(2T(n/4) + cn \log (n/2)/2) + cn \log n \\
&= 4T(n/4) + c(n \log (n/2) + n \log n) \\
&\quad \vdots \\
&\le nT(1) + c(n \log n + n \log (n/2) + n \log (n/4) + \cdots + n \log 2) \\
&= nT(1) + cn\left(\frac{1+\log n}{2} \log n\right) \\
&= c_3 n + \frac{c}{2} n \log^2 n + \frac{c}{2} n \log n.
\end{aligned}
$$

Therefore $T(n) = O(n \log^2 n)$.

The above time complexity is for both the worst case and the average case because the lowest time complexity $O(n \log n)$ for sorting is valid for the average case also. Similarly, the time complexity of $O(n)$ for median finding is also valid for the average case.

Note that this algorithm of finding the rank of each point is much better than an algorithm using exhaustive examining. If an exhaustive examining of all pairs is conducted, we need $O(n^2)$ steps to complete the process.

2–3 THE LOWER BOUND OF A PROBLEM

In the previous section, many examples of finding time complexities of algorithms taught us how to determine the goodness of an algorithm. In this section, we shall view the complexity problem from an entirely different point of view.

Consider the problem of finding ranks, or the traveling salesperson problem. We now ask: How do we measure the difficulty of a problem?

The above problem can be answered in a very intuitive way. If a problem can be solved by an algorithm with a low time complexity, then this problem is an easy one; otherwise, it is a difficult one. This intuitive definition leads to the concept of a lower bound of a problem.

Definition

A lower bound of a problem is the least time complexity required for any algorithm which can be used to solve this problem.

To describe the lower bound, we shall use a notation Ω, which is defined as follows:

Definition

$f(n) = \Omega(g(n))$ if and only if there exist positive constants c and n_0 such that for all $n > n_0$, $|f(n)| \geq c\,|g(n)|$.

The time complexity used in the above definition usually refers to the worst-case time complexity, although the average-case time complexity may also be used. If the average-case time complexity is used, the lower bound is called the average-case lower bound; otherwise, it is called the worst-case lower bound. Throughout this book, unless otherwise stated, when we mention lower bound, we mean the worst-case lower bound.

From the above definition, it almost appears that we must enumerate all possible algorithms, determine the time complexity of each algorithm and find the minimum time complexity. This is of course impossible because we simply cannot enumerate all possible algorithms; we can never be sure that a new algorithm will not be invented tomorrow and this new algorithm may possibly lead to a better bound.

The reader should note that the lower bound, by definition, is not unique. A famous lower bound is the lower bound for sorting, which is $\Omega(n \log n)$. Let us imagine that before this lower bound is found, someone may prove that a lower bound for sorting is $\Omega(n)$ because every data element must be examined before sorting is completed. In fact, we may be even more extreme. For instance, before $\Omega(n)$ is suggested as a lower bound for sorting, someone may suggest $\Omega(1)$ as a lower bound because it takes at least one step to complete any algorithm.

From the above discussion, we know that there may be three lower bounds: $\Omega(n \log n)$, $\Omega(n)$ and $\Omega(1)$ for sorting. They are all correct. But, obviously, $\Omega(n \log n)$ is the only significant one. The other two are both trivial lower bounds. Since a lower bound is trivial if it is too low, we like our lower bound to be as high as possible. The research of lower bounds always starts with a scientist suggesting a rather low lower bound. Then someone would improve this lower bound by finding a higher lower bound. This higher lower bound replaces the old one and becomes the currently significant lower bound of this problem. This status will remain so until an even higher lower bound is found.

We must understand that each higher lower bound is found by theoretical analysis, not by pure guessing. As the lower bound goes higher and higher, we will inevitably wonder whether there is an upper limit of the lower bound? Let us consider, for example, the lower bound for sorting. Is there any possibility that

$\Omega(n^2)$ is suggested as a lower bound for sorting? No, because there is a sorting algorithm, namely heap sort, which has $O(n \log n)$ as its worst-case time complexity. We therefore know that the lower bound for sorting is at most $\Omega(n \log n)$.

Let us consider the following two cases:

Case 1. At present, the highest lower bound of a problem is found to be $\Omega(n \log n)$ and the time complexity of the best available algorithm to solve this problem is $O(n^2)$.

In this case, there are three possibilities:

(1) The lower bound of the problem is too low. We may therefore try to find a sharper, or higher, lower bound. In other words, we should try to move the lower bound upward.
(2) The best available algorithm is not good enough. We may therefore try to find a better algorithm, an algorithm with a lower time complexity. In other words, we should try to move the best time complexity downward.
(3) The lower bound may be improved and the algorithm may also be improved. We may therefore try to improve both.

Case 2. The present lower bound is $\Omega(n \log n)$ and there is indeed an algorithm with time complexity $O(n \log n)$.

In this case, we shall say that the lower bound as well as the algorithm cannot be improved any further. In other words, we have found an optimal algorithm to solve the problem as well as a truly significant lower bound of the problem.

Let us emphasize the above point. Note that at the very beginning of this chapter, we asked a question: How do we know that an algorithm is optimal? We now have the answer. *An algorithm is optimal if its time complexity is equal to a lower bound of this problem. Neither the lower bound nor the algorithm can be improved further*.

The following is a summary of some important concepts of lower bounds:

(1) The lower bound of a problem is the minimum time complexity of all algorithms which can be used to solve this problem.
(2) The lower bound that is higher is better.
(3) If the currently-known lower bound of a problem is lower than the time complexity of the best available algorithm to solve this problem, then it is possible that the lower bound can be improved by moving it up. The algorithm can be improved by moving its time complexity down, or both.

(4) If the currently-known lower bound is equal to the time complexity of an available algorithm, then both the algorithm and the lower bound cannot be improved any more. The algorithm is an optimal algorithm and the lower bound is the highest lower bound, the truly significant one.

In the following sections, we shall discuss some methods to find lower bounds.

2–4 THE WORST-CASE LOWER BOUND OF SORTING

For many algorithms, we can describe the execution of an algorithm as binary trees. For example, consider the case of straight insertion sort. Let us assume that the number of inputs is three and all data elements are distinct. In such a case, there are six distinct permutations:

a_1	a_2	a_3
1	2	3
1	3	2
2	1	3
2	3	1
3	1	2
3	2	1

When we apply the straight insertion sort to the above set of data, each permutation evokes a distinct response. For example, imagine that the input is (2, 3, 1). The straight insertion sort now behaves as follows:

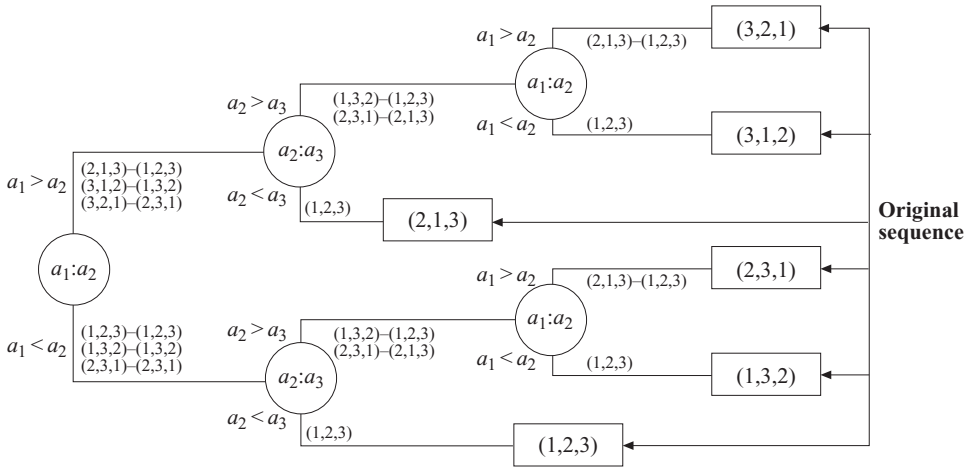
(1) Compare $a_1 = 2$ with $a_2 = 3$. Since $a_2 > a_1$, no exchange of data elements takes place.
(2) Compare $a_2 = 3$ with $a_3 = 1$. Since $a_2 > a_3$, exchange a_2 and a_3. That is, $a_2 = 1$ and $a_3 = 3$.
(3) Compare $a_1 = 2$ with $a_2 = 1$. Since $a_1 > a_2$, exchange a_1 and a_2. That is, $a_1 = 1$ and $a_2 = 2$.

If the input data is (2, 1, 3), the algorithm behaves as follows:

(1) Compare $a_1 = 2$ with $a_2 = 1$. Since $a_1 > a_2$, exchange a_1 and a_2. That is, $a_1 = 1$ and $a_2 = 2$.
(2) Compare $a_2 = 2$ with $a_3 = 3$. Since $a_2 < a_3$, no exchange of data takes place.

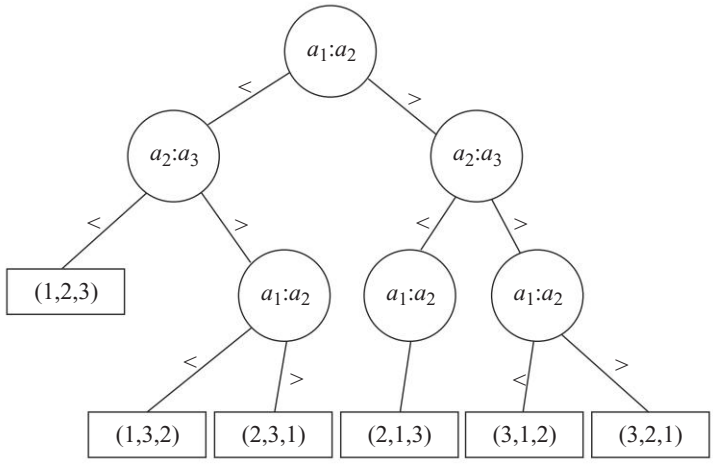
Figure 2–6 shows how the straight insertion sort can be described by a binary tree when three data elements are being sorted. This binary tree can be easily modified to handle the case of four data elements. It is easy to see that the straight insertion sort, when it is applied to any number of data elements, can be described by a binary decision tree.

FIGURE 2–6 Straight insertion sort with three elements represented by a tree.



In general, any sorting algorithm whose basic operation is a compare and exchange operation can be described by a binary decision tree. Figure 2–7 shows how the bubble sort can be described by a binary decision tree. In Figure 2–7, we assume that there are three data elements and they are all distinct. Figure 2–7 is also very much simplified so that it does not occupy a large space. We shall not analyze the bubble sort here because it is so well known.

FIGURE 2–7 The binary decision tree describing the bubble sort.



The action of a sorting algorithm based on compare and exchange operations on a particular input data set corresponds to one path from the top of the tree to a leaf node. Each leaf node therefore corresponds to a particular permutation. *The longest path from the top of the tree to a leaf node, which is called the depth of the tree, represents the worst-case time complexity of this algorithm*. To find the lower bound of the sorting problem, we have to find the smallest depth of some tree, among all possible binary decision trees modeling sorting algorithms.

Let us note the following important points:

(1) For every sorting algorithm, its corresponding binary decision tree will have $n!$ leaf nodes as there are $n!$ distinct permutations.
(2) The depth of a binary tree with a fixed number of leaf nodes will be the smallest if the tree is balanced.
(3) When a binary tree is balanced, the depth of the tree is $\lceil \log X \rceil$ where X is the number of leaf nodes.

Based on the above reasoning, we can easily conclude that a *lower bound of the sorting problem is* $\lceil \log n! \rceil$. That is, the number of comparisons needed to sort in the worst case is at least $\lceil \log n! \rceil$.

This $\log n!$ is perhaps a little puzzling to many of us. We just do not know how large this number is, and we shall now discuss two methods to approximate $\log n!$.

Method 1.

We use the fact that

$$\begin{aligned}
\log n! &= \log(n(n-1)\ldots 1) \\
&= \sum_{i=1}^{n} \log i \\
&= (2-1)\log 2 + (3-2)\log 3 + \cdots + (n-n+1)\log n \\
&> \int_1^n \log x dx \\
&= \log e \int_1^n \log_e x dx \\
&= \log e[x \log_e x - x]_1^n \\
&= \log e(n \log_e n - n + 1) \\
&= n \log n - n \log e + 1.44 \\
&\geq n \log n - 1.44n \\
&= n \log n\left(1 - \frac{1.44}{\log n}\right).
\end{aligned}$$

If we let $n = 2^2$, $n \log n\left(1 - \frac{1.44}{2}\right) = 0.28n \log n$.

Thus, by letting $n_0 = 2^2$ and $c = 0.28$, we have

$$\log n! \geq cn \log n \qquad \text{for } n \geq n_0.$$

That is, a worst-case lower bound of sorting is $\Omega(n \log n)$.

Method 2. The Stirling Approximation

The Stirling approximation approximates the value $n!$ as n is very large by the following formula:

$$n! \cong \sqrt{2\pi n}\left(\frac{n}{e}\right)^n.$$

This formula can be found in almost any advanced calculus book. Table 2–2 illustrates how closely the Stirling approximation approximates $n!$. In the table, we shall let S_n be the Stirling approximation.

TABLE 2–2 Some values of Stirling approximation.

n	$n!$	S_n
1	1	0.922
2	2	1.919
3	6	5.825
4	24	23.447
5	120	118.02
6	720	707.39
10	3,628,800	3,598,600
20	2.433×10^{18}	2.423×10^{18}
100	9.333×10^{157}	9.328×10^{157}

Using the Stirling approximation, we shall have

$$\begin{aligned}\log n! &= \log\sqrt{2\pi} + \frac{1}{2}\log n + n\log\frac{n}{e} \\ &= \log\sqrt{2\pi} + \frac{1}{2}\log n + n\log n - n\log e \\ &\geq n\log n - 1.44n.\end{aligned}$$

From both methods, we may claim that the minimum number of comparisons needed by sorting is $\Omega(n \log n)$, in the worst case.

We must note at this point that the above statement does not mean that a higher lower bound cannot be found. In other words, it is possible that some new discovery may inform us that the lower bound of sorting is actually higher. For example, it is possible that someone may find out that the lower bound of sorting is $\Omega(n^2)$.

In the next section, we shall show a sorting algorithm whose worst case time complexity is equal to the lower bound which we just derived. Because of the existence of such an algorithm, this lower bound cannot be made higher any more.

2–5 HEAP SORT: A SORTING ALGORITHM WHICH IS OPTIMAL IN WORST CASES

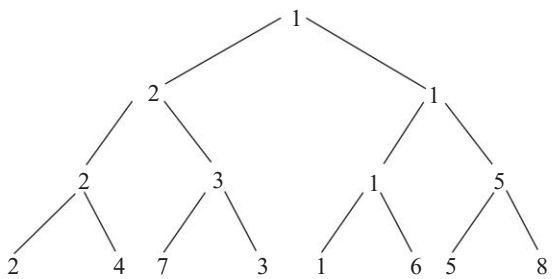
Heap sort is one of the sorting algorithms whose time complexities are $O(n \log n)$. Before introducing heap sort, let us examine the straight selection sort to see why it is not optimal in the worst case. For straight selection sort, we need $(n - 1)$ steps to obtain the smallest number, then $(n - 2)$ steps to obtain the

second smallest number and so on (all in worst cases). Therefore, in the worst case, $O(n^2)$ steps are needed for the straight selection sort. If we take a more careful look at the straight selection sort, we note that when we try to find the second smallest number, the information we may have extracted by finding the first smallest number is not used at all. This is why the straight insertion sort behaves so clumsily.

Let us consider another sorting algorithm, called knockout sort, which is much better than the straight selection sort. Knockout sort is similar to straight selection sort in the sense that it finds the smallest number, the second smallest and so on. However, it keeps some information after it finds the first smallest number so that it is quite efficient at finding the second smallest number.

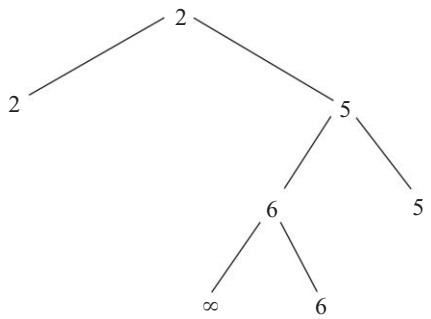
Consider the input sequence 2, 4, 7, 3, 1, 6, 5, 8. We may construct a knockout tree to find the smallest number, shown in Figure 2–8.

FIGURE 2–8 A knockout tree to find the smallest number.



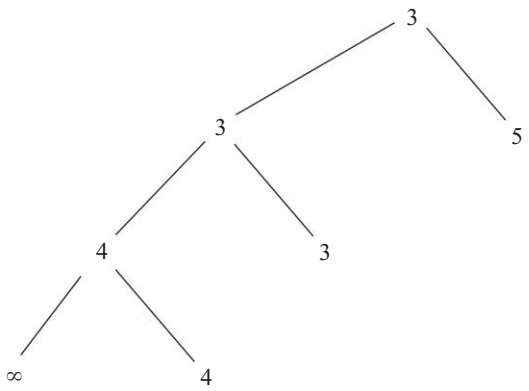
After the smallest number is found, we may start to find the second smallest number by substituting 1 with ∞. Then, only a small part of the knockout tree needs to be examined, shown in Figure 2–9.

FIGURE 2–9 Finding the second smallest number.



Every time a smallest number is found, we replace it by ∞ and the next smallest number can be easily found. For instance, we have now found the first two smallest numbers. The third smallest number can now be found as shown in Figure 2–10.

FIGURE 2–10 Finding the third smallest number in knockout sort.



To find the time complexity of knockout sort:

The first smallest number is found after $(n - 1)$ comparisons. For all of the other selections, only $\lceil \log n \rceil - 1$ comparisons are needed. Therefore, the total number of comparisons is

$$(n - 1) + (n - 1)(\lceil \log n \rceil - 1).$$

Thus, the time complexity of knockout sort is $O(n \log n)$, which is equal to the lower bound found in Section 2–4. Knockout sort is therefore an optimal sorting algorithm. We must note that the time complexity $O(n \log n)$ is valid for best, average and worst cases.

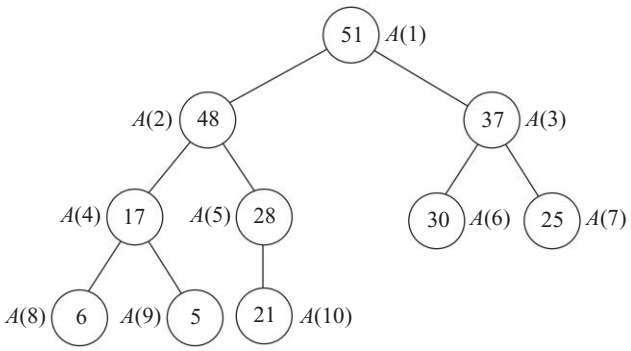
The reason that knockout sort is better than the straight selection sort is that it uses previous information. Unfortunately, the knockout tree needs extra space. Roughly $2n$ locations are needed to implement the knockout sort. The knockout sort can be improved to heap sort, which will be discussed in the rest of this section.

Similar to the knockout sort, the heap sort uses a special data structure to store the data. This data structure is called a heap. A heap is a binary tree satisfying the following conditions:

(1) This tree is completely balanced.
(2) If the height of this binary tree is h, then leaves can be at level h or level $h - 1$.
(3) All leaves at level h are as far to the left as possible.
(4) The data associated with all descendants of a node are smaller than the datum associated with this node.

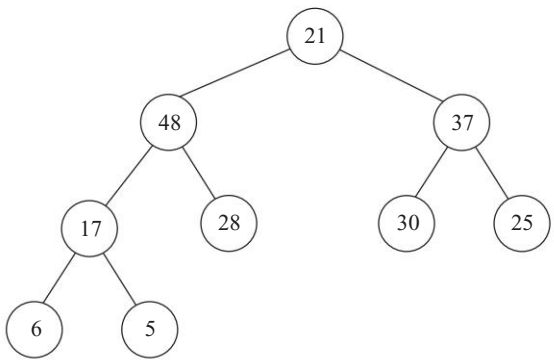
Figure 2–11 shows a heap for 10 numbers.

FIGURE 2–11 A heap.



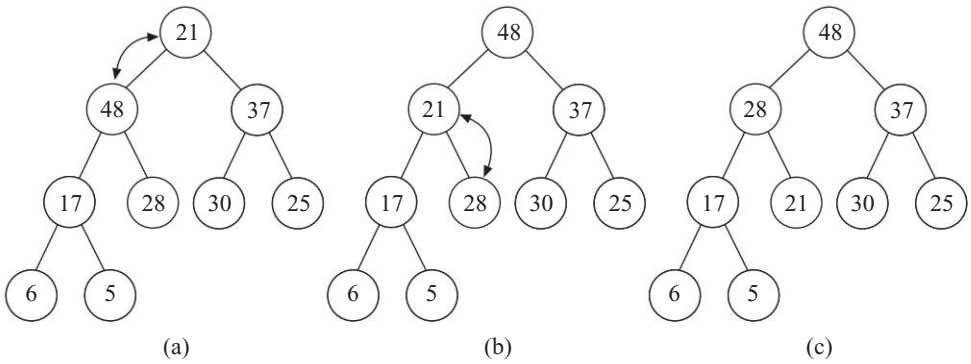
By definition, the root of the heap, $A(1)$, is the largest number. Let us assume that the heap is already constructed (the construction of a heap will be discussed later). Then we can output $A(1)$. After we output $A(1)$, which is the largest number, the original heap is not a heap any more. We now replace $A(1)$ by $A(n) = A(10)$. Thus, we have the tree shown in Figure 2–12.

FIGURE 2–12 Replacing $A(1)$ by $A(10)$.



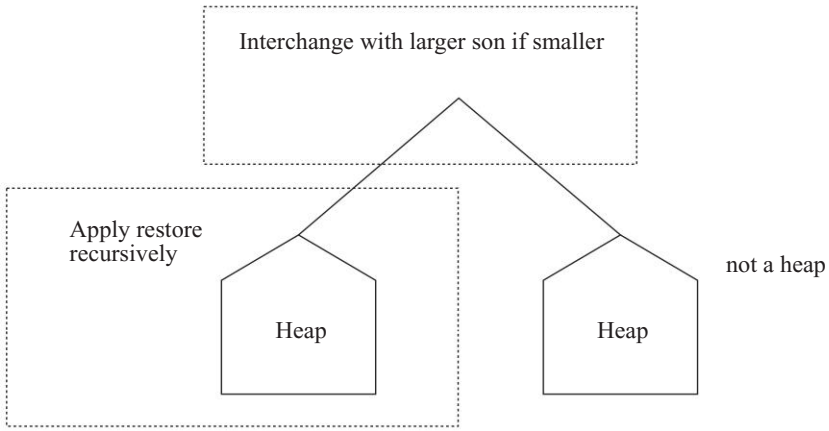
The balanced binary tree in Figure 2–12 is not a heap. But it can be easily restored, shown in Figure 2–13. The binary tree in Figure 2–13(c) is a heap.

FIGURE 2–13 The restoration of a heap.



The restore routine can be best understood by Figure 2–14.

FIGURE 2–14 The restore routine.



Algorithm 2–6 □ Restore(i, j)

Input: $A(i), A(i+1), \ldots, A(j)$.

Output: $A(i), A(i+1), \ldots, A(j)$ as a heap.

If $A(i)$ is not a leaf and if a son of $A(i)$ contains a larger element than $A(i)$, then

Begin

Let $A(h)$ be a son of $A(i)$ with the largest element

Interchange $A(i)$ and $A(h)$

Restore(h, j)

End

The parameter j is used to determine whether $A(i)$ is a leaf or not and whether $A(i)$ has two sons. If $i > j/2$, then $A(i)$ is a leaf and restore(i, j) need not do anything because $A(i)$ is already a heap itself.

We may say that there are two important elements of heap sort:

(1) Construction of the heap.
(2) Removing the largest number and restoring the heap.

Assuming that the heap is already constructed, then heap sort can be illustrated as follows:

Algorithm 2–7 □ Heap sort

Input: $A(1), A(2), \ldots, A(n)$ where each $A(i)$ is a node of a heap already constructed.

Output: The sorted sequence of $A(i)$'s.

For $i := n$ down to 2 do

Begin

Output $A(1)$

$A(1) := A(i)$

Delete $A(i)$

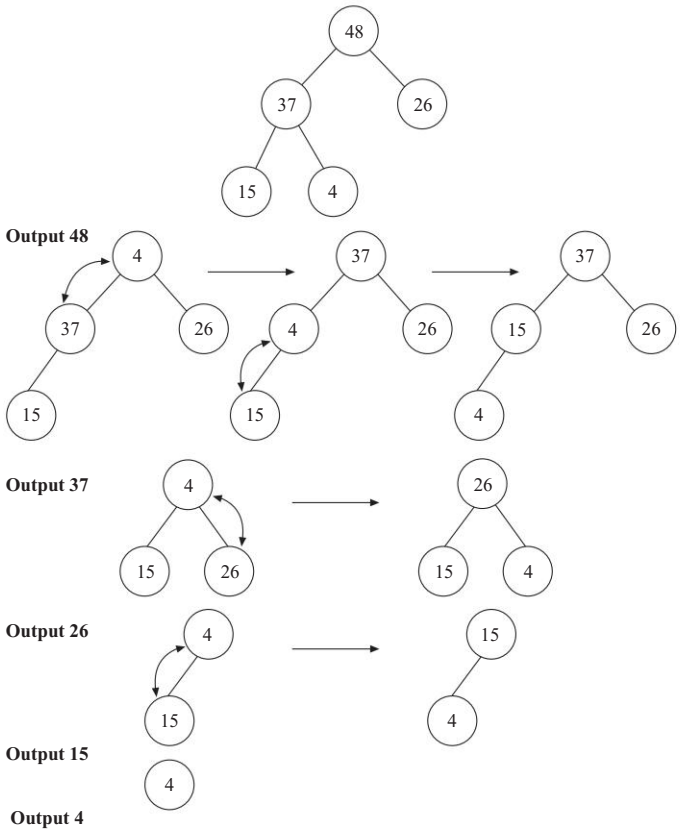
Restore $(1, i - 1)$

End

Output $A(1)$

Example 2–6 Heap sort

The following steps show a typical case of heap sort.



A beautiful characteristic of a heap is that we can represent a heap by an array. That is, we do not need pointers because a heap is a completely balanced binary tree. Each node and its descendants can be uniquely determined. The rule is rather simple: The descendants of $A(h)$ are $A(2h)$ and $A(2h + 1)$ if they exist.

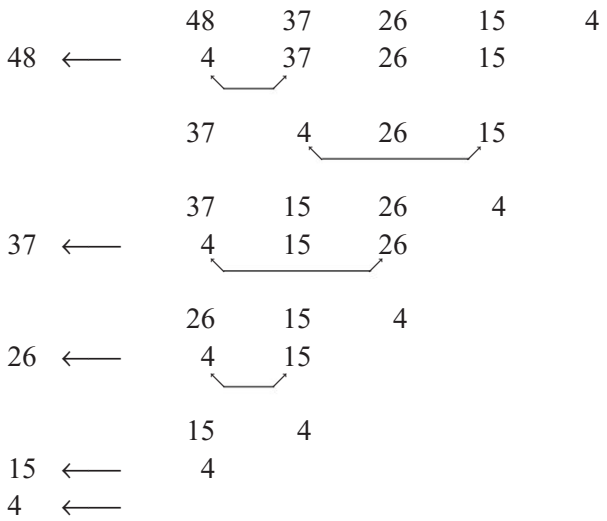
The heap in Figure 2–11 is now stored in an array:

$A(1)$	$A(2)$	$A(3)$	$A(4)$	$A(5)$	$A(6)$	$A(7)$	$A(8)$	$A(9)$	$A(10)$
51	48	37	17	28	30	25	6	5	21

Consider, for example, $A(2)$. Its left son is $A(4) = 17$.

Consider, $A(3)$. Its right son is $A(7) = 25$.

The entire process of heap sort can therefore be operated on an array. For example, the heap sort in Example 2–6 can now be described as follows:



The Construction of a Heap

To construct a heap, consider Figure 2–14 where the binary tree is not a heap. However, both subtrees below the top of the tree are heaps. For this kind of trees, a heap can be "constructed" by using the restore routine. The construction is based on the above idea. We start with any arbitrary completely balanced binary tree and gradually transform it into a heap by repeatedly invoking the restore routine.

Let $A(1), A(2), \ldots, A(n)$ be a completely balanced binary tree whose leaf nodes at the highest level are as far to the left as possible. For this binary tree, we can see that $A(i), i = 1, 2, \ldots, \lfloor n/2 \rfloor$ must be an internal node with descendants and $A(i), i = \lfloor n/2 \rfloor + 1, \ldots, n$ must be a leaf node without descendants. All leaf nodes can be trivially considered as heaps. So we do not have to perform any operation on them. The construction of a heap starts from restoring the subtree rooted at $\lfloor n/2 \rfloor$. The algorithm to construct a heap is as follows:

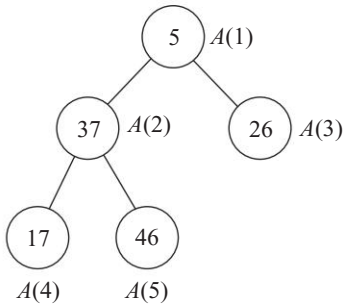
Algorithm 2–8 □ Construction of a heap

Input: $A(1), A(2), \ldots, A(n)$.
Output: $A(1), A(2), \ldots, A(n)$ as a heap.

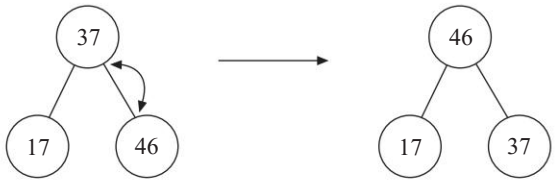
```
For i := ⌊n/2⌋ down to 1 do
    Restore (i, n)
```

Example 2–7 The Construction of a Heap

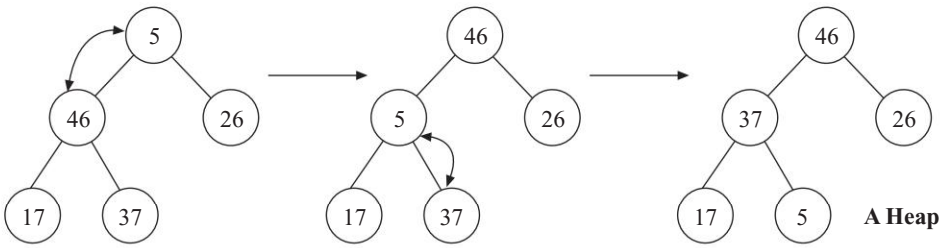
Consider the following binary tree which is not a heap.



In this heap, $n = 5$ and $\lfloor n/2 \rfloor = 2$. We therefore restore the subtree rooted at $A(2)$ as follows:



We then restore $A(1)$:



Heap sort is an improvement of knockout sort because it uses a linear array to represent the heap. The time complexity of heap sort is analyzed next.

Worst-Case Analysis of the Construction of a Heap

Let there be n numbers to be sorted. The depth d of a completely balanced binary tree is therefore $\lfloor \log n \rfloor$. For each internal node, two comparisons have to be

made. Let the level of an internal node be L. Then, in the worst case, $2(d - L)$ comparisons have to be made to perform the restore routine. The maximum number of nodes at level L is 2^L. Then the total number of comparisons for the construction stage is at most

$$\sum_{L=0}^{d-1} 2(d-L)2^L = 2d \sum_{L=0}^{d-1} 2^L - 4 \sum_{L=0}^{d-1} L2^{L-1}.$$

In Section 2–2, we proved in Equation (2–1) that

$$\sum_{L=0}^{k} L2^{L-1} = 2^k(k-1)+1.$$

Therefore,

$$\begin{aligned}
\sum_{L=0}^{d-1} 2(d-L)2^L &= 2d \sum_{L=0}^{d-1} 2^L - 4 \sum_{L=0}^{d-1} L2^{L-1} \\
&= 2d(2^d - 1) - 4(2^{d-1}(d - 1 - 1) + 1) \\
&= 2d(2^d - 1) - 4(d2^{d-1} - 2^d + 1) \\
&= 4 \cdot 2^d - 2d - 4 \\
&= 4 \cdot 2^{\lfloor \log n \rfloor} - \lfloor 2 \log n \rfloor - 4 \\
&= cn - \lfloor 2 \log n \rfloor - 4 \qquad \text{where } 2 \le c \le 4 \\
&\le cn.
\end{aligned}$$

Thus, the total number of comparisons needed to construct a heap in the worst case is $O(n)$.

The Time Complexity of Deleting Elements from a Heap

As we showed before, after a heap is constructed, the top of the heap will be the largest number and can be deleted (or output) now. We now analyze the number of comparisons needed to output all of the number elements from a heap consisting of n elements. Note that after deleting a number, in the worst case, $2\lfloor \log i \rfloor$ comparisons are needed to restore the heap if there are i elements remaining. So the total number of steps needed to delete all numbers is

$$2 \sum_{i=1}^{n-1} \lfloor \log i \rfloor.$$

To evaluate this formula, let us consider the case of $n = 10$.

$\lfloor \log 1 \rfloor = 0$
$\lfloor \log 2 \rfloor = \lfloor \log 3 \rfloor = 1$
$\lfloor \log 4 \rfloor = \lfloor \log 5 \rfloor = \lfloor \log 6 \rfloor = \lfloor \log 7 \rfloor = 2$
$\lfloor \log 8 \rfloor = \lfloor \log 9 \rfloor = 3.$

We observe that there are

2^1 numbers equal to $\lfloor \log 2^1 \rfloor = 1$
2^2 numbers equal to $\lfloor \log 2^2 \rfloor = 2$

and $10 - 2^{\lfloor \log 10 \rfloor} = 10 - 2^3 = 2$ numbers equal to $\lfloor \log n \rfloor$.

In general,

$$2\sum_{i=1}^{n-1} \lfloor \log i \rfloor = 2\sum_{i=1}^{\lfloor \log n \rfloor - 1} i2^i + 2(n - 2^{\lfloor \log n \rfloor})\lfloor \log n \rfloor$$

$$= 4\sum_{i=1}^{\lfloor \log n \rfloor - 1} i2^{i-1} + 2(n - 2^{\lfloor \log n \rfloor})\lfloor \log n \rfloor.$$

Using $\sum_{i=1}^{k} i2^{i-1} = 2^k(k-1) + 1$ (Equation 2–1 in Section 2–2)
we have

$$2\sum_{i=1}^{n-1} \lfloor \log i \rfloor$$

$$= 4\sum_{i=1}^{\lfloor \log n \rfloor - 1} i2^{i-1} + 2(n - 2^{\lfloor \log n \rfloor})\lfloor \log n \rfloor$$

$$= 4(2^{\lfloor \log n \rfloor - 1}(\lfloor \log n \rfloor - 1 - 1) + 1) + 2n\lfloor \log n \rfloor - 2\lfloor \log n \rfloor 2^{\lfloor \log n \rfloor}$$

$$= 2 \cdot 2^{\lfloor \log n \rfloor}\lfloor \log n \rfloor - 8 \cdot 2^{\lfloor \log n \rfloor - 1} + 4 + 2n\lfloor \log n \rfloor - 2 \cdot 2^{\lfloor \log n \rfloor}\lfloor \log n \rfloor$$

$$= 2 \cdot n\lfloor \log n \rfloor - 4 \cdot 2^{\lfloor \log n \rfloor} + 4$$

$$= 2n\lfloor \log n \rfloor - 4cn + 4 \qquad \text{where } 2 \le c \le 4$$

$$= O(n \log n).$$

Therefore, the worst-case time complexity of producing all elements from a heap in a sorted order is $O(n \log n)$.

In summary, we conclude that the worst case time complexity of heap sort is $O(n \log n)$. We emphasize here that heap sort achieves this $O(n \log n)$ time complexity essentially because it uses a data structure in such a way that each output operation will take at most $\lfloor \log i \rfloor$ steps where i is the number of remaining elements. This clever design of data structure is quite essential to heap sort.

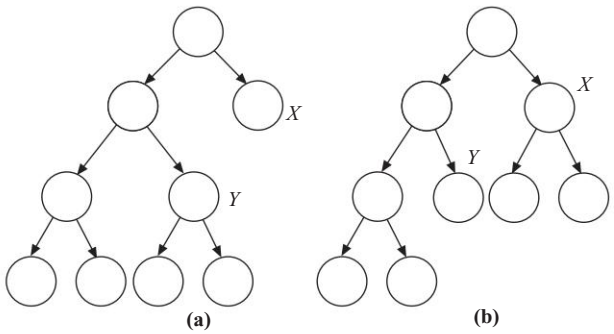
2–6 THE AVERAGE-CASE LOWER BOUND OF SORTING

In Section 2–4, we investigated the worst-case lower bound of sorting. In this section, we shall derive the average-case lower bound of sorting problem. We shall still use the binary decision tree model.

As we discussed before, every sorting algorithm based on comparisons can be described by a binary decision tree. In such a binary decision tree, the path from the root of the tree to a leaf node corresponds to the action of the algorithm in response to a particular case of input. Furthermore, the length of this path is equal to the number of comparisons executed for this input data set. Let us define *the external path length of a tree as the sum of the lengths of paths from the root to each of the leaf nodes*. Then the average time complexity of a sorting algorithm based on comparisons is equal to the external path length of the binary decision tree corresponding to this algorithm divided by the number of leaf nodes, which is $n!$.

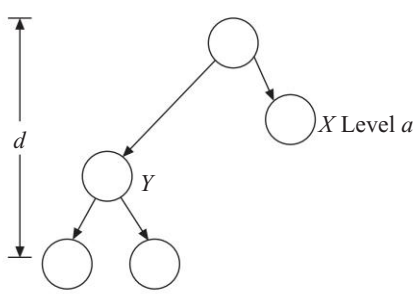
To find the lower bound of the average time complexity of sorting, we must find the minimum external path length of all possible binary trees with $n!$ leaf nodes. Among all possible binary trees with a fixed number of leaf nodes, the external path length is minimized if this tree is balanced. That is, all leaf nodes are on level d or level $d - 1$ for some d. Consider Figure 2–15. In Figure 2–15(a), the binary tree is not balanced. The external path length of this tree is $4 \times 3 + 1 = 13$. We can now decrease its external path length by removing the two descendants of Y and attaching them to X. The external path length now becomes $2 \times 3 + 3 \times 2 = 12$.

FIGURE 2–15 The modification of an unbalanced binary tree.



The general case is now described in Figure 2–16. In Figure 2–16, suppose that there is a leaf node on level a and the depth of the tree is d where $a \leq d - 2$. This tree can be modified in such a way that the external path length is decreased without changing the number of leaf nodes. To modify the tree, select any node on level $d - 1$ with descendants on level d. Let the leaf node on level a and the node on level $d - 1$ be denoted as X and Y respectively. Remove the descendants of Y and attach them to X. For node Y, it originally has two descendants and the sum of their path length is $2d$. Now Y becomes a leaf node and its path length is $d - 1$. This removal decreases the external path length by $2d - (d - 1) = d + 1$. For X, it originally was a leaf node. Now it becomes an internal node and its two descendant nodes become leaf nodes. Originally, the path length was a. Now the sum of the two new path lengths is $2(a + 1)$. Thus, this attachment increases the external path length by $2(a + 1) - a = a + 2$. The net change is $(d + 1) - (a + 2) = (d - a) - 1 \geq 2 - 1 = 1$. That is, the net change is decreasing the external path length.

FIGURE 2–16 An unbalanced binary tree.



Therefore, we conclude that an unbalanced binary tree can be modified so that the external path length is decreased and the external path length of a binary tree is minimized if and only if the tree is balanced.

Let there be in total x leaf nodes. We shall now calculate the external path length of a balanced binary tree with c leaf nodes. This external path length is found through the following reasoning:

(1) The depth of the tree is $d = \lceil \log c \rceil$. Leaf nodes can appear only on level d or level $d - 1$.

(2) Let there be x_1 leaf nodes on level $d - 1$ and x_2 leaf nodes on level d. Then

$$x_1 + x_2 = c.$$

(3) To simplify the discussion, let us assume that the number of nodes on level d is even. The reader may easily note that if the number of nodes on level d is odd, the following result still holds. For each two nodes on level d, there is one parent node on level $d - 1$. This parent node is not a leaf node. Thus, we have the following equation

$$x_1 + \frac{x_2}{2} = 2^{d-1}.$$

(4) Solving these equations, we obtain

$$\frac{x_2}{2} = c - 2^{d-1}$$

$$x_2 = 2(c - 2^{d-1})$$

$$x_1 = 2^d - c.$$

(5) The external path length is

$$x_1(d - 1) + x_2 d$$

$$= (2^d - c)(d - 1) + (2c - 2^d)d$$

$$= c + cd - 2^d.$$

(6) Since $d = \lceil \log c \rceil$, substituting $\log c \leq d < \log c + 1$ into the above equation, we have $c + cd - 2^d \geq c + c(\log c) - 2.2^{\log c} = c \log c - c$. Thus, the external path length is greater than $c \log c - c = n! \log n! - n!$. The average-case time complexity of sorting is therefore greater than

$$\frac{n! \log n! - n!}{n!} = \log n! - 1.$$

Using the result discussed in Section 2–4, we now conclude that *the average-case lower bound of the sorting problem is* $\Omega(n \log n)$.

In Section 2–2, Example 2–4, we showed that the average-case time complexity of quick sort is $O(n \log n)$. Thus, quick sort is optimal so far as its average case performance is concerned.

In Section 2–2, Example 2–3, it was shown that the average-case time complexity of straight selection sort is also $O(n \log n)$. However, we must understand that this time complexity is in terms of the changing of flag. The number of comparisons for straight selection sort is $n(n - 1)/2$ in average and worst cases. Since the number of comparisons is a dominating time factor in practical programming, the straight selection sort is quite slow in practice.

The famous bubble sort, as well as straight insertion sort, has average-case time complexity of $O(n^2)$. Experience tells us that bubble sort is much slower than quick sort.

Finally, let us take a look at heap sort discussed in Section 2–5. The worst-case time complexity of heap sort is $O(n \log n)$ and the average-case time complexity of heapsort has never been found. However, we know that it must be higher than or equal to $O(n \log n)$ because of the lower bound found in this section. But it cannot be higher than $O(n \log n)$ because its worst-case time complexity is $O(n \log n)$. We therefore may deduce the fact that the average-case time complexity of heap sort is $O(n \log n)$.

2–7 IMPROVING A LOWER BOUND THROUGH ORACLES

In the previous sections, we showed how to use the binary decision tree model to obtain a lower bound for sorting. It is just fortunate that the lower bound was found to be a good one. That is, there exists an algorithm whose worst-case time complexity is exactly equal to this lower bound. Therefore, we can be sure that this lower bound cannot be made any higher.

In this section, we shall show a case where the binary decision tree model does not produce a very meaningful lower bound. That is, we shall show that we can still improve the lower bound obtained by using the binary decision tree model.

Consider the merging problem. If our merging algorithm is based upon the compare and exchange operation, then the decision tree model can be used. A lower bound of merging can be derived by using the reasoning for deriving the lower bound of sorting. In sorting, the number of leaf nodes is $n!$ and the lower bound of sorting is thus $\lfloor \log_2 n! \rfloor$. In merging, the number of leaf nodes is again the number of distinct cases that we want to distinguish. So, given two sorted sequences A and B of m and n elements respectively, how many possible different

merged sequences are there? Again, to simplify the discussion, let us assume that all $(m + n)$ elements are distinct. After n elements are merged into m elements, there are totally $\binom{m+n}{n}$ ways that they are merged without disturbing the original ordering of sequences A and B. This means that we can obtain a lower bound for merging as

$$\left\lceil \log \binom{m+n}{n} \right\rceil.$$

However, we have never found any merging algorithm which achieves this lower bound.

Let us consider a conventional merging algorithm which compares the top elements of two sorted lists and outputs the smaller one. For this merging algorithm, the worst-case time complexity is $m + n - 1$ which is higher than or equal to

$$\left\lceil \log \binom{m+n}{n} \right\rceil.$$

That is, we have the following inequality

$$\left\lceil \log \binom{m+n}{n} \right\rceil \leq m + n - 1.$$

How do we bridge the gap? As discussed earlier, we can either increase the lower bound or find a better algorithm with a lower time complexity. In fact, it is interesting to point out that when $m = n$, another lower bound of merging is $m + n - 1 = 2n - 1$.

We shall show this by the oracle approach. An oracle will give us a very hard case (a particular input data). If we apply any algorithm to this data set, the algorithm will have to work very hard to solve the problem. Using this data set, we can therefore derive a worst-case lower bound.

Suppose we have two sequences $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_n$. Furthermore, consider the very hard case where $a_1 < b_1 < a_2 \ldots a_n < b_n$. Suppose some merging algorithm has already merged $a_1, a_2, \ldots, a_{i-1}$ with $b_1, b_2, \ldots, b_{i-1}$ correctly and produces the following sorted sequence:

$$a_1, b_1, \ldots, a_{i-1}, b_{i-1}.$$

However, assume that this merging algorithm will not compare a_i with b_i. Obviously, there is no way for the algorithm to make a correct decision on whether a_i, or b_i, should be put next to b_{i-1}. Thus,a_i and b_i must be compared. Using similar reasoning, we can prove that b_i and a_{i+1} must be compared after a_i is put next to b_{i-1}. In summary, every b_i must be compared with a_i and a_{i+1}. Therefore, totally $2n - 1$ comparisons are needed for any merging algorithm when $m = n$. We would like to remind the reader that this lower bound $2n - 1$ for merging is only valid for the case $m = n$.

Since the conventional merging algorithm needs $2n - 1$ comparisons for the case $m = n$, we may conclude that the conventional merging algorithm is optimal because its worst-case time complexity is equal to this lower bound.

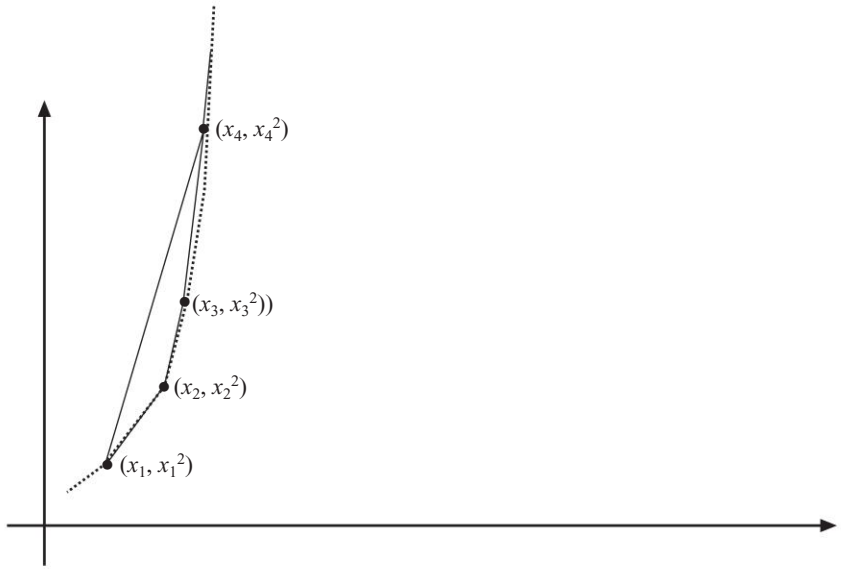
The above discussion shows that we can sometimes improve a lower bound to a higher one.

2–8 FINDING THE LOWER BOUND BY PROBLEM TRANSFORMATION

In the previous sections, we found lower bounds by directly analyzing the problems. Sometimes, this appears to be difficult. For instance, the convex hull problem is to find the smallest convex polygon (hull) of a set of planar points. What is the lower bound of the convex hull problem? It appears rather difficult for us to find a meaningful lower bound of this problem directly. However, we will next demonstrate that we can easily obtain a very meaningful lower bound by transforming the sorting problem, whose lower bound is known, to this problem.

Let the set of numbers to be sorted be $x_1, x_2, \ldots, x_n$ and without losing generality, we may assume that $x_1 < x_2 \ldots < x_n$. We then associate each x_i with x_i^2 to form a 2-dimensional point (x_i, x_i^2). All these newly created points lie on the parabola $y = x^2$. Consider the convex hull constructed out of these $n(x_i, x_i^2)$ points. As shown in Figure 2–17, this convex hull will consist of a list of sorted numbers. In other words, by solving the convex hull problem, we can also solve the sorting problem. The total time of sorting is equal to the time needed for the transformation plus the time needed to solve the convex hull problem. Thus,the lower bound of the convex hull problem is equal to the lower bound of sorting problem minus the time needed for transformation. That is, the lower bound of the convex hull problem is no less than $\Omega(n \log n) - \Omega(n) = \Omega(n \log n)$ as the transformation takes $\Omega(n)$ steps. The reader will see that this lower bound cannot be made higher because there is an algorithm to solve the convex hull problem in $\Omega(n \log n)$ steps.

FIGURE 2–17 A convex hull constructed out of data from a sorting problem.



In general, suppose that we want to find a lower bound for problem P_1. Let there be a problem P_2 whose lower bound is known. Furthermore, suppose that P_2 can be transformed into P_1 in such a way that P_2 can be solved after P_1 is solved. Let $\Omega(f_1(n))$ and $\Omega(f_2(n))$ denote the lower bounds of P_1 and P_2 respectively. Let $O(g(n))$ denote the time needed to transform P_2 into P_1. Then

$$\Omega(f_1(n)) + O(\mathrm{g}(n)) \geq \Omega(f_2(n))$$
$$\Omega(f_1(n)) \geq \Omega(f_2(n)) - O(\mathrm{g(n)}).$$

We now give another example to demonstrate the feasibility of this approach. Suppose that we want to find the lower bound of the Euclidean minimum spanning tree problem. Since it is hard to obtain the lower bound of P_1 directly, we consider P_2 which is again the sorting problem. We define the transformation thus: for each x_i, let $(x_i, 0)$ be a 2-dimensional point. We can see that sorting will be completed as soon as the minimum spanning tree is constructed out of $(x_i, 0)$'s. Let us again assume that $x_1 < x_2 < \cdots < x_n$. Then there will be an edge between $(x_i, 0)$ and $(x_j, 0)$ on the minimum spanning tree if and only if $j = i + 1$. Therefore, the solution of the Euclidean minimum spanning tree problem is also a solution for the sorting problem. Again, we can see that a meaningful lower bound of the Euclidean minimum spanning tree is $\Omega(n \log n)$.

2–9 NOTES AND REFERENCES

This chapter introduces some basic concepts of the analysis of algorithms. For more study on the subject of algorithm analysis, consult Basse and Van Gelder (2000); Aho, Hopcroft and Ullman (1974); Greene and Knuth (1981); Horowitz, Sahni and Rajasekaran (1998) and Purdom and Brown (1985a). Several Turing award winners are excellent algorithm researchers. In 1987, the ACM Press published a collection of 20 Turing award lectures (Ashenhurst, 1987). In this volume, lectures by Rabin, Knuth, Cook and Karp all talk about complexities of algorithms. The 1986 Turing award went to Hopcroft and Tarjan. Tarjan's Turing award lecture was about algorithm design and can be found in Tarjan (1987). Weide (1977) also gave a survey of algorithm analysis techniques.

This chapter introduces several sorting algorithms. To have a more comprehensive discussion of sorting and searching, readers may read Knuth (1973). For more on analysis of the straight insertion sort, the binary search and the straight selection sort, see Section 5.2.1, Section 6.2.1 and Section 5.2.3 of Knuth (1973) respectively. Quick sort was due to Hoare (1961, 1962). Heap sort was discovered by Williams (1964). For more on finding the lower bound of sorting, see Section 5.3.1 of Knuth (1973). For more information about the knockout sort, see Section 5.2.3 of Knuth (1973).

The basic terminologies of a tree can be found in many textbooks of data structure. For example, see Sections 5.1–5.2 of Horowitz and Sahni (1976). The depth of a tree is also called the height of a tree. To know more about the analysis of the external path length of a tree and the effect of the complete binary tree, see Section 2.3.4.5 of Knuth (1969).

A study of the minimum spanning tree problem can be found in Section 6.1 of Preparata and Shamos (1985). There are more information for the rank finding problem in Section 4.1 of Shamos (1978) and in Section 8.8.3 of Preparata and Shamos (1985). The proof that the average-time complexity for median finding is $O(n)$ can be found in Section 3.6 of Horowitz and Sahni (1978).

For material on improving a lower bound through oracles, see Section 5.3.2 of Knuth (1973) and also Section 10.2 of Horowitz and Sahni (1978). For material on finding lower bounds by problem transformation, see Sections 3.4 and 6.1.4 of Shamos (1978), and also see Sections 3.2 and 5.3 of Preparata and Shamos (1985). There are many examples of proving lower bounds by transformation in Shamos (1978) and Preparata and Shamos (1985).

2–10 FURTHER READING MATERIALS

Lower bound theories have continuously attracted researchers. Some recently published papers on this subject include: Dobkin and Lipton (1979); Edwards and Elphick (1983); Frederickson (1984); Fredman (1981); Grandjean (1988); Hasham and Sack (1987); Hayward (1987); John (1988); Karp (1972); McDiarmid (1988); Mehlhorn, Naher and Alt (1988); Moran, Snir and Manber (1985); Nakayama, Nishizeki and Saito (1985); Rangan (1983); Traub and Wozniakowski (1984); Yao (1981); and Yao (1985).

For some very interesting newly published papers, consult Berman, Karpinski, Larmore, Plandowski and Rytter (2002); Blazewicz and Kasprzak (2003); Bodlaender, Downey, Fellows and Wareham (1995); Boldi and Vigna (1999); Bonizzoni and Vedova (2001); Bryant (1999); Cole (1994); Cole and Hariharan (1997); Cole, Farach-Colton, Hariharan, Przytycka and Thorup (2000); Cole, Hariharan, Paterson and Zwick (1995); Crescenzi, Goldman, Papadimitriou, Piccolboni and Yannakakis (1998); Darve (2000); Day (1987); Decatur, Goldreich and Ron (1999); Demri and Schnoebelen (2002); Downey, Fellows, Vardy and Whittle (1999); Hasegawa and Horai (1991); Hoang and Thierauf (2003); Jerrum (1985); Juedes and Lutz (1995); Kannan, Lawler and Warnow (1996); Kaplan and Shamir (1994); Kontogiannis (2002); Leoncini, Manzini and Margara (1999); Maes (1990); Maier (1978); Marion (2003); Martinez and Roura (2001); Matousek (1991); Naor and Ruah (2001); Novak and Wozniakowski (2000); Owolabi and McGregor (1988); Pacholski, Szwast and Tendera (2000); and Peleg and Rubinovich (2000); and Ponzio, Radhakrishnan and Venkatesh (2001).

Exercises

2.1 Give the best, worst and average numbers of exchanges needed in bubble sort, whose definition can be found in almost any textbook on algorithms. The best case and worst case analyses are trivial. The average case analysis can be done by the following process:

(1) Define inversion of a permutation. Let $a_1, a_2, \ldots, a_n$ be a permutation of the set $(1, 2, \ldots, n)$. If $i < j$ and $a_j < a_i$, then

(a_i, a_j) is called an inversion of this permutation. For instance, (3, 2) (3, 1) (2, 1) (4, 1) are all inversions of the permutation (3, 2, 4, 1).

(2) Find out the relationship between the probability that a given permutation has exactly k inversions and the number of permutations of n elements having exactly k inversions.

(3) Using induction, prove that the average number of exchanges needed for bubble sort is $n(n-1)/4$.

2.2 Write a program for bubble sort. Run an experiment to convince yourself that the average performance of it is indeed $O(n^2)$.

2.3 Find the algorithm of Ford-Johnson algorithm for sorting, which is reproduced in many textbooks in algorithms. It was shown that this algorithm is optimal for $n \leq 12$. Implement this algorithm on a computer and compare it with any other sorting algorithm. Do you like this algorithm? If not, try to determine what is wrong with the analysis.

2.4 Show that to sort five numbers, we need at least seven comparisons. Then demonstrate that the Ford–Johnson algorithm does achieve this lower bound.

2.5 Show that to find the largest number in a list of n numbers requires at least $n - 1$ comparisons.

2.6 Show that to find the second largest one of a list of n numbers, we need at least $n - 2 + \lceil \log n \rceil$ comparisons.

Hint: We cannot determine the second largest element without having determined the largest element. Thus,the analysis can be done by the following:

(1) Show that at least $n - 1$ comparisons are necessary to find the largest element.

(2) Show that there is always some sequence of comparisons which forces the second largest one to be found in $\lceil \log n \rceil - 1$ additional comparisons.

2.7 Show that if $T(n) = aT\left(\frac{n}{b}\right) + n^c$, then for n a power of b and

$$T(1) = k,\ T(n) = ka^{\log_b n} + n^c\left(\frac{b}{a - b^c}\right)\left(\left(\frac{a}{b^c}\right)^{\log_b n} - 1\right).$$

2.8 Show that if $T(n) = \sqrt{n}\,T(\sqrt{n}) + n$, $T(m) = k$ and $m = n^{1/2^i}$, then $T(n) = kn^{(2^i - 1)/2^i} + in$.

2.9 Read Theorem 10.5 of Horowitz and Sahni 1978. The proof of this theorem gives a good method to find a lower bound.

2.10 Show that binary search is optimal for all searching algorithm performing comparisons only.

2.11 Given the following pairs of functions, what is the smallest value of n such that the first function is larger than the second one.
(a) 2^n, $2n^2$.
(b) $n^{1.5}$, $2n \log_2 n$.
(c) n^3, $5n^{2.81}$.

2.12 Is $\Omega(n \log n)$ time a lower bound for the problem of sorting n integers ranging from 1 to C, where C is a constant? Why?

chapter

3

The Greedy Method

The greedy method is a strategy to solve some optimization problems. Let us suppose that we can solve a problem by a sequence of decisions. The greedy method employs the following approach: In each stage, our decision is a locally-optimal one. For some problems, as we shall see, these locally-optimal solutions will finally add up to a globally-optimal solution. We emphasize here that only a few optimization problems can be solved by this greedy method. In cases where these locally-optimal decisions do not result in a globally-optimal solution, the greedy method might still be recommended because as we shall see later, it at least produces a solution which is usually acceptable.

Let us now describe the spirit of greedy method by an example. Consider the case where we are given a set of n numbers and we are asked to pick out k numbers such that the sum of these k numbers is the largest, among all possible ways of picking out these k numbers.

To solve this problem, one may test all possible ways of picking k numbers out of these n numbers. That is, of course, a foolish way of solving this problem because we may simply pick out the k largest numbers and these k largest numbers must constitute our solution. Or, we may say that our algorithm to solve this problem is as follows:

For $i := 1$ to k do

    Pick out the largest number and delete this number from the input.

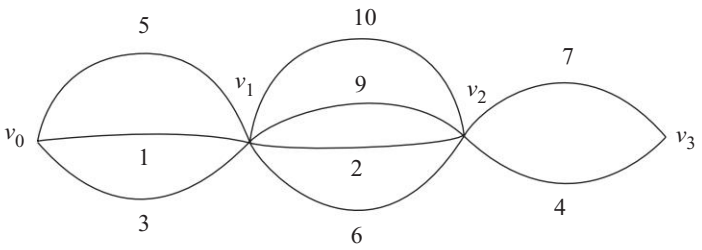
Endfor

The above algorithm is a typical greedy method. At each stage, the largest number is selected.

Let us consider another case where we can also use the greedy method. In Figure 3–1, we are asked to find a shortest path from v_0 to v_3. For this particular graph, we may solve the problem by finding a shortest path between v_i and v_{i+1}, $i = 0$ to 2. That

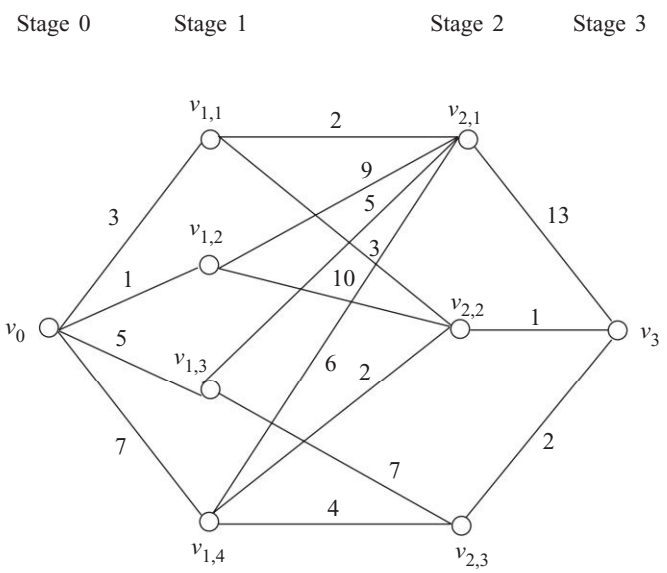
is, we first determine the shortest path between v_0 and v_1, then between v_1 and v_2 and so on. It should be obvious that this will finally produce an optimal solution.

FIGURE 3–1 A case where the greedy method works.



However, we can easily give an example in which the greedy method will not work. Consider Figure 3–2.

FIGURE 3–2 A case where the greedy method will not work.



In Figure 3–2, we are again asked to produce a shortest path from v_0 to v_3. If the greedy method is used, we shall find a shortest path from v_0 to some node

in Stage 1. Thus, $v_{1,2}$ is selected. In the next move, we shall find the shortest path between $v_{1,2}$ and some node in Stage 2. $v_{2,1}$ will be selected. The final solution will be

$$v_0 \rightarrow v_{1,2} \rightarrow v_{2,1} \rightarrow v_3.$$

The total length of this path is $1 + 9 + 13 = 23$.

This solution, although obtained in a fast way, is not an optimal solution. In fact, the optimum solution is

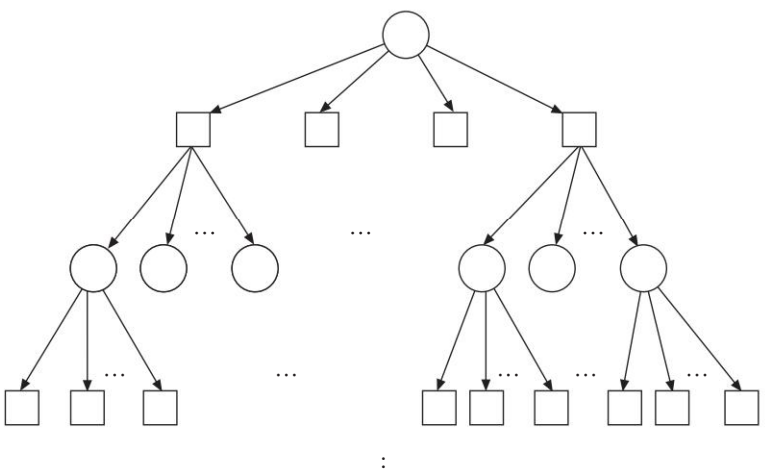
$$v_0 \rightarrow v_{1,1} \rightarrow v_{2,2} \rightarrow v_3$$

whose cost is $3 + 3 + 1 = 7$.

We shall show methods to solve the problem in later chapters.

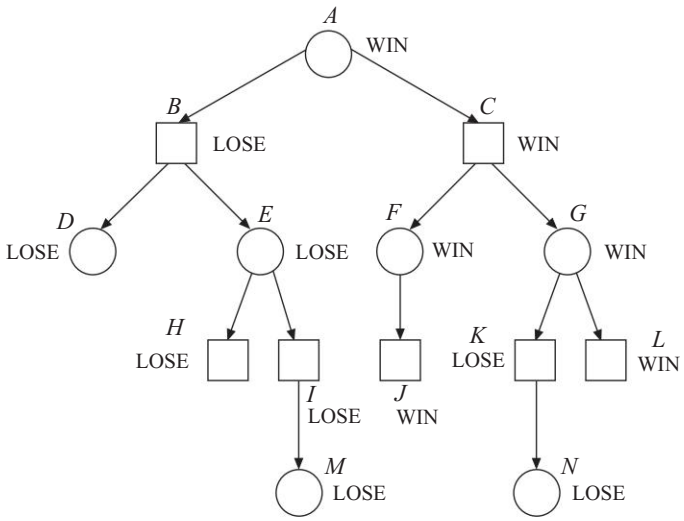
What is wrong when we apply the greedy method to solve the problem shown in Figure 3–2? To answer this question, let us briefly direct our mind to chess playing. A good chess player never simply looks at the chess board and makes a move which is the best for him at that time; he will "look ahead." That is, he will have to figure out most of the possible moves which he can make and then imagine how his opponent may react. He must understand that his opponent will also look ahead. Thus, the entire game may look like the tree shown in Figure 3–3.

FIGURE 3–3 A game tree.



In Figure 3–3, a circle represents a player and a box represents his opponent. Only when we can prune such a game tree can we make our first intelligent, and correct, move. Consider Figure 3–4, which is a fictitious end-game tree.

FIGURE 3–4 An end-game tree.



From this end-game tree, we can decide which move we should make by reasoning that:

(1) Since the opponent always wants us to lose, we label *I* and *K* as LOSE; if any of these states is reached, we definitely will lose.
(2) Since we always want to win, we label *F* and *G* as WIN. Besides, we must label *E* as LOSE.
(3) Again, since the opponent always wants us to lose, we label *B* and *C* as LOSE and WIN respectively.
(4) Since we want to win, we label *A* as WIN.

From the above discussion, *we understand that when we make decisions, we often have to look ahead. The greedy method, however, never does any work of looking ahead*. Consider Figure 3–2. To find a shortest path from v_0 to v_3, we also need to look ahead. To select a node from the Stage 1 nodes, we must know the shortest distance from each node in Stage 1 to v_3. Let $dmin(i, j)$ denote the minimum distance between nodes i and j. Then

$$dmin(v_0, v_3) = \begin{cases} 3 + dmin(v_{1,1}, v_3) \\ 1 + dmin(v_{1,2}, v_3) \\ 5 + dmin(v_{1,3}, v_3) \\ 7 + dmin(v_{1,4}, v_3). \end{cases}$$

The reader can now see that since the greedy method does not look ahead, it may fail to produce an optimal solution. Yet, in the rest of this chapter, we shall show many interesting examples in which the greedy method works.

3-1 KRUSKAL'S METHOD TO FIND A MINIMUM SPANNING TREE

One of the most famous problems which can be solved by the greedy method is the minimum spanning tree problem. In this section, we shall introduce Kruskal's method to find a minimum spanning tree. Minimum spanning trees can be defined either on Euclidean space points or on a graph. For Kruskal's method, minimum spanning trees are defined on graphs.

Definition

Let $G = (V, E)$ be a weighted connected undirected graph where V represents the set of vertices and E represents the set of edges. A spanning tree of G is an undirected tree $S = (V, T)$ where T is a subset of E. The total weight of a spanning tree is the sum of all weights of T. A minimum spanning tree of G is a spanning tree of G with the smallest total weight.

Example 3–1 Minimum Spanning Tree

Figure 3–5 shows a graph consisting of five vertices and eight edges. Figure 3–6 shows some of the spanning trees of this graph. A minimum spanning tree of the graph is shown in Figure 3–7 whose total weight is 50 + 80 + 60 + 70 = 260.

FIGURE 3–5 A weighted connected undirected graph.

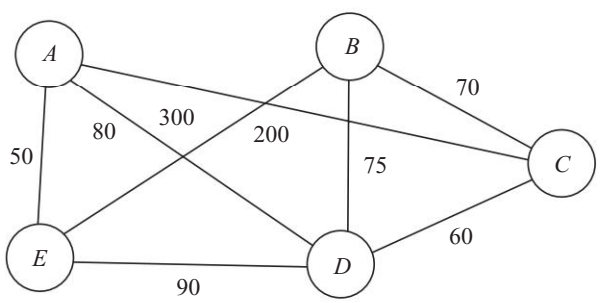


FIGURE 3–6 Some spanning trees.

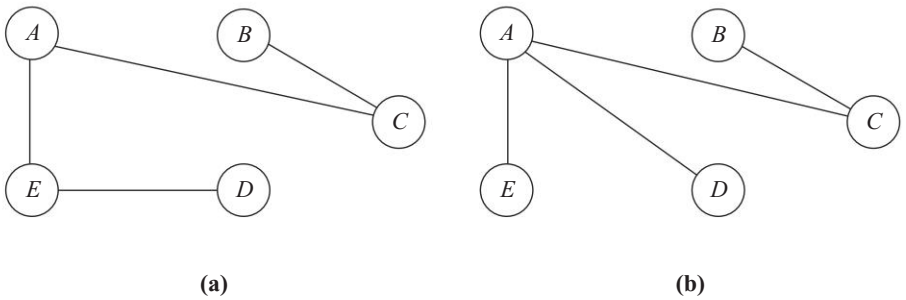
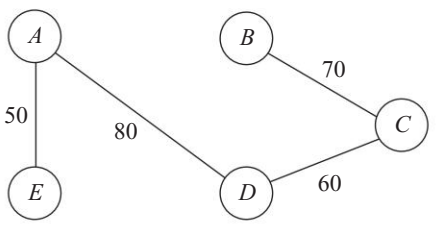


FIGURE 3–7 A minimum spanning tree.



Kruskal's method to construct a minimum spanning tree can be briefly described as follows:

(1) Select the edge with the smallest weight edge from the set of edges. This constitutes the initial partially constructed subgraph which will later be developed into a minimum spanning tree.
(2) Add the next smallest weight edge to this partially constructed graph if this will not cause a cycle to be formed. Discard the selected edge if otherwise.
(3) Terminate if the spanning tree contains $n - 1$ edges. Otherwise, go to (2).

Algorithm 3–1 □ Kruskal's minimum spanning tree algorithm

Input: A weighted, connected and undirected graph $G = (V, E)$.
Output: A minimum spanning tree for G.

$T := \phi$
While T contains less than $n - 1$ edges do
Begin
Choose an edge (v, w) from E of the smallest weight

```
        Delete (v, w) from E
        If (the adding of (v, w) to T does not create a cycle in T) then
            Add (v, w) to T
        Else
            Discard (v, w)
    End
```

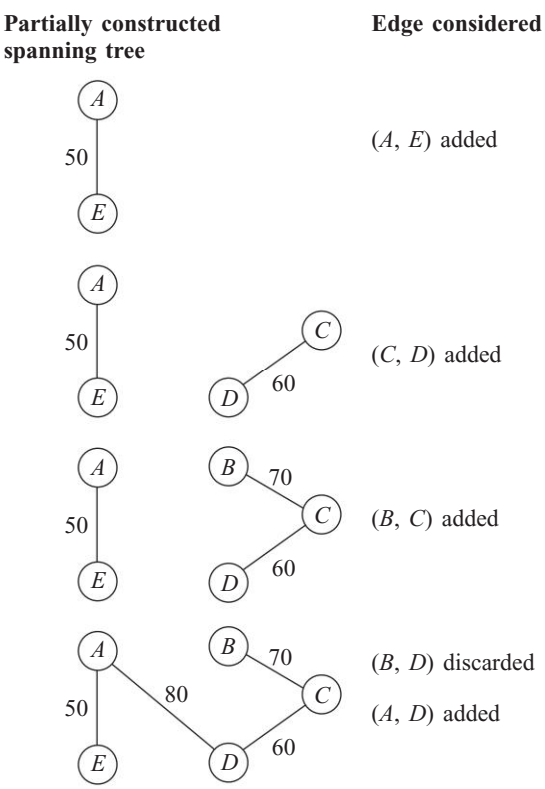
Example 3–2 Kruskal's Algorithm

Consider the graph of Figure 3–5. The edges are sorted into the following sequence:

(A, E) (C, D) (B, C) (B, D) (A, D) (E, D) (E, B) (A, C).

The minimum spanning tree is now constructed, shown in Figure 3–8.

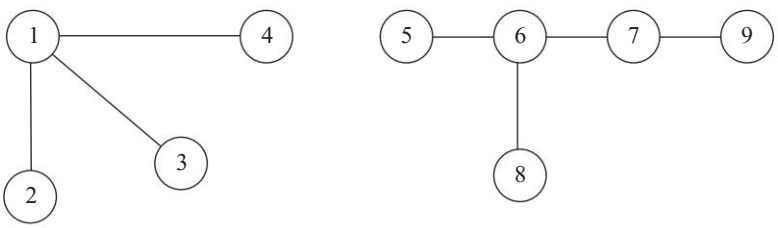
FIGURE 3–8 Finding a minimum spanning tree by Kruskal's algorithm.



The reader should note that we never have to sort the edges. In fact, the heap considered in Section 2-5 can be used to select the next edge with the smallest weight.

There is still one problem to solve: How can we determine efficiently whether or not the added edge will form a cycle? This can be found very easily. During the process of Kruskal's algorithm, the partially constructed subgraph is a spanning forest consisting of many trees. We may therefore keep each set of vertices in a tree in an individual set. Let us consider Figure 3–9. In Figure 3–9, we have two trees and they can be represented as $S_1 = \{1, 2, 3, 4\}$ and $S_2 = \{5, 6, 7, 8, 9\}$. Suppose the next edge to be added is (3, 4). Since both vertices 3 and 4 are in S_1, this will cause a cycle to be formed. Therefore, (3, 4) cannot be added. Similarly, we cannot add (8, 9) because both vertices 8 and 9 are in S_2. However, we can add (4, 8).

FIGURE 3–9 A spanning forest.



Based on the above discussion, we can see that Kruskal's algorithm is dominated by the following actions:

(1) Sorting, which takes $O(m \log m)$ steps where m is the number of edges in the graph.
(2) The union of two sets. This is necessary when we merge two trees. When we insert an edge linking two subtrees, we are essentially finding the union of two sets. For instance, suppose we add edge (4, 8) into the spanning tree in Figure 3–9, we are merging the two sets {1, 2, 3, 4} and {5, 6, 7, 8, 9} into {1, 2, 3, 4, 5, 6, 7, 8, 9}. Thus, we have to perform an operation, namely the union of two sets.
(3) The finding of an element in a set. Note that when we check whether an edge can be added or not, we must check whether two vertices are in a set of vertices or not. Thus, we have to perform an operation, called the find operation, which determines whether an element is in a set or not.

In Chapter 10, we shall show that both union and find operations will take $O(m)$ steps. Thus, the total time of Kruskal's algorithm is dominated by sorting, which is $O(m \log m)$. In the worst case, $m = n^2$. Therefore, the time complexity of Kruskal's algorithm is $O(n^2 \log n)$.

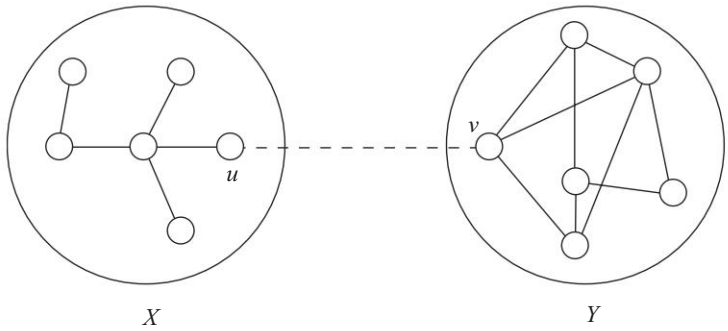
We now prove that Kruskal's algorithm is correct. That is, it produces a minimum spanning tree. Let us assume that the weights associated with all edges are distinct and $|e_1| < |e_2| < \cdots < |e_m|$ where $m = |E|$. Let T be the spanning tree produced by Kruskal's algorithm and T' be a minimum spanning tree. We shall prove that $T = T'$. Assume otherwise. Then let e_i be the minimum weight edge in T which does not appear in T'. Obviously, $i \neq 1$. Add e_i to T'. This will necessarily form a cycle in T'. Let e_j be an edge in this cycle which is not an edge in T. This e_j must exist. Otherwise, all edges in this cycle belong to T and this means that there is a cycle in T which is impossible. There are two possible cases. Case 1: e_j has a smaller weight than e_i, i.e., $j < i$. Let T_k denote the tree produced by Kruskal's algorithm after it checks whether e_k can be added or not. Clearly, T_k, for $k < i$, is a subtree of both T and T' because e_i is the smallest weight edge in T which does not appear in T', as assumed. Since $j < i$ and e_j is not an edge in T, e_j must have not been selected by Kruskal's algorithm. This must be due to the fact that adding e_j to T_{j-1} will form a cycle. However, since T_{j-1} is also a subtree of T' and e_j is an edge in T', adding e_j to T_{j-1} cannot form a cycle. Thus, this case is impossible. Case 2: e_j has a larger weight than e_i, i.e., $j > i$. In this case, remove e_j. This will create a new spanning tree whose total weight is smaller than T'. Thus, T' must be the same as T.

It is obvious that Kruskal's algorithm employs the greedy method. In every step, the next edge to be added is locally optimal. It is interesting to see that the final result is a globally-optimal one.

3–2 PRIM'S METHOD TO FIND A MINIMUM SPANNING TREE

In Section 3–1, we introduced Kruskal's algorithm to find a minimum spanning tree. In this section, we shall introduce an algorithm independently discovered by Dijkstra and Prim. Prim's algorithm builds a minimum spanning tree step by step. At any moment, let X denote the set of vertices contained in the partially constructed minimum spanning tree. Let $Y = V - X$. The next edge (u, v) to be added is an edge between X and Y ($u \in X$ and $v \in Y$) with the smallest weight. The situation is described in Figure 3–10. The next edge added will be (u, v) and after this edge is added, v will be added to X and deleted from Y. An important point in Prim's method is that we can start with any vertex, which is quite convenient.

FIGURE 3–10 An illustration of Prim's method.



Next, we shall briefly present Prim's algorithm. This algorithm will be described in detail later.

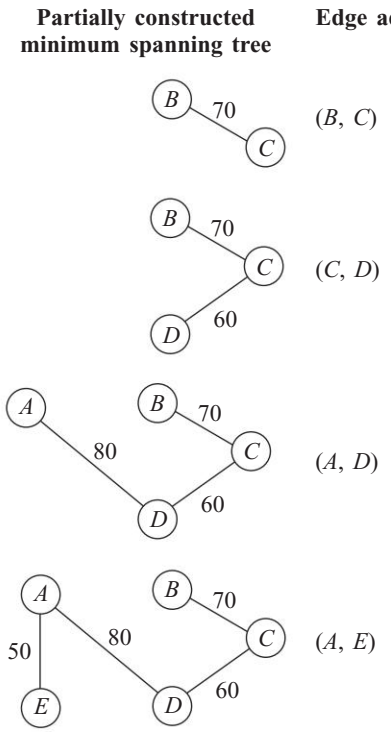
Algorithm 3–2 □ The basic Prim's algorithm to find a minimum spanning tree

Input: A weighted, connected and undirected graph $G = (V, E)$.
Output: A minimum spanning tree for G.
Step 1. Let x be any vertex in V. Let $X = \{x\}$ and $Y = V - \{x\}$.
Step 2. Select an edge (u, v) from E such that $u \in X$, $v \in Y$ and (u, v) has the smallest weight among edges between X and Y.
Step 3. Connect u to v. Let $X = X \bigcup \{v\}$ and $Y = Y - \{v\}$.
Step 4. If Y is empty, terminate and the resulting tree is a minimum spanning tree. Otherwise, Go to Step 2.

Example 3–3 Basic Prim's Algorithm

Consider Figure 3–5 again. Next, we shall show how Prim's algorithm would generate a minimum spanning tree. We shall assume that vertex B is selected as the initial point. Figure 3–11 illustrates this process. At each step of the process, the next edge to be added minimally increases the total cost. For instance, when the partially constructed tree contains vertices B, C and D, the remaining vertex set is $\{A, E\}$. The edge connecting $\{A, E\}$ and $\{B, C, D\}$ with the smallest weight is (A, D). Thus, (A, D) is added. Since vertex A is not contained in the partially constructed minimum spanning tree, this new edge will not form any cycle.

FIGURE 3–11 Finding a minimum spanning tree by basic Prim's algorithm with starting vertex *B*.



The reader will be interested in knowing whether we can start with some other vertex. This is indeed the case. Suppose we initially start with vertex *C*. Figure 3–12 illustrates the minimum spanning tree that is constructed.

FIGURE 3–12 Finding a minimum spanning tree by basic Prim's algorithm with starting vertex *C*.

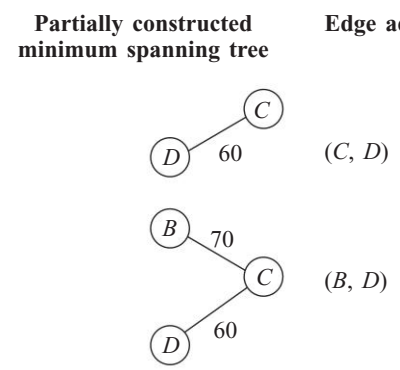
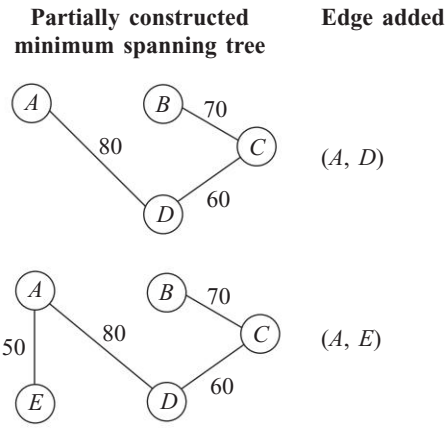
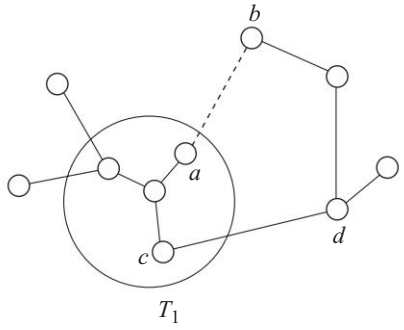


FIGURE 3–12 (*cont'd*)



We now prove that Prim's algorithm is correct. That is, the spanning tree it produces is indeed a minimum spanning tree. Let $G = (V, E)$ denote a weighted connected graph. Without losing generality, we may assume that the weights of all edges are distinct. Let T be a minimum spanning tree for G. Let T_1 denote a subtree of T, as shown in Figure 3–13. Let V_1 denote the set of vertices in T_1 and $V_2 = V - V_1$. Let (a, b) be a minimum weight edge in E such that $a \in V_1$ and $b \in V_2$. We shall show that (a, b) must be in T. Assume otherwise. Then there must be a path in T from a to b because a tree is connected. Let (c, d) be an edge in that path such that $c \in V_1$ and $d \in V_2$. The weight of (c, d) must be larger than the weight of (a, b). We may therefore create another smaller spanning tree by deleting (c, d) and adding (a, b). This shows that T must not be a minimum spanning tree. Therefore, (a, b) must be in T and Prim's algorithm is correct.

FIGURE 3–13 A minimum spanning tree to explain the correctness of Prim's algorithm.



The above algorithm is only a brief sketch of Prim's algorithm. It was presented in such a way that it would be easily understood. Let us start from the very beginning: $X = \{x\}$ and $Y = V - \{x\}$. To find the minimum weighted edge between X and Y, we must examine all the edges incident on x. In the worst case, this will take $n - 1$ steps where n is the number of vertices in V. Assume that y is added to X. That is, assume that $X = \{x, y\}$ and $Y = V - \{x, y\}$. To find the minimum weighted edge between X and Y, we seem to have one problem: Do we need to examine the edges incident on x again? (Of course, we do not need to examine the edge between x and y because they are both in X.) Prim suggested a clever way to avoid this trouble by keeping two vectors.

Let there be n vertices, labeled as $1, 2, \ldots, n$. Let there be two vectors C_1 and C_2. Let X denote the set of vertices in the partially constructed tree in Prim's algorithm and $Y = V - X$. Let i be a vertex in Y. Among all edges incident on vertices in X and vertex i in Y, let edge (i, j), $j \in X$, be the edge with the smallest weight. Vectors C_1 and C_2 are used to store this information. Let $w(i, j)$ denote the weight of edge (i, j). Then, at any step of Prim's algorithm,

$$C_1(i) = j$$

and $$C_2(i) = w(i, j).$$

We now show that these two vectors can be used to avoid repeatedly examining edges. Without losing generality, let us assume $X = \{1\}$ and $Y = \{2, 3, \ldots, n\}$ initially. Obviously, for each vertex i in Y, $C_1(i) = 1$ and $C_2(i) = w(i, 1)$ if edge $(i, 1)$ exists. The smallest $C_2(i)$ determines the next vertex to be added to X.

Again, we may assume that vertex 2 is selected as the point added to X. Thus, $X = \{1, 2\}$ and $Y = \{3, 4, \ldots, n\}$. Prim's algorithm requires the determination of the minimum weighted edge between X and Y. But, with the help of $C_1(i)$ and $C_2(i)$, we do not need to examine edges incident on vertex i any more. Suppose i is a vertex in Y. If $w(i, 2) < w(i, 1)$, we change $C_1(i)$ from 1 to 2 and $C_2(i)$ from $w(i, 1)$ to $w(i, 2)$. If $w(i, 2) \geq w(i, 1)$, do nothing. After updating is completed for all vertices in Y, we may choose a vertex to be added to X by examining $C_2(i)$. The smallest $C_2(i)$ determines the next vertex to be added. As the reader can see, we have now successfully avoided repeatedly examining all edges. Each edge is examined only once.

Prim's algorithm is given in more detail on the next page.

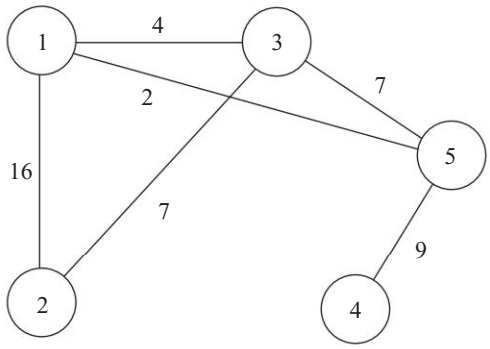
Algorithm 3–3 □ Prim's algorithm to construct a minimum spanning tree

Input: A weighted, connected and undirected graph $G = (V, E)$.
Output: A minimum spanning tree of G.
Step 1. Let $X = \{x\}$ and $Y = V - \{x\}$ where x is any vertex in V.
Step 2. Set $C_1(y_j) = x$ and $C_2(y_j) = \infty$ for every vertex y_j in V.
Step 3. For every vertex y_j in V, examine whether y_j is in Y and edge (x, y_j) exists. If y_j is in Y, edge (x, y_j) exists and $w(x, y_j) = b < C_2(y_j)$, set $C_1(y_j) = x$ and set $C_2(y_j) = b$; otherwise, do nothing.
Step 4. Let y be a vertex in Y such that $C_2(y)$ is minimum. Let $z = C_1(y)$ (z must be in X). Connect y with edge (y, z) to z in the partially constructed tree T. Let $X = X + \{y\}$ and $Y = Y - \{y\}$. Set $C_2(y) = \infty$.
Step 5. If Y is empty, terminate and the resulting tree T is a minimum spanning tree; otherwise, set $x = y$ and go to Step 3.

Example 3–4 Prim's Algorithm

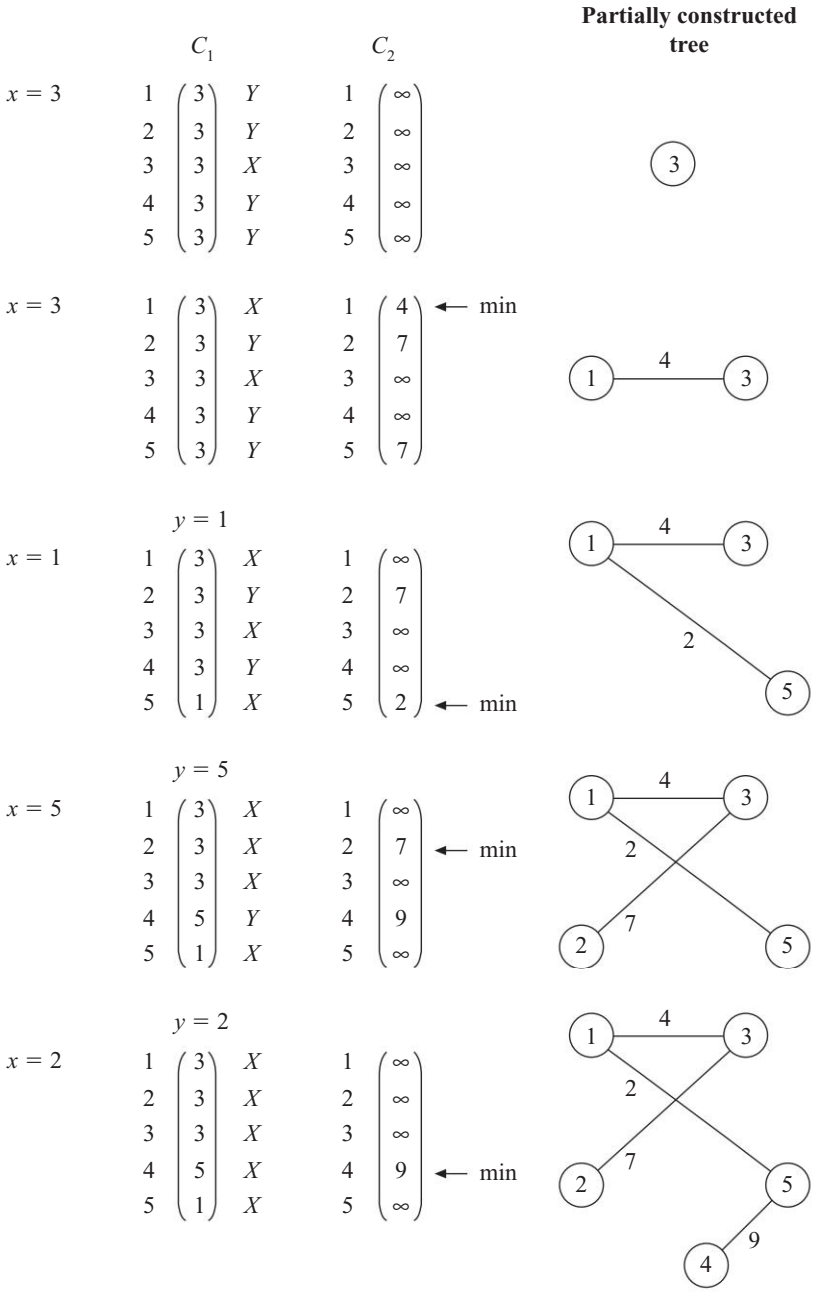
Let us consider the graph shown in Figure 3–14.

FIGURE 3–14 A graph to demonstrate Prim's algorithm.



With Figure 3–15, we shall show how Prim's algorithm works to generate a minimum spanning tree. We shall assume that vertex 3 is initially selected.

FIGURE 3–15 Finding a minimum spanning tree by basic Prim's algorithm with starting vertex 3.



For Prim's algorithm, whenever a vertex is added to the partially constructed tree, every element of C_1 must be examined. Therefore, the time complexity of Prim's algorithm, in both worst and average cases, is $O(n^2)$ where n is the number of vertices in V. Note that the time complexity of Kruskal's algorithm is $O(m \log m)$ where m is the number of edges in E. In case that m is small, Kruskal's method is preferred. In the worst case, as mentioned before, m may be equal to $O(n^2)$ and the worst-case time complexity of Kruskal's algorithm becomes $O(n^2 \log n)$, which is larger than that of Prim's algorithm.

3–3 THE SINGLE-SOURCE SHORTEST PATH PROBLEM

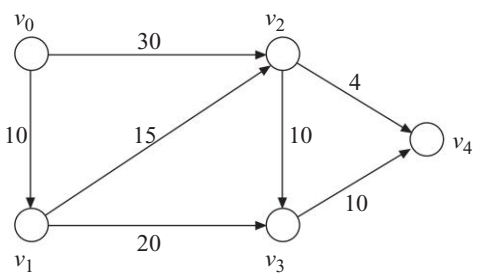
In the shortest path problem, we are given a directed graph $G = (V, E)$ in which each edge is associated with a non-negative weight. This weight can be considered as the length of this edge. The length of a path in G is defined to be the sum of lengths of the edges in this path. The single-source shortest path problem is to find all the shortest paths from a specified source vertex, denoted as v_0, to all other vertices in V.

We shall show in this section that we can solve the single-source shortest path problem by the greedy method. This was pointed out by Dijkstra and the algorithm we shall present is therefore called Dijkstra's algorithm. The spirit of Dijkstra's algorithm is quite similar to that of the minimum spanning tree algorithm presented earlier. The basic idea is rather simple: Shortest paths from v_0 to all of the other vertices are found one by one. The nearest neighbor of v_0 is first found. Then the second nearest neighbor of v_0 is found. This process is repeated until the nth nearest neighbor of v_0 is found where n is the number of vertices in the graph other than v_0.

Example 3–5 Single-Source Shortest Path Problem

Consider the directed graph in Figure 3–16. Find all the shortest paths from v_0.

FIGURE 3–16 A graph to demonstrate Dijkstra's method.



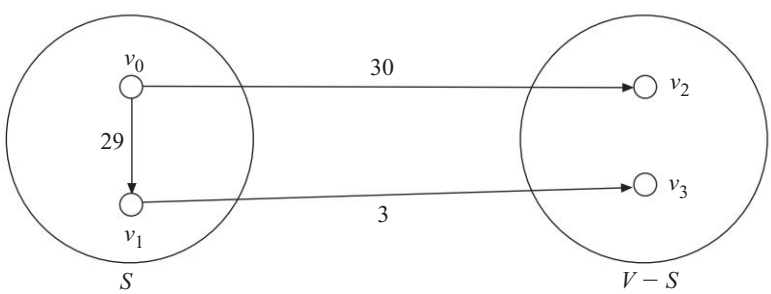
Our algorithm first determines that the nearest neighbor of v_0 is v_1. The shortest path from v_0 to v_1 is v_0v_1. Then the second nearest neighbor of v_0 is determined to be v_2 and its corresponding shortest path is $v_0v_1v_2$. Note that although this path consists of two edges, it is still shorter than the path v_0v_2, which contains only one edge. The third and the fourth nearest neighbors of v_0 will be found to be v_4 and v_3 respectively. The entire process can be illustrated by Table 3–1.

TABLE 3–1 An illustration to find the shortest paths from v_0.

i	*i*th nearest neighbor of v_0	Shortest path from v_0 to the *i*th nearest neighbor (length)
1	v_1	v_0v_1 (10)
2	v_2	$v_0v_1v_2$ (25)
3	v_4	$v_0v_1v_2v_4$ (29)
4	v_3	$v_0v_1v_3$ (30)

As in the minimum spanning tree algorithm, in Dijkstra's algorithm we also divide the set of vertices into two sets: S and $V - S$ where S contains all the i nearest neighbors of v_0 which have been found in the first i steps. Thus, in the $(i + 1)$th step, our job is to find the $(i + 1)$th nearest neighbor of v_0. At this point, it is perhaps quite important that we should not take any incorrect action. Consider Figure 3–17. In Figure 3–17, it is shown that we have already found the first nearest neighbor of v_0, which is v_1. It may appear that since v_1v_3 is the shortest link between S and $V - S$, we should choose v_1v_3 as the next edge and v_3 to be the second nearest neighbor. This is wrong because we are interested in finding shortest paths from v_0. For this case, v_2 is the second nearest neighbor, not v_3.

FIGURE 3–17 Two vertex sets S and $V - S$.



Let us explain an important trick used in Dijkstra's algorithm to find the next nearest neighbor of v_0 in an efficient way. This can be explained by considering Figure 3–16. Let us denote $L(v_i)$ as the shortest distance from v_0 to v_i presently found. At the very beginning, $S = \{v_0\}$ and we have

$$L(v_1) = 10$$
$$\text{and} \quad L(v_2) = 30$$

as v_1 and v_2 are connected to v_0.

Since $L(v_1)$ is the shortest, v_1 is the first nearest neighbor of v_0. Let $S = \{v_0, v_1\}$. Now, only v_2 and v_3 are connected to S. For v_2, its previous $L(v_2)$ was equal to 30. However, after v_1 is put into S, we may use the path $v_0v_1v_2$ whose length is $10 + 15 = 25 < 30$. Thus, so far as v_2 is concerned, its $L(v_2)'$ will be calculated as:

$$\begin{aligned} L(v_2)' &= \min\{L(v_2), L(v_1) + \text{length of } v_1v_2\} \\ &= \min\{30, 10 + 15\} \\ &= 25. \end{aligned}$$

The above discussion shows that the shortest distance from v_0 to v_2 presently found may be not short enough because of the newly-added vertex. If this situation occurs, this shortest distance must be updated.

Let u be the latest vertex added to S. Let $L(w)$ denote the presently found shortest distance from v_0 to w. Let $c(u, w)$ denote the length of the edge connecting u and w. Then $L(w)$ will need to be updated according to the following formula:

$$L(w) = \min(L(w), L(u) + c(u, w)).$$

Dijkstra's algorithm to solve the single-source shortest path problem is summarized below.

Algorithm 3–4 □ Dijkstra's algorithm to generate single-source shortest paths

Input: A directed graph $G = (V, E)$ and a source vertex v_0. For each edge $(u, v) \in E$, there is a non-negative number $c(u, v)$ associated with it. $|V| = n + 1$.

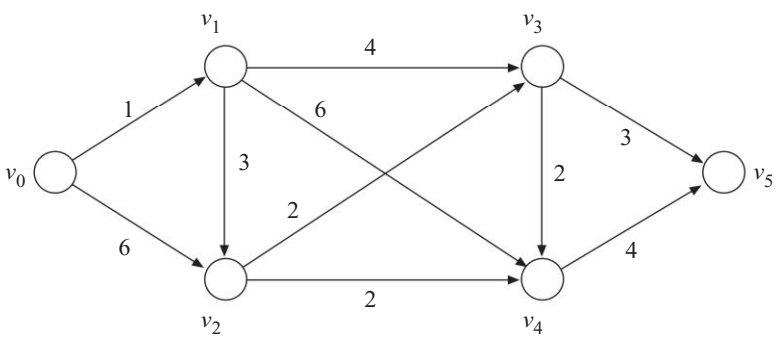
Output: For each $v \in V$, the length of a shortest path from v_0 to v.

```
S := {v0}
For i := 1 to n do
Begin
    If (v0, vi) ∈ E then
        L(vi) := c(v0, vi)
    else
        L(vi) := ∞
End
For i := 1 to n do
Begin
    Choose u from V − S such that L(u) is the smallest
    S := S ⋃ {u} (* Put u into S*)
    For all w in V − S do
        L(w) := min(L(w), L(u) + c(u, w))
End
```

Example 3–6 Dijkstra's Algorithm

Consider the directed graph shown in Figure 3–18.

FIGURE 3–18 A weighted directed graph.



The Dijkstra's algorithm will proceed as follows:

(1) $S = \{v_0\}$

$L(v_1) = 1$

$L(v_2) = 6$

All of the other $L(v_i)$'s are equal to ∞.

$L(v_1)$ is the smallest. v_0v_1 is the shortest path from v_0 to v_1.

$S = \{v_0, v_1\}$

(2) $L(v_2) = \min(6, L(v_1) + c(v_1, v_2))$

$= \min(6, 1 + 3)$

$= 4$

$L(v_3) = \min(\infty, L(v_1) + c(v_1, v_3))$

$= 1 + 4$

$= 5$

$L(v_4) = \min(\infty, L(v_1) + c(v_1, v_4))$

$= 1 + 6$

$= 7$

$L(v_2)$ is the smallest. $v_0v_1v_2$ is the shortest path from v_0 to v_2.

$S = \{v_0, v_1, v_2\}$

(3) $L(v_3) = \min(5, L(v_2) + c(v_2, v_3))$

$= \min(5, 4 + 2)$

$= 5$

$L(v_4) = \min(7, L(v_2) + c(v_2, v_4))$

$= \min(7, 4 + 2)$

$= 6$

$L(v_3)$ is the smallest. $v_0v_1v_3$ is the shortest path from v_0 to v_3.

$S = \{v_0, v_1, v_2, v_3\}$

(4) $L(v_4) = \min(6, L(v_3) + c(v_3, v_4))$

$= \min(6, 5 + 2)$

$= 6$

$L(v_5) = \min(\infty, L(v_3) + c(v_3, v_5))$

$= 5 + 3$

$= 8$

$L(v_4)$ is the smallest. $v_0v_1v_2v_4$ is the shortest path from v_0 to v_4.
$S = \{v_0, v_1, v_2, v_3, v_4\}$

(5) $L(v_5) = \min(8, L(v_4) + c(v_4, v_5))$
$= \min(8, 6 + 4)$
$= 8$

$v_0v_1v_3v_5$ is the shortest path from v_0 to v_5.
Table 3–2 summarizes the output.

TABLE 3–2 Shortest paths from vertex v_0.

Vertex	Shortest distance to v_0 (length)
v_1	v_0v_1 (1)
v_2	$v_0v_1v_2$ (1 + 3 = 4)
v_3	$v_0v_1v_3$ (1 + 4 = 5)
v_4	$v_0v_1v_2v_4$ (1 + 3 + 2 = 6)
v_5	$v_0v_1v_3v_5$ (1 + 4 + 3 = 8)

Time Complexity of Dijkstra's Algorithm

It can be easily seen that the worst case time complexity of Dijkstra's algorithm is $O(n^2)$ due to the repeated operations to calculate $L(w)$. From another point of view, the minimum number of steps to solve the single-source shortest path problem is $\Omega(e)$ where e is the number of edges in the graph because every edge must be examined. In the worst case, $\Omega(e) = \Omega(n^2)$. Therefore, in this sense, Dijkstra's algorithm is optimal.

3–4 THE 2-WAY MERGE PROBLEM

We are given two sorted lists L_1 and L_2, $L_1 = (a_1, a_2, \ldots, a_{n_1})$ and $L_2 = (b_1, b_2, \ldots, b_{n_2})$. $L_1$and L_2 can be merged into one sorted list by applying the linear merge algorithm described on the next page.

Algorithm 3–5 □ Linear merge algorithm

Input: Two sorted lists, $L_1 = (a_1, a_2, \ldots, a_{n_1})$ and $L_2 = (b_1, b_2, \ldots, b_{n_2})$.
Output: A sorted list consisting of elements in L_1 and L_2.

```
Begin
    i := 1
    j := 1
do
    compare a_i and b_j
    if a_i > b_j then output b_j and j := j + 1
    else output a_i and i := i + 1
while (i ≤ n_1 and j ≤ n_2)
    if i > n_1 then output b_j, b_{j+1}, ... , b_{n_2},
    else output a_i, a_{i+1}, ... , a_{n_1}.
End.
```

It can be easily seen that the number of comparisons required is $m + n - 1$ in the worst case. When m and n are equal, it can be shown that the number of comparisons for the linear merge algorithm is optimal. If more than two sorted lists are to be merged, we can still apply the linear merge algorithm, which merges two sorted lists, repeatedly. These merging processes are called 2-way merge because each merging step only merges two sorted lists. Suppose that we are given three sorted lists L_1, L_2 and L_3 consisting of 50, 30 and 10 elements respectively. Thus, we can merge L_1 and L_2 to obtain L_4. This merging step requires $50 + 30 - 1 = 79$ comparisons in the worst case. Then we merge L_4 and L_3 by using $80 + 10 - 1 = 89$ comparisons. The number of comparisons required in this merging sequence is 168. Alternatively, we can first merge L_2 and L_3 and then L_1. The number of comparisons required is only 128. There will be many different merging sequences and they will require different numbers of comparisons. We are now concerned with the following problem: There are m sorted lists. Each of them consists of n_i elements. What is the optimal sequence of merging process to merge these sorted lists together by using the minimum number of comparisons?

In the following, to simplify the discussion, we shall use $n + m$, instead of $n + m - 1$, as the number of comparisons needed to merge two lists with sizes n and m respectively, as this will obviously not affect our algorithm design. Let us consider an example where we have $(L_1, L_2, L_3, L_4, L_5)$ with sizes (20, 5, 8, 7, 4). Imagine that we merge these lists as follows:

Merge L_1 and L_2 to produce Z_1 with $20 + 5 = 25$ comparisons
Merge Z_1 and L_3 to produce Z_2 with $25 + 8 = 33$ comparisons
Merge Z_2 and L_4 to produce Z_3 with $33 + 7 = 40$ comparisons
Merge Z_3 and L_5 to produce Z_4 with $40 + 4 = 44$ comparisons
Total = 142 comparisons.

The merging pattern can be easily represented by a binary tree as shown in Figure 3–19(a).

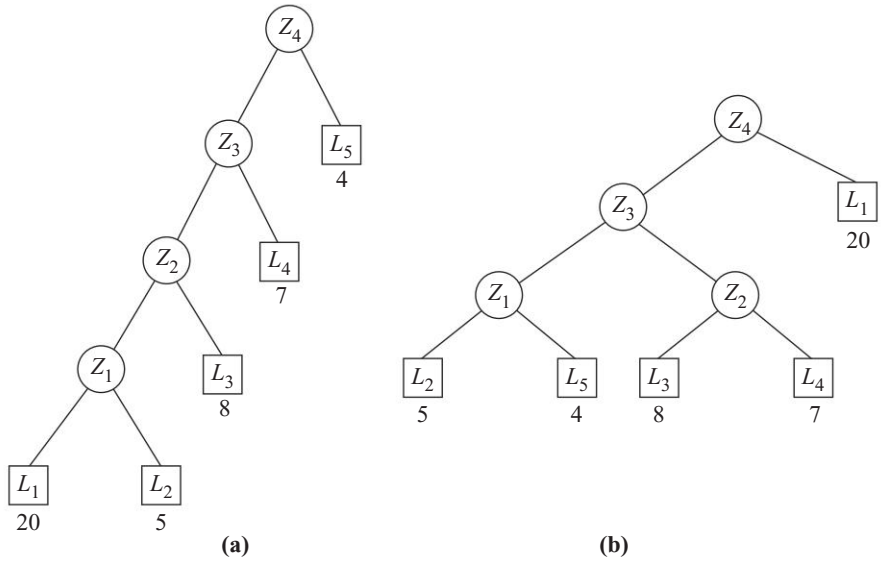
Let d_i be the depth of a leaf node of the binary tree. Let n_i be the size of the list L_i associated with this leaf node. Then the total number of comparisons corresponding to this merging process can easily be seen to be $\sum_{i=1}^{5} d_i n_i$. In our case, $d_1 = d_2 = 4$, $d_3 = 3$, $d_4 = 2$ and $d_5 = 1$. Thus, the total number of comparisons needed can be computed to be $4 \cdot 20 + 4 \cdot 5 + 3 \cdot 8 + 2 \cdot 7 + 1 \cdot 4 = 80 + 20 + 24 + 14 + 4 = 142$ which is correct.

Suppose that we use a greedy method in which we always merge two presently shortest lists. Then the merging pattern will be

Merge L_2 and L_5 to produce Z_1 with $5 + 4 = 9$ comparisons
Merge L_3 and L_4 to produce Z_2 with $8 + 7 = 15$ comparisons
Merge Z_1 and Z_2 to produce Z_3 with $9 + 15 = 24$ comparisons
Merge Z_3 and L_1 to produce Z_4 with $24 + 20 = 44$ comparisons
Total = 92 comparisons.

The above merging process is now shown as a binary tree in Figure 3–19(b).

FIGURE 3–19 Different merging sequences.



Again, the formula $\sum_{i=1}^{5} d_i n_i$ can be used. In this case, $d_1 = 1$ and $d_2 = d_3 = d_4 = d_5 = 3$. The total number of comparisons can be computed to be $1 \cdot 20 + 3 \cdot (5 + 4 + 8 + 7) = 20 + 3 \cdot 24 = 92$ which is smaller than that corresponding to Figure 3–19(a).

A greedy method to find an optimal 2-way merge tree follows:

Algorithm 3–6 □ A greedy algorithm to generate an optimal 2-way merge tree

Input: m sorted lists, L_i, $i = 1, 2, \ldots, m$, each L_i consisting of n_i elements.
Output: An optimal 2-way merge tree.
Step 1. Generate m trees, where each tree has exactly one node (external node) with weight n_i.
Step 2. Choose two trees T_1 and T_2 with minimal weights.
Step 3. Create a new tree T whose root has T_1 and T_2 as its subtrees and weight is equal to the sum of weights of T_1 and T_2.
Step 4. Replace T_1 and T_2 by T.
Step 5. If there is only one tree left, stop and return; otherwise, go to Step 2.

Example 3–7

There are six sorted lists with lengths 2, 3, 5, 7, 11 and 13. Find an extended binary tree with the minimum weighted path length for these lists.

First we merge 2 and 3, and look for the solution of the problem to merge 5 sorted lists with length 5, 5, 7, 11 and 13. And then we merge 5 and 5, and so on. The merging sequence is shown in Figure 3–20.

FIGURE 3–20 An optimal 2-way merging sequence.

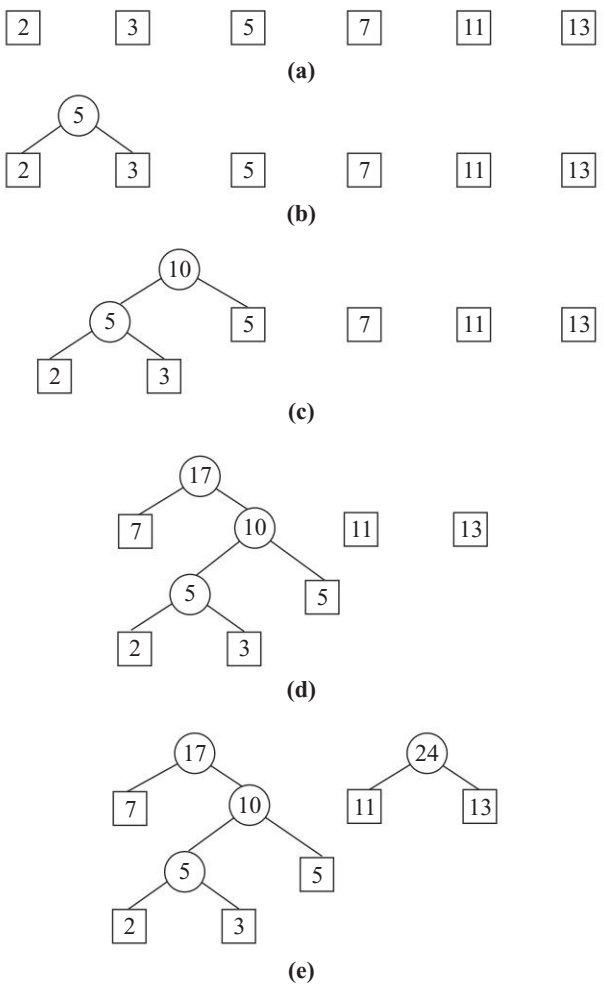
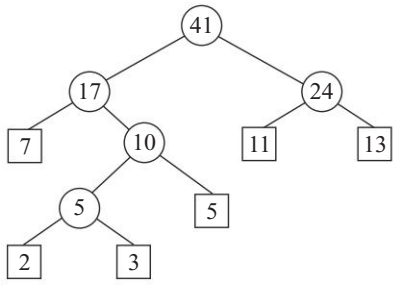


FIGURE 3–20 (*cont'd*)



(f)

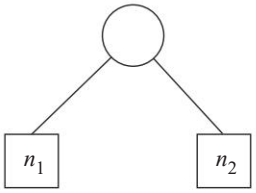
To prove the correctness of the greedy algorithm above, we first prove that there exists an optimal 2-way merge tree in which the two leaf nodes with minimum sizes are assigned to brothers. Let A be an internal node of maximum distance from the root. For instance, the node labeled with 5 in Figure 3–20 is such a node. The sons of A, say L_i and L_j, must be leaf nodes. Suppose that the sizes of the sons of A are n_i and n_j. If n_i and n_j are not the two smallest, we may interchange L_i and L_j with the two smallest nodes without increasing the weights of the 2-way merge tree. Thus, we obtain an optimal 2-way merge tree in which the two leaf nodes with the smallest sizes are assigned to be brothers.

Based on the above discussion, we may assume that T is an optimal 2-way merge tree for $L_1, L_2, \ldots, L_m$ with lengths $n_1, n_2, \ldots, n_m$ respectively and without losing generality, $n_1 \le n_2 \ldots \le n_m$, in which the two lists of the shortest lengths, namely L_1 and L_2, are brothers. Let A be the parent of L_1 and L_2. Let T_1 denote the tree where A is replaced by a list with length $n_1 + n_2$. Let $W(X)$ denote the weight of a 2-way merge tree X. Then we have

$$W(T) = W(T_1) + n_1 + n_2. \qquad \textbf{(3–1)}$$

We can now prove the correctness of our greedy algorithm by induction. It is obvious that this algorithm produces an optimal 2-way merge tree for $m = 2$. Now, assume that the algorithm produces an optimal 2-way merge tree for $m - 1$ lists. For the problem instance involving m lists $L_1, L_2, \ldots, L_m$, we combine the first two lists, namely L_1 and L_2. Then we apply our algorithm to this problem instance with $m - 1$ lists. Let the optimal 2-way merge tree produced by our algorithm be denoted as T_2. In T_2, there is a leaf node with length $n_1 + n_2$. We split this node so that it has two sons, namely L_1 and L_2 with lengths n_1 and n_2 respectively, as shown in Figure 3–21. Let this newly created tree be denoted as T_3. We have

FIGURE 3–21 A subtree.



$$W(T_3) = W(T_2) + n_1 + n_2. \tag{3–2}$$

We claim that T_3 is an optimal 2-way merge tree for $L_1, L_2, \ldots, L_m$. Suppose otherwise. Then

$$W(T_3) > W(T),$$

which implies

$$W(T_2) > W(T_1).$$

This is impossible because T_2 is an optimal 2-way merge tree for $m - 1$ lists by the induction hypothesis.

Time Complexity of the Greedy Algorithm to Generate an Optimal 2-Way Merge Tree

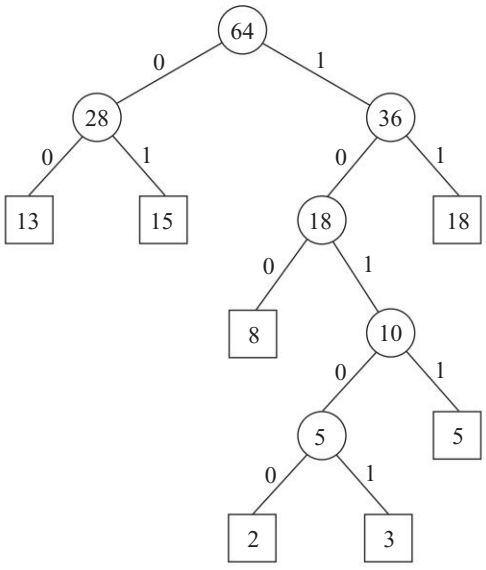
For the given m numbers $n_1, n_2, \ldots, n_m$, we can construct a min-heap to represent these numbers where the root value is smaller than the values of its sons. Then, tree reconstruction after removing the root, which has the smallest value, can be done in $O(\log n)$ time. And the insertion of a new node into a min-heap also can be done in $O(\log n)$ time. Since the main loop is executed $n - 1$ times, the total time to generate an optimal extended binary tree is $O(n \log n)$.

Example 3–8 Huffman Codes

Consider a problem in telecommunication in which we want to represent a set of messages by a sequence of 0's and 1's. Therefore, to send a message, we simply transmit a string of 0's and 1's. An application of the extended binary tree with the optimal weighted external path length is to generate an optimal set of codes, that is, binary strings, for these messages. Let us assume that there are seven

messages whose access frequencies are 2, 3, 5, 8, 13, 15 and 18. To minimize the transmission and decoding costs, we may use short strings to represent frequently used messages. Then we can construct an optimal extended binary tree by merging 2 and 3 first and then 5 and 5, etc. The resulting extended binary tree is shown in Figure 3–22.

FIGURE 3–22 A Huffman code tree.



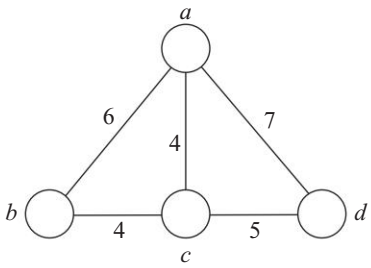
Then the codes corresponding to the messages with frequencies 2, 3, 5, 8, 13, 15 and 18 are 10100, 10101, 1011, 100, 00, 01 and 11 respectively. The frequently used messages will be coded by short codes.

3–5 THE MINIMUM CYCLE BASIS PROBLEM SOLVED BY THE GREEDY ALGORITHM

In this section, we shall introduce the minimum cycle basis problem and also show how this problem can be solved by the greedy algorithm.

Consider Figure 3–23. In the undirected graph shown in Figure 3–23, there are three cycles: namely $\{ab, bc, ca\}$, $\{ac, cd, da\}$ and $\{ab, bc, cd, da\}$.

FIGURE 3–23 A graph consisting of cycles.



By a proper operation, we may combine two cycles into another cycle. This operation is the ring sum operation defined as follows: Let A and B be two cycles. Then $C = A \oplus B = (A \bigcup B) - (A \bigcap B)$ may be a cycle. For the cycles in the above case, let

$$A_1 = \{ab, bc, ca\}$$
$$A_2 = \{ac, cd, da\}$$
and $$A_3 = \{ab, bc, cd, da\}.$$

It can be easily shown that

$$A_3 = A_1 \oplus A_2$$
$$A_2 = A_1 \oplus A_3$$
and $$A_1 = A_2 \oplus A_3.$$

This example shows that $\{A_1, A_2\}$, $\{A_1, A_3\}$ and $\{A_2, A_3\}$ can all be considered as cycle basis for the graph in Figure 3–23 because for each cycle basis, all cycles of this graph can be generated by using it. Formally, *a cycle basis of a graph is a set of cycles such that every cycle in the graph can be generated by applying ring sum operation on some cycles of this basis*.

We assume that each edge is associated with a weight. The weight of a cycle is the total weight of all edges in this cycle. *The weight of a cycle basis is the total weight of all cycles in the cycle basis*. The weighted cycle basis problem is defined as follows: *Given a graph, find a minimum cycle basis of this graph*. For the graph in Figure 3–23, the minimum cycle basis is $\{A_1, A_2\}$; it has the smallest weight.

Our greedy algorithm to solve the minimum cycle basis problem is based on the following concepts:

(1) We can determine the size of the minimum cycle basis. Let this be denoted by K.
(2) Assume that we can find all the cycles. (Finding all cycles in a graph is by no means easy. But it is irrelevant to this problem and we shall not discuss that technique here.) Sort all the cycles according to their weights into a non-decreasing sequence.
(3) From the sorted sequence of cycles, add cycles to the cycle basis one by one. For each cycle added, check if it is a linear combination of some cycles already existing in the partially constructed cycle basis. If it is, delete this cycle.
(4) Stop the process if the cycle basis has K cycles.

Our greedy method is a typical greedy approach. Suppose we want to find a set of objects minimizing some parameter. Sometimes, we can determine the minimum size K of this set and then use the greedy method to add objects one by one to this set until the size of the set becomes K. In the next section, we shall show another example with the same greedy algorithm approach.

Let us now come back to our original problem. We shall first show that we can determine the size of a minimum cycle basis. We shall not give a formal proof. Instead, we shall informally illustrate the concept through an example. Consider Figure 3–24. Suppose that we construct a spanning tree out of this graph as shown in Figure 3–25(a). This spanning tree has no cycle. Yet, if we add an edge to the spanning tree, as shown in Figure 3–25(b), a cycle will be formed.

In fact, as shown in Figure 3–25, each addition of an edge creates a new independent cycle. The number of independent cycles is equal to the number of edges which can be added to the spanning tree. Since the number of edges in a spanning tree is $|V| - 1$, the total number of edges which can be added is equal to

$$|E| - (|V| - 1) = |E| - |V| + 1.$$

FIGURE 3–24 A graph illustrating the dimension of a minimum cycle basis.

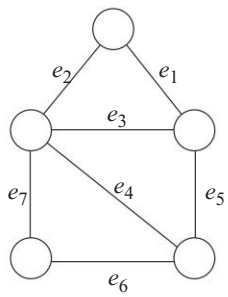
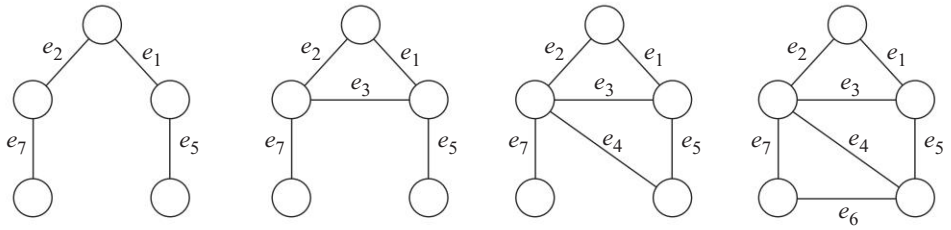


FIGURE 3–25 The relationship between a spanning tree and the cycles.

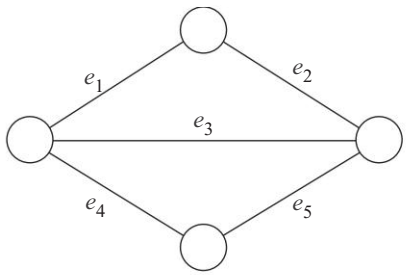


We have shown the formula to find the size of a minimum cycle basis. The only thing left for us to do is to show how to determine whether a cycle is a linear combination of a set of cycles. But we do not intend to describe a formal method. We only describe the techniques through an example. This action can be done by representing the cycles with an incidence matrix. Each row corresponds to a cycle and each column corresponds to an edge. The dependence, or independence, checking can be done by Gaussian elimination except each operation involving two rows is a ring sum (exclusive-or) operation. We now illustrate this by an example. Consider Figure 3–26. There are three cycles: $C_1 = \{e_1, e_2, e_3\}$, $C_2 = \{e_3, e_4, e_5\}$ and $C_3 = \{e_1, e_2, e_5, e_4\}$.

The matrix representing the first two cycles is shown below:

$$\begin{array}{c} \\ C_1 \\ C_2 \end{array} \begin{array}{c} \begin{array}{ccccc} e_1 & e_2 & e_3 & e_4 & e_5 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & & \\ & & 1 & 1 & 1 \end{bmatrix} \end{array}$$

FIGURE 3–26 A graph illustrating the independence checking of cycles.



If we add C_3, the matrix becomes the following:

$$\begin{array}{c} \\ C_1 \\ C_2 \\ C_3 \end{array} \begin{array}{c} \begin{array}{ccccc} e_1 & e_2 & e_3 & e_4 & e_5 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & & \\ & & 1 & 1 & 1 \\ 1 & 1 & & 1 & 1 \end{bmatrix} \end{array}$$

The exclusive-or operation on row 1 and row 3 produces the following matrix:

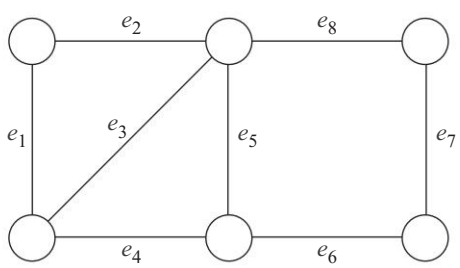
$$\begin{array}{c} \\ C_1 \\ C_2 \\ C_3 \end{array} \begin{array}{c} \begin{array}{ccccc} e_1 & e_2 & e_3 & e_4 & e_5 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & & \\ & & 1 & 1 & 1 \\ & & 1 & 1 & 1 \end{bmatrix} \end{array}$$

The exclusive-or operation on C_2 and C_3 produces an empty row which shows that C_3 is a linear combination of C_1 and C_2.

We now complete our discussion by considering Figure 3–27. We assume that every edge has weight 1. There are six cycles shown below:

$$C_1 = \{e_1, e_2, e_3\}$$
$$C_2 = \{e_3, e_5, e_4\}$$
$$C_3 = \{e_2, e_5, e_4, e_1\}$$
$$C_4 = \{e_8, e_7, e_6, e_5\}$$
$$C_5 = \{e_3, e_8, e_7, e_6, e_4\}$$
$$C_6 = \{e_2, e_8, e_7, e_6, e_4, e_1\}.$$

FIGURE 3–27 A graph illustrating the process of finding a minimum cycle basis.



For this case, $|E| = 8$ and $|V| = 6$. Thus, $K = 8 - 6 + 1 = 3$. This greedy method will work as follows:

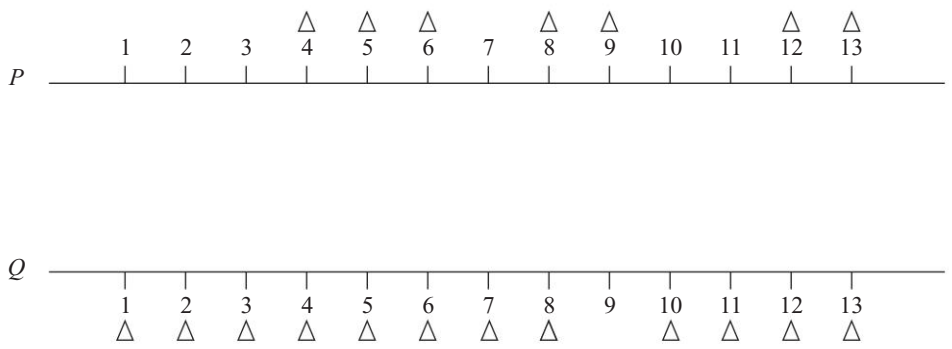
(1) C_1 is added.
(2) C_2 is added.
(3) C_3 is added and discarded because it is found that C_3 is a linear combination of C_1 and C_2.
(4) C_4 is added. Since now the cycle basis already has three cycles, we stop as $K = 3$. A minimum cycle basis is found to be $\{C_1, C_2, C_4\}$.

3–6 THE 2-TERMINAL ONE TO ANY PROBLEM SOLVED BY THE GREEDY METHOD

In VLSI design, there is a channel routing problem. There are many versions of this problem and in Chapter 5, we shall also introduce a channel routing problem which can be solved by the A^* algorithm approach. The channel routing problem which we shall discuss in this section is a much simplified version of the general channel routing problem. That it is called the 2-terminal one to any problem will be made clear in the next paragraph.

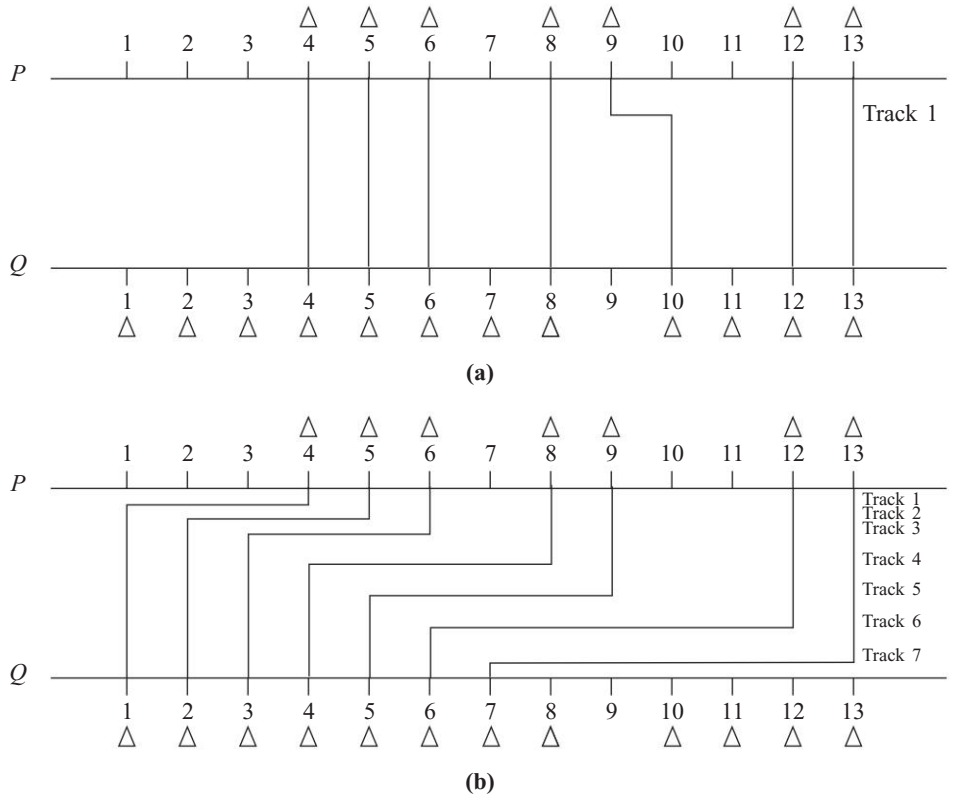
Consider Figure 3–28, where some terminals are marked. Each marked upper row terminal must be connected to a marked lower row terminal in a one-to-one fashion. It is required that no two lines can intersect. Besides, all the lines are either vertical or horizontal. Every horizontal line corresponds to a track.

FIGURE 3–28 A 2-terminal one to any problem instance.



There may be several solutions to the same problem. In Figure 3–29, two feasible solutions for the problem instance of Figure 3–28 are shown. It can be seen that the solution in Figure 3–29(a) uses fewer tracks than that in Figure 3–28(b).

FIGURE 3–29 Two feasible solutions for the problem instance in Figure 3–24.

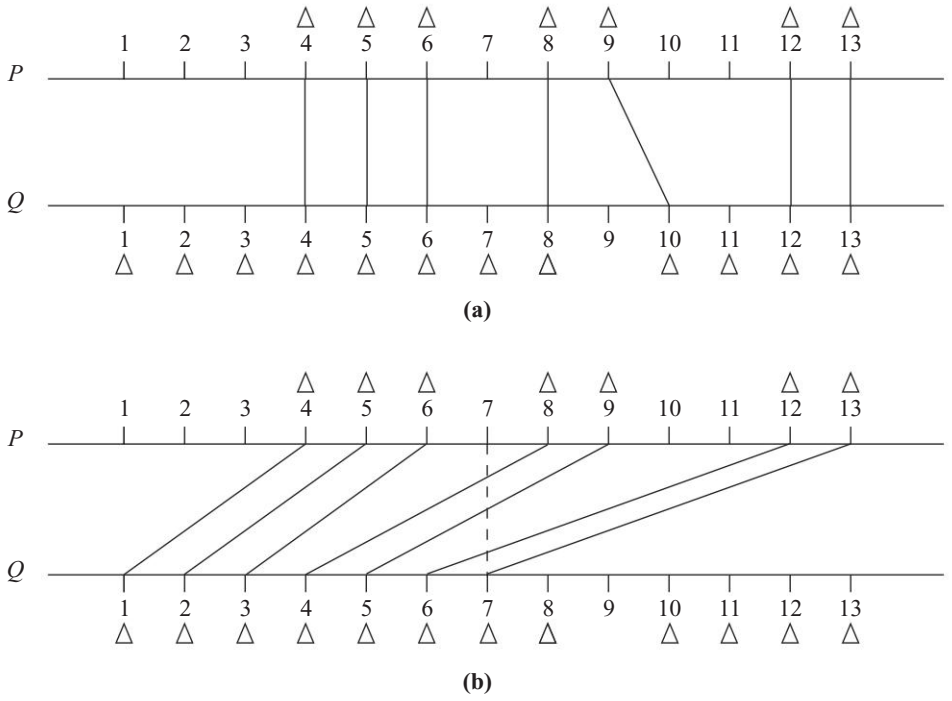


For a real channel routing problem, of course we would like to minimize the number of tracks. In this simplified version, we only try to minimize the density of a solution.

To explain the meaning of density, we redraw the two solutions (Figure 3–30) in Figure 3–29. We now imagine that we use a vertical line to scan from left to right. This vertical line will intersect with the lines of the solution. At each point, the local density of the solution is the number of lines the vertical line intersects.

For instance, as shown in Figure 3–30(b), the vertical line intersects four lines at terminal 7. Thus, the local density at terminal 7 of the solution in Figure 3–30(b) is 4. The density of a solution is the maximum local density. It can be easily seen that the densities of solutions in Figure 3–30(a) and Figure 3–30(b) are 1 and 4 respectively.

FIGURE 3–30 The redrawn Figure 3–29.



Why do we use density as the parameter which we try to minimize? The answer is simple: This simplifies the problem. If we use the number of tracks as the parameter, the problem becomes much more difficult to solve. On the other hand, the density is a lower bound of the number of tracks. Therefore, we can use it as an indication of how good our solution is.

To find a solution with the minimum density, we use the same trick as in the last section when we tried to solve the minimum cycle basis problem. That is, we have a method to determine the minimum density of a problem instance. Once this is determined, we can use the greedy method to find a solution with this minimum density. We shall not discuss how we determine this minimum density as it is very complicated and is irrelevant to the discussion of the greedy method. The most important part of our discussion is to show that our greedy method always finds such a solution with minimum density.

We now illustrate how our greedy algorithm works. We are given a problem instance where we have already determined the minimum density for this problem instance. For the problem instance in Figure 3–28, the minimum density is 1.

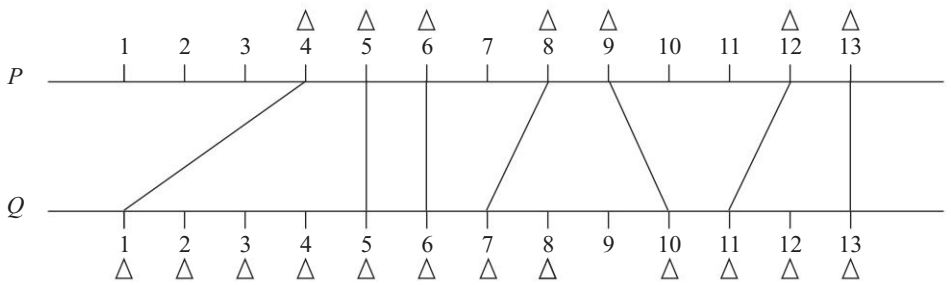
Let the upper row terminals be labeled as $P_1, P_2, \ldots, P_n$ and let the lower row terminals which can be connected be denoted as $Q_1, Q_2, \ldots, Q_m$, $m > n$. We further assume that the P_i's and Q_i's are all labeled from left to right. That is, if $j > i$, then $P_j(Q_j)$ is to the right of $P_i(Q_i)$.

Given the minimum density d and the above notations, our greedy algorithm proceeds as follows:

(1) P_1 is connected to Q_1.
(2) After P_i is connected to, say Q_j, we check whether P_{i+1} can be connected to Q_{j+1}. If the addition of the line connecting P_{i+1} and Q_{j+1} increases the density to $d + 1$, then try to connect P_{i+1} with Q_{j+2}.
(3) Repeat the above step until all P_i's are connected.

Let us now illustrate how our greedy algorithm works by applying it to the problem instance in Figure 3–28. We first decide that $d = 1$. With this information, the terminals are connected as in Figure 3–31. We note that P_5 cannot be connected to Q_2, Q_3 and Q_4 because these connections will increase the density to 2, which exceeds our minimum density. Similarly, we cannot connect P_9 to Q_8 for exactly the same reason.

FIGURE 3–31 The problem instance in Figure 3–28 solved by the greedy algorithm.



Note that our algorithm will never produce any solution where line segments connecting terminals intersect each other, as in Figure 3–32. Actually, it can be easily seen that we will never have such a connection because we can transform this connection to another connection with a smaller or equal density, as shown in Figure 3–33.

FIGURE 3–32 A crossing intersection.

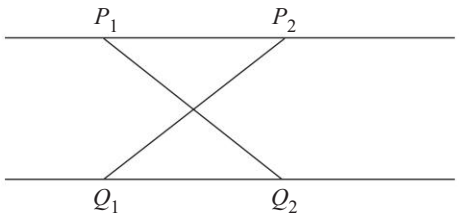
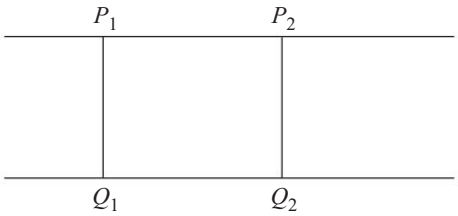


FIGURE 3–33 An interconnection transformed from the intersections in Figure 3–33.



Finally, we shall show that our greedy algorithm works. Since the minimum density is assumed to be d, there exists a feasible S_1 solution, $((P_1, Q_{j_1}), (P_2, Q_{j_2}), \ldots, (P_n, Q_{j_n}))$, with density d. We shall show that this solution can be transformed to the solution S_2, $((P_2, Q_{i_1}), (P_2, Q_{i_2}), \ldots, (P_n, Q_{i_n}))$, obtained by our greedy algorithm. We shall prove this by induction.

Let us assume that the first k connections in S_1 can be transformed to the first k connections in S_2 without violating the density d requirement. Then we shall prove that this transformation can be done for $k = k + 1$. This hypothesis is trivially true for $k = 1$. If this is true for k, then we have a partial solution $((P_1, Q_{i_1}), (P_2, Q_{i_2}), \ldots, (P_k, Q_{i_k}))$ without violating the density d requirement. Consider P_{k+1}. P_{k+1} is connected to $Q_{j_{k+1}}$ in S_1. Suppose that there is a terminal $Q_{i_{k+1}}$ to the left of $Q_{j_{k+1}}$ to which P_{k+1} can be connected without violating the minimum density requirement, then we can connect P_{k+1} with this terminal and this is what we will obtain by using our greedy algorithm. If otherwise, then our greedy algorithm will also connect P_{k+1} with $Q_{j_{k+1}}$.

The above discussion shows that any feasible solution may be transformed to a solution obtained by using our greedy algorithm, and therefore our greedy algorithm works.

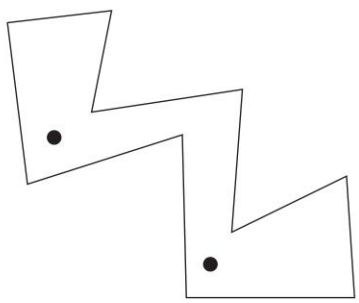
In this last example of using the greedy algorithm approach, we determine the minimum value of the parameter which we try to optimize and then use the greedy

method straightforwardly to achieve this goal. In the cycle basis problem, we add cycles one by one until the minimum size of the cycle basis is obtained. In this 2-terminal one to any problem, we first calculate the minimum density. The greedy algorithm connects terminals greedily, and at any time our algorithm ensures that the resulting density does not exceed d.

3–7 THE MINIMUM COOPERATIVE GUARDS PROBLEM FOR 1-SPIRAL POLYGONS SOLVED BY THE GREEDY METHOD

The minimum cooperative guards problem is a variation of the art gallery problem, which is defined as follows: We are given a polygon, which represents an art gallery, and we are required to place a minimum number of guards in the polygon such that every point of the polygon is visible to at least one guard. For instance, consider Figure 3–34. For this polygon, the minimum number of guards required is two. The art gallery problem is an NP-hard problem.

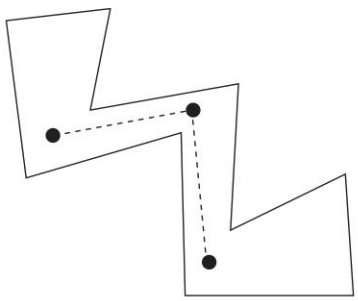
FIGURE 3–34 A solution of the art gallery problem.



The minimum cooperative guards problem puts more constraint on the art gallery problem. Note that it may be quite dangerous for a guard to be stationed in an art gallery if he cannot be seen by any other guards. We represent the relationship among guards by a visibility graph G of the guards. In G, every guard is represented by a vertex and there is an edge between two vertices if and only if the corresponding guards can see each other. In addition to finding a minimum set of guards who can see the given polygon, we further require that the visibility graph of these guards is connected. In other words, we require that no guard is isolated and there is a path between every pair of guards. We call such a problem the *minimum cooperative guards problem*.

Consider Figure 3–34 again. The visibility graph corresponding to the two guards is obviously a set of two vertices which are isolated. To satisfy the requirement of the minimum cooperative guards problem, we must add another guard, shown in Figure 3–35.

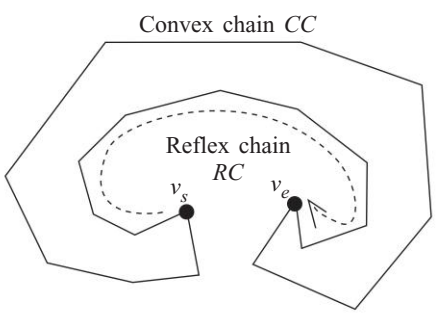
FIGURE 3–35 A solution of the minimum cooperative guards problem for the polygon in Figure 3–34.



It can again be shown that the minimum cooperative guards problem is NP-hard. Thus, it is quite unlikely that this minimum cooperative guards problem can be solved by any polynomial algorithm on general polygons. But we shall show that there is a greedy algorithm for this problem for 1-spiral polygons.

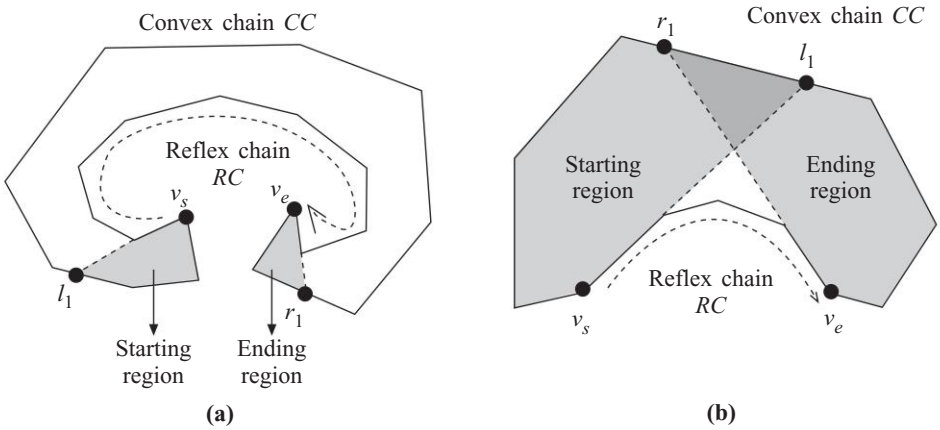
Before defining 1-spiral polygons, let us first define reflex (convex) chains. *A reflex (convex) chain, denoted as RC (CC), of a simple polygon is a chain of edges of this polygon if all of the vertices on this chain are reflex (convex) with respect to the interior of the polygon*. We also stipulate that a reflex chain refers to a *maximal* reflex chain; that is, it is not contained in any other reflex chain. A *1-spiral polygon P* is a simple polygon whose boundary can be partitioned into a reflex chain and a convex chain. Figure 3–36 shows a typical 1-spiral polygon.

FIGURE 3–36 A typical 1-spiral polygon.



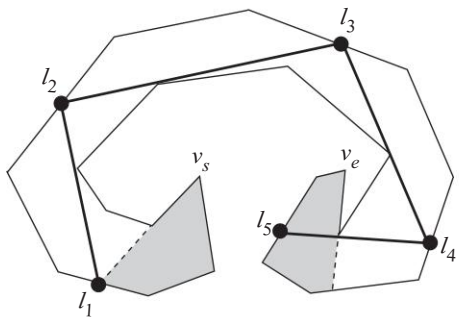
By traversing the boundary of a 1-spiral polygon counterclockwise, we call the starting (ending) vertex of the reflex chain $v_s(v_e)$. Given v_s and v_e, we can now define two regions, called the starting region and the ending region. Let us draw a line, starting from $v_s(v_e)$ along the first (last) edge of the reflex chain until it hits the boundary of the polygon at $l_1(r_1)$. This line segment $\overline{v_s l_1}(\overline{v_e r_1})$ and the first (last) part of the convex chain starting from $v_s(v_e)$ form a region and we call this region the *starting (ending) region*. Figure 3–37 shows two examples. Note that the starting and ending regions may overlap and there must be a guard stationed in both the starting and ending regions.

FIGURE 3–37 The starting and ending regions in 1-spiral polygons.



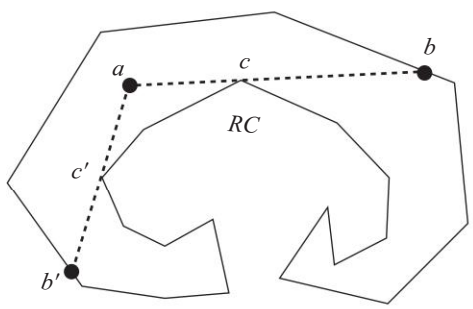
Our greedy algorithm to solve the minimum cooperative guards problem for 1-spiral polygons is as follows. We first place a guard at l_1. Then we ask: How much can this guard see? This can be answered by drawing a tangent line of *RC* starting from l_1 until it hits a point l_2 on *CC*. We place a guard at l_2. This is quite reasonable. Note that if there is no guard to the left of $\overline{l_1 l_2}$, then the resulting visibility graph will not be connected. We repeat this process until we reach the ending region. Figure 3–38 shows a typical case. In Figure 3–38, the guards stationed at l_1, l_2, l_3, l_4 and l_5 constitute an optimal solution for the minimum cooperative guards problem.

FIGURE 3–38 A set of guards $\{l_1, l_2, l_3, l_4, l_5\}$ in a 1-spiral polygon.



Before giving the formal algorithm, we must define some new terms. Consider Figure 3–39. Let a be a point in a 1-spiral polygon P. We may draw two tangents with respect to RC from a. If the exterior of RC lies entirely on the right-hand (left-hand) side of a tangent drawing from a, we call it the *left* (*right*) *tangent* of a with respect to RC. Let us draw the left (right) tangent of a with respect to RC until it hits the boundary of P at $b(b')$. Then, $\overline{ab}(\overline{ab'})$ is called the *left* (*right*) *supporting line segment* with respect to a. This is illustrated in Figure 3–39. If point a is in the starting (ending) region, we define the right (left) supporting line segment with respect to a as $\overline{av_s}(\overline{av_e})$.

FIGURE 3–39 The left and right supporting line segments with respect to a.



Next we introduce an algorithm to solve the minimum cooperative guards problem for 1-spiral polygons.

Algorithm 3–7 □ An algorithm to solve the minimum cooperative guards problem for 1-spiral polygons

Input: A 1-spiral polygon P.

Output: A set of points which is the solution of the minimum cooperative guards problem.

Step 1. Find the reflex chain RC and convex chain CC of P.

Step 2. Find the intersection points l_1 and r_1 of CC with the directed lines starting from v_s and v_e along the first and last edges of RC, respectively.

Step 3. Let $k = 1$.

While l_k is not on the ending region do

Draw the left tangent of l_k with respect to RC until it hits CC at l_{k+1}. ($\overline{l_k l_{k+1}}$ is a left supporting line segment with respect to l_k.)

Let $k = k + 1$.

End While.

Step 4. Report $\{l_1, l_2, \ldots, l_k\}$.

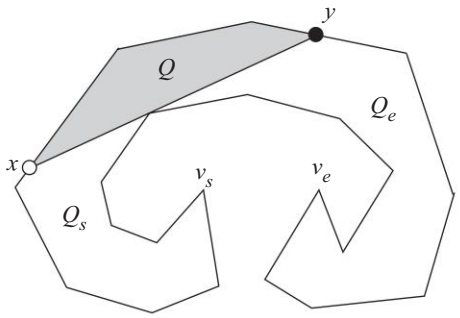
The correctness of Algorithm 3–7 will now be established. We first introduce some notations and terms. A subchain of the boundary of polygon P from point a to point b in counterclockwise order is denoted as $C[a, b]$.

Assume that A is a set of cooperative guards which can see the entire simple polygon P and the visibility graph of A is connected, then we shall call A a feasible solution of the minimum cooperative guards problem for P. Note that A is not necessarily minimum.

Let points x and y be on the convex chain of P and $\overline{xy}$ be a supporting line segment. $\overline{xy}$ partitions P into three subpolygons, Q, Q_s and Q_e. Q is the subpolygon bounded by $C[y, x]$ and $\overline{xy}$, but exclusive of x or y. Q_s and Q_e are the other two subpolygons containing v_s and v_e, respectively. See Figure 3–40 for illustration.

Assume that A is a feasible solution of the minimum cooperative guards problem for P. Then, there must be at least one guard of A placed inside Q; otherwise, the visibility graph of A will not be connected because any two guards stationed inside Q_s and Q_e, respectively, will not be visible to each other.

FIGURE 3–40 A supporting line segment $\overline{xy}$ and the regions Q, Q_s and Q_e.



Let P be a 1-spiral polygon and $\{l_1, l_2, \ldots, l_k\}$ be the set of points resulting from Algorithm 3–7 applied on P. Let L_i be the region bounded by $C[l_i, l_{i-1}]$ and $\overline{l_{i-1}l_i}$, but exclusive of l_{i-1}, for $1 \le i \le k$. Let A be any feasible solution of the minimum cooperative guards problem for P. Then, there is at least one guard of A stationed in each L_i, for $1 \le i \le k$. Besides, it is clear that L_i's are disjoint. Let N be the size of A, then, $N \ge k$. This means that k is the minimum number of guards in any feasible solution of the minimum cooperative guards problem for P.

After establishing the minimality of $\{l_1, l_2, \ldots, l_k\}$, we shall now establish the visibility of $\{l_1, l_2, \ldots, l_k\}$. That is, we must prove that each point in P is visible from at least one guard in $\{l_1, l_2, \ldots, l_k\}$. It can be seen that a collection of guards see the 1-spiral polygon if and only if they can see all edges of the reflex chain. Let the supporting line segment $\overline{l_i l_{i+1}}$ contact with the reflex chain at c_i, for $1 \le i \le k-1$, and let $c_0 = v_s$ and $c_k = v_e$. It is clear that l_i can see $C[c_i, c_{i-1}]$, for $1 \le i \le k-1$. Thus, $\{l_1, l_2, \ldots, l_k\}$ can see the entire reflex chain $C[v_s, v_e]$ and therefore can see the entire 1-spiral polygon. Besides, it is clear that the visibility graph of $\{l_1, l_2, \ldots, l_k\}$ is connected according to Algorithm 3–7 itself. This means that the output of Algorithm 3–7 is a feasible solution of the minimum cooperative guards problem. Since the number of guards is minimal, we conclude that Algorithm 3–7 produces an optimal solution.

As for the time-complexity analysis of Algorithm 3–7, let the number of vertices of a 1-spiral polygon P be n. Step 1 and Step 2 can be done in $O(n)$ time by a linear scan of the boundary of P. For Step 3, we can conduct a linear scan on the reflex chain counterclockwise and on the convex chain clockwise to find all the required left supporting line segments. Step 3 can also be done in $O(n)$ time. Thus, Algorithm 3–7 is linear.

3–8 THE EXPERIMENTAL RESULTS

To show the power of the greedy method strategy, we implemented Prim's algorithm for minimum spanning trees on an IBM PC with C language. We also implemented a straightforward method which would generate all possible sets of $(n - 1)$ edges out of a graph $G = (V, E)$. If these edges form a cycle, they are ignored; otherwise, the total length of this spanning tree is calculated.

The minimum spanning tree will be found after all spanning trees have been examined. This straightforward method was also implemented by C language. Each set of data corresponds to a randomly generated graph $G = (V, E)$. Table 3–3 summarizes the experimental results. It is obvious that this minimum spanning tree problem cannot be solved by the straightforward method and Prim's algorithm is quite effective.

TABLE 3–3 Experimental results of testing the effectiveness of Prim's algorithm.

Execution time (seconds) (Average of 20 times)							
$\|V\|$	$\|E\|$	Straightforward	Prim	$\|V\|$	$\|E\|$	Straightforward	Prim
10	10	0.305	0.014	50	200	–	1.209
10	20	11.464	0.016	50	250	–	1.264
10	30	17.629	0.022	60	120	–	1.648
15	30	9738.883	0.041	60	180	–	1.868
20	40	–	0.077	60	240	–	1.978
20	60	–	0.101	60	300	–	2.033
20	80	–	0.113	70	140	–	2.527
20	100	–	0.118	70	210	–	2.857
30	60	–	0.250	70	280	–	2.967
30	90	–	0.275	70	350	–	3.132
30	120	–	0.319	80	160	–	3.791
30	150	–	0.352	80	240	–	4.176
40	80	–	0.541	80	320	–	4.341
40	120	–	0.607	80	400	–	4.560
40	160	–	0.646	90	180	–	5.275
40	200	–	0.698	90	270	–	5.714
50	100	–	1.030	90	360	–	6.154
50	150	–	1.099	90	450	–	6.264

3–9 NOTES AND REFERENCES

For more discussion of the greedy method, consult Horowitz and Sahni (1978) and Papadimitriou and Steiglitz (1982). Korte and Louasz (1984) introduced a structural framework for the greedy method.

Kruskal's algorithm and Prim's algorithm for minimum spanning trees can be found in Kruskal (1956) and Prim (1957) respectively. Dijkstra's algorithm originally appeared in Dijkstra (1959).

The greedy algorithm generating optimal merge trees first appeared in Huffman (1952). Schwartz (1964) gave an algorithm to produce the Huffman code set. The minimum cycle basis algorithm was due to Horton (1987).

The greedy method to solve the 2-terminal one to many channel assignment problem was proposed by Atallah and Hambrusch (1986). The linear algorithm to solve the minimum cooperative guards problem for 1-spiral polygon was given in Liaw and Lee (1994).

3–10 FURTHER READING MATERIALS

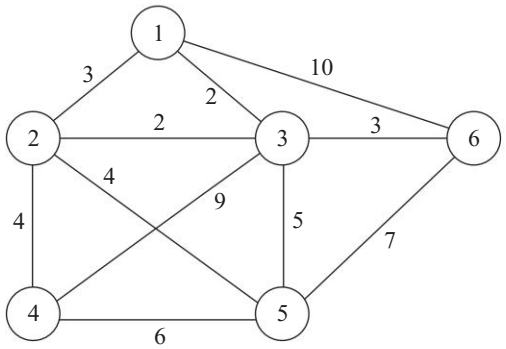
Although the greedy method concept was discovered a long time ago, there are still many interesting papers published discussing it. The following papers are highly recommended for anyone interested in doing further research in the greedy method: Bein and Brucker (1986); Bein, Brucker and Tamir (1985); Blot, Fernandez de la Vega, Paschos and Saad (1995); Coffman, Langston and Langston (1984); Cunningham (1985); El-Zahar and Rival (1985); Faigle (1985); Fernandez-Baca and Williams (1991); Frieze, McDiarmid and Reed (1990); Hoffman (1988); and Rival and Zaguia (1987).

Greedy method, of course, can be used for approximate algorithms (Chapter 9 of this book). The following papers are good reference materials: Gonzalez and Lee (1987); Levcopoulos and Lingas (1987); and Tarhio and Ukkonen (1988).

For some very interesting newly published papers, consult Akutsu, Miyano and Kuhara (2003); Ando, Fujishige and Naitoh (1995); Bekesi, Galambos, Pferschy and Woeginger (1997); Bhagavathi, Grosch and Olariu (1994); Cidon, Kutten, Mansour and Peleg (1995); Coffman, Langston and Langston (1984); Cowureur and Bresler (2000); Csur and Kao (2001); Erlebach and Jansen (1999); Gorodkin, Lyngso and Stormo (2001); Gudmundsson, Levcopoulos and Narasimhan (2002); Hashimoto and Barrera (2003); Iwamura (1993); Jorma and Ukkonen (1988); Krogh, Brown, Mian, Sjolander and Haussler (1994); Maggs and Sitaraman (1999); Petr (1996); Slavik (1997); Tarhio and Ukkonen (1986); Tarhio and Ukkonen (1988); Tomasz (1998); Tsai, Tang and Chen (1994); Tsai, Tang and Chen (1996); and Zhang, Schwartz, Wagner and Miller (2003).

Exercises

3.1 Use Kruskal's algorithm to find a minimum spanning tree of the following graph.



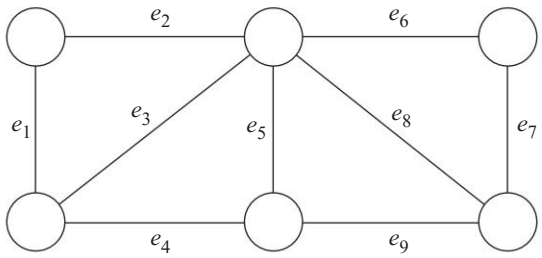
3.2 Use Prim's algorithm to find the minimum spanning tree of the graph in Problem 3.1.

3.3 Prove the correctness of Dijkstra's algorithm.

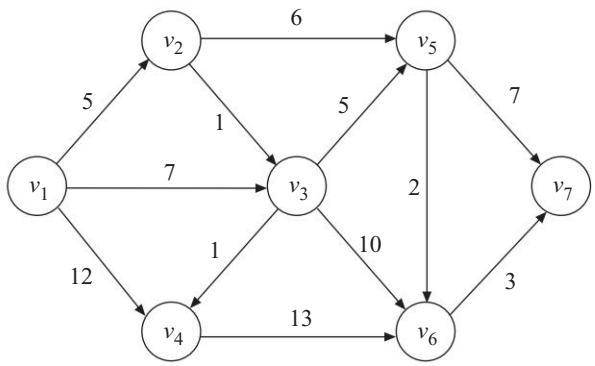
3.4 (a) Show why Dijkstra's algorithm will not work properly when the considered graph contains negative cost edges.
(b) Modify Dijkstra's algorithm so that it can compute the shortest path from source node to each node in an arbitrary graph with negative cost edges, but no negative cycles.

3.5 Obtain a set of optimal Huffman codes for the eight messages $(M_1, M_2, \ldots, M_8)$ with access frequencies $(q_1, q_2, \ldots, q_8) = (5, 10, 2, 6, 3, 7, 12, 14)$. Draw the decode tree for this set of code.

3.6 Obtain a minimum cycle basis for the following graph.



3.7 Use Dijkstra's algorithm to find the shortest paths from V_1 to all other nodes.



3.8 Give a greedy method, which is heuristic, to solve the 0/1 knapsack problem and also give an example to show that it does not always yield an optimal solution.

3.9 Implement both Prim's algorithm and Kruskal's algorithm. Perform experiments to compare the performances of these two algorithms.

3.10 Read Section 12–4 of Papadimitriou and Steiglitz (1982), for the relationship between matroids and the greedy algorithm.

3.11 The knapsack problem is defined as follows:
Given positive integers $P_1, P_2, \ldots, P_n, W_1, W_2, \ldots, W_n$ and M.
Find $X_1, X_2, \ldots, X_n, 0 \le X_i \le 1$ such that

$$\sum_{i=1}^{n} P_i X_i$$

is maximized subject to

$$\sum_{i=1}^{n} W_i X_i \le M.$$

Give a greedy method to find an optimal solution of the knapsack problem and prove its correctness.

3.12 Consider the problem of scheduling n jobs on one machine. Describe an algorithm to find a schedule such that its average completion time is minimum. Prove the correctness of your algorithm.

3.13 Sollin's algorithm was first proposed in Boruvka (1926) for finding a minimum spanning tree in a connected graph G. Initially, every vertex in G is regarded as a single-node tree and no edge is selected. In each step, we select a minimum cost edge e for each tree T such that e has exactly one vertex in T. Eliminate the copies of selected edges if necessary. The algorithm terminates if only one tree is obtained or all edges are selected. Prove the correctness of the algorithm and find the maximum number of steps of the algorithm.

chapter

4

The Divide-and-Conquer Strategy

The divide-and-conquer strategy is a powerful paradigm for designing efficient algorithms. This approach first divides a problem into two smaller subproblems and each subproblem is identical to its original problem, except its input size is smaller. Both subproblems are then solved and the subsolutions are finally merged into the final solution.

A very important point about the divide-and-conquer strategy is that it partitions the original problem into two subproblems which are identical to the original problem. Thus, these two subproblems themselves can be solved by the divide-and-conquer strategy again. Or, to put it in another way, these two subproblems are solved recursively.

Let us consider the simple problem of finding the maximum of a set S of n numbers. The divide-and-conquer strategy would solve this problem by dividing the input into two sets, each set consisting of $n/2$ numbers. Let us call these two sets S_1 and S_2. We now find the maximums of S_1 and S_2 respectively. Let the maximum of S_i be denoted as X_i, $i = 1, 2$. Then the maximum of S can be found by comparing X_1 and X_2. Whichever is the larger is the maximum of S.

In the above discussion, we casually mentioned that we have to find the maximum X_i. But, how are we going to find X_i? We may use the divide-and-conquer strategy again. That is, we divide S_i into two subsets, find the maximums of these subsets and later merge the results.

In general, a divide-and-conquer algorithm consists of three steps.

Step 1. If the problem size is small, solve this problem by some straightforward method; otherwise split the original problem into two subproblems, preferably in equal sizes.

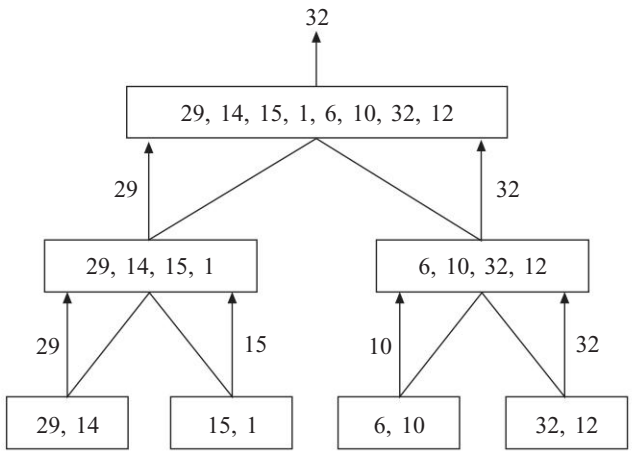
Step 2. Recursively solve these two subproblems by applying the divide-and-conquer algorithm to these subproblems.

Step 3. Merge the solutions of the subproblems into a solution of the original one.

The recursion of Step 2 creates smaller and smaller subproblems. Eventually, these subproblems will be so small that each problem can be solved directly and easily.

Let us demonstrate the meaning of divide-and-conquer by applying it to solve the problem of finding the maximum. Imagine that we have eight numbers: 29, 14, 15, 1, 6, 10, 32 and 12. The divide-and-conquer strategy finds the maximum of these eight numbers, shown in Figure 4–1.

FIGURE 4–1 Divide-and-conquer strategy to find the maximum of eight numbers.



As shown in Figure 4–1, the eight numbers are first partitioned into two subsets and each subset is further divided into subsets consisting of two numbers. As a subset contains only two numbers, we do not divide this subset any more. A simple comparison of these two numbers will determine the maximum.

After the four maximums are found, the merging starts. Thus, 29 is compared with 15 and 10 is compared with 32. Finally, 32 is found to be the maximum by comparing 32 and 29.

In general, the complexity $T(n)$ of a divide-and-conquer algorithm is determined by the following formulas:

$$T(n) = \begin{cases} 2T(n/2) + S(n) + M(n) & n \geq c \\ b & n < c \end{cases}$$

where $S(n)$ denotes the time steps needed to split the problem into two subproblems, $M(n)$ denotes the time steps needed to merge two subsolutions and b is a constant. For the problem of finding the maximum, the splitting and merging take constant number of steps. Therefore, we have

$$T(n) = \begin{cases} 2T(n/2) + 1 & n > 2 \\ 1 & n = 2. \end{cases}$$

Assuming $n = 2^k$, we have

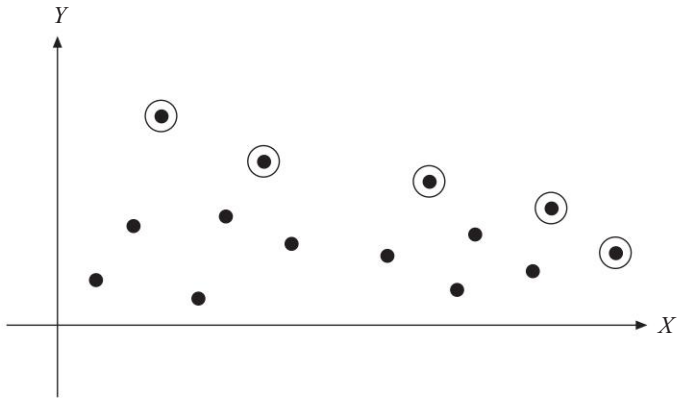
$$\begin{aligned} T(n) &= 2T(n/2) + 1 \\ &= 2(2T(n/4) + 1) + 1 \\ &\vdots \\ &= 2^{k-1}T(2) + \sum_{j=0}^{k-2} 2^j \\ &= 2^{k-1} + 2^{k-1} - 1 \\ &= 2^k - 1 \\ &= n - 1. \end{aligned}$$

The reader should note that we applied the divide-and-conquer strategy to find the maximum merely for illustrative purposes. It is obvious that in this case, the divide-and-conquer strategy is not exceedingly effective because a straightforward scanning through these n numbers and making $n - 1$ comparisons will also find the maximum and is equally efficient as the divide-and-conquer strategy. However, in the following sections, we shall show that in many cases, especially in geometry problems, the divide-and-conquer strategy is indeed a very good one.

4–1 THE 2-DIMENSIONAL MAXIMA FINDING PROBLEM

In the 2-dimensional space, we shall say that a point (x_1, y_1) dominates (x_2, y_2) if $x_1 > x_2$ and $y_1 > y_2$. A point is called a maxima if no other point dominates it. Given a set of n points, the maxima finding problem is to find all of the maximal points among these n points. For example, the circled points in Figure 4–2 are maximal points.

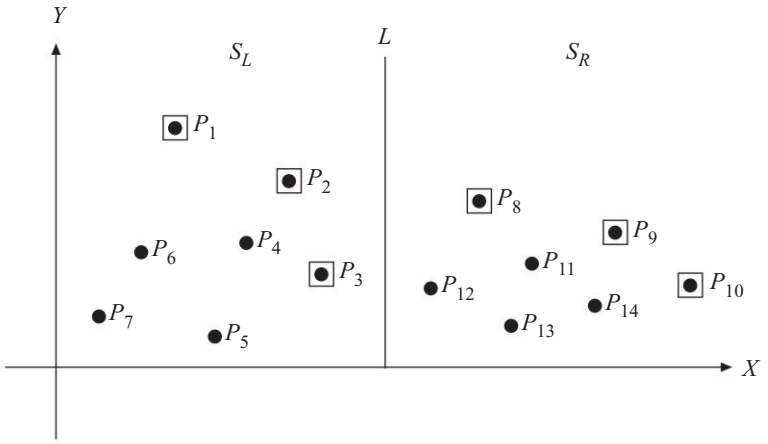
FIGURE 4–2 Maximal points.



A straightforward method to solve the maxima finding problem is to compare every pair of points. This straightforward way requires $O(n^2)$ comparisons of points. As we shall show below, if the divide-and-conquer strategy is used, we can solve the problem in $O(n \log n)$ steps, a good improvement indeed.

If the divide-and-conquer strategy is used, we first find the median line L perpendicular to the x-axis which divides the entire set of points into two subsets, as shown in Figure 4–3. Denote the set of points to the left of L and the set of points to the right of L by S_L and S_R respectively.

FIGURE 4–3 The maximal of S_L and S_R.



Our next step is to find the maximal points of S_L and S_R recursively. As shown in Figure 4–3, P_1, P_2 and P_3 are maximal points of S_L and P_8, P_9 and P_{10} are maximal points of S_R.

The merging process is rather simple. Since the x-value of a point in S_R is always larger than the x-value of every point in S_L, a point in S_L is a maxima if and only if its y-value is not less than the y-value of a maxima of S_R. For instance, for the case shown in Figure 4–3, P_3 should be discarded because it is dominated by P_8 in S_R. The set of maximal points of S are P_1, P_2, P_8, P_9 and P_{10}.

Based upon the above discussion, we may now summarize the divide-and-conquer algorithm to solve the 2-dimensional maxima finding problem as follows:

Algorithm 4–1 □ A divide-and-conquer approach to find maximal points in a plane

Input: A set of n planar points.
Output: The maximal points of S.
Step 1. If S contains only one point, return it as the maxima. Otherwise, find a line L perpendicular to the x-axis which separates the set of points into two subsets S_L and S_R, each consisting of $n/2$ points.
Step 2. Recursively find the maximal points of S_L and S_R.
Step 3. Project the maximal points of S_L and S_R onto L and sort these points according to their y-values. Conduct a linear scan on the projections and discard each of the maximal points of S_L if its y-value is less than the y-value of some maximal point of S_R.

The time complexity of this algorithm depends upon the time complexities of the following steps:

(1) In Step 1, there is an operation which finds the median on n numbers. We shall show later that this can be accomplished in $O(n)$ steps.
(2) In Step 2, we have two subproblems, each with $n/2$ points.
(3) In Step 3, the projection and the linear scanning can be accomplished in $O(n \log n)$ steps after sorting the n points according to their y-values.

Let $T(n)$ denote the time complexity of the divide-and-conquer algorithm to find maximal points of n points in plane. Then

$$T(n) = \begin{cases} 2T(n/2) + O(n) + O(n \log n) & n > 1 \\ 1 & n = 1. \end{cases}$$

Let $n = 2^k$. It is easy to prove that

$$\begin{aligned} T(n) &= O(n \log n) + O(n \log^2 n) \\ &= O(n \log^2 n). \end{aligned}$$

Thus, we have obtained an $O(n \log^2 n)$ algorithm. The reader should be reminded that if a straightforward method of examining every pair of points exhaustively is conducted, $O(n^2)$ steps are needed.

We note that our divide-and-conquer strategy is dominated by sorting in the merging steps. Somehow, we are not doing a very efficient job because sorting should be done once and for all. That is, we should conduct a presorting. If this is done, the merging complexity is $O(n)$ and the total number of time steps needed is

$$O(n \log n) + T(n)$$

where

$$T(n) = \begin{cases} 2T(n/2) + O(n) + O(n) & n > 1 \\ 1 & n = 1 \end{cases}$$

and $T(n)$ can be easily found to be $O(n \log n)$. Thus, the total time complexity of using the divide-and-conquer strategy to find maximal points with presorting is $O(n \log n)$. In other words, we have obtained an even more efficient algorithm.

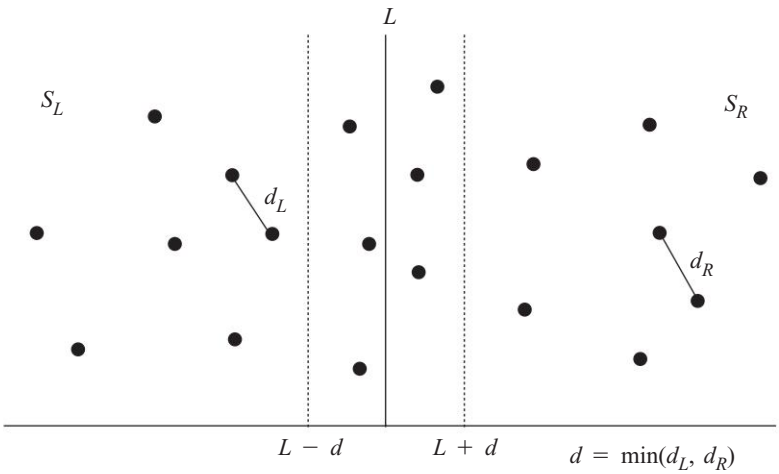
4–2 THE CLOSEST PAIR PROBLEM

The closest pair problem is defined as follows: Given a set S of n points, find a pair of points which are closest together. The 1-dimensional closest pair problem can be solved in $O(n \log n)$ steps by sorting these n numbers and conducting a linear scanning. By examining the distance between every adjacent pair of points through the sorted list, we can determine the closest pair.

In the 2-dimensional space, the divide-and-conquer strategy can be used to design an efficient algorithm to find the closest pair. As we did in solving the maxima finding problem, we first partition the set S into S_L and S_R such that

every point in S_L lies to the left of every point in S_R and the number of points in S_L is equal to that in S_R. That is, we find a vertical line L perpendicular to the x-axis such that S is cut into two equal sized subsets. Solving the closest pair problems in S_L and S_R respectively, we shall obtain d_L and d_R where d_L and d_R denote the distances of the closest pairs in S_L and S_R respectively. Let $d = \min(d_L, d_R)$. If the closest pair (P_a, P_b) of S consists of a point in S_L and a point in S_R, then P_a and P_b must lie within a slab centered at line L and bounded by lines $L - d$ and $L + d$, as shown in Figure 4–4.

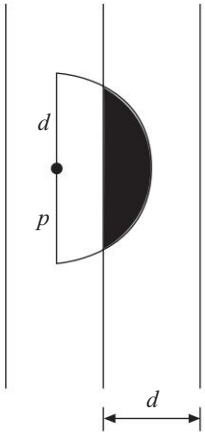
FIGURE 4–4 The closest pair problem.



The above discussion indicates that during the merging step, we may examine only points in the slab. Although on average the number of points within the slab may not be too large, in the worst case there can be as many as n points within the slab. Thus, the brute-force way to find the closest pair in the slab needs calculating $n^2/4$ distances and comparisons. This kind of merging step will not be good for our divide-and-conquer algorithm. Fortunately, as will be shown in the following, the merging step can be accomplished in $O(n)$ time.

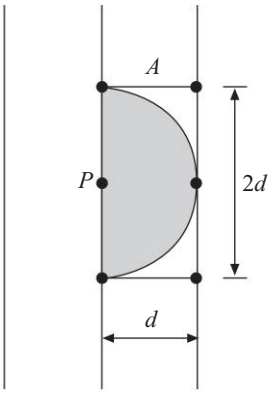
If a point P in S_L and a point Q in S_R constitute a closest pair, the distance between P and Q must be less than d. Hence, we do not have to consider a point too far away from P. Consider Figure 4–5. We only have to examine the shaded area in Figure 4–5. If P is exactly on line L, this shaded area will be the largest. Even in this case, this shaded area will be contained in the $d \times 2d$ rectangle A, shown in Figure 4–6. Thus, we only have to examine points within this rectangle A.

FIGURE 4–5 The limited region to examine in the merging process of the divide-and-conquer closest pair algorithm.



The remaining question is: How many points are there in the rectangle A? Since we know that the distance between every pair of points in both S_L and S_R is greater than or equal to d, there can be at most six points in this rectangle, shown in Figure 4–6. Based on this observation, we know that in the merging step, for each point P in the slab, we only need to examine a limited number of points in the other half of the slab. Without losing generality, we may assume that P is within the left half of the slab. Let the y-value of P be denoted as y_p. For P, we only need to examine points in the other half of the slab whose y-values are within $y_p + d$ and $y_p - d$. As discussed above, there will be at most six such points.

FIGURE 4–6 Rectangle A containing possible nearest neighbors of P.



As discussed before, we need to sort the n points according to their y-values before we apply the divide-and-conquer algorithm. After presorting, the merging step will take only $O(n)$ steps.

Next is the divide-and-conquer algorithm to solve the 2-dimensional closest pair problem.

Algorithm 4–2 □ A divide-and-conquer algorithm to solve the 2-dimensional closest pair problem

Input: A set S of n points in the plane.
Output: The distance between two closest points.
Step 1. Sort points in S according to their y-values and x-values.
Step 2. If S contains only one point, return ∞ as its distance.
Step 3. Find a median line L perpendicular to the x-axis to divide S into two equal sized subsets S_L and S_R. Every point in S_L lies to the left of L and every point in S_L lies to the right of L.
Step 4. Recursively apply Step 2 and Step 3 to solve the closest pair problems of S_L and S_R. Let $d_L(d_R)$ denote the distance between the closest pair in $S_L(S_R)$. Let $d = \min(d_L, d_R)$.
Step 5. Project all points within the slab bounded by $L - d$ and $L + d$ onto the line L. For a point P in the half slab bounded by $L - d$ and L, let its y-value be denoted as y_P. For each such P, find all points in the half slab bounded by L and $L + d$ whose y-values fall within $y_P + d$ and $y_P - d$. If the distance d' between P and a point in the other half slab is less than d, let $d = d'$. The final value of d is the answer.

The time complexity of the entire algorithm is equal to $O(n \log n) + T(n)$ and

$$T(n) = 2T(n/2) + S(n) + M(n)$$

where $S(n)$ and $M(n)$ are the time complexities of the splitting step in Step 3 and merging step in Step 4. The splitting step takes $O(n)$ steps because points are already sorted by their x-values. The merging step first has to find all of the points within $L - d$ and $L + d$. Again, because points are sorted on the x-axis, this can be accomplished in $O(n)$ steps. For each point, at most 12 other points need to be examined, 6 of them from each half slab. Thus, this linear scanning takes $O(n)$ steps. In other words, the merging step is of $O(n)$ time complexity.

Thus,

$$T(n)=\begin{cases}2T(n/2)+O(n)+O(n) & n>1\\ 1 & n=1\end{cases}$$

It can be easily proved that

$T(n) = O(n \log n)$.

The total time complexity is

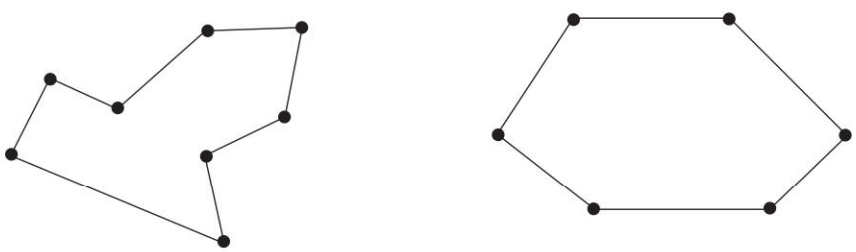
$O(n \log n)$(presorting complexity) + $O(n \log n) = O(n \log n)$.

4–3 THE CONVEX HULL PROBLEM

The convex hull problem was first mentioned in Chapter 2. In this section, we shall show that this convex hull problem can be solved elegantly by the divide-and-conquer strategy.

A convex polygon is a polygon with the property that any line segment connecting any two points inside the polygon must itself lie inside the polygon. Figure 4–7(a) shows a polygon which is not convex and Figure 4–7(b) shows a polygon which is convex.

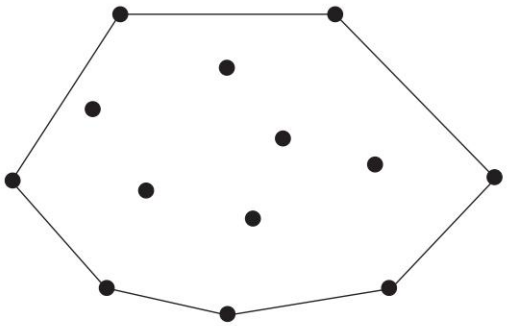
FIGURE 4–7 Concave and convex polygons.



(a) (b)

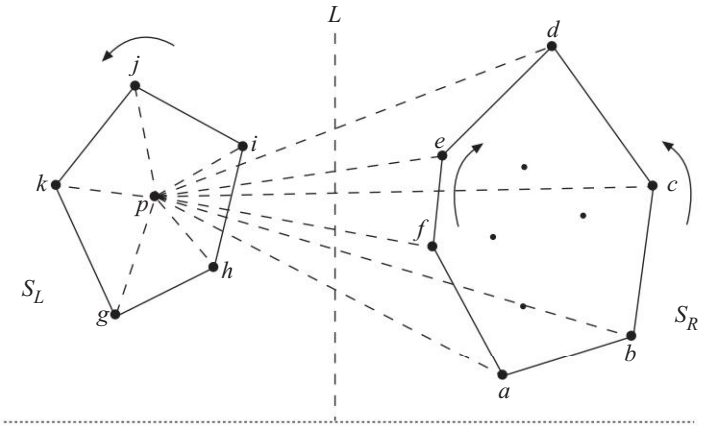
The convex hull of a set of planar points is defined as the smallest convex polygon containing all the points. For instance, Figure 4–8 shows a typical convex hull.

FIGURE 4–8 A convex hull.



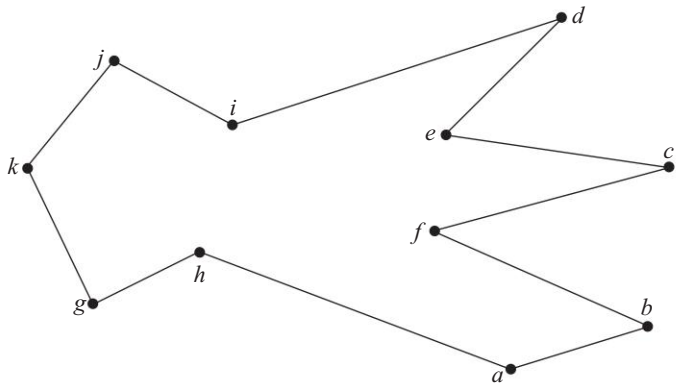
To find a convex hull, we may use the divide-and-conquer strategy. Consider Figure 4–9. In Figure 4–9, the set of planar points have been divided into two subsets S_L and S_R by a line perpendicular to the x-axis. Convex hulls for S_L and S_R are now constructed and they are denoted as Hull(S_L) and Hull(S_R) respectively. To combine Hull(S_L) and Hull(S_R) into one convex hull, we may use the Graham's scan.

FIGURE 4–9 The divide-and-conquer strategy to construct a convex hull.



To conduct the Graham's scan, an interior point is selected. Consider this point as the origin. Then each other point forms a polar angle with respect to this interior point. All the points are now sorted with respect to these polar angles. The Graham's scan examines the points one by one and eliminates the points which cause reflexive angles, as illustrated in Figure 4–10. In Figure 4–10, *h*, *f*, *e* and *i* will be eliminated. The remaining points are convex hull vertices.

FIGURE 4–10 The Graham's scan.



To conduct a Graham's scan after Hull(S_L) and Hull(S_R) have been constructed, we may select an interior point P of Hull(S_L). It should be obvious that points inside the convex hulls need not be examined. As seen from P, Hull(S_R) lies in a wedge whose apex angle is equal to or smaller than π. This wedge is defined by two vertices u and v of Hull(S_R), which can be found in linear time by the following procedure: Construct a horizontal line through P. If this line intersects Hull(S_R), then Hull(S_R) lies in the wedge determined by the two vertices of Hull(S_R) that have the greatest polar angle $< \pi/2$ and the least polar angle $> 3\pi/2$ respectively. If the horizontal line through P does not intersect Hull(S_R), the wedge is determined by the vertices that subtend the largest and smallest polar angles about P. In Figure 4–9, they are points a and d. Then there are three sequences of points which have increasing polar angles with respect to P, an interior point of S_L. These three sequences are:

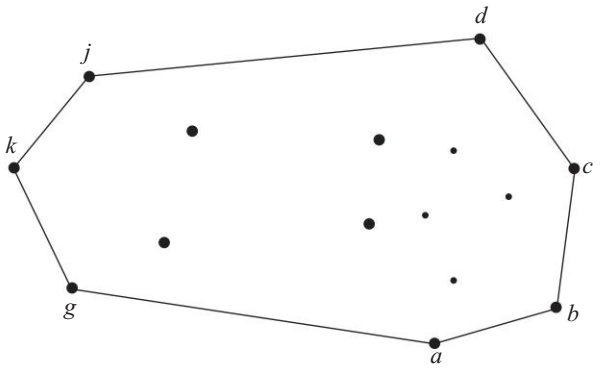
(1) g, h, i, j, k
(2) a, b, c, d
and (3) f, e.

We may merge these three sequences into one sequence. In our case, the merged sequence is

$g, h, a, b, f, c, e, d, i, j, k.$

A Graham's scan may now be applied to this sequence to eliminate points which cannot be convex hull vertices. For the case shown in Figure 4–9, the resulting convex hull is shown in Figure 4–11.

FIGURE 4–11 The convex hull for points in Figure 4–9.



The divide-and-conquer algorithm to construct a convex hull follows.

Algorithm 4–3 □ An algorithm to construct a convex hull based on the divide-and-conquer strategy

Input: A set S of planar points.

Output: A convex hull for S.

Step 1. If S contains no more than five points, use exhausive searching to find the convex hull and return.

Step 2. Find a median line perpendicular to the x-axis which divides S into S_L and S_R; S_L lies to the left of S_R.

Step 3. Recursively construct convex hulls for S_L and S_R. Denote these convex hulls by Hull(S_L) and Hull(S_R) respectively.

Step 4. Find an interior point P of S_L. Find the vertices v_1 and v_2 of Hull(S_R) that divide the vertices of Hull(S_R) into two sequences of vertices which have increasing polar angles with respect to P. Without loss of generality, let us assume that v_1 has y-value greater than v_2. Then form three sequences as follows:

(a) Sequence 1: all of the convex hull vertices of Hull(S_L) in counterclockwise direction.

(b) Sequence 2: the convex hull vertices of Hull(S_R) from v_2 to v_1 in counterclockwise direction.

(c) Sequence 3: the convex hull vertices of Hull(S_R) from v_2 to v_1 in clockwise direction.

Merge these three sequences and conduct the Graham's scan. Eliminate the points which are reflexive and the remaining points form the convex hull.

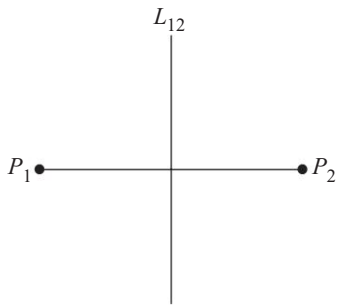
The time complexity of the above algorithm is essentially determined by the splitting process and the merging process. The time complexity of the splitting process is $O(n)$ because this is a median finding process. The time complexity of the merging process is $O(n)$ because the search for the interior point, the determination of extreme points, the merging and the Graham's scan all take $O(n)$ steps. Thus, $T(n) = 2T(n/2) + O(n) = O(n \log n)$.

4–4 THE VORONOI DIAGRAMS CONSTRUCTED BY THE DIVIDE-AND-CONQUER STRATEGY

The Voronoi diagram is, as the minimal spanning tree mentioned in Chapter 3, a very interesting data structure. This data structure can be used to store important information about the nearest neighbors of points on a plane.

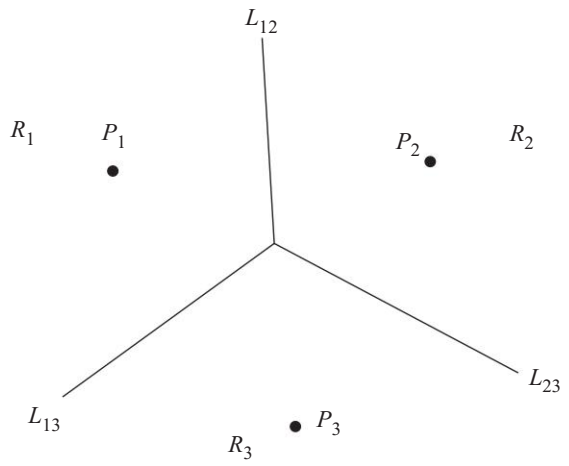
Let us consider the case of two points as shown in Figure 4–12. In Figure 4–12, line L_{12} is a perpendicular bisector of the line connecting P_1 and P_2. L_{12} is the Voronoi diagram for P_1 and P_2. For every point X located in the half plane to the left (right) of L_{12}, the nearest neighbor of X, among P_1 and P_2, is $P_1(P_2)$. In a certain sense, given any point X, to find a nearest neighbor of X, we do not have to calculate the distance between X and P_1 and the distance between X and P_2. Instead, we only have to determine which side of L_{12} is located. This can be done by substituting the coordinates of X into L_{12}. Depending upon the sign of the result of this substitution, we can determine where X is located.

FIGURE 4–12 A Voronoi diagram for two points.



Consider Figure 4–13, where each L_{ij} is a perpendicular bisector of the straight line connecting P_i and P_j. The three hyperplanes L_{12}, L_{13} and L_{23} constitute the Voronoi diagram for points P_1, P_2 and P_3. If a point is located in R_i, then its nearest neighbor among P_1, P_2 and P_3 is P_i.

FIGURE 4–13 A Voronoi diagram for three points.



To define the Voronoi diagram of a set of points in the plane, we shall first define the Voronoi polygon.

Definition

Given two points P_i and P_j in a set S of n points, let $H(P_i, P_j)$ denote the half plane containing the set of points closer to P_i. The Voronoi polygon associated with P_i is a convex polygon region having no more than $n - 1$ sides, defined by $V(i) = \bigcap_{i \neq j} H(P_i, P_j)$.

A Voronoi polygon is now shown in Figure 4–14. Given a set of n points, the Voronoi diagram comprises all the Voronoi polygons of these points. The vertices of the Voronoi diagram are called Voronoi points, and its segments are called Voronoi edges. (The name Voronoi refers to a famous Russian mathematician.) A Voronoi diagram for six points is shown in Figure 4–15.

The straight line dual of a Voronoi diagram is called the Delaunay triangulation, in honor of a famous French mathematician. There is a line

FIGURE 4–14 A Voronoi polygon.

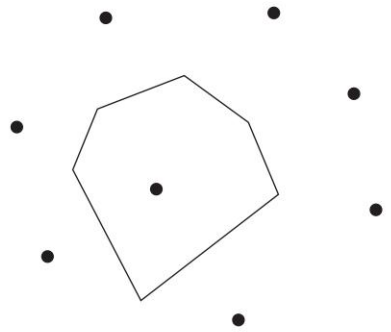
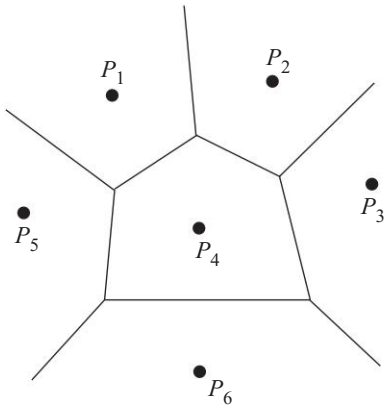


FIGURE 4–15 A Voronoi diagram for six points.

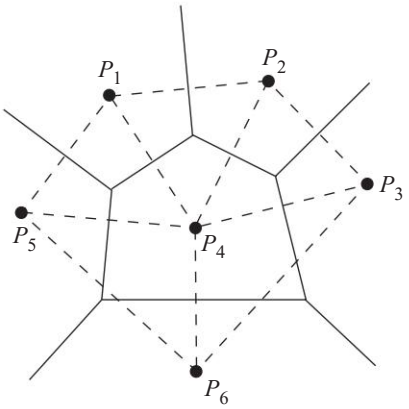


segment connecting P_i and P_j in a Delaunay triangulation if and only if the Voronoi polygons of P_i and P_j share the same edge. For instance, for the Voronoi diagram in Figure 4–15, its Delaunay triangulation is shown in Figure 4–16.

Voronoi diagrams are very useful for many purposes. For instance, we can solve the so-called all closest pairs problem by extracting information from the Voronoi diagram. A minimum spanning tree can also be found from the Voronoi diagram.

A Voronoi diagram can be constructed in the divide-and-conquer manner with the algorithm on page 135.

FIGURE 4–16 A Delaunay triangulation.



Algorithm 4–4 □ A divide-and-conquer algorithm to construct Voronoi diagrams

Input: A set S of n planar points.
Output: The Voronoi diagram of S.
Step 1. If S contains only one point, return.
Step 2. Find a median line L perpendicular to the x-axis which divides S into S_L and S_R such that $S_L(S_R)$ lies to the left (right) of L and the sizes of S_L and S_R are equal.
Step 3. Construct Voronoi diagrams of S_L and S_R recursively. Denote these Voronoi diagrams by $VD(S_L)$ and $VD(S_R)$.
Step 4. Construct a dividing piecewise linear hyperplane HP which is the locus of points simultaneously closest to a point in S_L and a point in S_R. Discard all segments of $VD(S_L)$ that lie to the right of HP and all segments of $VD(S_R)$ that lie to the left of HP. The resulting graph is the Voronoi diagram of S.

This algorithm can be understood by considering Figure 4–17. To merge $VD(S_L)$ and $VD(S_R)$, let us observe that only those parts of $VD(S_L)$ and $VD(S_R)$ close to L interfere with one another. Those points which are far away from L will not be affected by the merging step and will remain intact.

The most important step of merging $VD(S_L)$ and $VD(S_R)$ is to construct the dividing piecewise linear hyperplane HP. This hyperplane has the following property: If a point P is within the left (right) side of HP, the nearest neighbor

of P must be a point in $S_L(S_R)$. In Figure 4–17, the HP for $VD(S_L)$ and $VD(S_R)$ is shown in Figure 4–18.

FIGURE 4–17 Two Voronoi diagrams after Step 2.

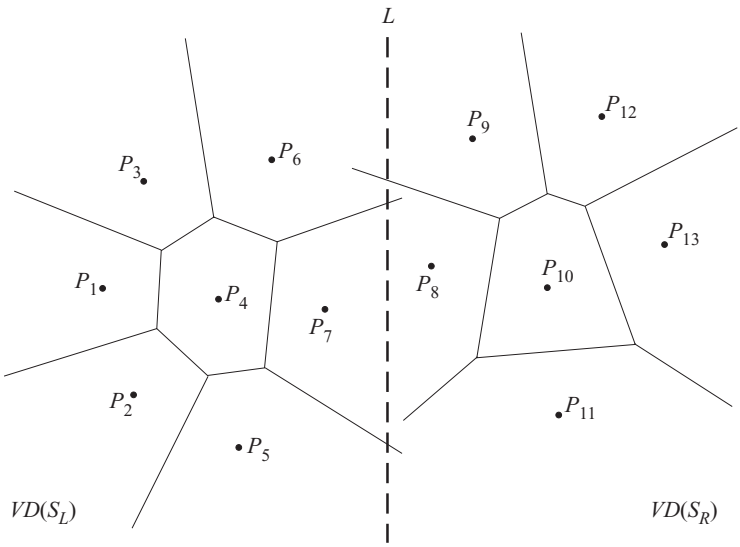
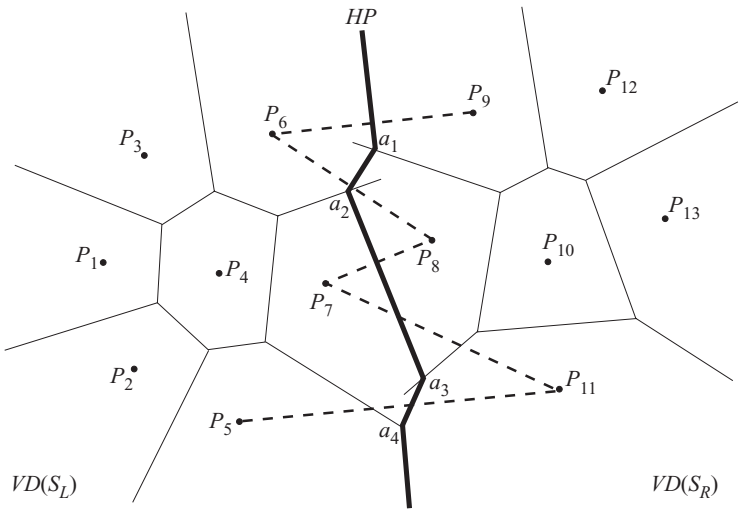
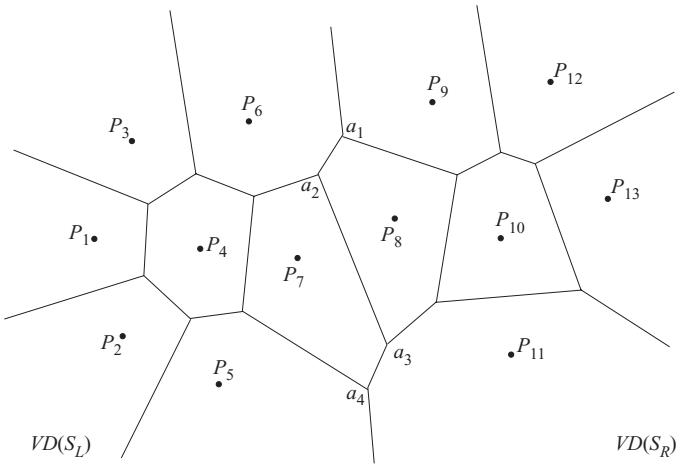


FIGURE 4–18 The piecewise linear hyperplane for the set of points shown in Figure 4–17.



After discarding all of $VD(S_L)$ to the right of HP and all of $VD(S_R)$ to the left of HP, we obtain the resulting Voronoi diagram shown in Figure 4–19.

FIGURE 4–19 The Voronoi diagram of the points in Figure 4–17.



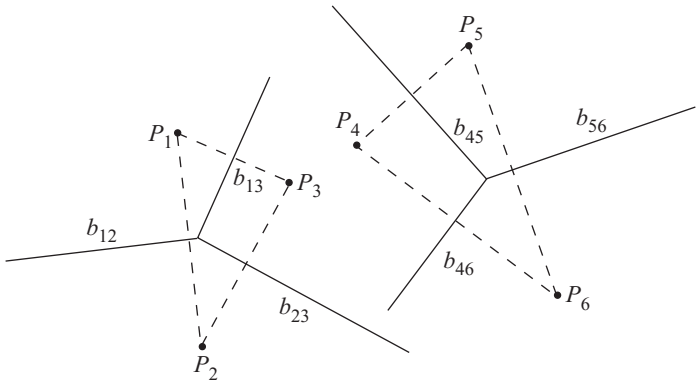
The remaining problem is how to find the HP efficiently. Let Hull(S_L) and Hull(S_R) denote the convex hulls of S_L and S_R, respectively. There are two line segments between S_L and S_R to join Hull(S_L) and Hull(S_R) into a convex hull. Let $\overline{P_aP_b}$ denote the upper segment, where P_a and P_b belong to S_L and S_R, respectively. The first step of constructing the HP is to find P_a and P_b. They are P_6 and P_9. A perpendicular bisector of line $\overline{P_6P_9}$ is now constructed. This is the starting segment of HP. Now, as shown in Figure 4–18, at a_1, HP intersects with the bisector of line $\overline{P_9P_8}$. Thus, we know that the locus will be closer to P_8 than to P_9. In other words, our next segment will be a bisector of $\overline{P_6P_8}$.

Moving downward, HP will intersect with $VD(S_L)$ at a_2. The segment of $VD(S_L)$ which intersects with HP is the bisector of $\overline{P_6P_7}$. Thus, the new segment will be the bisector of $\overline{P_7P_8}$.

Let us illustrate the process of constructing a Voronoi diagram by another example. Consider the six points in Figure 4–20.

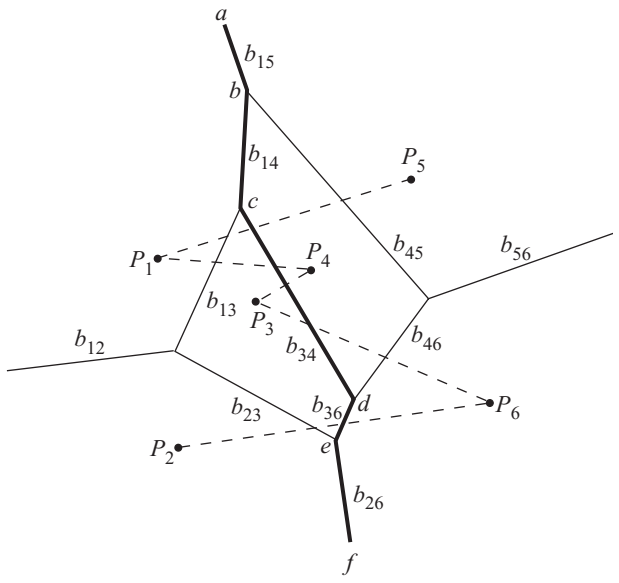
Figure 4–20 shows two Voronoi diagrams constructed for $\{P_1, P_2, P_3\}$ and $\{P_4, P_5, P_6\}$ respectively, with every ray labeled. For example, b_{13} is the perpendicular bisector of line $\overline{P_1P_3}$. The process of constructing a Voronoi diagram requires constructing *convex hulls*. The two convex hulls are shown in Figure 4–20. They are both triangles. After constructing the two convex hulls, we conclude that $\overline{P_1P_5}$ and $\overline{P_2P_6}$ are the two segments joining the two convex hulls.

FIGURE 4–20 Another case illustrating the construction of Voronoi diagrams.



Thus, our merging step starts from finding the perpendicular bisector of $\overline{P_1P_5}$, as shown in Figure 4–21.

FIGURE 4–21 The merging step of constructing a Voronoi diagram.



The entire merging step consists of the following steps:

The entire merging step consists of the following steps:

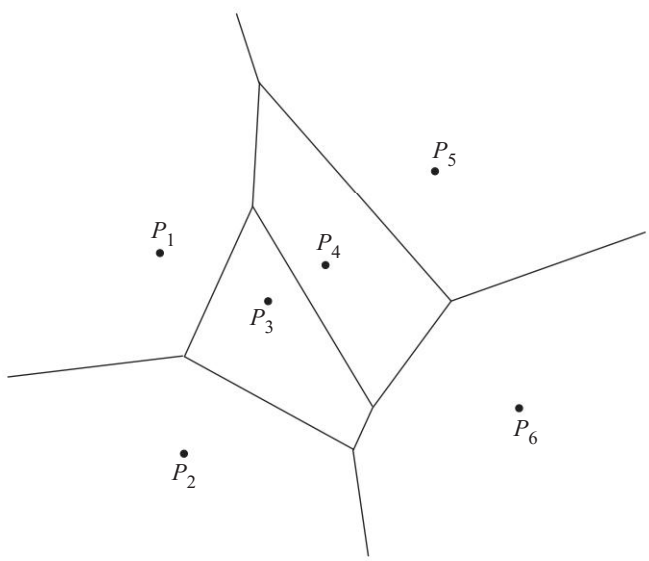
(1) The perpendicular bisector of $\overline{P_1P_5}$ intersects with b_{45} at b. We now find the perpendicular bisector of $\overline{P_1P_4}$. This line is labeled as b_{14}.
(2) b_{14} intersects with b_{13} at c. The next perpendicular bisector of $\overline{P_3P_4}$ will be b_{34}.
(3) b_{34} intersects with b_{46} at d. The next perpendicular bisector of $\overline{P_3P_6}$ will be b_{36}.
(4) b_{36} intersects with b_{23} at e. The next perpendicular bisector of $\overline{P_2P_6}$ will be b_{26}.

Since P_2 and P_6 are the lowest points of the two convex hulls, ray b_{26} extends to the infinity and the dividing piecewise linear hyperplane HP is found. Thus, HP is defined by $\overline{ab}$, $\overline{bc}$, $\overline{cd}$, $\overline{de}$ and $\overline{ef}$. If a point falls to the right (left) of this HP, its nearest neighbor must be one of $\{P_4, P_5, P_6\}(\{P_1, P_2, P_3\})$.

Figure 4–21 shows all the lines whose perpendicular bisectors constitute the dividing hyperplane. They are labeled as dashed-lines. These line segments are $\overline{P_5P_1}$, $\overline{P_1P_4}$, $\overline{P_4P_3}$, $\overline{P_3P_6}$ and $\overline{P_6P_2}$.

In the merging step, our final step is to discard all $VD(S_L)$ ($VD(S_R)$) which extend to the right (left) of HP. The resulting Voronoi diagram is shown in Figure 4–22.

FIGURE 4–22 The resulting Voronoi diagram.



We now present the algorithm to merge two Voronoi diagrams.

Algorithm 4–5 □ An algorithm which merges two Voronoi diagrams into one Voronoi diagram

Input: (a) S_L and S_R where S_L and S_R are divided by a perpendicular line L. (b) $VD(S_L)$ and $VD(S_R)$.

Output: $VD(S)$ where $S = S_L \bigcup S_R$.

Step 1. Find the convex hulls of S_L and S_R. Let them be denoted as Hull(S_L) and Hull(S_R) respectively. (A special algorithm for finding a convex hull in this case will be given later.)

Step 2. Find segments $\overline{P_aP_b}$ and $\overline{P_cP_d}$ which join Hull(S_L) and Hull(S_R) into a convex hull (P_a and P_c belong to S_L and P_b and P_d belong to S_R.) Assume that $\overline{P_aP_b}$ lies above $\overline{P_cP_d}$. Let $x = a$, $y = b$, $SG = \overline{P_xP_y}$ and $HP = \phi$.

Step 3. Find the perpendicular bisector of SG. Denote it by BS. Let $HP = HP \bigcup BS$. If $SG = \overline{P_cP_d}$, go to Step 5; otherwise, go to Step 4.

Step 4. Let BS first intersect with a ray from $VD(S_L)$ or $VD(S_R)$. This ray must be a perpendicular bisector of either $\overline{P_xP_z}$ or $\overline{P_yP_z}$ for some z. If this ray is the perpendicular bisector of $\overline{P_yP_z}$, then let $SG = \overline{P_xP_z}$; otherwise, let $SG = \overline{P_zP_y}$. Go to Step 3.

Step 5. Discard the edges of $VD(S_L)$ which extend to the right of HP and discard the edges of $VD(S_R)$ which extend to the left of HP. The resulting graph is the Voronoi diagram of $S = S_L \bigcup S_R$.

Next, we shall describe in detail the properties of HP. Before doing that, we first define the distance between a point and a set of points.

Definition

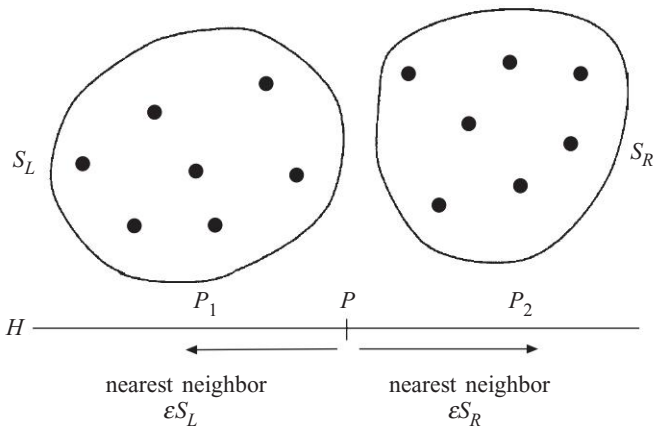
Given a point P and a set S of points $P_1, P_2, \ldots, P_n$, let P_i be a nearest neighbor of P. The distance between P and S is the distance between P and P_i.

Using this definition, we can state: The HP obtained from Algorithm 4–5 is the locus of points which keep equal distances to S_L and S_R. That is, for each point P on HP, let P_i and P_j be a nearest neighbor of S_L and S_R of P respectively. Then the distance between P and P_i is equal to that between P and P_j.

For instance, consider Figure 4–21. Let P be a point on line segment $\overline{cd}$. For S_L, the nearest neighbor of P is P_3 and for S_R, the nearest neighbor of P is P_4. Since $\overline{cd}$ is the perpendicular bisector of $\overline{P_3P_4}$, each point on $\overline{cd}$ is of equal distance to P_3 and P_4. Thus, for each point on $\overline{cd}$, it has equal distance to S_L and S_R.

Utilizing the above property of *HP*, we can easily see that *each horizontal line H intersects with HP at least once*. That each horizontal line H intersects with *HP* at least once can be seen by considering Figure 4–23.

FIGURE 4–23 The relationship between a horizontal line H and S_L and S_R.

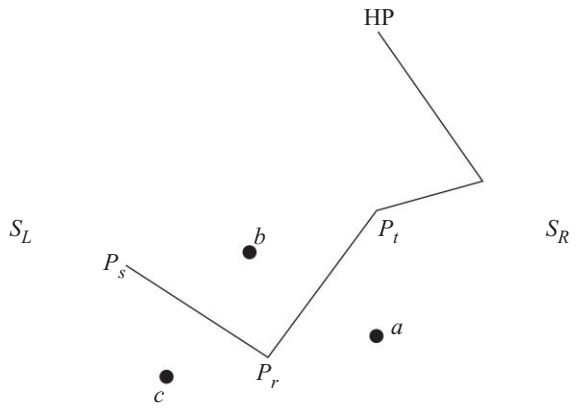


Imagine that we move along line H from left to right. At the beginning, for a point such as P_1, the nearest neighbor of P_1 must be a member of S_L. Let us also imagine that we move from right to left. For a point such as P_2, the nearest neighbor of P_2 must be a member of S_R. Since H is a contiguous line, there must be a point P such that to the left (right) of P, each point has a member of $S_L(S_R)$ as its nearest neighbor. *Thus, P is of equal distance to S_L and S_R. Or, the distance between P and its nearest neighbor in S_L must be equal to that between P and its nearest neighbor in S_R.* Because *HP* is the locus of points of equal distances to S_L and S_R, this P must be also on *HP*. Thus, we have shown that every horizontal line must intersect with *HP* at least once. The reader may consult Figure 4–21 to convince himself about the validity of the above remark.

Considering Figure 4–21, we can see that the *HP* has another interesting property; it is monotonic in y. Next, we shall prove that *each horizontal line H intersects with HP at most once*.

Suppose otherwise. Then there is a point P_r where *HP* turns upwardly, as shown in Figure 4–24.

FIGURE 4–24 An illustration of the monotonicity of HP.



In Figure 4–24, $\overline{P_tP_r}$ is the perpendicular bisector of $\overline{ab}$ and $\overline{P_rP_s}$ is the perpendicular bisector of $\overline{bc}$. Since *HP* is the locus that separates S_L and S_R, either *a* and *c* belong to S_L and *b* belongs to S_R or *a* and *c* belong to S_R and *b* belongs to S_L. Both are impossible because we have a line perpendicular to the *X*-axis to divide *S* into S_L and S_R. That is, P_r cannot exist.

Since each horizontal line *H* intersects with *HP* at least once and at most once, we conclude that *each horizontal line H intersects with HP at one and only one point. That is, HP is monotonic in y*.

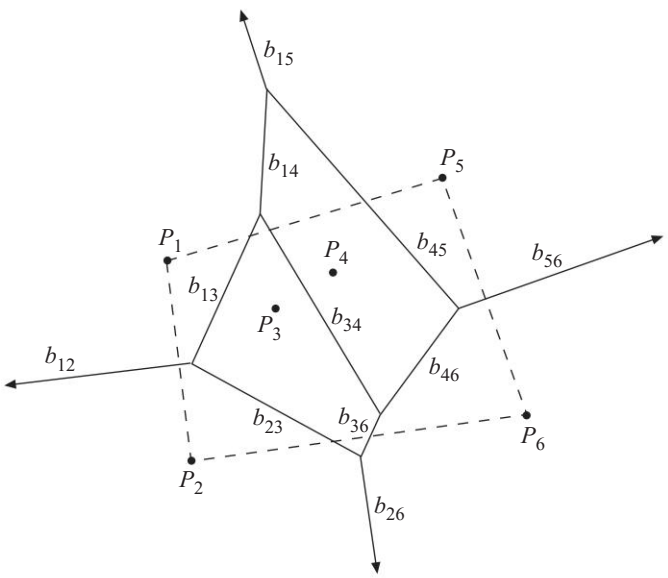
There is another important property of Voronoi diagrams which is useful for us to derive the time complexity of our algorithm to construct a Voronoi diagram. This property is: *The number of edges of a Voronoi diagram is at most 3n − 6, where n is the number of points*. Note that each Voronoi diagram corresponds to a Delaunay triangulation. Since a Delaunay triangulation is a planar graph, it can contain at most $3n - 6$ edges. Because there is a one-to-one correspondence among edges in the Voronoi diagram and edges in the Delaunay triangulation, the number of edges in a Voronoi diagram is at most $3n - 6$.

Having obtained the upper bound of Voronoi edges, we can now derive the upper bound of Voronoi vertices. Note that each Voronoi vertex corresponds to a triangle in the Delaunay triangulation. Since a Delaunay triangulation is a planar graph, we may consider a triangulation as a face of this planar graph. Let *F*, *E* and *V* denote the number of faces, edges and vertices of the Delaunay triangulation. Then, according to Euler's relation: $F = E - V + 2$. In a Delaunay triangulation, $V = n$ and *E* is at most $3n - 6$. Therefore, *F* is at most $(3n - 6) - n + 2 = 2n - 4$. That is, *the number of Voronoi vertices is at most 2n − 4*.

We are now almost ready to derive the time complexity of the merging process. Remember that in the merging process, we must find two convex hulls. The algorithm introduced in Section 4–3 cannot be used here because its time complexity is $O(n \log n)$, which is too high for our purpose. We shall now show that these convex hulls can be easily found because $VD(S_L)$ and $VD(S_R)$ can be used to construct these convex hulls.

Consider Figure 4–25 showing a Voronoi diagram with four infinite rays, namely b_{12}, b_{15}, b_{56} and b_{26}. The points associated with these infinite rays are P_2, P_1, P_5 and P_6. In fact, the convex hull can now be constructed by connecting these points, shown by the dashed-lines in Figure 4–25.

FIGURE 4–25 Constructing a convex hull from a Voronoi diagram.



Having constructed a Voronoi diagram, we may construct the convex hull by examining all the Voronoi edges until an infinite ray is found. Let P_i be the point to the left of this infinite ray. Then P_i is a convex hull vertex. We now examine the Voronoi edges of the Voronoi polygon of P_i until an infinite ray is found. The above process is repeated until we return to the starting ray. Since each edge occurs in exactly two Voronoi polygons, no edge is examined more than twice. *Consequently, after a Voronoi diagram is constructed, a convex hull can be found in O(n) time.*

The time complexity of the merging step of the algorithm to construct a Voronoi diagram based upon the divide-and-conquer strategy is now derived as follows:

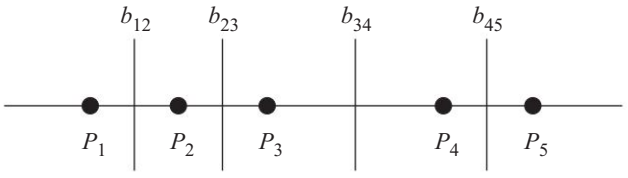
(1) The convex hulls can be found in $O(n)$ time because $VD(S_L)$ and $VD(S_R)$ have already been constructed and a convex hull can be found by finding the infinite rays.
(2) The edges joining two convex hulls to form a new convex hull can be determined in $O(n)$ time. This was explained in Section 4.3.
(3) Since there are at most $3n - 6$ edges in $VD(S_L)$ and $VD(S_R)$ and HP contains at most n segments (because of the monotonicity of HP in y), the entire construction process of HP takes at most $O(n)$ steps.

Thus, the merging process of the Voronoi diagram construction algorithm is $O(n)$. Thus,

$$\begin{aligned} T(n) &= 2T(n/2) + O(n) \\ &= O(n \log n). \end{aligned}$$

We conclude that the *time complexity of the divide-and-conquer algorithm to construct a Voronoi diagram is O(n log n)*. We now show that this is an optimal algorithm. Let us consider a set of points on a straight line. The Voronoi diagram of such a set of points consists of a set of bisecting lines as shown in Figure 4–26. After these lines have been constructed, a linear scanning of these Voronoi edges will accomplish the function of sorting. In other words, *the Voronoi diagram problem cannot be easier than the sorting problem*. A lower bound of the Voronoi diagram problem is therefore $\Omega(n \log n)$ and the algorithm is consequently optimal.

FIGURE 4–26 The Voronoi diagram for a set of points on a straight line.



4–5 APPLICATIONS OF THE VORONOI DIAGRAMS

We have discussed the concept of Voronoi diagrams and we have shown how the divide-and-conquer strategy can be used to construct a Voronoi diagram. In this section, we shall show that there are many interesting applications of Voronoi diagrams. These applications, of course, have nothing to do with the divide-and-conquer strategy. By introducing applications of Voronoi diagrams we hope this will stimulate your interest in computational geometry.

The Euclidean Nearest Neighbor Searching Problem

The first application is to solve the Euclidean nearest neighbor searching problem. The Euclidean nearest neighbor searching problem is defined as follows: *We are given a set of n planar points:* $P_1, P_2, \ldots, P_n$ *and a testing point P. Our problem is to find a nearest neighbor of P among* P_i*'s and the distance used is the Euclidean distance*.

A straightforward method is to conduct an exhaustive search. This algorithm would be an $O(n)$ algorithm. Using the Voronoi diagram, we can reduce the searching time to $O(\log n)$ with preprocessing time $O(n \log n)$.

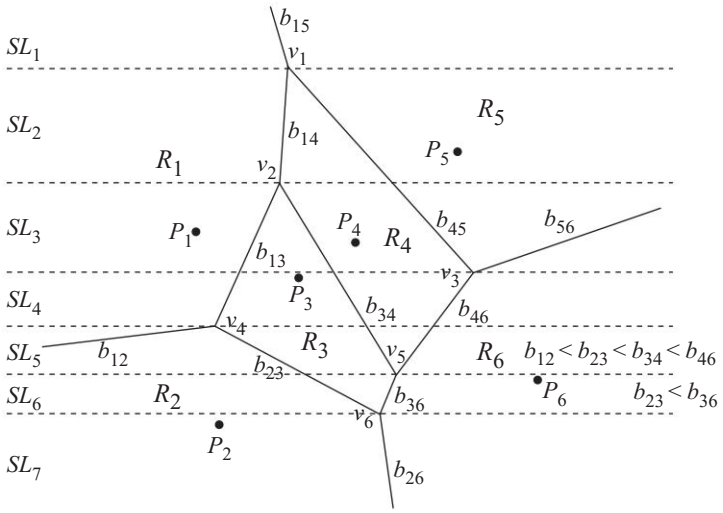
We can use the Voronoi diagram to solve the Euclidean nearest neighbor searching problem because of the fundamental properties of Voronoi diagrams. Note that the Voronoi diagram divides the entire plane into regions $R_1, R_2, \ldots, R_n$. Within each region R_i, there is a point P_i. If a testing point falls within region R_i, then its nearest neighbor, among all points, is P_i. Therefore, we may avoid an exhaustive search by simply transforming the problem into a region location problem. That is, if we can determine which region R_i a testing point is located, we can determine a nearest neighbor of this testing point.

A Voronoi diagram is a planar graph. We may therefore utilize the special properties of a planar graph as illustrated in Figure 4–27. In Figure 4–27, the Voronoi diagram in Figure 4–22 is redrawn. Note that there are six Voronoi vertices. Our first step is to sort these vertices according to their y-values. Thus, as shown in Figure 4–27, the Voronoi vertices are labeled $V_1, V_2, \ldots, V_6$ according to their decreasing y-values. For each Voronoi vertex, a horizontal line is drawn passing this vertex. These horizontal lines divide the entire space into slabs as shown in Figure 4–27.

Within each slab, there are Voronoi edges which can be linearly ordered and these Voronoi edges again divide each slab into regions. Consider SL_5 in Figure 4–27. Within SL_5, there are three Voronoi edges: b_{23}, b_{34} and b_{46}. They can be ordered as

$$b_{23} < b_{34} < b_{46}.$$

FIGURE 4–27 The application of Voronoi diagrams to solve the Euclidean nearest neighbor searching problem.



These three edges divide the slab into four regions: R_2, R_3, R_4 and R_6. If a point is found to be between edges b_{23} and b_{34}, it must be in R_3. If it is found to be to the right of b_{46}, it must be within region R_6. Since these edges are ordered, a binary search can be used to determine the location of a testing point so far as the Voronoi edges are concerned.

Our Euclidean nearest neighbor searching algorithm essentially consists of two major steps:

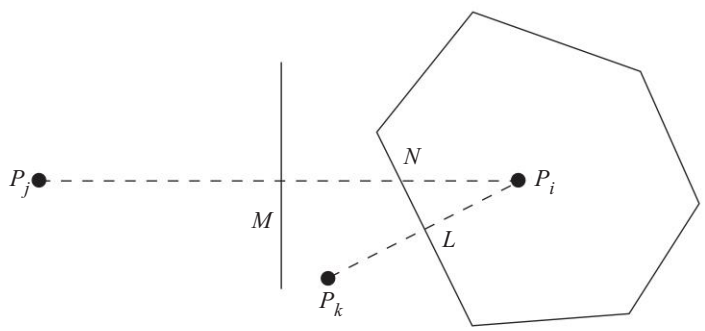
(1) Conduct a binary search to determine which slab this testing point is located. Since there are at most $O(n)$ Voronoi vertices, this can be done in $O(\log n)$ time.
(2) Within each slab, conduct a binary search to determine which region this point is located in. Since there are at most $O(n)$ Voronoi edges, this can be done in $O(\log n)$ time.

The total searching time is $O(\log n)$. It is easy to see that the preprocessing time is $O(n \log n)$, essentially the time needed to construct the Voronoi diagram.

The Euclidean All Nearest Neighbor Problem

Our next application is to solve the Euclidean all nearest neighbor problem. We are given a set of n planar points $P_1, P_2, \ldots, P_n$. The Euclidean closest pair problem is to find a nearest neighbor of every P_i. This problem can be easily solved by using the Voronoi diagram because of the following property of Voronoi diagrams. *If P_j is a nearest neighbor of P_i, then P_i and P_j share the same Voronoi edge. Moreover, the midpoint of $\overline{P_iP_j}$ is located exactly on this commonly shared Voronoi edge*. We shall show this property by contradiction. Suppose that P_i and P_j do not share the same Voronoi edge. By the definition of Voronoi polygons, the perpendicular bisector of $\overline{P_iP_j}$ must be outside of the Voronoi polygon associated with P_i. Let $\overline{P_iP_j}$ intersect the bisector at M and some Voronoi edge at N, as illustrated in Figure 4–28.

FIGURE 4–28 An illustration of the nearest neighbor property of Voronoi diagrams.



Suppose that the Voronoi edge is between P_i and P_k and $\overline{P_iP_k}$ intersects the Voronoi edge at L. Then we have

$$\overline{P_iN} < \overline{P_iM}$$

and $$\overline{P_iL} < \overline{P_iN}.$$

This means that

$$\overline{P_iP_k} = 2\overline{P_iL} < 2\overline{P_iN} < 2\overline{P_iM} = \overline{P_iP_j}.$$

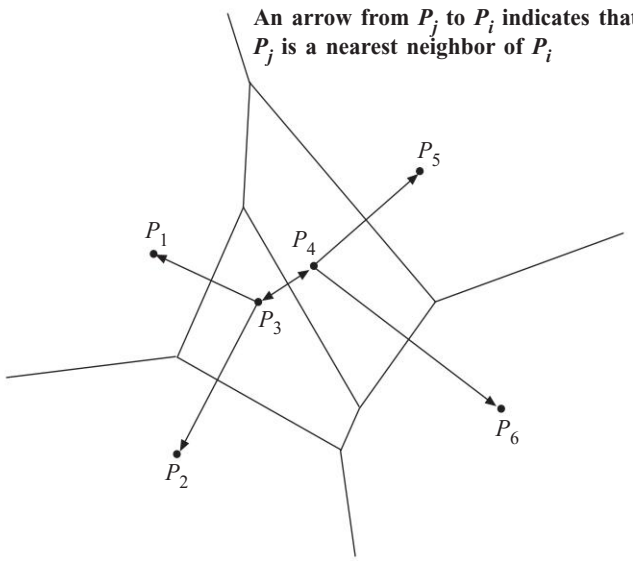
This is impossible because P_j is a nearest neighbor of P_i.

Given the above property, the Euclidean all nearest neighbor problem can be solved by examining every Voronoi edge of each Voronoi polygon. Since each

Voronoi edge is shared by exactly two Voronoi polygons, no Voronoi edge is examined more than twice. That is, this Euclidean all nearest neighbor problem can be solved in linear time after the Voronoi diagram is constructed. Thus, this problem can be solved in $O(n \log n)$ time.

Let us redraw Figure 4–22 in Figure 4–29. For P_4, four edges will have to be examined and the nearest neighbor of P_4 will be found to be P_3.

FIGURE 4–29 The all nearest neighbor relationship.



4–6 THE FAST FOURIER TRANSFORM

The Fast Fourier Transform problem is to calculate the following:

$$A_j = \sum_{k=0}^{n-1} a_k e^{\frac{2\pi ijk}{n}}, 0 \le j \le n-1$$

where $i = \sqrt{-1}$ and $a_0, a_1, \ldots, a_{n-1}$ are given numbers. A straightforward approach would require $O(n^2)$ steps. If divide-and-conquer strategy is used, we can reduce the time complexity to $O(n \log n)$.

To simplify our discussion, let $w_n = e^{\frac{2\pi i}{n}}$. Thus, the Fourier Transform is to calculate the following:

$$A_j = a_0 + a_1 w_n^j + a_2 w_n^{2j} + \cdots + a_{n-1} w_n^{(n-1)j}.$$

Let $n = 4$. Then, we have

$$\begin{aligned}
A_0 &= a_0 + a_1 + a_2 + a_3 \\
A_1 &= a_0 + a_1 w_4 + a_2 w_4^2 + a_3 w_4^3 \\
A_2 &= a_0 + a_1 w_4^2 + a_2 w_4^4 + a_3 w_4^6 \\
A_3 &= a_0 + a_1 w_4^3 + a_2 w_4^6 + a_3 w_4^9.
\end{aligned}$$

We may rearrange the above equations into the following form:

$$\begin{aligned}
A_0 &= (a_0 + a_2) + (a_1 + a_3) \\
A_1 &= (a_0 + a_2 w_4^2) + w_4(a_1 + a_3 w_4^2) \\
A_2 &= (a_0 + a_2 w_4^4) + w_4^2(a_1 + a_3 w_4^4) \\
A_3 &= (a_0 + a_2 w_4^6) + w_4^3(a_1 + a_3 w_4^6)
\end{aligned}$$

Since $w_n^2 = w_{n/2}$ and $w_n^{n+k} = w_n^k$, we have

$$\begin{aligned}
A_0 &= (a_0 + a_2) + (a_1 + a_3) \\
A_1 &= (a_0 + a_2 w_2) + w_4(a_1 + a_3 w_2) \\
A_2 &= (a_0 + a_2) + w_4^2(a_1 + a_3) \\
A_3 &= (a_0 + a_2 w_4^2) + w_4^3(a_1 + a_3 w_4^2) \\
&= (a_0 + a_2 w_2) + w_4^3(a_1 + a_3 w_2).
\end{aligned}$$

Let

$$\begin{aligned}
B_0 &= a_0 + a_2 \\
C_0 &= a_1 + a_3 \\
B_1 &= a_0 + a_2 w_2 \\
C_1 &= a_1 + a_3 w_2.
\end{aligned}$$

Then

$$A_0 = B_0 + w_4^0 C_0$$
$$A_1 = B_1 + w_4^1 C_1$$
$$A_2 = B_0 + w_4^2 C_0$$
$$A_3 = B_1 + w_4^3 C_1.$$

The above equation shows that the divide-and-conquer strategy can be elegantly applied to solve the Fourier Transform problem. We merely have to calculate B_0, C_0, B_1 and C_1. The A_j's can be easily obtained. In other words, once A_0 is obtained, A_2 can be obtained immediately. Similarly, once A_1 is obtained, A_3 can be obtained directly.

But, what are B_i's and C_i's? Note that B_i's are the Fourier Transform of the odd numbered input data items and the C_i's are the Fourier Transform of the even numbered input data items. This is the basis of applying the divide-and-conquer strategy to the Fourier Transform problem. We break a large problem into two subproblems, solve these subproblems recursively and then merge the solutions.

Consider A_j in the general case.

$$\begin{aligned} A_j &= a_0 + a_1 w_n^j + a_2 w_n^{2j} + \cdots + a_{n-1} w_n^{(n-1)j} \\ &= (a_0 + a_2 w_n^{2j} + a_4 w_n^{4j} + \cdots + a_{n-2} w_n^{(n-2)j}) \\ &\quad + w_n^j (a_1 + a_3 w_n^{2j} + a_5 w_n^{4j} + \cdots + a_{n-1} w_n^{(n-2)j}) \\ &= (a_0 + a_2 w_{n/2}^j + a_4 w_{n/2}^{2j} + \cdots + a_{n-2} w_{n/2}^{\frac{(n-2)j}{2}}) \\ &\quad + w_n^j (a_1 + a_3 w_{n/2}^j + a_5 w_{n/2}^{2j} + \cdots + a_{n-1} w_{n/2}^{\frac{(n-2)j}{2}}). \end{aligned}$$

Define B_j and C_j as follows:

$$B_j = a_0 + a_2 w_{n/2}^j + a_4 w_{n/2}^{2j} + \cdots + a_{n-2} w_{n/2}^{\frac{(n-2)j}{2}}$$

and $$C_j = a_1 + a_3 w_{n/2}^j + a_5 w_{n/2}^{2j} + \cdots + a_{n-1} w_{n/2}^{\frac{(n-2)j}{2}}.$$

Then

$$A_j = B_j + w_n^j C_j.$$

It can also be proved that

$$A_{j+n/2} = B_j + w_n^{j+n/2} C_j.$$

For $n = 2$, the Fourier Transform is as follows:

$$A_0 = a_0 + a_1$$
$$A_1 = a_0 - a_1.$$

The Fast Fourier Transform algorithm based upon the divide-and-conquer approach is presented below:

Algorithm 4–6 □ A Fast Fourier Transform algorithm based on the divide-and-conquer strategy

Input: $a_0, a_1, \ldots, a_{n-1}, n = 2^k$.

Output: $A_j = \sum_{k=0}^{n-1} a_k e^{\frac{2\pi ijk}{n}}$ for $j = 0, 1, 2, \ldots, n-1$.

Step 1. If $n = 2$,

$A_0 = a_0 + a_1$

$A_1 = a_0 - a_1$

Return.

Step 2. Recursively find the coefficients of the Fourier Transform of $a_0, a_2, \ldots, a_{n-2}(a_1, \ldots, a_{n-1})$. Let the coefficients be denoted as $B_0, B_1, \ldots, B_{n/2}(C_0, C_1, \ldots, C_{n/2})$.

Step 3. For $j = 0$ to $j = \frac{n}{2} - 1$

$A_j = B_j + w_n^j C_j$

$A_{j+n/2} = B_j + w_n^{j+n/2} C_j.$

The time complexity of the above algorithm is obviously $O(n \log n)$. There is an inverse Fourier Transform which transforms $A_0, A_1, \ldots, A_{n-1}$ back to $a_0, a_1, \ldots, a_{n-1}$ as follows:

$$a_j = \frac{1}{n} \sum_{k=0}^{n-1} A_k e^{\frac{-2\pi ijk}{n}} \quad \text{for} \quad j = 0, 1, 2, \ldots, n-1.$$

Let us consider one example. Let $a_0 = 1$, $a_1 = 0$, $a_2 = -1$, $a_3 = 0$. Then

$$B_0 = a_0 + a_2 = 1 + (-1) = 0$$
$$B_1 = a_0 - a_2 = 1 - (-1) = 2$$
$$C_0 = a_1 + a_3 = 0 + 0 = 0$$
$$C_1 = a_1 - a_3 = 0 - 0 = 0$$
$$w_4 = e^{\frac{2\pi i}{4}} = i$$
$$w_4^2 = -1$$
$$w_4^3 = -i.$$

Thus,

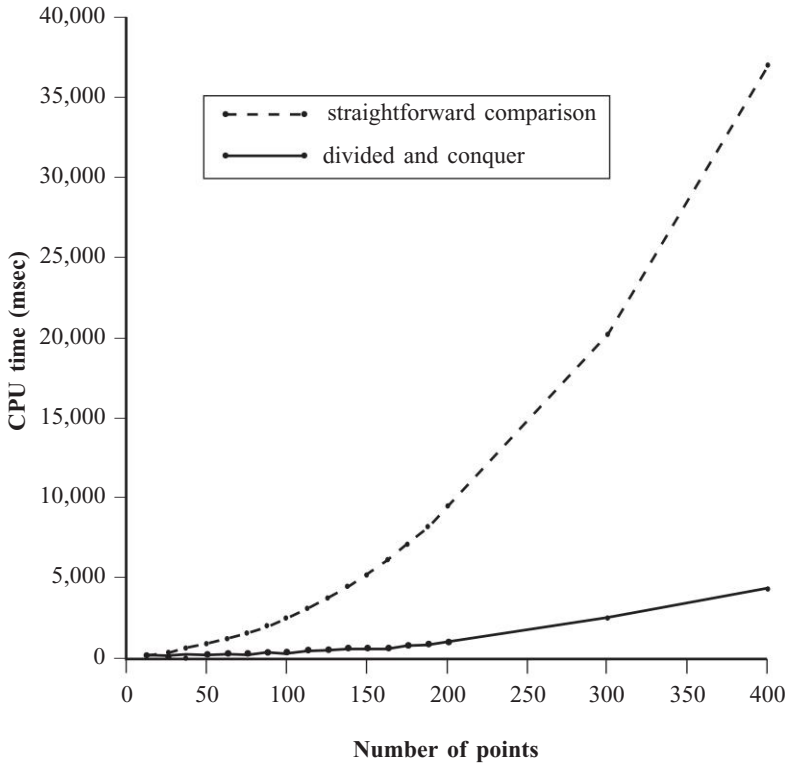
$$A_0 = B_0 + w_4^0 C_0 = B_0 + C_0 = 0 + 0 = 0$$
$$A_1 = B_1 + w_4 C_1 = 2 + 0 = 2$$
$$A_2 = B_0 + w_4^2 C_0 = 0 - 0 = 0$$
$$A_3 = B_1 + w_4^3 C_1 = 2 + 0 = 2.$$

It should be easy to see that the inverse Fourier Transform would transform A_i's back to a_i's correctly.

4–7 THE EXPERIMENTAL RESULTS

To show that the divide-and-conquer strategy is a useful strategy, we implemented the closest pair algorithm based upon the divide-and-conquer and the straightforward algorithms exhaustively examining every pair of points. Both programs were implemented on an IBM personal computer. Figure 4–30 summarizes the experimental results. As shown in Figure 4–30, as n is small the straightforward method performs better. However, as n becomes larger, the divide-and-conquer algorithm is much faster. For instance, when n is equal to 7,000, the divide-and-conquer algorithm is nearly 200 times faster than the straightforward method. The results are quite predictable because the straightforward algorithm time complexity is $O(n^2)$ and the divide-and-conquer algorithm time complexity is $O(n \log n)$.

FIGURE 4–30 Experimental results of the closest pair finding problem.



4–8 NOTES AND REFERENCES

The divide-and-conquer strategy is often applied to solve computational geometry problems. Consult Preparata and Shamos (1985) and Lee and Preparata (1984). The divide-and-conquer strategy is also discussed in Horowitz and Sahni (1978) and Aho, Hopcroft and Ullman (1974).

For a complete discussion of the maxima problem, consult Pohl (1972). The closest pair problem and the maxima finding problem were solved by Bentley (1980).

Voronoi diagrams were first discussed in Voronoi (1908). Again, more information about Voronoi diagrams can be found in Preparata and Shamos (1985). For a generalization of Voronoi diagrams to higher order and higher dimension, see Lee (1982) and Brown (1979).

Different convex hull algorithms can be found in Levcopoulos and Lingas (1987) and Preparata and Shamos (1985). The Graham's scan was proposed by Graham (1972). A three-dimensional convex hull can also be found by the divide-and-conquer strategy. This was pointed out by Preparata and Hong (1977).

That the Fourier Transform can be solved by the divide-and-conquer approach was pointed out by Cooley and Tukey (1965). Gentleman and Sande (1966) discuss applications of the Fast Fourier Transform. Brigham's (Brigham, 1974) book is totally devoted to the Fast Fourier Transform.

Divide-and-conquer can be used to produce an efficient matrix multiplication algorithm. See Strassen (1969).

4–9 FURTHER READING MATERIALS

Divide-and-conquer strategy continues to be an attractive topic for many researchers. It is especially powerful for computational geometry. The following papers are recommended for further research: Aleksandrov and Djidjev (1996); Bentley (1980); Bentley and Shamos (1978); Blankenagel and Gueting (1990); Bossi, Cocco and Colussi (1983); Chazelle, Drysdale and Lee (1986); Dwyer (1987); Edelsbrunner, Maurer, Preparata, Rosenberg, Welzl and Wood (1982); Gilbert, Hutchinson and Tarjan (1984); Gueting (1984); Gueting and Schilling (1987); Karp (1994); Kumar, Kiran and Pandu (1987); Lopez and Zapata (1994); Monier (1980); Oishi and Sugihara (1995); Reingold and Supowit (1983); Sykora and Vrto (1993); Veroy (1988); Walsh (1984); and Yen and Kuo (1997).

For some very interesting newly published papers, consult Abel (1990); Derfel and Vogl (2000); Dietterich (2000); Even, Naor, Rao and Schieber (2000); Fu (2001); Kamidoi, Wakabayashi and Yoshida (2002); Lee and Sheu (1992); Liu (2002); Lo, Rajopadhye, Telle and Zhong (1996); Melnik and Garcia-Molina (2002); Messinger, Rowe and Henry (1991); Neogi and Saha (1995); Rosler (2001); Rosler and Ruschendorf (2001); Roura (2001); Tisseur and Dongarra (1999); Tsai and Katsaggelos (1999); Verma (1997); Wang (1997); Wang (2000); Wang, He, Tang and Wee (2003); Yagle (1998); and Yoo, Smith and Gopalarkishnan (1997).

Exercises

4.1 Does binary search use the divide-and-conquer strategy?

4.2 Multiplying two n bit numbers u and v straightforwardly requires $O(n^2)$ steps. By using the divide-and-conquer approach, we can split the number into two equal parts and compute the product by the following method:

$$uv = (a \cdot 2^{n/2} + b) \cdot (c \cdot 2^{n/2} + d) = ac \cdot 2^n + (ad + bc) \cdot 2^{n/2} + bd.$$

If $ad + bc$ is computed as $(a + b)\ (c + d) - ac - bd$, what is the computing time?

4.3 Prove that in quick sort, the maximum stack needed is $O(\log n)$.

4.4 Implement the Fast Fourier Transform algorithm based upon the divide-and-conquer approach. Compare it with the straightforward approach.

4.5 Implement the rank finding algorithm based upon the divide-and-conquer approach. Compare it with the straightforward approach.

4.6 Let $T\left(\frac{n^2}{2^r}\right) = nT(n) + bn^2$, where r is an integer and $r \geq 1$. Find $T(n)$.

4.7 Read Section 3–7 of Horowitz and Sahni (1978) for Strassen's matrix multiplication method based upon divide-and-conquer.

4.8 Let

$$T(n) = \begin{cases} b & \text{for } n = 1 \\ aT(n/c) + bn & \text{for } n > 1. \end{cases}$$

where a, b and c are non-negative constants.
Prove that if n is a power of c then

$$T(n) = \begin{cases} O(n) & \text{if } a < c \\ O(n \log_a n) & \text{if } a = c \\ O(n^{\log_c a}) & \text{if } a > c. \end{cases}$$

4.9 Prove that if $T(n) = mT(n/2) + an^2$, then $T(n)$ is satisfied by

$$T(n) = \begin{cases} O(n^{\log m}) & \text{if } m > 4 \\ O(n^2 \log n) & \text{if } m = 4 \\ O(n^2) & \text{if } m < 4. \end{cases}$$

4.10 A very special kind of sorting algorithm, also based upon divide-and-conquer, is the odd-even merge sorting, invented by Batcher (1968). Read Section 7.4 of Liu (1977) for this sorting algorithm. Is this sorting algorithm suitable as a sequential algorithm? Why? (This is a famous parallel sorting algorithm.)

4.11 Design an $O(n \log n)$ time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers.

chapter

5

Tree Searching Strategy

In this chapter, we shall show that the solutions of many problems may be represented by trees and therefore solving these problems becomes a tree searching problem. Let us consider the satisfiability problem which will be discussed in Chapter 8. Given a set of clauses, one method of determining whether this set of clauses is satisfiable is to examine all possible assignments. That is, if there are n variables $x_1, x_2, \ldots, x_n$, then we simply examine all 2^n possible assignments. In each assignment, x_i is assigned either T or F. Suppose that $n = 3$. Then the following assignments need be examined:

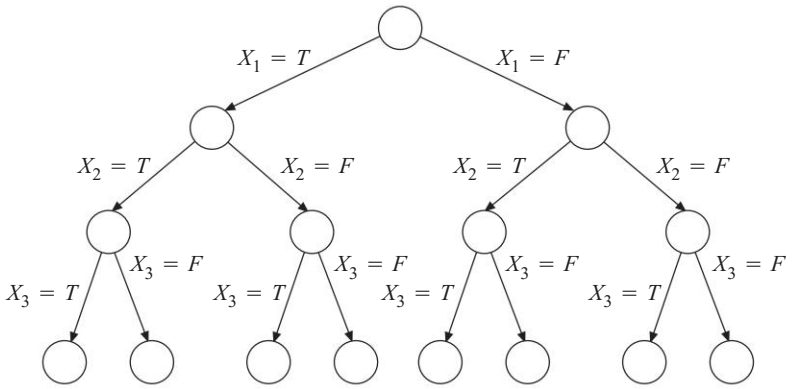
x_1	x_2	x_3
F	F	F
F	F	T
F	T	F
F	T	T
T	F	F
T	F	T
T	T	F
T	T	T

The above eight assignments, on the other hand, may be represented by a tree, shown in Figure 5–1. Why is the tree informative? Well, a tree representation of the assignments shows that at the top level, there are actually two classes of assignments:

Class 1: Those assignments where $x_1 = T$.

Class 2: Those assignments where $x_1 = F$.

FIGURE 5–1 Tree representation of eight assignments.

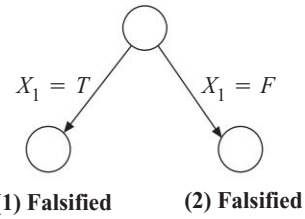


We can thus recursively classify each class of assignments into two subclasses. With this view point, we can manage to determine the satisfiability without examining all assignments, only all classes of assignments. For example, let us suppose that we have the following set of clauses:

$$-x_1 \quad \textbf{(1)}$$
$$x_1 \quad \textbf{(2)}$$
$$x_2 \vee x_5 \quad \textbf{(3)}$$
$$x_3 \quad \textbf{(4)}$$
$$-x_2. \quad \textbf{(5)}$$

We can now develop a partial tree, shown in Figure 5–2.

FIGURE 5–2 A partial tree to determine the satisfiability problem.



From the tree in Figure 5–2, we can easily see that the assignment of $x_1 = T$ will falsify clause (1) and the assignment of $x_1 = F$ will falsify clause (2). Since in every assignment, x_1 is assigned either T or F, we need not examine every assignment. The unsatisfiability of the above set of clauses is now established.

Many other similar problems can also be solved by tree searching techniques. Consider the famous 8-puzzle problem. In Figure 5–3, we show a square frame which can hold nine items. However, only eight items exist and therefore there is an empty spot. Our problem is to move these numbered tiles around so that the final state is reached, shown in Figure 5–4. The numbered tiles can be moved only horizontally or vertically to the empty spot. Thus, for the initial arrangement shown in Figure 5–3, there are only two possible moves, shown in Figure 5–5.

FIGURE 5–3 An 8-puzzle initial arrangement.

2	3	
5	1	4
6	8	7

FIGURE 5–4 The goal state of the 8-puzzle problem.

1	2	3
8		4
7	6	5

The 8-puzzle problem becomes a tree searching problem because we can gradually develop the solution tree. Our problem is solved as soon as a node representing the final goal occurs. We shall elaborate this in later sections.

Finally, we shall show that the solution space of the Hamiltonian cycle problem can also be conveniently represented by a tree. Given a graph $G = (V, E)$, which is a connected graph with n vertices, a Hamiltonian cycle is a round trip path along n edges of G which visits every vertex once and returns to its starting position. Consider Figure 5–6. The path represented by the following sequence is a Hamiltonian cycle: 1, 2, 3, 4, 7, 5, 6, 1. Consider Figure 5–7. There is no Hamiltonian cycle in this graph.

The Hamiltonian cycle problem is to determine whether a given graph contains any Hamiltonian cycle or not. This is an NP-complete problem. Still, it

FIGURE 5–5 Two possible moves for an initial arrangement of an 8-puzzle problem.

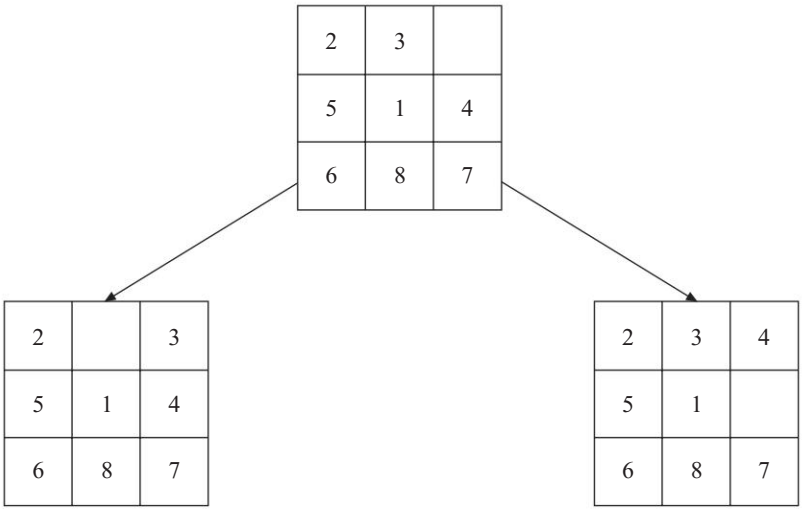


FIGURE 5–6 A graph containing a Hamiltonian cycle.

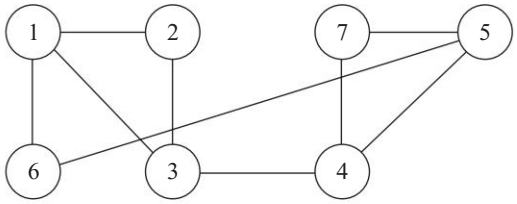
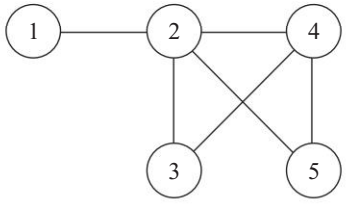


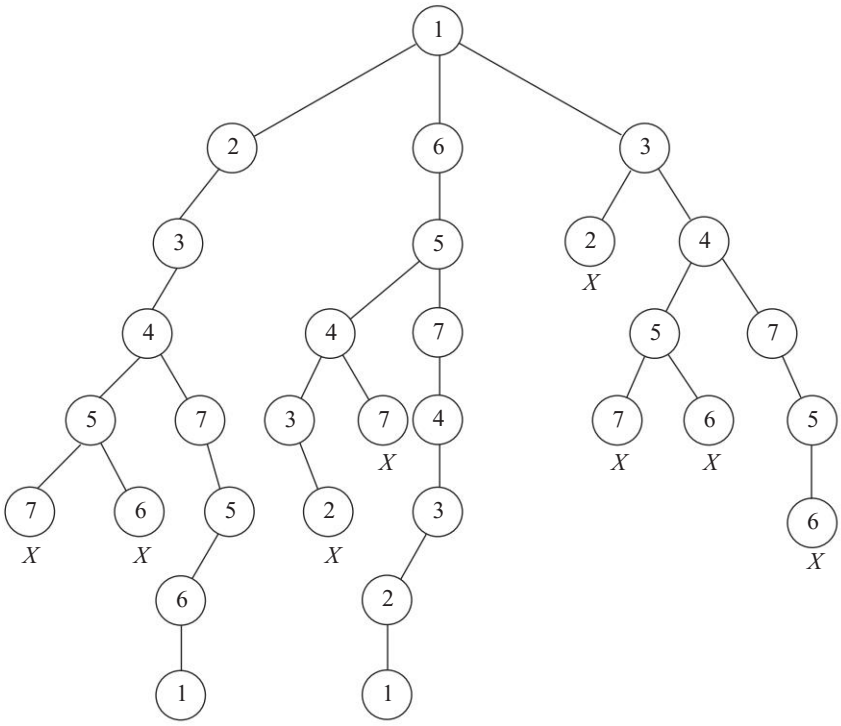
FIGURE 5–7 A graph containing no Hamiltonian cycle.



can be solved by examining all possible solutions and these solutions can be conveniently represented by a tree. Note that a Hamiltonian cycle must visit every node. We therefore may assume that node 1 is our starting node. Consider

Figure 5–6 again. The searching of a Hamiltonian cycle can be described by a tree, shown in Figure 5–8. It shows that there exists one and only one Hamiltonian cycle, as the cycle 1, 2, 3, 4, 7, 5, 6, 1 is equivalent to 1, 6, 5, 7, 4, 3, 2, 1.

FIGURE 5–8 The tree representation of whether there exists a Hamiltonian cycle of the graph in Figure 5–6.



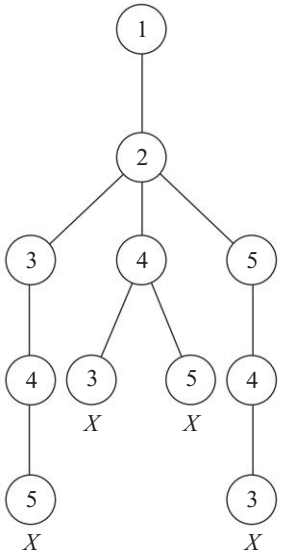
If we examine Figure 5–7, its tree representation of whether there exists a Hamiltonian cycle of the corresponding graph is shown in Figure 5–9. This time, it can be easily seen that there exists no Hamiltonian cycle.

We have shown that many problems can be represented by trees. In the rest of this chapter, we shall discuss the strategies of pruning trees to solve the problems.

5-1 BREADTH-FIRST SEARCH

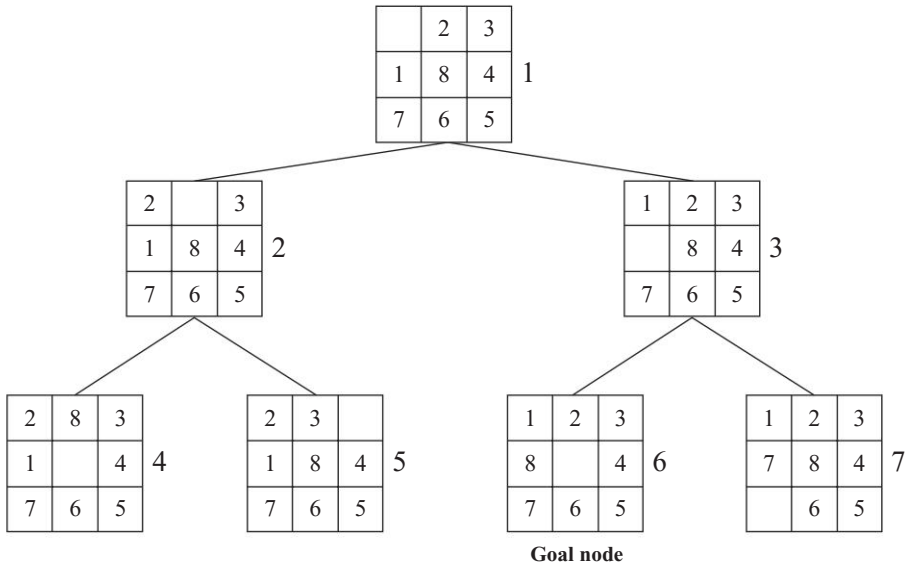
The breadth-first search is perhaps the most straightforward way of pruning a tree. In breadth-first search, all the nodes on one level of the tree are examined

FIGURE 5–9 A tree showing the non-existence of any Hamiltonian cycle.



before the nodes on the next level are examined. Figure 5–10 shows a typical breadth-first search which solves an 8-puzzle problem.

FIGURE 5–10 A search tree produced by a breadth-first search.



We note that Node 6 represents a goal node. Therefore, the searching would stop.

The basic data structure of the breadth-first search is a queue which holds all the expanded nodes. The following scheme illustrates the breadth-first search.

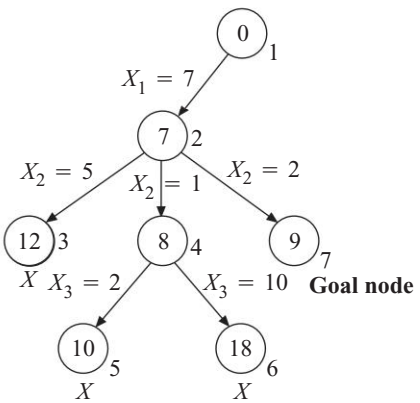
Breadth-first search

Step 1. Form a 1-element queue consisting of the root node.
Step 2. Test to see if the first element in the queue is a goal node. If it is, stop; otherwise, go to Step 3.
Step 3. Remove the first element from the queue. Add the first element's descendants, if any, to the end of the queue.
Step 4. If the queue is empty, then failure. Otherwise, go to Step 2.

5–2 DEPTH-FIRST SEARCH

The depth-first search always selects the deepest node for expansion. Let us consider the following sum of subset problem. We are given $S = \{7, 5, 1, 2, 10\}$ and we have to determine whether there exists a subset S' such that the sum of elements in S' is equal to 9. This problem can be easily solved by depth-first tree searching, shown in Figure 5–11. In Figure 5–11, many nodes are terminated because it is obvious that they are not leading to solutions. The number inside each circle in Figure 5–11 represents the sum of a subset. Note that we always select the deepest node to expand in the process.

FIGURE 5–11 A sum of subset problem solved by depth-first search.



Let us now consider the Hamiltonian cycle problem. For the graph shown in Figure 5–12, we can find a Hamiltonian cycle by depth-first search, shown in Figure 5–13.

FIGURE 5–12 A graph containing a Hamiltonian cycle.

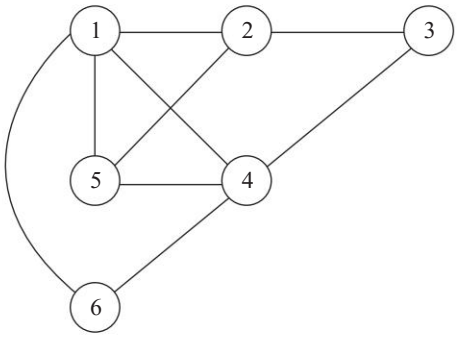
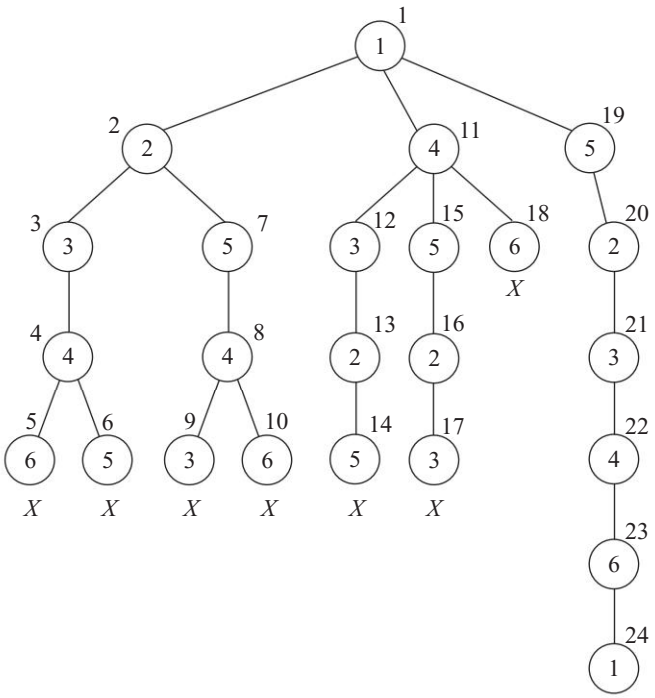


FIGURE 5–13 A Hamiltonian cycle produced by depth-first search.



The depth-first search is now summarized as follows:

Depth-first search

Step 1. Form a 1-element stack consisting of the root node.
Step 2. Test to see if the top element in the stack is a goal node. If it is, then stop; otherwise, go to Step 3.
Step 3. Remove the top element from the stack and add its descendants, if any, to the top of the stack.
Step 4. If the stack is empty, then failure. Otherwise, go to Step 2.

5–3 HILL CLIMBING

After reading the section about depth-first search, the reader may wonder about one problem: Among all the descendants, which node should be selected by us to expand? In this section, we shall introduce a scheme, called hill climbing. Hill climbing is a variant of depth-first search in which some greedy method is used to help us decide which direction to move in the search space. Usually, the greedy method uses some heuristic measure to order the choices. And, the better the heuristics, the better the hill climbing is.

Let us consider the 8-puzzle problem again. Assume that the greedy method uses the following simple evaluation function $f(n)$ to order the choices:

$$f(n) = w(n)$$

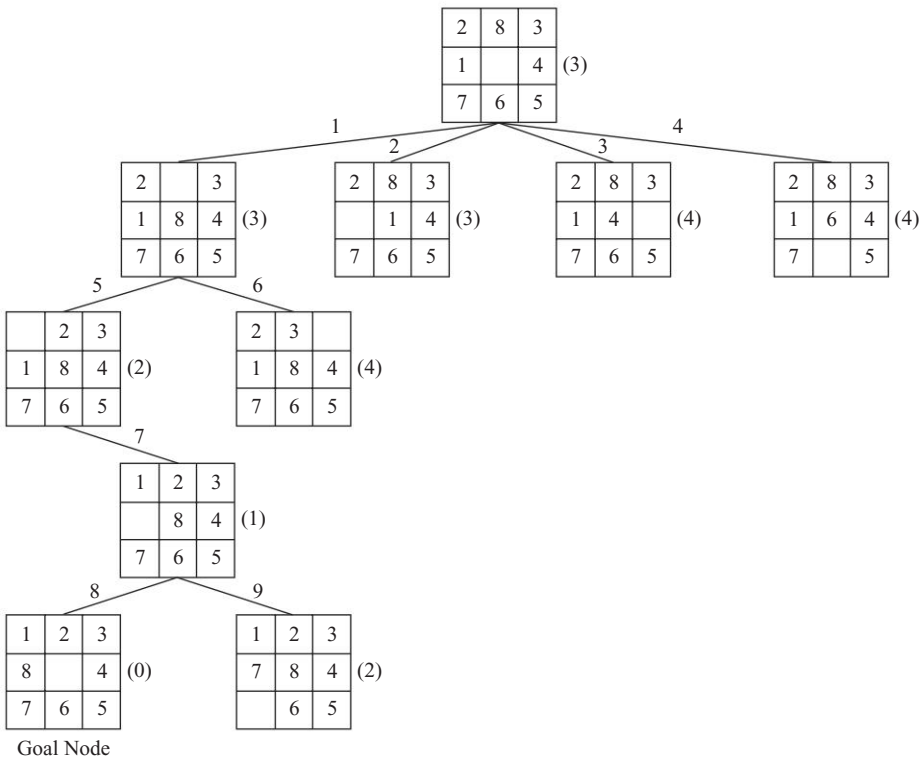
where $w(n)$ is the number of misplaced tiles in node n. Thus, if the starting node is positioned as shown in Figure 5–14, then $f(n)$ is equal to 3 because 1, 2 and 8 are misplaced.

FIGURE 5–14 A starting node of an 8-puzzle problem.

2	8	3
1		4
7	6	5

Figure 5–15 shows the result of applying the hill climbing method using f as a heuristic to order the choices among all the descendants of a node. The value of f for each node is shown in Figure 5–15. On each edge, the order in which the nodes are expanded is also indicated. Note that the hill climbing method still employs the depth-first search, except that among all the descendants of a node, the hill climbing method selects the locally optimal one to expand.

FIGURE 5–15 An 8-puzzle problem solved by the hill climbing method.



The reader should not get a wrong impression that the hill climbing method is extremely efficient because of the example shown in Figure 5–15. In Figure 5–15, among the first expanded nodes, there are two nodes with the same value of the evaluation function. If the other nodes were expanded, it would take much longer to obtain the solution.

Scheme of hill climbing

Step 1. Form a 1-element stack consisting of the root node.
Step 2. Test to see if the top element in the stack is a goal node. If it is, stop; otherwise, go to Step 3.
Step 3. Remove the top element from the stack and expand the element. Add the descendants of the element into the stack ordered by the evaluation function.
Step 4. If the list is empty, then failure. Otherwise, go to Step 2.

5–4 BEST-FIRST SEARCH STRATEGY

The best-first search strategy is a way of combining the advantages of both depth-first and breadth-first search into a single method. In best-first search, there is an evaluation function and we select the node with the least cost among all nodes which we have generated so far. It can be seen that the best-first search approach, unlike the hill climbing approach, has a global view.

Best-first search scheme

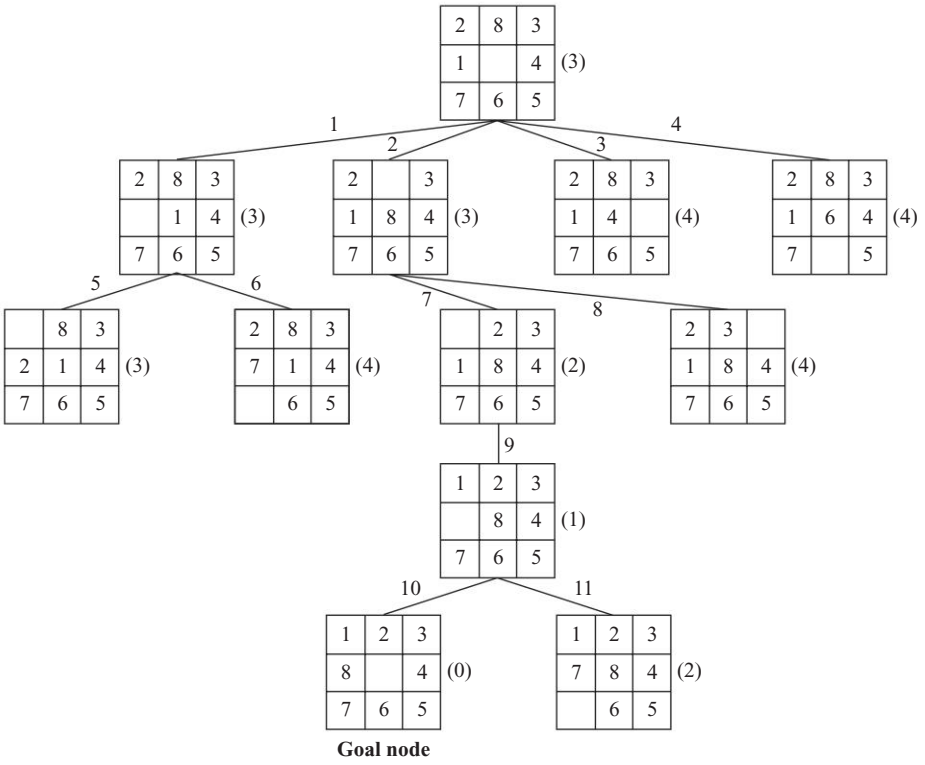
Step 1. Construct a heap by using the evaluation function. First, form a 1-element heap consisting of the root node.
Step 2. Test to see if the root element in the heap is a goal node. If it is, stop; otherwise, go to Step 3.
Step 3. Remove the root element from the heap and expand the element. Add the descendants of the element into the heap.
Step 4. If the heap is empty, then failure. Otherwise, go to Step 2.

If we use the same heuristics as we did in hill climbing for this best-first search, the 8-puzzle problem will be solved as shown in Figure 5–16.

5–5 BRANCH-AND-BOUND STRATEGY

In the previous sections, we showed that many problems can be solved by using the tree searching techniques. Note that none of the problems is an optimization problem. It should be interesting for readers to note that none of the above schemes can be used to solve any optimization problem. In this section, we shall introduce the branch-and-bound strategy, which is perhaps one of the most efficient strategies to solve a large

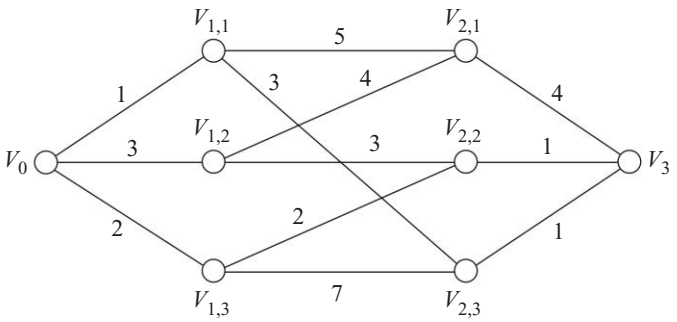
FIGURE 5–16 An 8-puzzle problem solved by a best-first search scheme.



combinatorial problem. Basically, it suggests that a problem may have feasible solutions. However, one should try to cut down the solution space by discovering that many feasible solutions cannot be optimal solutions.

Let us now explain the basic principles of the branch-and-bound strategy by considering Figure 5–17.

FIGURE 5–17 A multi-stage graph searching problem.



In Figure 5–17, our problem is to find a shortest path from V_0 to V_3. This problem can be solved efficiently by first transforming this problem into a tree-searching problem as shown in Figure 5–18.

FIGURE 5–18 A tree representation of solutions to the problem in Figure 5–17.

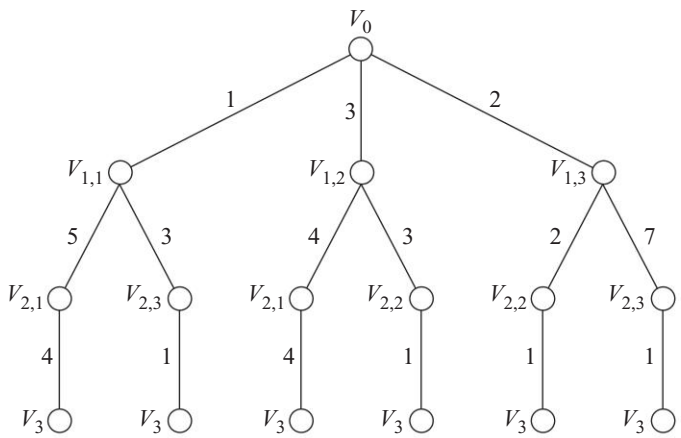


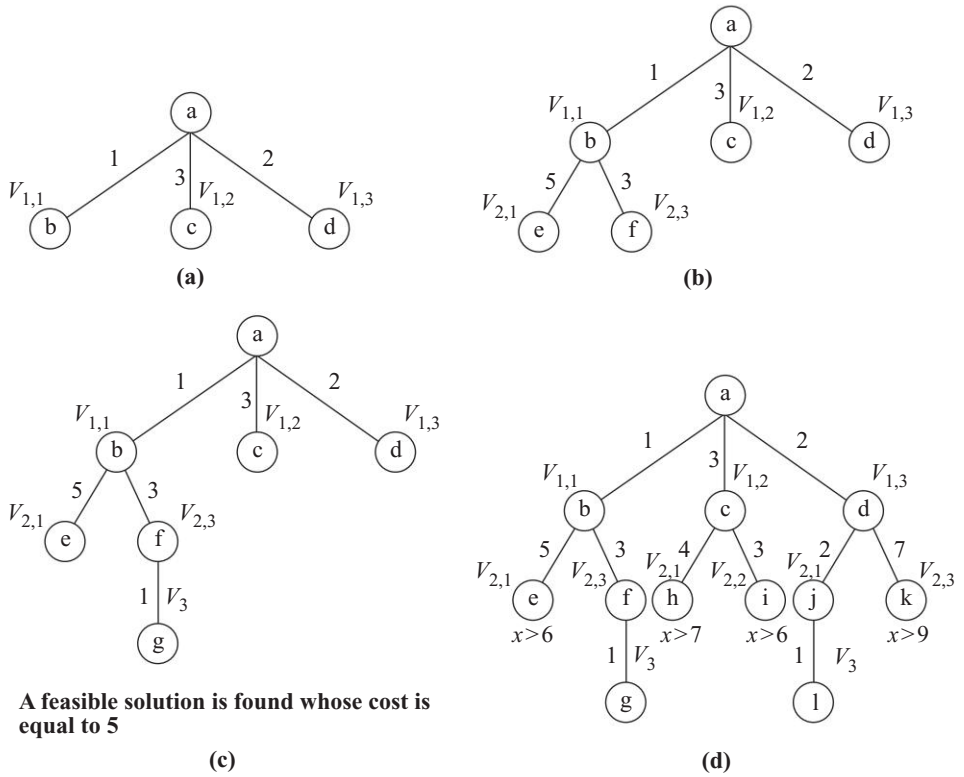
Figure 5–18 shows all six feasible solutions. How will the branch-and-bound strategy help us find the shortest path without an exhaustive search? Consider Figure 5–19, which illustrates the process of using some kind of hill climbing scheme. In Figure 5–19(a), three nodes are expanded from the root of the searching tree. Among these three nodes, we have to select one node to be expanded. There may be many ways to select the next node to be expanded.

In our case, let us assume that the hill climbing method is used. That is, among the nodes most recently expanded, we always choose the node associated with the least cost as the node to be expanded next.

Using this principle, node *b* will be expanded. Its two descendants are shown in Figure 5–19(b). Since node *f* corresponds to $V_{2,3}$ and its associated cost is the smallest, among nodes *e* and *f*, *f* will be the node to be expanded. Since node *g* is a goal node, we have found a feasible solution whose cost is equal to 5, as shown in Figure 5–19(c).

The cost of this feasible solution, which is equal to 5, serves as an upper bound of our optimal solution. Any solution with cost greater than 5 cannot be an optimal solution. This bound can therefore be used to terminate many

FIGURE 5–19 An illustration of the branch-and-bound strategy.



branches prematurely. For instance, node *e* will never lead to any optimal solution because any solution with node *e* will have a cost greater than 6.

As shown in Figure 5–19, an exhaustive searching of the entire solution space can be avoided. Of course, we must point out that there is another solution which is also optimal.

The above example illustrates the basic principle of the branch-and-bound strategy. This strategy consists of two important mechanisms: A mechanism to generate branchings and a mechanism to generate a bound so that many branchings can be terminated. Although the branch-and-bound strategy is usually very efficient, in worst cases, a very large tree may still be generated. Thus, we must realize that the branch-and-bound strategy is efficient in the sense of average cases.

5–6 A PERSONNEL ASSIGNMENT PROBLEM SOLVED BY THE BRANCH-AND-BOUND STRATEGY

We now show how a personnel assignment problem, which is NP-complete, can be solved efficiently by the branch-and-bound strategy. Let there be a linearly ordered set of persons $P = \{P_1, P_2, \dots, P_n\}$, where $P_1 < P_2 < \cdots < P_n$. We may imagine that the ordering of persons is determined by some criterion, such as height, age, seniority and so on. Let there also be a set of jobs $J = \{J_1, J_2, \dots, J_n\}$ and we assume that these jobs are partially ordered. Each person can be assigned to a job. Let P_i and P_j be assigned to jobs $f(P_i)$ and $f(P_j)$ respectively. We require that if $f(P_i) \leq f(P_j)$, then $P_i \leq P_j$. The function f can be interpreted as a feasible assignment which maps persons to their appropriate jobs. We also require that if $i \neq j$, then $f(P_i) \neq f(P_j)$.

Consider the following example. $P = \{P_1, P_2, P_3\}$, $J = \{J_1, J_2, J_3\}$ and the partial ordering of J is $J_1 \leq J_3$ and $J_2 \leq J_3$. In this case, $P_1 \rightarrow J_1$, $P_2 \rightarrow J_2$ and $P_3 \rightarrow J_3$ are feasible assignments while $P_1 \rightarrow J_1$, $P_2 \rightarrow J_3$ and $P_3 \rightarrow J_2$ are not.

We further assume that there is a cost C_{ij} incurred for a person P_i being assigned to job J_j. Let X_{ij} be 1 if P_i is assigned to J_j and 0 if otherwise. Then the total cost corresponding to a feasible assignment f is

$$\sum_{i,j} C_{ij} X_{ij}.$$

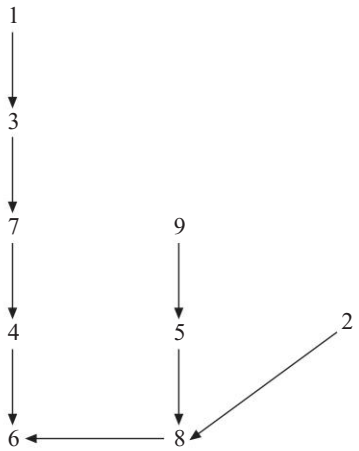
Our personnel assignment problem is precisely defined as follows: We are given a linearly ordered set of persons $P = \{P_1, P_2, \dots, P_n\}$, where $P_1 < P_2 < \cdots < P_n$ and a partially ordered set of jobs $J = \{J_1, J_2, \dots, J_n\}$. Cost C_{ij} is equal to the cost of assigning P_i to J_j. Each person is assigned to a job and no two persons are assigned to the same job. Our problem is to find an optimal feasible assignment which minimizes the following quantity

$$\sum_{i,j} C_{ij} X_{ij}.$$

Thus, our problem is an optimization problem and can be proved to be NP-hard. We shall not discuss the NP-hardness here.

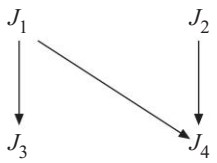
To solve this problem, we shall use the notion "topological sorting". For a given partial ordering set S, a linear sequence $S_1, S_2, \dots, S_n$ is topologically sorted with respect to S if $S_i \leq S_j$ in the partial ordering implies that S_i is located before S_j in the sequence. For instance, for the partial ordering shown in Figure 5–20, a corresponding topologically sorted sequence is 1, 3, 7, 4, 9, 2, 5, 8, 6.

FIGURE 5–20 A partial ordering.



Let $P_1 \rightarrow J_{k_1}$, $P_2 \rightarrow J_{k_2}$, … , $P_n \rightarrow J_{k_n}$ be a feasible assignment. According to our problem definition, the jobs are partially ordered and persons are linearly ordered. Therefore, $J_{k_1}, J_{k_2}, \ldots, J_{k_n}$ must be a topologically sorted sequence with respect to the partial ordering of jobs. Let us illustrate our idea by an example. Consider $J = \{J_1, J_2, J_3, J_4\}$ and $P = \{P_1, P_2, P_3, P_4\}$. The partial ordering of J is illustrated in Figure 5–21.

FIGURE 5–21 A partial ordering of jobs.



The following are all the topologically sorted sequences:

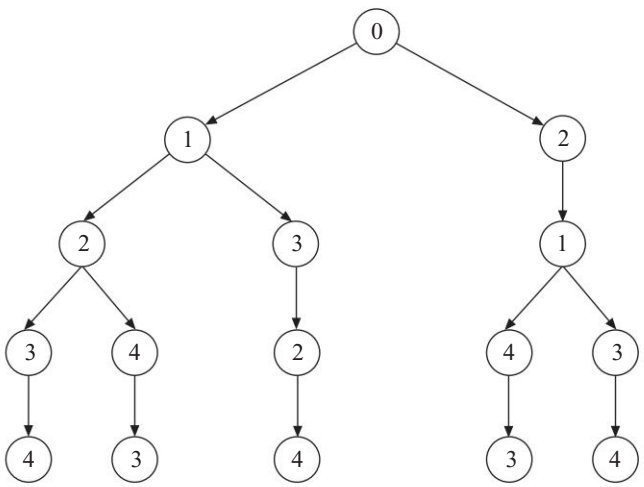
J_1, J_2, J_3, J_4
J_1, J_2, J_4, J_3
J_1, J_3, J_2, J_4
J_2, J_1, J_3, J_4
J_2, J_1, J_4, J_3.

Each sequence represents a feasible assignment. For instance, for the first sequence, it corresponds to the feasible assignment

$P_1 \rightarrow J_1,\ P_2 \rightarrow J_2,\ P_3 \rightarrow J_3,\ P_4 \rightarrow J_4.$

Tree searching techniques can be easily used to find all the topologically sorted sequences. For example, for the partial ordering shown in Figure 5–21, a tree showing all the topologically sorted sequences is shown in Figure 5–22.

FIGURE 5–22 A tree representation of all topologically sorted sequences corresponding to Figure 5–21.



The tree in Figure 5–22 is generated by using three basic steps.

(1) Take an element which is not preceded by any other element in the partial ordering.
(2) Select this element as an element in a topologically sorted sequence.
(3) Remove this element just selected from the partial ordering set. The resulting set is still partially ordered.

For instance, for the partial ordering shown in Figure 5–21, at the very beginning, J_1 and J_2 are the elements without predecessors. Thus, they are on the same level of the tree. Consider the node corresponding to 1. If we remove 1, the partially ordered set now contains 2, 3 and 4. Only 2 and 3 do not have predecessors in this new set. Therefore, 2 and 3 are generated.

Having described how the solution space of our problem can be described by a tree, we can now proceed to show how the branch-and-bound strategy can be used to find an optimal solution.

Given a cost matrix, we can compute a lower bound of our solutions immediately. This lower bound is obtained by reducing the cost matrix in such a way that it will not affect the solutions and there will be at least one zero in every row and every column and all remaining entries of the cost matrix are non-negative.

Note that if a constant is subtracted from any row or any column of the cost matrix, an optimal solution does not change. Consider Table 5–1 in which a cost matrix is shown. For this set of data, we can subtract 12, 26, 3 and 10 from rows 1, 2, 3 and 4, respectively. We can also subtract 3 from column 2 afterwards. The resulting matrix is a reduced cost matrix in which every row and every column contains at least one zero and the remaining entries of the matrix are all non-negative, as shown in Table 5–2. The total cost subtracted is $12 + 26 + 3 + 10 + 3 = 54$. This is a lower bound of our solutions.

TABLE 5–1 A cost matrix for a personnel assignment problem.

Persons \ Jobs	1	2	3	4
1	29	19	17	12
2	32	30	26	28
3	3	21	7	9
4	18	13	10	15

TABLE 5–2 A reduced cost matrix.

Persons \ Jobs	1	2	3	4	
1	17	4	5	0	
2	6	1	0	2	Total = 54
3	0	15	4	6	
4	8	0	0	5	

Figure 5–23 shows an enumeration tree associated with this reduced cost matrix. If the least lower bound is used, subsolutions which cannot lead to optimal solutions will be pruned in a much earlier stage, shown in Figure 5–24.

FIGURE 5–23 An enumeration tree associated with the reduced cost matrix in Table 5–2.

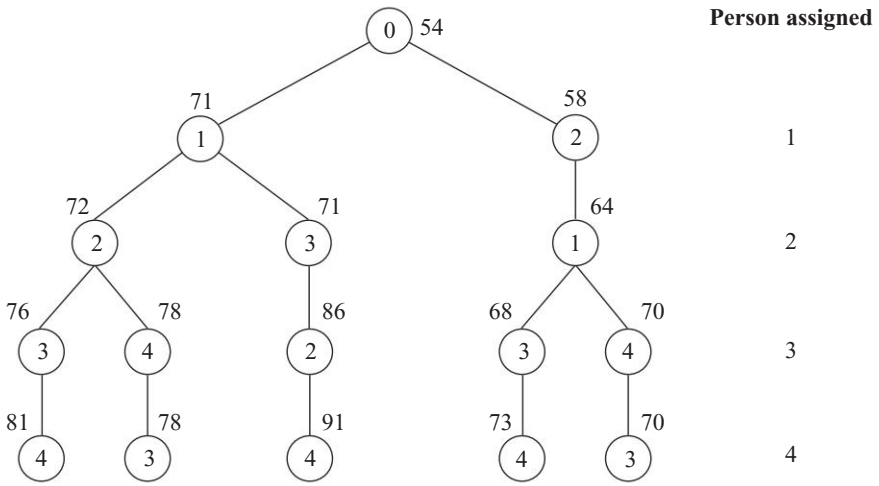
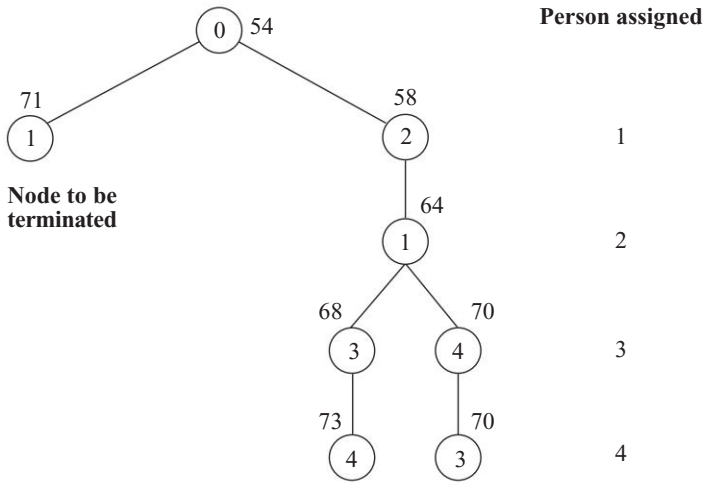


FIGURE 5–24 The bounding of subsolutions.



In Figure 5–24, we can see that after a solution with cost 70 is found, we can immediately bound all solutions starting with assigning P_1 to J_1 because its cost is 71 which is larger than 70.

Why did we subtract costs from the cost matrix? Suppose that we did not. Then, consider the node corresponding to assigning $P_1 \rightarrow J_1$. The cost associated with this node is only 29. Let us imagine that we have found a feasible solution with cost 70, namely assigning $P_1 \rightarrow J_2$, $P_2 \rightarrow J_1$, $P_3 \rightarrow J_4$ and $P_4 \rightarrow J_3$. Although we have found an upper bound, we cannot use it to bound the node corresponding to $P_1 \rightarrow J_1$ because its cost is only 29, lower than 70.

Look at Figure 5–24 again. We can now see that the cost associated with P_1 to J_1 is 71, instead of 29. Thus, a bound occurs. Why can we have such a high cost? This is because we have subtracted costs from the original cost matrix so that each row and each column contains a zero. Thus, after subtracting, we have a better lower bound for all solutions, namely 54. In other words, no solution can have a cost lower than 54. With this information, we know that the lower bound of assigning P_1 to J_1 is $54 + 17 = 71$, instead of only 29. A higher lower bound will of course lead to an earlier termination.

5–7 THE TRAVELING SALESPERSON OPTIMIZATION PROBLEM SOLVED BY THE BRANCH-AND-BOUND STRATEGY

The traveling salesperson decision problem is an NP-complete problem. Thus, the traveling salesperson problem is hard to solve in worst cases. But, as will be shown in this section, the traveling salesperson problem can be solved by using the branch-and-bound strategy. That is, if we are lucky, an exhaustive search through the solution space may be avoided.

The basic principle of using the branch-and-bound strategy to solve the traveling salesperson optimization problem consists of two parts.

(1) There is a way to split the solution space.
(2) There is a way to predict a lower bound for a class of solutions. There is also a way to find an upper bound of an optimal solution. If the lower bound of a solution exceeds this upper bound, this solution cannot be optimal. Thus, we should terminate the branching associated with this solution.

The traveling salesperson problem can be defined on a graph, or planar points. If the traveling salesperson problem is defined on a set of planar points, many tricks can be used so that the algorithm can be even more efficient. In this

section, we shall assume that the problem is defined on a graph. To simplify our discussion, our example assumes that there is no arc between a vertex and itself and there is an arc between every pair of vertices which is associated with a non-negative cost. The traveling salesperson problem is to find a tour, starting from any vertex, visiting every other vertex and returning to this vertex, with minimum cost.

Consider the cost matrix in Table 5–3.

TABLE 5–3 A cost matrix for a traveling salesperson problem.

i \ j	1	2	3	4	5	6	7
1	∞	3	93	13	33	9	57
2	4	∞	77	42	21	16	34
3	45	17	∞	36	16	28	25
4	39	90	80	∞	56	7	91
5	28	46	88	33	∞	25	57
6	3	88	18	46	92	∞	7
7	44	26	33	27	84	39	∞

Our branch-and-bound strategy splits a solution into two groups: one group including a particular arc and the other excluding this arc. Each splitting incurs a lower bound and we shall traverse the searching tree with the "lower" lower bound.

First of all, let us note, as we discussed in the above section, that if a constant is subtracted from any row or any column of the cost matrix, an optimal solution does not change. For the Table 5–3 example, if we subtract the minimum cost of each row from the cost matrix, the total amount that we subtract will be a lower bound for the traveling salesperson solution. Thus, we may subtract 3, 4, 16, 7, 25, 3 and 26 from rows 1 to 7 respectively. The total cost subtracted is $3 + 4 + 16 + 7 + 25 + 3 + 26 = 84$. Through this way, we may obtain a reduced matrix, shown in Table 5–4.

In the matrix shown in Table 5–4, each row contains one zero. Yet, some columns, namely columns 3, 4 and 7, still do not contain any zero. Thus, we further subtract 7, 1 and 4 from columns 3, 4 and 7 respectively. (The cost subtracted is $7 + 1 + 4 = 12$.) The resulting reduced matrix is shown in Table 5–5.

TABLE 5–4 A reduced cost matrix.

i \ j	1	2	3	4	5	6	7
1	∞	0	90	10	30	6	54
2	0	∞	73	38	17	12	30
3	29	1	∞	20	0	12	9
4	32	83	73	∞	49	0	84
5	3	21	63	8	∞	0	32
6	0	85	15	43	89	∞	4
7	18	0	7	1	58	13	∞

TABLE 5–5 Another reduced cost matrix.

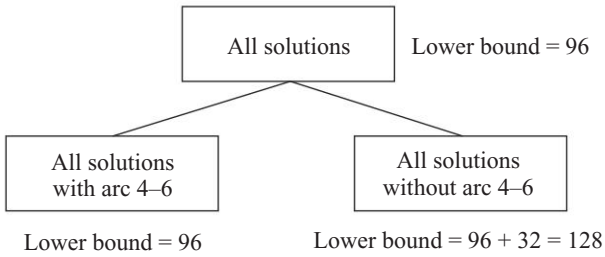
i \ j	1	2	3	4	5	6	7
1	∞	0	83	9	30	6	50
2	0	∞	66	37	17	12	26
3	29	1	∞	19	0	12	5
4	32	83	66	∞	49	0	80
5	3	21	56	7	∞	0	28
6	0	85	8	42	89	∞	0
7	18	0	0	0	58	13	∞

Since a total cost of $84 + 12 = 96$ is subtracted, we know that a lower bound of this traveling salesperson problem is 96.

Let us consider the following problem: Suppose we know that a tour does include arc 4–6, whose cost is zero. What is the lower bound of the cost of this tour? The answer is very easy to find: The lower bound is still 96.

Suppose we know that the tour excludes arc 4–6. What will the new lower bound be? Examine Table 5–5. If a tour does not include arc 4–6, then it must include some other arc which emanates from 4. The arc with the least cost emanating from 4 is arc 4–1 whose cost is 32. The arc with the least cost going into 6 is 5–6 whose cost is 0. Thus the new lower bound is $96 + (32 + 0) = 128$. We therefore have the binary tree shown in Figure 5–25.

FIGURE 5–25 The highest level of a decision tree.



Why did we choose arc 4–6 to split the solution? This is due to the fact that arc 4–6 will cause the largest increase of lower bound. Suppose that we use arc 3–5 to split. In such a case, we can only increase the lower bound by $1 + 17 = 18$.

We now consider the left subtree. In this subtree, arc 4–6 is included. So we must delete the fourth row and the sixth column from the cost matrix. Furthermore, since arc 4–6 is used, arc 6–4 cannot be used. We must set c_{6-4} to be ∞. The resulting matrix is shown in Table 5–6.

TABLE 5–6 A reduced cost matrix if arc 4–6 is included.

i \ j	1	2	3	4	5	7
1	∞	0	83	9	30	50
2	0	∞	66	37	17	26
3	29	1	∞	19	0	5
5	3	21	56	7	∞	28
6	0	85	8	∞	89	0
7	18	0	0	0	58	∞

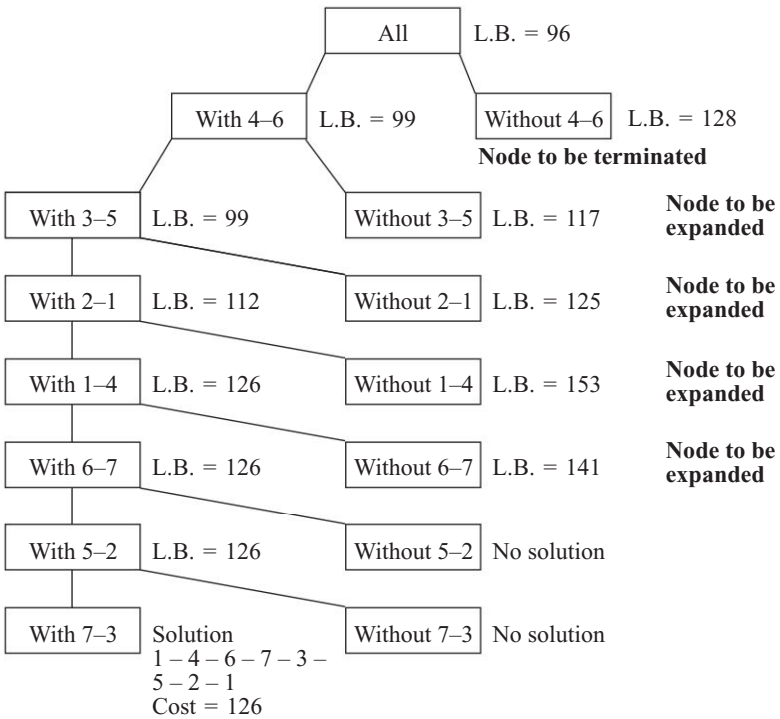
Again, we note that row 5 does not contain any zero yet. So we may subtract 3 from row 5. The reduced cost matrix for the left subtree is shown in Table 5–7. We must also add 3 to the lower bound of the left subtree (solutions with arc 4–6).

As for the cost matrix of the right subtree, solutions without arc 4–6, we only have to set c_{4-6} to be ∞. The splitting process would continue and would produce the tree shown in Figure 5–26. In this process, if we follow the path with the least

TABLE 5–7 A reduced cost matrix for that in Table 5–6.

i \ j	1	2	3	4	5	7
1	∞	0	83	9	30	50
2	0	∞	66	37	17	26
3	29	1	∞	19	0	5
5	0	18	53	4	∞	25
6	0	85	8	∞	89	0
7	18	0	0	0	58	∞

FIGURE 5–26 A branch-and-bound solution of a traveling salesperson problem.



cost, we will obtain a feasible solution with cost 126. This cost 126 serves as an upper bound and many branchings may be terminated because their lower bounds exceed this bound.

Another point needs to be mentioned here. This point can be explained by considering the reduced cost matrix of all solutions with arcs 4–6, 3–5 and 2–1 included as shown in Table 5–8.

TABLE 5–8 A reduced cost matrix.

i \ j	2	3	4	7
1	∞	74	0	41
5	14	∞	0	21
6	85	8	∞	0
7	0	0	0	∞

We may use arc 1–4 to split and the resulting subtree will be as shown in Table 5–9.

TABLE 5–9 A reduced cost matrix.

i \ j	2	3	7
5	14	∞	21
6	85	8	0
7	0	0	∞

Note that we have already decided that arcs 4–6 and 2–1 are included in the solution. Now, arc 1–4 is to be added. Clearly, we must prevent arc 6–2 from being used. If arc 6–2 is used, there will be a loop which is forbidden. So we must set c_{6-2} to be ∞. We shall therefore have the cost matrix for the left subtree as shown in Table 5–10.

In general, if paths i_1–i_2–…–i_m and j_1–j_2–…–j_n have already been included and a path from i_m to j_1 is to be added, then path from j_n to i_1 must be prevented.

TABLE 5–10 A reduced cost matrix.

i \ j	2	3	7
5	14	∞	21
6	∞	8	0
7	0	0	∞

5–8 THE 0/1 KNAPSACK PROBLEM SOLVED BY THE BRANCH-AND-BOUND STRATEGY

The 0/1 knapsack problem is defined as follows: We are given positive integers $P_1, P_2, \ldots, P_n, W_1, W_2, \ldots, W_n$ and M. Our problem is to find $X_1, X_2, \ldots, X_n$, $X_i = 0$ or 1, $i = 1, 2, \ldots, n$, such that

$$\sum_{i=1}^{n} P_i X_i$$

is maximized subject to

$$\sum_{i=1}^{n} W_i X_i \leq M.$$

This problem is an NP-hard problem. However, as we shall see later, we can still use the branch-and-bound strategy to solve this problem. Of course, in the worst cases, even the branch-and-bound strategy will take an exponential number of steps to solve this problem.

The original 0/1 knapsack problem is a maximization problem which cannot be solved by the branch-and-bound strategy. To solve the 0/1 knapsack problem, we must modify the original 0/1 knapsack problem into a minimization problem as follows: Given positive integers $P_1, P_2, \ldots, P_n, W_1, W_2, \ldots, W_n$ and M, find $X_1, X_2, \ldots, X_n, X_i = 0$ or 1, $i = 1, \ldots, n$, such that

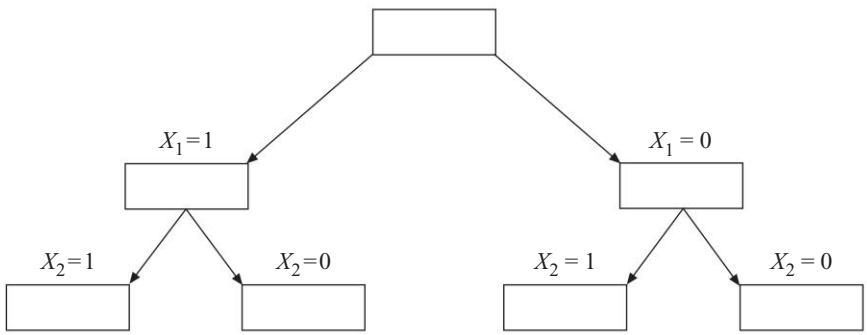
$$-\sum_{i=1}^{n} P_i X_i$$

is minimized subject to

$$\sum_{i=1}^{n} W_i X_i \le M.$$

Any branch-and-bound strategy will need a mechanism for branching. In the 0/1 knapsack problem, the branching mechanism is illustrated by Figure 5–27. The first branching splits all the solutions into two groups: The solutions with $X_1 = 0$ and the solutions with $X_1 = 1$. For each group, X_2 is used to split the solutions. As can be seen, after n X_i's are enumerated, a feasible solution will be found.

FIGURE 5–27 The branching mechanism in the branch-and-bound strategy to solve the 0/1 knapsack problem.



Before explaining the branch-and-bound strategy to solve the 0/1 knapsack problem, let us recall how it is used to solve the traveling salesperson problem. When we use the branch-and-bound strategy to solve the traveling salesperson problem, we use the following basic principle: We split solutions into two groups. For each group, a lower bound is found. At the same time, we try to search for a feasible solution. Whenever a feasible solution is found, an upper bound is found. Our branch-and-bound strategy terminates the expansion of a node if and only if one of the following conditions is satisfied:

(1) The node itself represents an infeasible solution. Then no further expansion makes any sense.
(2) The lower bound of this node is higher than or equal to the presently found lowest upper bound.

We now show a further improvement of the branch-and-bound strategy to solve the 0/1 knapsack problem. We still split solutions into two groups. *For each group, not only a lower bound is found, but also an upper bound is found by finding a feasible solution*. As we expand a node, we hope to find a solution with lower cost. This means that we wish to find a lower upper bound as we expand a node. If we know that our upper bound cannot be lowered because it is already equal to its lower bound, then we should not expand this node any more. In general, we terminate the branching if and only if one of the following conditions is satisfied:

(1) *The node itself represents an infeasible solution.*
(2) *The lower bound of this node is higher than or equal to the presently found lowest upper bound.*
(3) *The lower bound of this node is equal to the upper bound of this node.*

Our question is: How can we find an upper bound and a lower bound of a node? Note that a lower bound can be considered as the value of the best solution you can achieve. A node of our tree corresponds to a partially constructed solution. A lower bound of this node therefore corresponds to the highest possible profit associated with this partially constructed solution. As for the upper bound of a node, we mean the cost of a feasible solution corresponding to this partially constructed solution. We shall illustrate our method through an example.

Consider the following data:

i	1	2	3	4	5	6
P_i	6	10	4	5	6	4
W_i	10	19	8	10	12	8

$M = 34$

We should note that $P_i/W_i \geq P_{i+1}/W_{i+1}$ for $i = 1, 2, \ldots, 5$. This ordering is necessary as we shall see later.

How can we find a feasible solution?

A feasible solution can be easily found by starting from the smallest available i, scanning towards the larger i's until M is exceeded. For instance, we may let $X_1 = X_2 = 1$. Then $\sum_{i=1}^{n} W_i X_i = 10 + 19 = 29 < M = 34$. This means that

$X_1 = X_2 = 1$ is a feasible solution. (Note that we cannot further let $X_3 = 1$ because $\sum_{i=1}^{3} W_i X_i > 34$.)

How can we find a lower bound?

To find a lower bound, let us recall that a lower bound corresponds to the best value that the cost function can achieve. Note that the 0/1 knapsack problem is a constrained optimization problem because X_i is restricted to 0 and 1.

If we relax this restriction, we shall obtain a better result and this better result will be used as our lower bound. We may let X_i be between 0 and 1. If we do this, the 0/1 knapsack problem becomes the knapsack problem which is defined as follows: Given positive integers $P_1, P_2, \ldots, P_n$, $W_1, W_2, \ldots, W_n$ and M, find $X_1, X_2, \ldots, X_n$, $0 \le X_i \le 1$, $i = 1, \ldots, n$, such that

$$-\sum_{i=1}^{n} P_i X_i$$

is minimized subject to

$$\sum_{i=1}^{n} W_i X_i \le M$$

Let $-\sum_{i=1}^{n} P_i X_i$ be an optimal solution for 0/1 knapsack problem and $-\sum_{i=1}^{n} P_i X_i'$ be an optimal solution for the knapsack problem. Let $Y = -\sum_{i=1}^{n} P_i X_i$ and $Y' = -\sum_{i=1}^{n} P_i X_i'$. It is easy to prove that $Y' \le Y$. That is, a solution for the knapsack problem can be served as a lower bound of the solution of the 0/1 knapsack problem.

Furthermore, there is a very interesting point concerning the knapsack optimization problem. That is, the greedy method can be used to find an optimal solution of the knapsack problem (see Exercise 3–11). Consider the set of data presented above. Assume that we have already set $X_1 = X_2 = 1$. We cannot let X_3 be 1 because $W_1 + W_2 + W_3$ will exceed M. However, if we let X_3 be somewhere between 0 and 1, we shall obtain an optimal solution for the knapsack problem,

which is a lower bound of the optimal solution for the 0/1 knapsack problem. The appropriate value of X_3 can be found as follows: Since $M = 34$ and $W_1 + W_2 = 10 + 19 = 29$, X_3 can at best be such that $W_1 + W_2 + W_3X_3 = 10 + 19 + 8X_3 = 34$. Thus, $X_3 = (34 - 29)/8 = 5/8$. With this value, a lower bound is found to be $-(6 + 10 + 5/8 \times 4) = -18.5$. We use the higher limit. Therefore, the lower bound is -18.

Let us consider the case where $X_1 = 1$, $X_2 = 0$ and $X_3 = 0$. Since $W_1 + W_4 + W_5 = 32 < 34$ and $W_1 + W_4 + W_5 + W_6 = 40 > 34$, the lower bound can be found by solving the following equation:

$$W_1 + W_4 + W_5 + W_6X_6 = 32 + 8X_6 = 34.$$

We have: $W_6X_6 = 34 - 32 = 2$, and

$X_6 = \dfrac{2}{8} = \dfrac{1}{4}$. This corresponds to a lower bound of

$$-\left(P_1 + P_4 + P_5 + \frac{1}{4}P_6\right) = -\left(6 + 5 + 6 + \frac{1}{4} \times 4\right) = -18.$$

Note that our method of finding a lower bound is correct because $P_i/W_i \geq P_{i+1}/W_{i+1}$ and our greedy method correctly finds an optimal solution for the knapsack (not 0/1 knapsack) problem under this condition.

How can we find an upper bound?

Consider the following case:

$$X_1 = 1, X_2 = 0, X_3 = 0, X_4 = 0.$$

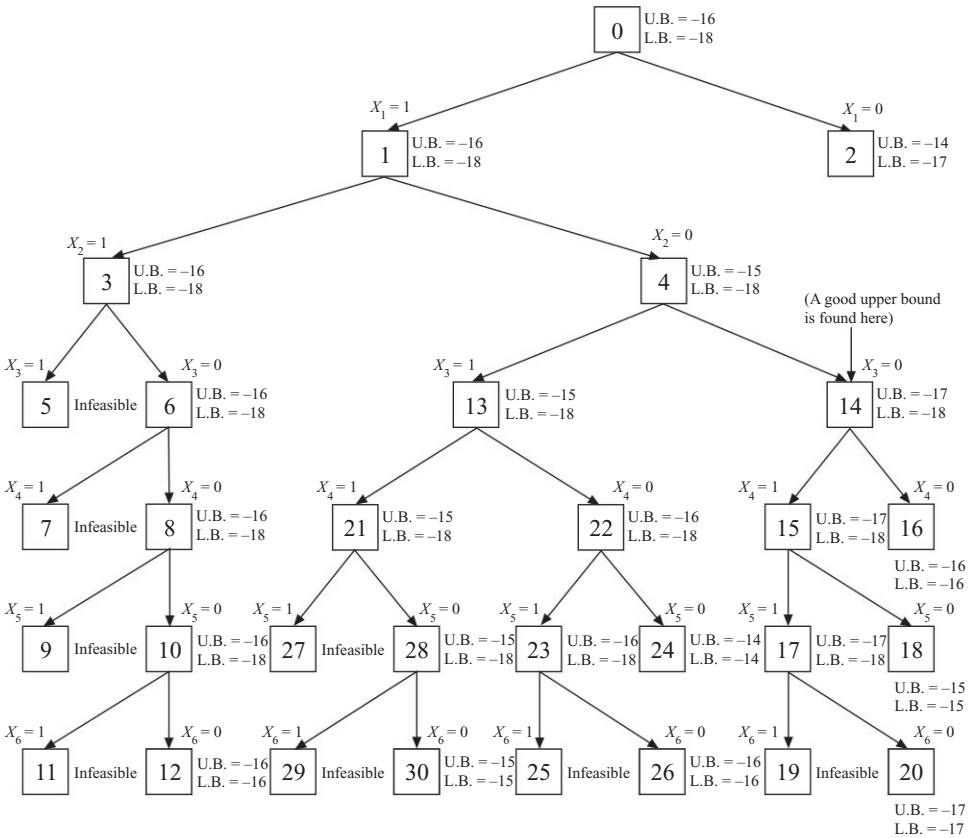
In this case, an upper bound corresponds to

$$X_1 = 1, X_2 = 0, X_3 = 0, X_4 = 0, X_5 = 1, X_6 = 1.$$

This upper bound is $-(P_1 + P_5 + P_6) = -(6 + 6 + 4) = -16$. This means that so far as this node is concerned, if we further expand it, we shall obtain a feasible solution with cost -16. That is why we call this -16 the upper bound of this node.

The entire problem can be solved as indicated by the tree shown in Figure 5–28. The number in each node indicates the sequence in which it is expanded. We use the best search rule. That is, expand the node with the best lower bound. If two nodes have the same lower bound, expand the node with the lower upper bound.

FIGURE 5–28 A 0/1 knapsack problem solved by the branch-and-bound strategy.



In the tree shown in Figure 5–28,

(1) Node 2 is terminated because its lower bound is equal to the upper bound of node 14.
(2) All other nodes are terminated because the local lower bound is equal to the local upper bound.

5–9 A JOB SCHEDULING PROBLEM SOLVED BY THE BRANCH-AND-BOUND APPROACH

While it is easy to explain the basic principles of the branch-and-bound strategy, it is by no means easy to use this strategy effectively. Clever branching and bounding rules still need to be devised. In this section, we shall show the importance of having clever bounding rules.

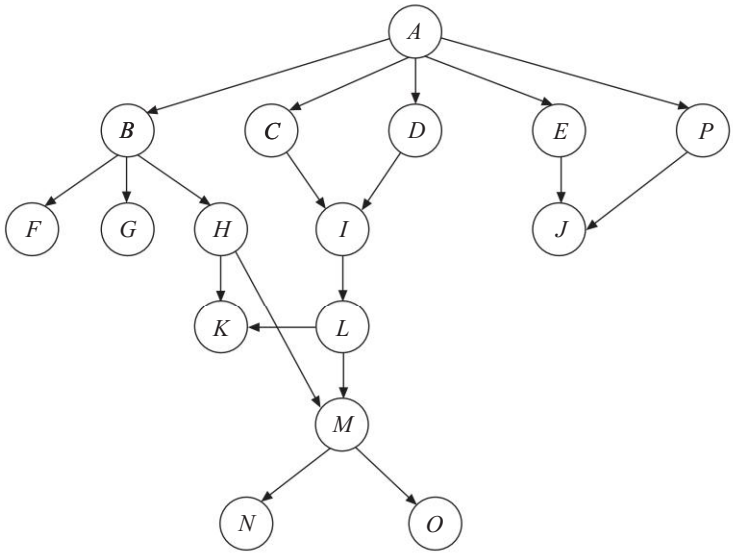
Our problem is a job scheduling problem based on the following assumptions:

(1) All the processors are identical and any job can be executed on any processor.
(2) There is a partial ordering of jobs. A job cannot be executed if one of its ancestor jobs, if it exists, has not been executed yet.
(3) Whenever a processor is idle and a job is available, this processor must start working on the job.
(4) Each job takes equal time for execution.
(5) A time profile is given which specifies the number of processors that can be used simultaneously in each time slot.

The object of the scheduling is to minimize the maximum completion time, which is the time slot in which the last job is finished.

In Figure 5–29, a partial ordering is given. As can be seen, job *I* must wait for jobs *C* and *D* and job *H* must wait for job *B*. At the very beginning, only job *A* can be executed immediately.

FIGURE 5–29 A partial ordering of a job scheduling problem.



Let us consider the time profile shown in Table 5–11.

TABLE 5–11 A time profile.

T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
3	2	2	2	4	5	3	2	3

This time profile indicates that at time $t = 1$, only three processors can be used and at $t = 5$, four processors can be active.

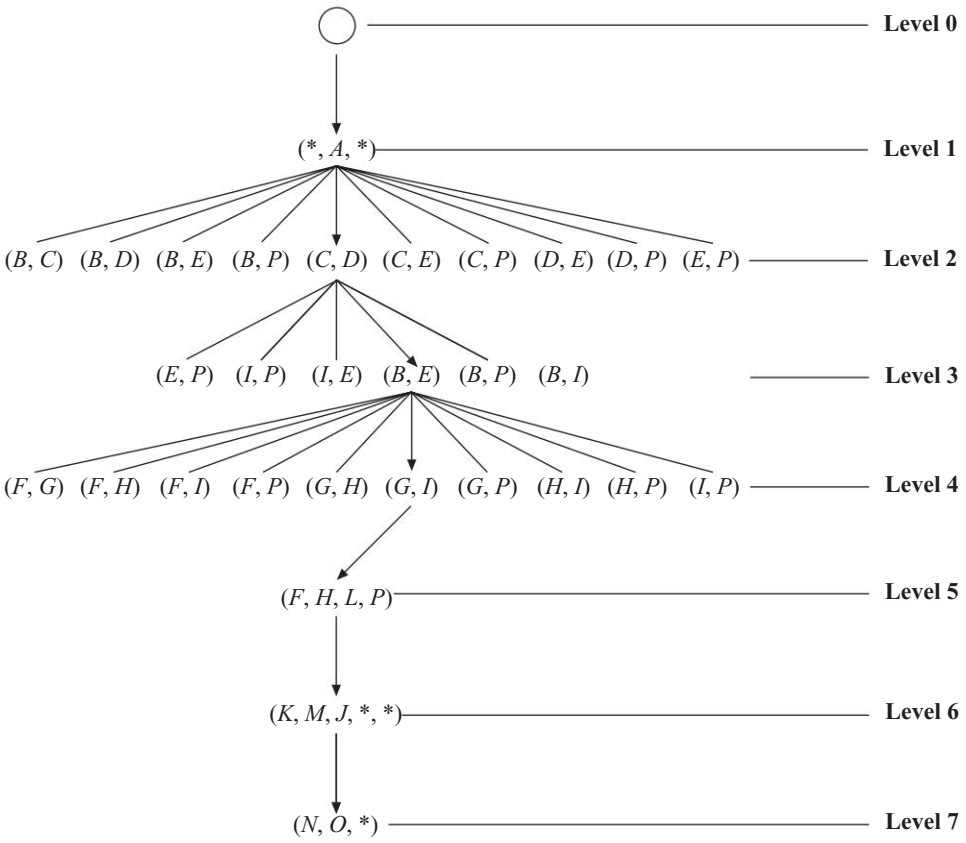
For the partial ordering in Figure 5–29 and the above time profile, there are two possible solutions:

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	
Solution 1:	*A*	*B*	*C*	*H*	*M*	*J*			
	*	*D*	*I*	*L*	*E*	*K*			Time = 6
	*				*F*	*N*			
					P	*O*			
						G			
Solution 2:	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	
	A	*B*	*D*	*F*	*H*	*L*	*M*	*N*	Time = 8
	*	*C*	*E*	*G*	*I*	*J*	*K*	*O*	
	*				*P*	*	*		
					*	*			
						*			

Obviously, Solution 1 is better than Solution 2. Our job scheduling is defined as follows: We are given a partial ordering graph and a time profile; find a solution with the minimal time steps to finish all the jobs. This problem is called equal execution-time job scheduling problem with precedence constraint and time profile. It was proved to be an NP-hard problem.

We shall first show that this job scheduling problem can be solved by tree searching techniques. Suppose that our job partial ordering is that shown in Figure 5–29 and the time profile is that in Table 5–11. Then a partial solution tree is shown in Figure 5–30.

FIGURE 5–30 Part of a solution tree.



* indicates an idle processor

The top of the tree is an empty node. From the time profile, we know that the maximum number of jobs that can be executed is three. However, since job *A* is the only job which can be executed at the very beginning, level 1 of the solution tree consists of only one node. After job *A* is executed, from the time profile, we know that two jobs may be executed now. Level 2 of the solution tree in Figure 5–30 shows all the possible combinations. Limited by space, we only show the descendants of nodes as a subset of nodes of the solution tree.

In the above paragraph, we showed how our job scheduling problem with time profile can be solved by the tree searching strategy. In the following sections, we shall show that in order to use the branch-and-bound strategy, we need to invent good branching and bounding rules, so that we avoid searching the entire solution tree.

The four rules below can be used by our branch-and-bound method. We shall not prove the validity of these rules. Instead, we shall only informally describe them.

RULE 1: Common successors effect

This rule can be informally explained by considering Figure 5–30 again. In this case, the root of the solution tree will be the node consisting of only one job, namely *A*. This node will have many immediate descendants and two of them are (*C*, *E*) and (*D*, *P*). As shown in Figure 5–29, jobs *C* and *D* share the same immediate descendant, namely job *I*. Similarly, *E* and *P* share the same descendant, namely job *J*. In this case, Rule 1 stipulates that only one node, either (*C*, *E*) or (*D*, *P*) needs to be expanded in the solution tree, because the length of any optimal solution headed by node (*C*, *E*) will be the same as that headed by node (*D*, *P*).

Why can we make this conclusion? Consider any feasible solution emanating from (*C*, *E*). Somewhere in the feasible solution, there are jobs *D* and *P*. Since *C* and *D* share the same descendant, namely job *I*, we can exchange *C* and *D* without changing the feasibility of the solution. By similar reasoning, we can also exchange *P* and *E*. Thus, for any feasible solution headed with (*C*, *E*), we can transform it into another feasible solution headed with (*D*, *P*), without changing the length of the solution. Thus, Rule 1 is valid.

RULE 2: Internal node first strategy

The internal node of the job precedence graph shall be processed earlier than the leaf node.

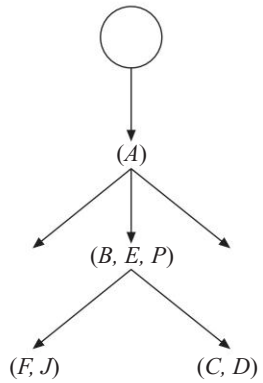
This rule can be informally explained by considering the job precedence graph in Figure 5–29 and the time profile in Table 5–12.

TABLE 5–12 A time profile.

T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
1	3	2	3	2	2	3	2	3

We show a part of the solution tree in Figure 5–31 for this case. After expanding node (*B*, *E*, *P*) of the solution tree, we have many immediate descendants for node (*B*, *E*, *P*) and two of them are (*C*, *D*) and (*F*, *J*) as illustrated in Figure 5–31.

FIGURE 5–31 A partial solution tree.



Since jobs C and D are internal nodes in the job precedence graph where F and J are leaf nodes, Rule 2 suggests that we should traverse node (C, D) and terminate node (F, J) for reasons similar to those used in explaining Rule 1.

Rule 2 suggests that the candidate set should be split into two subsets, one for the active internal nodes and one for the active leaf nodes of the job precedence graph. The former has a higher priority to be processed. Since this set has a smaller size, it reduces the number of possible choices. The set of active leaf nodes will be chosen only when the size of the set of active internal nodes is smaller than the number of active processors. Since leaf nodes have no successors, it makes no difference how they are selected. Therefore, we may arbitrarily choose any group of them. As illustrated in Figure 5–30, after traversing node (B, E) at level 3, we have five jobs in the current candidate set (F, G, H, I, P) and there are two processors active. So we generate the total ten possible combinations. But, if we only consider the internal nodes, then we have only three nodes generated. They are (H, I), (H, P) and (I, P). The other seven nodes will never be generated.

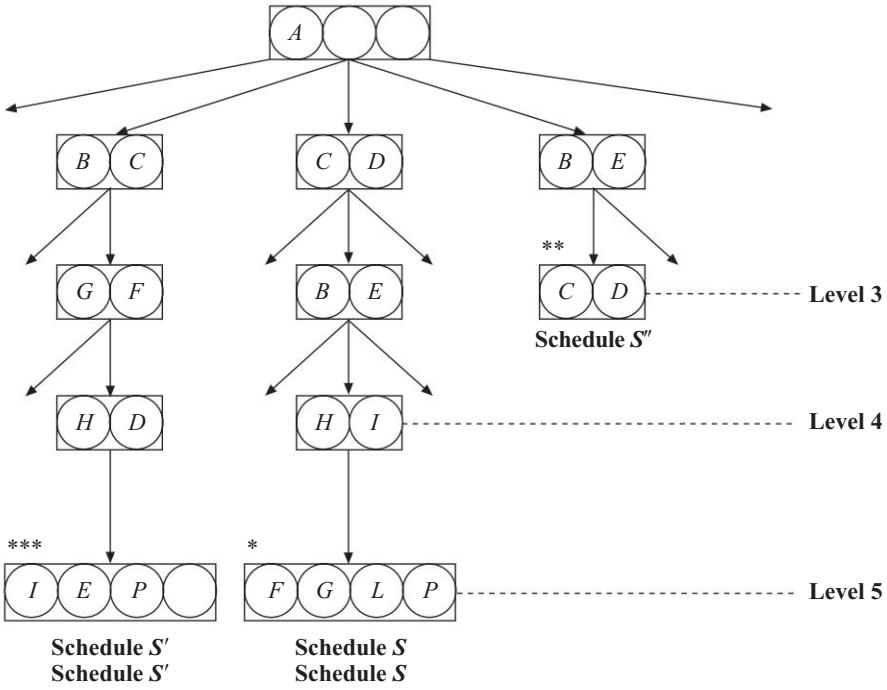
RULE 3: Maximizing the number of processed jobs

Let $P(S, i)$ be a set of already processed jobs in the partially developed or completely developed schedule S, from time slot 1 to time slot i. Rule 3 can be stated as follows:

A schedule S will not be an optimal solution if P(S, i) is contained in P(S′, i) for some other schedule S′. Rule 3 is obviously correct. Figure 5–32 shows

how Rule 3 can be used to terminate some nodes of the solution tree. For Figure 5–32, we claim that the schedules from node "***" and from "**" cannot be better than the schedule from "*". This is due to the fact $P(S'', 3) = P(S, 3)$ and $P(S', 5)$ is contained in $P(S, 5)$.

FIGURE 5–32 Processed jobs effect.



$P(S'', 3) = (A, B, C, D, E)$
$P(S, 3) = (A, B, C, D, E)$
$P(S'', 3) = P(S, 3)$
$P(S', 5) = (A, B, C, D, E, F, G, H, I, P)$
$P(S, 5) = (A, B, C, D, E, F, G, H, I, P, L)$
$P(S', 5)$ is contained by $P(S, 5)$

RULE 4: Accumulated idle processors strategy

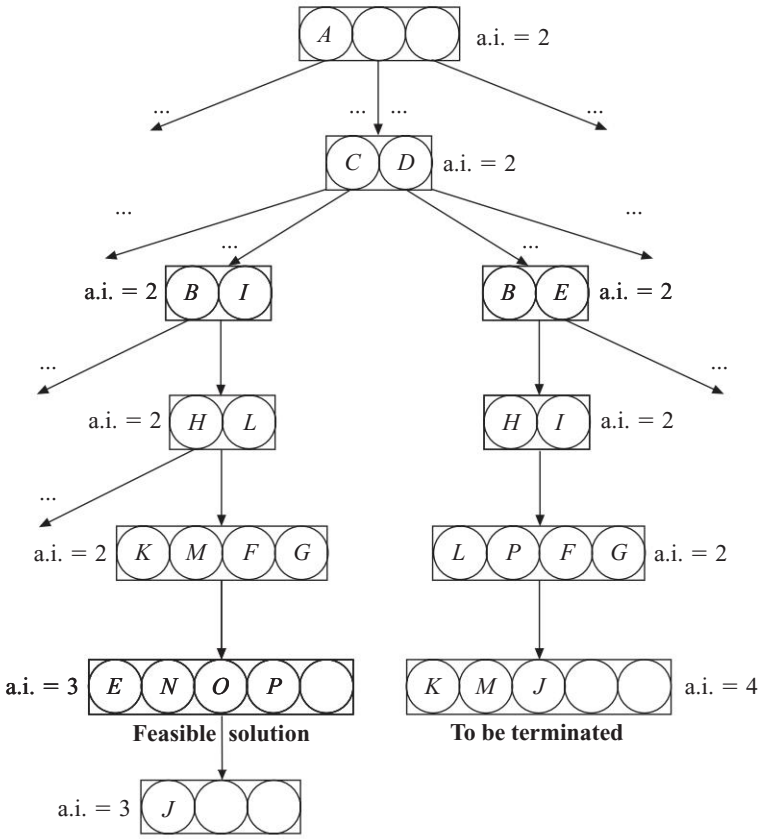
Rule 4 can be stated as follows:

Any partially developed schedule with the number of accumulated idle processors greater than that of a feasible schedule cannot be better than this feasible solution and can therefore be terminated.

Rule 4 shows that if we have already found a feasible solution, then we can use the total number of idle processors to terminate other nodes of the solution tree.

Consider Figure 5–33. Node (*K*, *M*, *J*) of the solution tree has the number of accumulated idle processors equal to 4 which is greater than that of the current feasible solution. So node (*K*, *M*, *J*, *, *) is bounded by using Rule 4.

FIGURE 5–33 Accumulated idle processors effect.



5–10 *A** ALGORITHM

The *A** algorithm is a good tree searching strategy very much favored by artificial intelligence researchers. It is quite regretful that it is often ignored by algorithm researchers.

Let us first discuss the main philosophy behind the *A** algorithm. The best way to do this is to compare it with the branch-and-bound strategy. Note that in

the branch-and-bound strategy, our main effort is to make sure that many solutions need not be further probed because they will not lead to optimal solutions. Thus the main tricks in the branch-and-bound strategy are in bounding.

The A^* algorithm approach emphasizes another viewpoint. *It will tell us that under certain situations, a feasible solution that we have obtained must be an optimal one*. Thus we may stop. Of course, we hope that this termination occurs at the early stage.

The A^* algorithm is often used to solve optimization problems. It uses the best-first (least-cost-first) strategy. The critical element of the A^* algorithm is the cost function. Consider Figure 5–34. Our job is to find a shortest path from S to T. Suppose that we use some tree searching algorithm to solve the problem. The first stage of the solution tree will be that shown in Figure 5–35. Consider node A in Figure 5–35. Node A is associated with the decision to select V_1 (edge a). The cost incurred by selecting V_1 is at least 2 because edge a has a cost 2. Let $g(n)$ denote the path length from the root of the decision tree to node n. *Let* $h^*(n)$ *denote the optimal path length from node n to a goal node. Then the cost of node* n *is* $f^*(n) = g(n) + h^*(n)$. For the tree in Figure 5–35, $g(A) = 2$, $g(B) = 4$ and $g(C) = 3$. The trouble is: What is the value of $h^*(n)$? Note that there are many paths which start from V_1 and end at T.

The following are all such paths:

$V_1 \rightarrow V_4 \rightarrow T$ (path length = 5)
$V_1 \rightarrow V_2 \rightarrow V_4 \rightarrow T$ (path length = 8)
$V_1 \rightarrow V_2 \rightarrow V_5 \rightarrow T$ (path length = 10)
$V_1 \rightarrow V_4 \rightarrow V_5 \rightarrow T$. (path length = 8)

FIGURE 5–34 A graph to illustrate A^* algorithm.

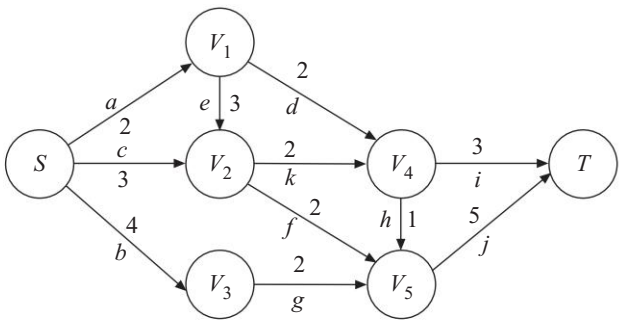
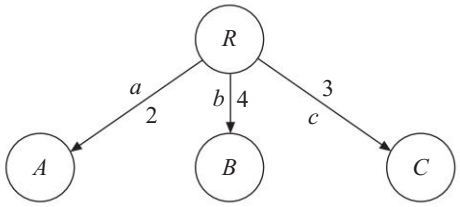


FIGURE 5–35 The first level of a solution tree.



If we start from V_1, the optimum path is $V_1 \rightarrow V_4 \rightarrow T$, which has path length 5. Thus, $h^*(A)$ is equal to 5, which is the path of length of $V_1 \rightarrow V_4 \rightarrow T$. Since $h^*(n)$ is, in general, unknown to us, $f^*(n)$ is generally unknown. The A^* algorithm assumes that we can still estimate $h^*(n)$. Consider node A of Figure 5–35 again. Node A corresponds to V_1 in Figure 5–34. Starting from V_1, there are two possible routes, either to V_2 or to V_4. The shorter one is to visit V_4, which has cost 2. This means that starting from node A, the shortest path from it has length at least equal to 2. Therefore, $h^*(A)$ is at least 2. *We now denote $h(n)$ to be an estimation of $h^*(n)$.*

Although there may be many ways to estimate *$h^*(n)$, the A^* algorithm stipulates that we should always use $h(n) \leq h^*(n)$. That is, we should always use a rather conservative estimation of $h^*(n)$. This means that in the A^* algorithm, we always use $f(n) = g(n) + h(n) \leq g(n) + h^*(n) = f^*(n)$ as our cost function*. This concept will become clear later after a complete example is shown.

Finally, we shall introduce a very important property of the A^* algorithm. Note that the A^* algorithm employs the best-first rule, which means that among all nodes expanded, the node with the least cost will be selected as the next node to be expanded. The A^* algorithm has the following termination rule: *If a selected node is also a goal node, then this node represents an optimal solution and the process may be terminated*.

We now explain why the above rule is a correct one. Let t be the selected goal node. Let n denote a node already expanded.

(1) Since the A^* algorithm uses the least cost rule, $f(t) \leq f(n)$ for all n.
(2) Since the A* algorithm uses a conservative estimation of h*(n), we have $f(t) \leq f(n) \leq f^*(n)$, for all n.
(3) But, one of the $f^*(n)$'s must be an optimal solution. In fact, let $f^*(s)$ denote the value of an optimal solution. Then we have

$$f(t) \leq f^*(s).$$

(4) Statement 3 indicates that at any time, for any selected node to expand, its cost function will not be more than the value of an optimal solution. Since t is a goal node, we have $h(t) = 0$ and

$$f(t) = g(t) + h(t) = g(t)$$

because $h(t)$ is zero. Therefore,

$$f(t) = g(t) \leq f^*(s).$$

(5) Yet, $f(t) = g(t)$ is the value of a feasible solution. Consequently, $g(t)$ cannot be smaller than $f^*(s)$, by definition. This means that $g(t) = f^*(s)$.

We have described the A^* algorithm. We now summarize the fundamental rules of the A* algorithm as follows:

(1) The A^* algorithm uses the best-first (or least-cost-first) rule. In other words, among all nodes expanded, the one with the smallest cost will be the next node to expand.
(2) In the A^* algorithm, the cost function $f(n)$ is defined as follows:

$$f(n) = g(n) + h(n)$$

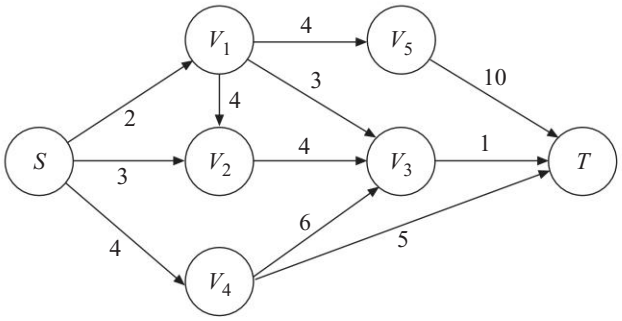
where $g(n)$ is the path length from the root of the tree to n and $h(n)$ is the estimation of $h^*(n)$, which is the optimal path length from n to some goal node.
(3) $h(n) \leq h^*(n)$ for all n.
(4) The A^* algorithm stops if and only if the selected node is also a goal node. It then returns an optimal solution.

The A^* algorithm can, of course, be used with other tree searching rules. For instance, consider Figure 5–36. For this graph, if the problem is to find a shortest path from S to T, then we observe that there are two paths from S to V_3:

$$S \xrightarrow{2} V_1 \xrightarrow{3} V_3$$
$$S \xrightarrow{3} V_2 \xrightarrow{4} V_3.$$

Since the path length of the second one is larger than that of the first one, the second path should be ignored as it will never lead to any optimal solution.

FIGURE 5–36 A graph illustrating the dominance rule.



This is called the dominance rule, which is used in dynamic programming. We shall introduce dynamic programming later in this book. The reader may note that if the dominance rule is used, the path

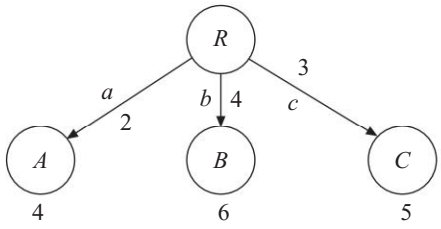
$$S \rightarrow V_1 \rightarrow V_2$$

can also be ignored.

Similarly, all the bounding rules used in the branch-and-bound strategy can also be employed in the A^* algorithm.

Let us now show how the A^* algorithm works to find a shortest path in the graph of Figure 5–34. Steps 1 to 7 show the entire process.

Step 1. Expand *R*

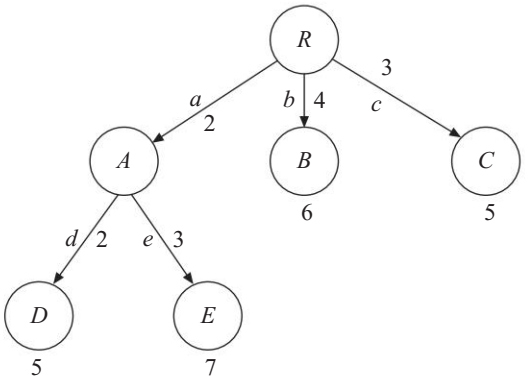


$g(A) = 2$ $\quad h(A) = \min\{2, 3\} = 2$ $\quad f(A) = 2 + 2 = 4$

$g(B) = 4$ $\quad h(B) = \min\{2\} = 2$ $\quad f(B) = 4 + 2 = 6$

$g(C) = 3$ $\quad h(C) = \min\{2, 2\} = 2$ $\quad f(C) = 3 + 2 = 5$

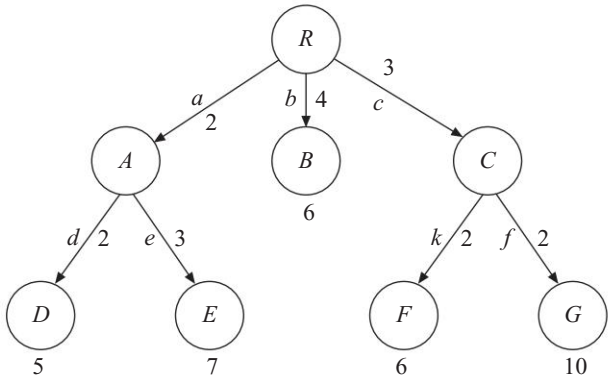
Step 2. Expand *A*



$g(D) = 2 + 2 = 4$ $h(D) = \min\{3, 1\} = 1$ $f(D) = 4 + 1 = 5$

$g(E) = 2 + 3 = 5$ $h(E) = \min\{2, 2\} = 2$ $f(E) = 5 + 2 = 7$

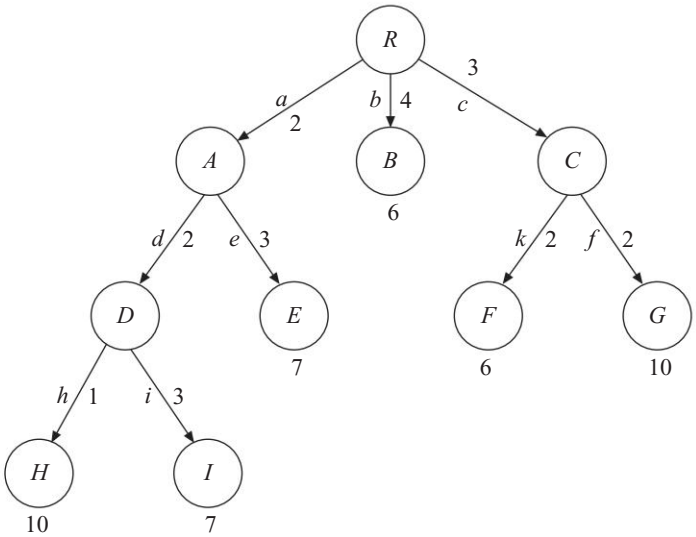
Step 3. Expand *C*



$g(F) = 3 + 2 = 5$ $h(F) = \min\{3, 1\} = 1$ $f(F) = 5 + 1 = 6$

$g(G) = 3 + 2 = 5$ $h(G) = \min\{5\} = 5$ $f(G) = 5 + 5 = 10$

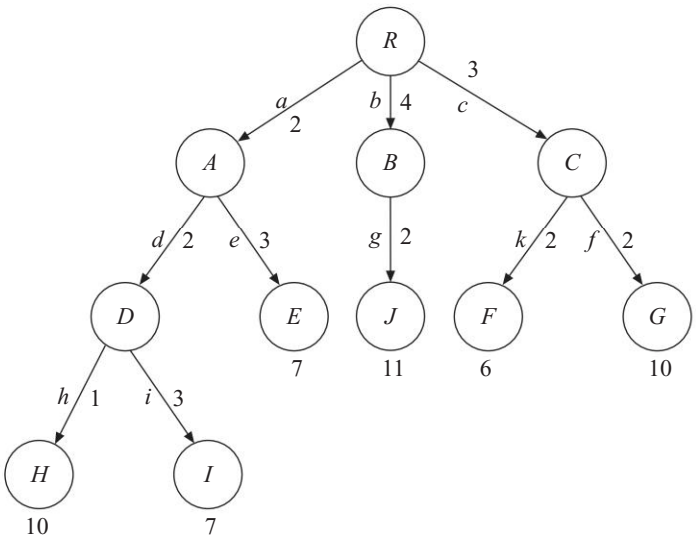
Step 4. Expand *D*



$g(H) = 2 + 2 + 1 = 5$ $h(H) = \min\{5\} = 5$ $f(H) = 5 + 5 = 10$

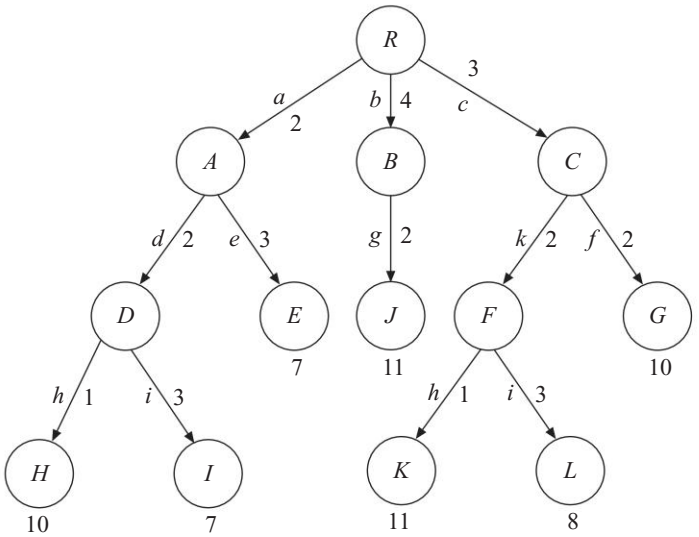
$g(I) = 2 + 2 + 3 = 7$ $h(I) = 0$ $f(I) = 7 + 0 = 7$

Step 5. Expand *B*



$g(J) = 4 + 2 = 6$ $h(J) = \min\{5\} = 5$ $f(J) = 6 + 5 = 11$

Step 6. Expand *F*



$g(K) = 3 + 2 + 1 = 6$ $\quad h(K) = \min\{5\} = 5$ $\quad f(K) = 6 + 5 = 11$

$g(L) = 3 + 2 + 3 = 8$ $\quad h(L) = 0$ $\quad f(L) = 8 + 0 = 8$

Step 7. Expand *I*

Since I is a goal node, we stop and return

$$S \xrightarrow[2]{a} V_1 \xrightarrow[2]{d} V_4 \xrightarrow[3]{i} T$$

as an optimal solution.

Perhaps it is important to ask the following question: *Can we view the A* algorithm as a special kind of branch-and-bound strategy in which the cost function is cleverly designed?* The answer is "Yes". Note that when the *A** algorithm is terminated, we essentially claim that all the other expanded nodes are now simultaneously bounded by this found feasible solution. For example, consider Step 6 of the example above. Node I corresponds to a feasible solution with cost 7. This means that we have found an upper bound of this problem instance, which is 7. However, all the other nodes have costs larger than, or equal to, 7. These costs are also lower bounds. That is why we can terminate the process by bounding all other nodes with this upper bound.

Note that the branch-and-bound strategy is a general strategy. It specifies neither the cost function nor the rule to select a node to expand. *The A* algorithm specifies the cost function $f(n)$ which is actually a lower bound of this node. The A* algorithm also specifies the node selection rule to be the least cost rule. Whenever a goal is reached, an upper bound is found*. If this goal is later selected to be expanded, since the least rule is used, this upper bound must be smaller than or equal to all costs of other nodes. Again, since the cost of each node is a lower bound of this node, this upper bound is smaller than or equal to all other lower bounds. Therefore, this upper bound must be the cost of an optimal solution.

5–11 A CHANNEL ROUTING PROBLEM SOLVED BY A SPECIALIZED *A** ALGORITHM

In this section, we shall introduce a channel routing problem which arises from very large system integrated computer aided design. We shall show that this problem can be elegantly solved by using a specialized A^* algorithm. It is important to note that in this specialized A^* algorithm, *as soon as a goal node is found, we may stop and return an optimal solution*. Note that this is not true in ordinary A^* algorithms because in an ordinary A^* algorithm, we cannot terminate the algorithm when a goal node is found. We can only terminate it when a goal node is selected to be expanded.

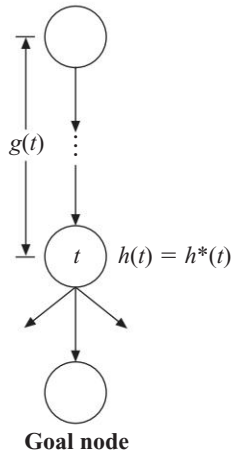
The specialized A^* algorithm can be explained by considering Figure 5–37. In this algorithm, whenever a node t is selected and produces a goal node as its immediate successor, then $h(t) = h^*(t) = g(goal) - g(t)$. Under this condition,

$$f(goal) = g(goal) + h(goal) = g(t) + h^*(t) + 0 = f^*(t)$$
$$g(goal) = f(goal) = f^*(t).$$

Note that t is a selected node to be expanded. Therefore, $f^*(t) \le f(n)$ if n is an expanded node because we use the best-first strategy. By definition, $f(n) \le f^*(n)$. Consequently, $g(goal) = f^*(t) \le f^*(n)$ and the first found goal node must be an optimal solution.

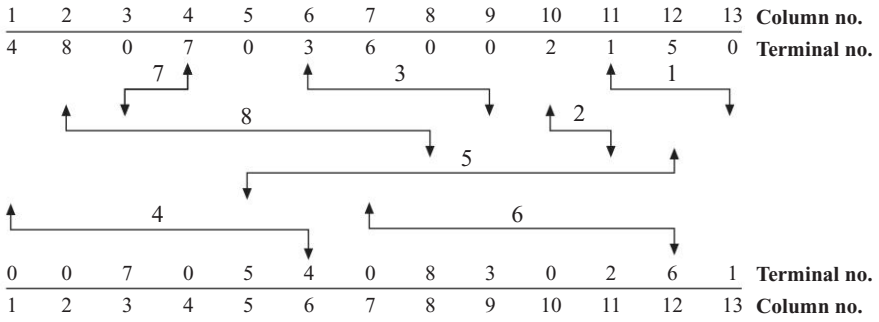
As can be expected, the function $h(n)$ must be carefully designed; otherwise, we will not obtain that good result. It will be surprising to the reader that for the channel routing problem defined in this section, $h(n)$ can be easily designed so that when the A^* algorithm is applied, the first found goal node, or putting it in another way, the first leave node, represents an optimal solution.

FIGURE 5–37 A special situation when the A^* algorithm is applied.



We now describe the channel routing problem. Consider Figure 5–38. In Figure 5–38, we have a channel and two rows of terminals, one on top and one at the bottom. There is also a set of nets labeled from 1 to 8. For instance, net 7 will connect terminal 4 on top with terminal 3 at the bottom. Similarly, net 8 connects terminal 2 on top with terminal 8 at the bottom. When we connect these terminals, we do not allow the illegal connections described in Figure 5–39.

FIGURE 5–38 A channel specification.



There are many ways to connect terminals. Two of them are shown in Figure 5–40 and Figure 5–41 respectively. For the layout in Figure 5–40, there are seven tracks while for that in Figure 5–41, there are only four tracks. In fact, the layout in Figure 5–41 has the minimum number of tracks.

FIGURE 5–39 Illegal wirings.

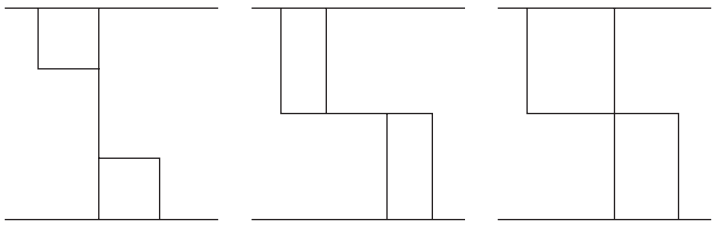


FIGURE 5–40 A feasible layout.

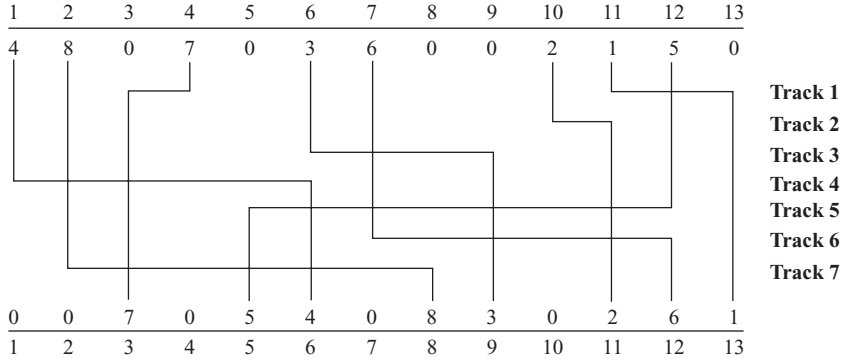
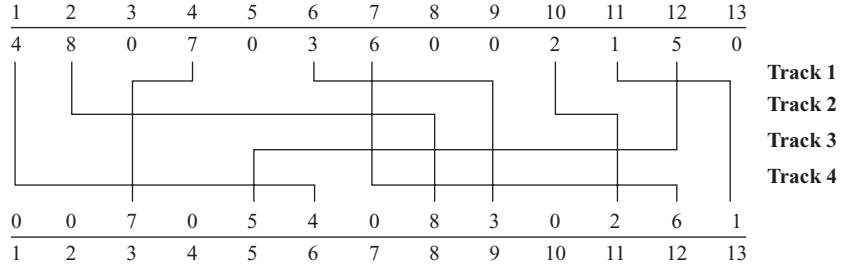


FIGURE 5–41 An optimal layout.

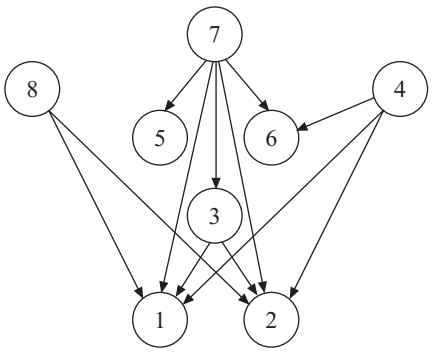


Our channel routing problem is to find a layout which minimizes the number of tracks. This has been proved to be an NP-hard problem. To design an A^* algorithm to solve this problem, we first observe that the nets must obey two kinds of constraints: horizontal constraints and vertical constraints.

Horizontal constraints

The horizontal constraints can be explained by considering the horizontal constraint graph shown in Figure 5–42. By consulting Figure 5–38, we note that net 7, for instance, must be wired to the left of net 3, net 5 and net 6 if they are in the same track. Similarly, net 8 must be to the left of net 1 and net 2. These relationships are summarized in the horizontal constraint graph shown in Figure 5–42.

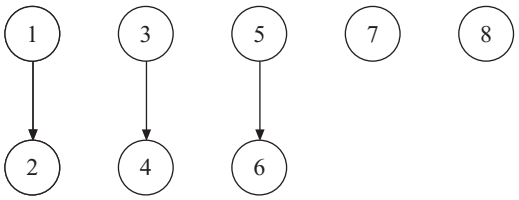
FIGURE 5–42 A horizontal constraint graph.



Vertical constraints

Vertical constraints can also be summarized in a vertical constraint graph, shown in Figure 5–43.

FIGURE 5–43 A vertical constraint graph.



Again, let us consider Figure 5–38. We note that net 1 must be realized before net 2 as they share the same terminal 11. Similarly, net 3 must be wired before net 4.

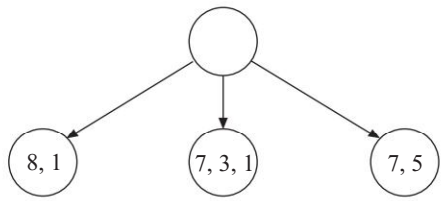
Since, in this book, we are only interested in showing how to apply the *A** algorithm, we are not going into the details of solving the channel routing problem. We shall not explain many steps in detail because they are irrelevant to the understanding of *A** algorithms.

Because of the existence of the vertical constraint graph, which essentially defines a partial ordering, only those nets without predecessors in the vertical constraint graph can be assigned to any track. For instance, initially, only nets 1, 3, 5, 7 and 8 can be assigned to a track. Suppose 1 and 3 have already been assigned, then 2 and 4 can be assigned consequently.

While the vertical constraint graph gives us information about which nets can be assigned, the horizontal constraint graph informs us which nets can be assigned to a track. Since our main interest is to introduce the *A** algorithm, not the approach to solve this channel routing problem, we shall introduce some operations, only in concepts, not the theory behind it. Let us consider Figure 5–43. From Figure 5–43, we note that nets 1, 3, 5, 7 and 8 may be assigned. Consulting Figure 5–42, we note that among nets 1, 3, 5, 7 and 8, there are three maximal cliques: {1, 8}, {1, 3, 7} and {5, 7}. (A clique of a graph is a subgraph in which every pair of vertices are connected. A maximal clique is a clique whose size cannot be made larger.) We can prove that each maximal clique can be assigned to a track. The reader may verify this point by consulting Figure 5–38.

Using these rules, we may use a tree searching approach to solve the channel routing problem, and the first level of the tree is shown in Figure 5–44.

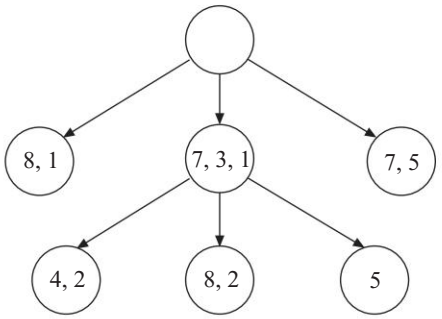
FIGURE 5–44 The first level of a tree to solve a channel routing problem.



To expand this subtree further, let us consider node (7, 3, 1). Consulting the vertical constraint graph shown in Figure 5–43, after nets 1, 3 and 7 are assigned, the nets which can be assigned become 2, 4, 5 and 8. Similarly, if nets 1 and 8 are assigned, nets 2, 3, 5 and 7 can be assigned. For node (7, 3, 1), the next set of nets which can be assigned is {2, 4, 5, 8}. Consulting the horizontal constraint

graph in Figure 5–42 again, we note that among 2, 4, 5 and 8, there are three maximal cliques, namely {4, 2}, {8, 2} and {5}. Thus, if we only expand node {7, 3, 1}, the solutions become those shown in Figure 5–45.

FIGURE 5–45 The tree in Figure 5–44 further expanded.

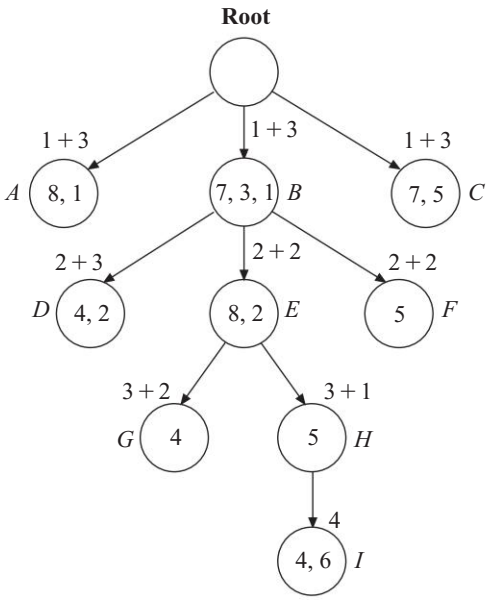


A critical question is: What is the cost function for this tree searching approach to solve the channel routing problem? Note that for A^* algorithm, we need two cost functions: $g(n)$ *and* $h(n)$. $g(n)$ *can be simply defined. Since the level of the tree exactly corresponds to the number of tracks,* $g(n)$ *can be conveniently defined to be the level of the tree. As for* $h(n)$*, we can estimate it by using the minimum number of tracks which must be used after a certain number of tracks have been assigned.*

Let us consider Figure 5–38. We note that for every terminal, we may draw a vertical line. If this vertical line intersects k horizontal lines, then the minimum number of tracks that we need is k. For instance, at terminal 3, the minimum number of tracks is 3 while at terminal 7, the minimum number of tracks is 4. This is called the local density functions at terminal i. The density function of the entire problem is the maximal local density function. For the problem shown in Figure 5–38, the density function is 4. The density function may be changed after some nets are assigned into tracks. For instance, after nets 7, 3 and 1 have been assigned, the density becomes 3. We now use this density function as $h(n)$. Figure 5–46 shows how the A^* algorithm works with our cost function.

It is easy to see that for our A^* algorithm, $h(t) = h^*(t) = 1 = g(goal) - g(t)$ because it needs at least one track to complete the layout. Therefore, node I must represent an optimal solution.

FIGURE 5–46 A partial solution tree for the channel routing problem by using the *A** algorithm.



This example shows that the *A** algorithm is a very good strategy to solve some combinatorially explosive problems provided that a good cost function can be designed.

5–12 THE LINEAR BLOCK CODE DECODING PROBLEM SOLVED BY THE *A** ALGORITHM

Imagine that we use binary codes to send eight numbers, from 0 to 7. We need three bits for each number. For example, 0 is sent by 000, 4 is sent by 100 and 7 is sent by 111. The problem is that if there is any error, the received signal will be decoded wrongly. For example, for 100, if it is sent and received as 000, it will cause a big error.

Let us imagine that instead of using three bits to code numbers, we use, say six digits. Consider the codes in Table 5–13. The right code is called a code word. Each number in its binary number form is sent in the form of its corresponding code. That is, instead of sending 100, we now send 100110 and instead of sending 101, we send 101101. The advantage is now obvious. We decode a received vector to a code word whose Hamming distance is the smallest among

TABLE 5–13 Code words.

000	000000
100	100110
010	010101
001	001011
110	110011
101	101101
011	011110
111	111000

all code words. Suppose the code word 000000 is sent as 000001. We can easily see the Hamming distance between 000001 and 000000 is the shortest among Hamming distances between 000001 and all eight codes. Thus, the decoding process will decode 0000001 as 000000. In other words, by enlarging the number of digits, we can tolerate more errors.

In this book, we shall not discuss how the codes are generated. This can be found in coding theory books. We merely assume that the code words already exist. And our job is to do the decoding. In our case, the coding scheme is called linear block code.

In practice, we do not send 1 and 0 directly. In this section, we shall assume that 1(0) is sent as $-1(1)$. That is, 110110 will be sent as $(-1, -1, 1, -1, -1, 1)$. The distance between a received vector $r = (r_1, r_2, \ldots, r_n)$ and a code word $(c_1, c_2, \ldots, c_n)$ is

$$d_E(r, c) = \sum_{i=1}^{n} (r_i - (-1)^{c_i})^2.$$

Suppose we have $r = (-2, -2, -2, -1, -1, 0)$ and $c = 111000$. Then the distance between them is

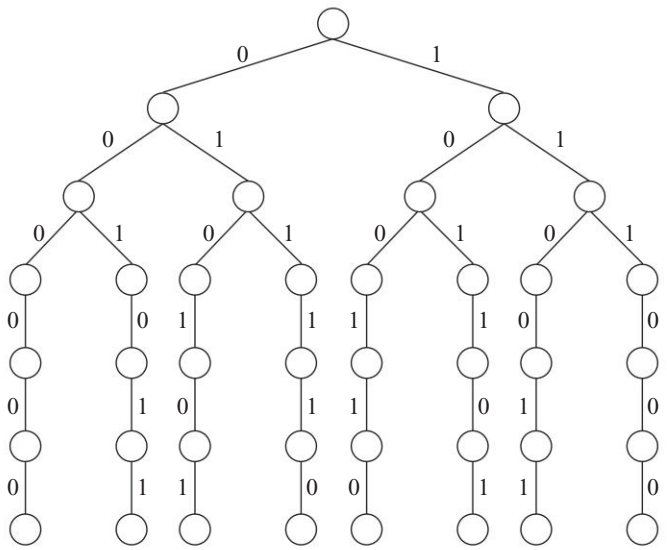
$$\begin{aligned} d_E(r, c) &= (-2 - (-1)^1)^2 + (-2 - (-1)^1)^2 + (-2 - (-1)^1)^2 \\ &\quad + (-1 - (-1)^0)^2 + (-1 - (-1)^0)^2 + (0 - (-1)^0)^2 \\ &= 12. \end{aligned}$$

To decode a received vector, we simply calculate the distances between this vector and all code words and decode this received vector as the particular code

word with the smallest distance. Note that in practice, the number of code words is more than 10^7, which is extremely large. Any exhaustive searching through all of the code words is impossible.

What we do is to use a code tree to represent all of the code words. For instance, for our case, we shall represent all of the code words in Table 5–13 as the tree in Figure 5–47.

FIGURE 5–47 A code tree.



Decoding a received vector, which is to find a code word closest to this received word, now becomes a tree searching problem. This problem can be formally described as follows: Find the path from the root of the code tree to a goal node such that the cost of the path is minimum among all paths from the root to a goal node where the cost of a path is the summation of the costs of branches in the path. The cost of the branch from a node at level $t-1$ to level t is $(r_t - (-1)^{c_t})^2$. We shall show that the A^* algorithm is an effective method to solve the problem.

Let the level of the root of the code tree be -1. Let n be a node at level t. The function $g(n)$ is defined as

$$g(n) = \sum_{i=0}^{t} (r_i - (-1)^{c_i})^2$$

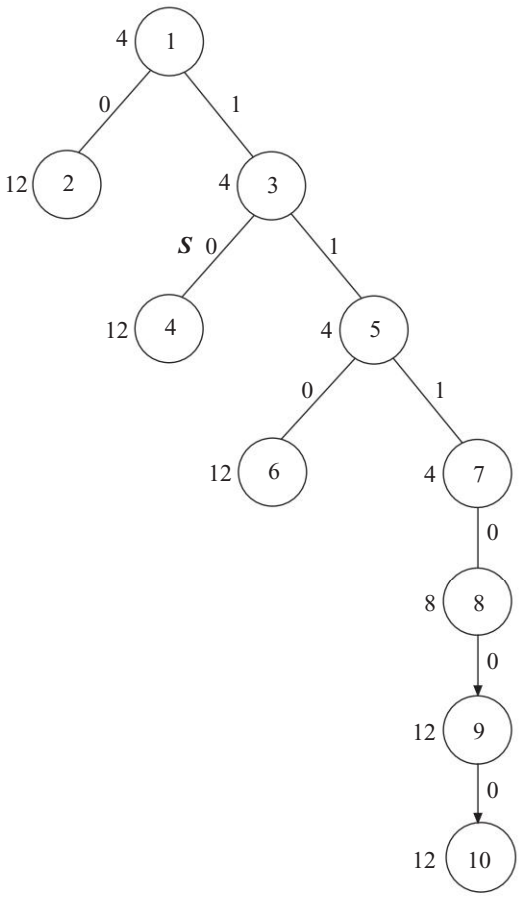
where $(c_1, c_2, \ldots, c_t)$ are the labels of branches associated with the path from the root to node n. Define the heuristic $h(n)$ as

$$h(n) = \sum_{i=t+1}^{n-1} (|r_i| - 1)^2 .$$

It can be easily seen that $h(n) \le h^*(n)$ for every node n.

Consider the code tree in Figure 5–47. Let the received vector be $(-2, -2, -2, -1, -1, 0)$. The decoding process applying the A^* process is illustrated in Figure 5–48.

FIGURE 5–48 The development of a solution tree.



At the very beginning of the decoding, we expand the root node and calculate the two newly generated nodes, which are nodes 2 and 3. The value of $f(2)$ is calculated as follows:

$$\begin{aligned} f(2) &= (-2 - (-1)^0)^2 + h(2) \\ &= 9 + \sum_{i=1}^{5} (|r_i| - 1)^2 \\ &= 9 + 1 + 1 + 0 + 0 + 1 \\ &= 12. \end{aligned}$$

The value of $f(3)$ can be found in the same way. Node 10 is a goal node. When it is selected to expand, the process is terminated and the closest code word is found to be 111000.

5–13 THE EXPERIMENTAL RESULTS

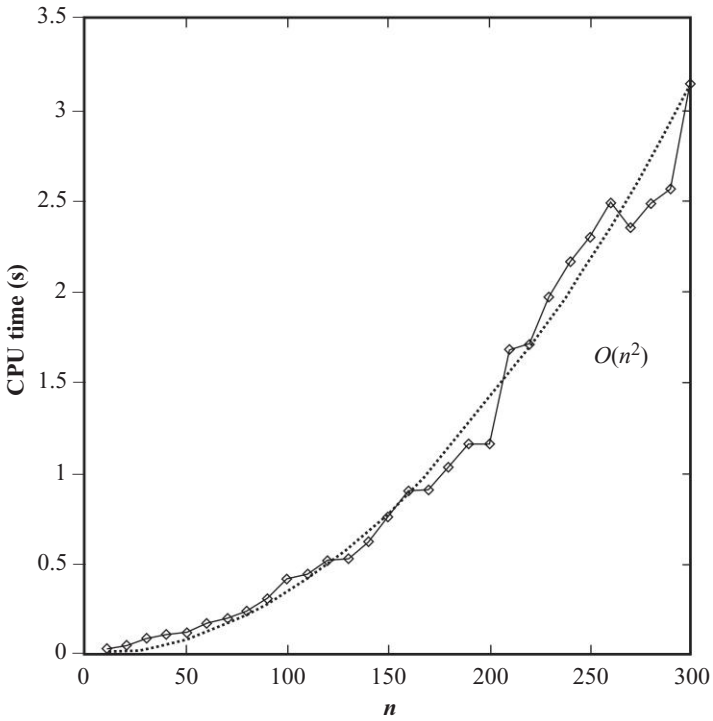
It is our experience that many NP-complete problems can be solved efficiently, in average case sense, by tree searching techniques. Consider the 0/1 knapsack problem, which is an NP-complete problem. We implemented the branch-and-bound approach to solve this problem on an IBM PC. The purpose is to test whether the average case performance is exponential or not. Table 5–14 summarizes the testing results. For each n, the number of elements, we used a random number generator to generate five sets of data. In each set of data, the values of both P_i's and W_i's are between 1 and 50. The integer M is determined as follows. Let W be the sum of all weights. Then M is set to be $W/4$, $(W/4) + 20$, $(W/4) + 40$, … . Of course, M should not exceed W. Then the average time of solving all sets of problem instances is found for each n.

It is rather obvious that the average case performance of the algorithm solving the 0/1 knapsack problem based upon the branch-and-bound strategy is far from exponential. In fact, Figure 5–49 shows that it is almost $O(n^2)$. The reader should be encouraged by these experimental results. Do not be afraid of NP-complete problems. There are efficient algorithms to solve many of them, in average cases, of course.

TABLE 5–14 Test results of using the branch-and-bound strategy to solve the 0/1 knapsack problem.

n	Average time
10	0.025
20	0.050
30	0.090
40	0.110
50	0.120
60	0.169
70	0.198
80	0.239
90	0.306
100	0.415
110	0.441
120	0.522
130	0.531
140	0.625
150	0.762
160	0.902
170	0.910
180	1.037
190	1.157
200	1.167
210	1.679
220	1.712
230	1.972
240	2.167
250	2.302
260	2.495
270	2.354
280	2.492
290	2.572
300	3.145

FIGURE 5–49 The experimental results of the 0/1 knapsack problem by branch-and-bound strategy.



5–14 NOTES AND REFERENCES

Depth-first, breadth-first and least-cost-first searching techniques can be found in Horowitz and Sahni (1976) and Knuth (1973). For *A** algorithms, consult Nilsson (1980) and Perl (1984). Excellent reviews of branch-and-bound strategies can be found in Lawler and Wood (1966) and Mitten (1970).

The personnel assignment problem was first discussed in Ramanan, Deogun and Liu (1984). Liang (1985) showed that this problem can be solved by the branch-and-bound approach. For using branch-and-bound to solve the traveling salesperson problem, consult Lawler, Lenstra, Rinnooy Kan and Shmoys (1985) and Little, Murty, Sweeney and Karel (1963). Using branch-and-bound to solve the traveling salesperson problem appeared in Yang, Wang and Lee (1989). Wang and Lee (1990) pointed out that *A** algorithm is a good technique to solve the channel routing problem. The applications of *A** algorithms to linear block code problems can be found in Ekroot and Dolinar (1996); Han, Hartmann and Chen

(1993); and Han, Hartmann and Mehrotra (1998). Related work can be found in Hu and Tucker (1971).

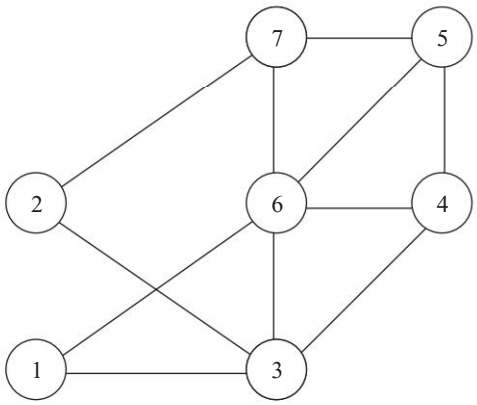
5-15 FURTHER READING MATERIALS

Tree searching techniques are quite natural and very easy to apply. For average case analysis of tree searching algorithms, we recommend Brown and Purdom (1981); Huyn, Dechter and Pearl (1980); Karp and Pearl (1983); and Purdom and Brown (1985). For branch-and-bound algorithms, we recommend Boffey and Green (1983); Hariri and Potts (1983); Ibaraki (1977); Sen and Sherali (1985); Smith (1984); and Wah and Yu (1985). For A^* algorithms, we recommend Bagchi and Mahanti (1983); Dechter and Pearl (1985); Nau, Kumar and Kanal (1984); Pearl (1983); and Srimani (1989).

For recent results, consult Ben-Asher, Farchi and Newman (1999); Devroye (2002); Devroye and Robson (1995); Gallant, Marier and Storer (1980); Giancarlo and Grossi (1997); Kirschenhofer, Prodinger and Szpankowski (1994); Kou, Markowsky and Berman (1981); Lai and Wood (1998); Lew and Mahmoud (1992); Louchard, Szpankowski and Tang (1999); Lovasz, Naor, Newman and Wigderson (1995); and Meleis (2001).

Exercises

5.1 Consider the following graph. Find a Hamiltonian cycle by some kind of tree searching technique.



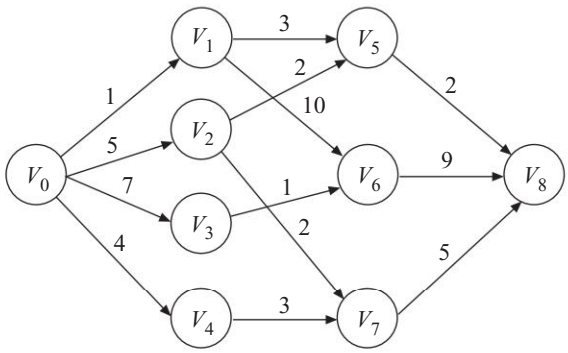
5.2 Solve the 8-puzzle problem which tests with the following initial state.

2	3	
8	1	4
7	5	6

Note that our final goal is

1	2	3
8		4
7	6	5

5.3 Find the shortest path from v_0 to v_8 by the branch-and-bound strategy.

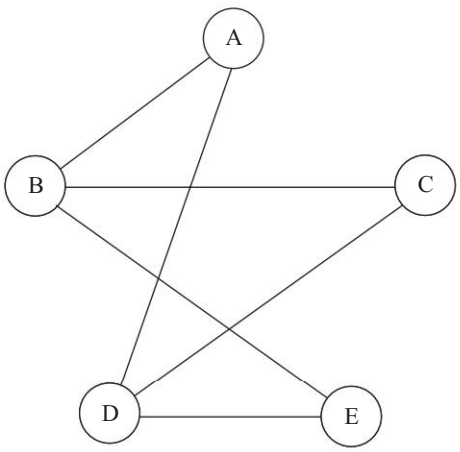


5.4 Solve the traveling salesperson problem characterized by the following distance matrix by the branch-and-bound strategy.

i \ *j*	1	2	3	4	5
1	∞	5	61	34	12
2	57	∞	43	20	7
3	39	42	∞	8	21
4	6	50	42	∞	8
5	41	26	10	35	∞

5.5 Solve the following sum of subset problem. $S = \{7, 1, 4, 6, 14, 25, 5, 8\}$ and $M = 18$. Find a sum whose elements add up to M. Use the branch-and-bound strategy.

5.6 Solve the vertex cover problem of the following graph by some tree searching technique.



5.7 Determine the satisfiability of the following Boolean formulas by tree searching technique.

(a) $-X_1 \vee X_2 \vee X_3$
$X_1 \vee X_3$
X_2

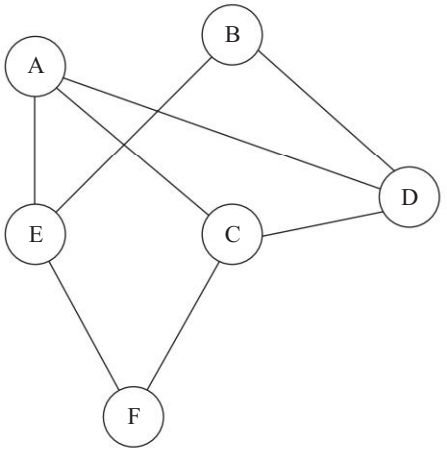
(b) $-X_1 \vee X_2 \vee X_3$
$X_1 \vee X_2$
$-X_2 \vee X_3$
$-X_3$

(c) $X_1 \vee X_2 \vee X_3$
$-X_1 \vee -X_2 \vee -X_3$

(d) $X_1 \vee X_2$
$-X_2 \vee X_3$
$-X_3$

5.8 Consider the graph in Problem 5.6. Is this graph 2-colorable? Answer this question by some kind of tree searching technique.

5.9 Consider the following graph. Prove that this graph contains a 3-clique by tree searching.



5.10 Design an experiment to test the average case performance of the algorithm solving the traveling salesperson problem based upon the branch-and-bound strategy.

chapter

6

Prune-and-Search

In this chapter, we shall discuss an elegant strategy of algorithm design, called prune-and-search. This approach can be used to solve a lot of problems, especially optimization problems, and usually gives efficient algorithms for solving these problems. For instance, when the dimension is fixed, we can use the prune-and-search strategy to solve the linear programming problem with n constraints in linear time. In the following sections, we shall first introduce the general steps of the prune-and-search strategy and then discuss some problems that can be solved efficiently by this strategy.

6–1 The General Method

The prune-and-search strategy always consists of several iterations. At each iteration, it prunes away a fraction, say f, of the input data, and then it invokes the same algorithm recursively to solve the problem for the remaining data. After p iterations, the size of input data will be q, which is so small that the problem can be solved directly in some constant time c'. The time-complexity analysis for this kind of algorithms is as follows: Assume that the time needed to execute the prune-and-search in each iteration is $O(n^k)$ for some constant k, and the worst case run time of the prune-and-search algorithm is $T(n)$. Then

$$T(n) = T((1-f)n) + O(n^k).$$

We have

$$\begin{aligned} T(n) &\le T((1-f)n) + cn^k \qquad \text{for sufficiently large } n. \\ &\le T((1-f)^2 n) + cn^k + c(1-f)^k n^k \\ &\ \ \vdots \end{aligned}$$

$$\leq c' + cn^k + c(1-f)^k n^k + c(1-f)^{2k} n^k + \cdots + c(1-f)^{pk} n^k$$
$$= c' + cn^k(1 + (1-f)^k + (1-f)^{2k} + \cdots + (1-f)^{pk}).$$

Since $1 - f < 1$, as $n \to \infty$,

$$T(n) = O(n^k).$$

The above formula indicates that the time complexity of the whole prune-and-search process is of the same order as the time complexity of the prune-and-search in each iteration.

6–2 THE SELECTION PROBLEM

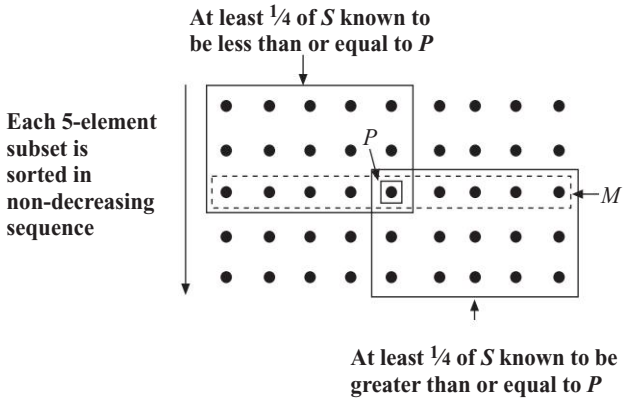
In this problem, we are given n elements and are required to determine the kth smallest element. One possible solution to this problem is to sort the n elements and locate the kth element in the sorted list. The worst-case time complexity of this approach is $O(n \log n)$ time because sorting time complexity is $O(n \log n)$ and this dominates. In this section, we shall show that the selection problem can be solved in linear time by the prune-and-search strategy. The median problem, which is to find the $\lceil n/2 \rceil$th smallest element, is a special case of the selection problem. Thus, the median problem can also be solved in linear time.

The basic idea of this linear-time prune-and-search selection algorithm is to determine a fraction of elements which will not contain the kth element and discard the fraction of elements during each iteration. From Section 6–1, we know that in order to have an $O(n)$ algorithm, we must have a method which prunes away a fraction of elements in $O(n)$ time in each iteration. We shall show that in each iteration, the time needed for searching is negligible.

Let S be the set of input data. Let p be an element of S and we can partition S into three subsets S_1, S_2 and S_3 such that S_1 contains all elements less than p, S_2 contains all elements equal to p, and S_3 contains all elements greater than p. If the size of S_1 is greater than k, then we can be sure that the kth smallest element of S is located in S_1, and we can prune away S_2 and S_3 during the next iteration. Otherwise, if the number of elements in S_1 and S_2 is greater than k, p is the kth smallest element of S. If none of the conditions stated above is satisfied, then the kth smallest element of S must be greater than p. Hence, we can discard S_1 and S_2 and start the next iteration which selects the $(k - |S_1| - |S_2|)$-th smallest element from S_3.

The key point is how to select p such that we can always discard a fraction of S, no matter whether we prune away S_1, S_2 or S_3, during each iteration. We can select p as follows: First of all, we divide the n elements into $\lceil n/5 \rceil$ subsets of five elements each, adding some dummy ∞ elements, if needed, to complete the last subset. Then we sort each of the 5-element subsets. The median of each subset is selected to form a new sequence $M = \{m_1, m_2, \ldots, m_{\lceil n/5 \rceil}\}$ and let p be the median of M. As shown in Figure 6–1, at least one-fourth of the elements in S are less than or equal to p and at least one-fourth of the elements in S are greater than or equal to p.

FIGURE 6–1 The pruning of points in the selection procedure.



Thus, if we choose p in this way, we can always prune away at least $n/4$ elements from S during each iteration. Now we can state the algorithm.

Algorithm 6–1 □ A prune-and-search algorithm to find the *k*th smallest element

Input: A set S of n elements.
Output: The kth smallest element of S.
Step 1. If $|S| \leq 5$, solve the problem by any brute force method.
Step 2. Divide S into $\lceil n/5 \rceil$ subsets. Each subset contains five elements. Add some dummy ∞ elements to the last subset if n is not a net multiple of 5.
Step 3. Sort each subset of elements.
Step 4. Recursively find the element p which is the median of the medians of the $\lceil n/5 \rceil$ subsets.
Step 5. Partition S into S_1, S_2 and S_3, which contain the elements less than, equal to, and greater than p, respectively.

Step 6. If $|S_1| \geq k$, then discard S_2 and S_3 and solve the problem that selects the kth smallest element from S_1 during the next iteration; else if $|S_1| + |S_2| \geq k$ then p is the kth smallest element of S; otherwise, let $k' = k - |S_1| - |S_2|$. Solve the problem that selects the k'th smallest element from S_3 during the next iteration.

Let $T(n)$ be the time complexity of the algorithm stated above to select the kth element from S. Since each subset contains a constant number of elements, for each subset, sorting takes constant time. Thus, Steps 2, 3 and 5 can be done in $O(n)$ time and Step 4 needs $T(\lceil n/5 \rceil)$ time if we use the same algorithm recursively to find the median of the $\lceil n/5 \rceil$ elements. Because we always prune away at least $n/4$ elements during each iteration, the problem remaining in Step 6 contains at most $3n/4$ elements and therefore can be accomplished in $T(3n/4)$ time. Hence,

$$T(n) = T(3n/4) + T(n/5) + O(n).$$

Let $T(n) = a_0 + a_1 n + a_2 n^2 + \cdots$, $a_1 \neq 0$.

We have

$$T\left(\frac{3}{4}n\right) = a_0 + \frac{3}{4}a_1 n + \frac{9}{16}a_2 n^2 + \cdots$$

$$T\left(\frac{1}{5}n\right) = a_0 + \frac{1}{5}a_1 n + \frac{1}{25}a_2 n^2 + \cdots$$

$$T\left(\frac{3}{4}n + \frac{1}{5}n\right) = T\left(\frac{19}{20}n\right) = a_0 + \frac{19}{20}a_1 n + \frac{361}{400}a_2 n^2 + \cdots.$$

Thus,

$$T\left(\frac{3}{4}n\right) + T\left(\frac{1}{5}n\right) \leq a_0 + T\left(\frac{19}{20}n\right).$$

Therefore,

$$T(n) = T\left(\frac{3}{4}n\right) + T\left(\frac{1}{5}n\right) + O(n)$$

$$\leq T\left(\frac{19}{20}n\right) + cn.$$

Applying the formula obtained in Section 6–1 to this inequality, we get

$T(n) = O(n)$.

Thus, we have a linear-time worst case algorithm for solving the selection problem based on the prune-and-search strategy.

6–3 LINEAR PROGRAMMING WITH TWO VARIABLES

The computational complexity of linear programming has been a subject of great interest to computer scientists for a very long time. Although a brilliant algorithm was proposed by Khachian, who showed that the linear programming problem can be solved in polynomial time, it is of theoretical interest because the constant involved is so large. Megiddo and Dyer independently proposed a prune-and-search strategy to solve the linear programming problem with a fixed number of variables in $O(n)$ time where n is the number of constraints.

In this section, we shall describe their elegant techniques to solve the linear programming problem with two variables. The special linear programming problem with two variables is defined as follows:

Minimize $ax + by$

Subject to $a_i x + b_i y \geq c_i, \quad i = 1, 2, \ldots, n.$

The basic idea of the prune-and-search strategy to solve the linear programming problem with two variables is that there are always some constraints which have nothing to do with the solution and therefore can be pruned away. In the prune-and-search method, a fraction of constraints are pruned away after every iteration. After several iterations, the number of constraints will be so small that the linear programming problem can be solved in some constant time.

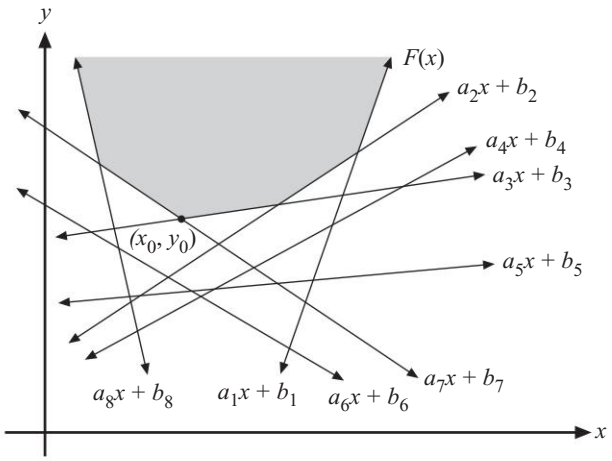
In order to simplify the discussion, let us describe the prune-and-search method to solve a simplified two-variable linear programming problem:

Minimize y

Subject to $y \geq a_i x + b_i, \quad i = 1, 2, \ldots, n.$

Consider Figure 6–2. In Figure 6–2, there are eight constraints in total and (x_0, y_0) is the optimum solution.

FIGURE 6–2 An example of the special 2-variable linear programming problem.



Since

$$y \geq a_i x + b_i \quad \text{for all } i,$$

we know that this optimum solution must be on the boundary surrounding the feasible region, as shown in Figure 6–2. For each x, the boundary $F(x)$ must have the highest value among all n constraints. That is,

$$F(x) = \max_{1 \leq i \leq n} \{a_i x + b_i\}$$

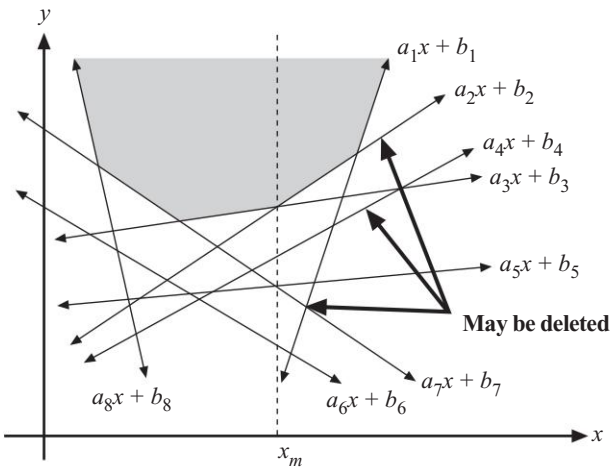
where $a_i x + b_i$ is an input constraint. The optimum solution x_0 satisfies the following equation:

$$F(x_0) = \min_{-\infty \leq x \leq \infty} F(x).$$

Let us now assume the following:

(1) We have picked a point x_m, as shown in Figure 6–3.
(2) We, by some reasoning, understand that $x_0 \leq x_m$.

FIGURE 6–3 Constraints which may be eliminated in the 2-variable linear programming problem.



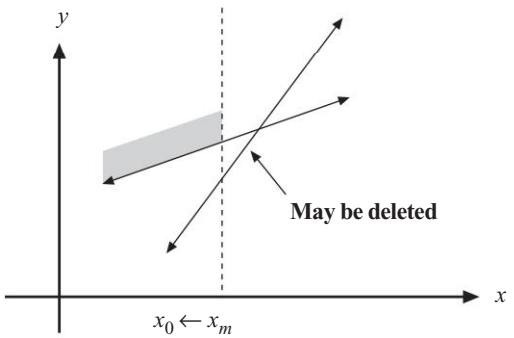
In this case, consider

$$y = a_1x + b_1,$$

and $y = a_2x + b_2$.

The intersection of these two lines is to the right of x_m. Among these two straight lines, one of them is less than the other for $x < x_m$. This constraint may be deleted because it cannot be a part of the boundary, as illustrated in Figure 6–4.

FIGURE 6–4 An illustration of why a constraint may be eliminated.



Thus, if x_m is selected and we know that $x_0 < x_m$, then $a_1x + b_1$ may be deleted.

Similarly, since the intersection of $a_3x + b_3$ and $a_4x + b_4$ is to the right of x_m, $x_0 < x_m$ and $a_4x + b_4 < a_3x + b_3$ for $x \le x_m$, we may delete the constraint $a_4x + b_4$.

In general, if there is an x_m, such that the optimum solution $x_0 < x_m$ and the intersection of two input constraints $a_1x + b_1$ and $a_2x + b_2$ is to the right of x_m, then one of these constraints is always smaller than the other for $x < x_m$. Therefore, this constraint may be deleted because it will not affect the search for the optimum solution.

The reader must note that the elimination of the constraint $a_1x + b_1$ may be done only after we know that $x_0 < x_m$. In other words, suppose that we are in a searching process and searching along the direction from x_m to x_0 along the boundary, then we can eliminate $a_1x + b_1$ because where $x < x_m$ is concerned, $a_1x + b_1$ will have no contribution to the result of the search. We must note that $a_1x + b_1$ is a part of the boundary of the feasible region. But this occurs for $x > x_m$.

If $x_0 > x_m$, then for every pair of constraints whose intersection is to the left of x_m, there is a constraint which is smaller than the other for $x > x_m$ and this constraint may be deleted without affecting the solution.

We still have to answer the following two questions:

(1) Suppose an x_m is known. How do we know the direction of searching? That is, how do we know whether $x_0 < x_m$ or $x_0 > x_m$?
(2) How do we choose x_m?

Let us answer the first question. Suppose x_m is chosen. Let

$$y_m = F(x_m) = \max_{1 \le i \le n} \{a_i x_m + b_i\}.$$

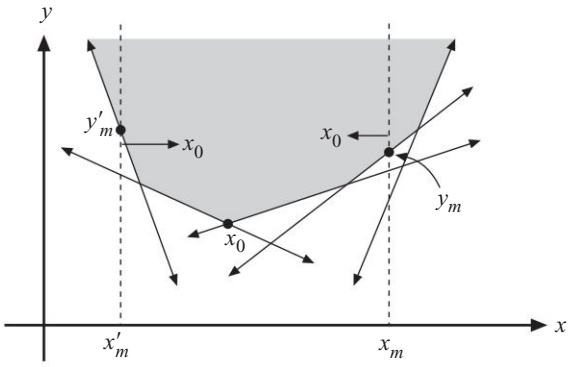
Obviously, (x_m, y_m) is a point on the boundary of the feasible region. There are two possibilities:

Case 1. y_m is on only one constraint.

Case 2. y_m is the intersection of several constraints.

For Case 1, we simply check the slope g of this constraint. If $g > 0$, then $x_0 < x_m$. If $g < 0$, then $x_0 > x_m$. These are illustrated in Figure 6–5.

FIGURE 6–5 The cases where x_m is on only one constraint.



For Case 2, we calculate the following:

$$g_{\max} = \max_{1 \le i \le n} \{a_i : a_i x_m + b_i = F(x_m)\}$$

$$g_{\min} = \min_{1 \le i \le n} \{a_i : a_i x_m + b_i = F(x_m)\}.$$

In other words, among all the constraints which intersect with one another at (x_m, y_m), let $g_{\min}$ and $g_{\max}$ be the minimum slope and the maximum slope of these constraints respectively. Then, there are three possibilities:

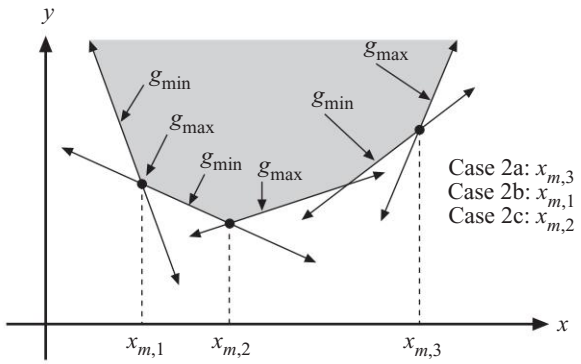
(1) Case 2a. $g_{\min} > 0$ and $g_{\max} > 0$. In this case, $x_0 < x_m$.
(2) Case 2b. $g_{\min} < 0$ and $g_{\max} < 0$. In this case, $x_0 > x_m$.
(3) Case 2c. $g_{\min} < 0$ and $g_{\max} > 0$. In this case, (x_m, y_m) is the optimum solution.

We show these three cases in Figure 6–6.

We have answered the first question. Now, let us answer the second question: How are we going to choose x_m? *The x_m should be so chosen that half the intersections between paired lines in the chosen pairing lie to the right of it and half of this kind of intersections lie to the left of it. This means that for n constraints, we may arbitrarily group them into n/2 pairs. For each pair, find their intersection. Among these n/2 intersections, let x_m be the median of their x-coordinates*.

In Algorithm 6–2, we shall give the precise algorithm based on the prune-and-search strategy to find an optimum solution for the special linear programming problem.

FIGURE 6–6 Cases of x_m on the intersection of several constraints.



Algorithm 6–2 □ A prune-and-search algorithm to solve a special linear programming problem

Input: Constraints: $S: a_i x + b_i, \quad i = 1, 2, \ldots, n.$

Output: The value x_0 such that y is minimized at x_0 subject to $y \geq a_i x + b_i, \quad i = 1, 2, \ldots, n.$

Step 1. If S contains no more than two constraints, solve this problem by a brute force method.

Step 2. Divide S into $n/2$ pairs of constraints. For each pair of constraints $a_i x + b_i$ and $a_j x + b_j$, find the intersection p_{ij} of them and denote its x-value as x_{ij}.

Step 3. Among the x_{ij}'s (at most $n/2$ of them), find the median x_m.

Step 4. Determine $y_m = F(x_m) = \max\limits_{1 \leq i \leq n} \{a_i x_m + b_i\}$

$$g_{\min} = \min_{1 \leq i \leq n} \{a_i : a_i x_m + b_i = F(x_m)\}$$

$$g_{\max} = \max_{1 \leq i \leq n} \{a_i : a_i x_m + b_i = F(x_m)\}.$$

Step 5. Case 5a: If $g_{\min}$ and $g_{\max}$ are not of the same sign, y_m is the solution and exit.

Case 5b: Otherwise, $x_0 < x_m$, if $g_{\min} > 0$, and $x_0 > x_m$, if $g_{\min} < 0$.

Step 6. Case 6a: If $x_0 < x_m$, for each pair of constraints whose x-coordinate intersection is larger than x_m, prune away the constraint which is always smaller than the other for $x \leq x_m$.

Case 6b: If $x_0 > x_m$, for each pair of constraints whose x-coordinate intersection is less than x_m, prune away the constraint which is always smaller than the other for $x \geq x_m$.
Let S denote the remaining of constraints. Go to Step 2.

The complexity of the above algorithm is analyzed as follows: Since the intersection of two straight lines can be found in constant time, Step 2 can be done in $O(n)$ time. The median can be found in $O(n)$ time as shown in Section 6–2. Step 4 can be done by linearly scanning all these constraints. Hence, Steps 4 and 5 can be completed in $O(n)$ time. Step 6 can also be done in $O(n)$ time by scanning all intersecting pairs.

Since we use the median of chosen intersections, half of them lie to the right of x_m. There are $\lfloor n/2 \rfloor$ intersections in total. For each intersection, one constraint is pruned away. Thus, $\lfloor n/4 \rfloor$ constraints are pruned away for each iteration. Based upon the result in Section 6–1, the time complexity of the above algorithm is $O(n)$.

We now return to the original 2-variable linear programming problem, which is defined as follows:

Minimize $Z = ax + by$
Subject to $a_i x + b_i y \geq c_i$ $(i = 1, 2, \ldots, n)$.

A typical example is now shown below:

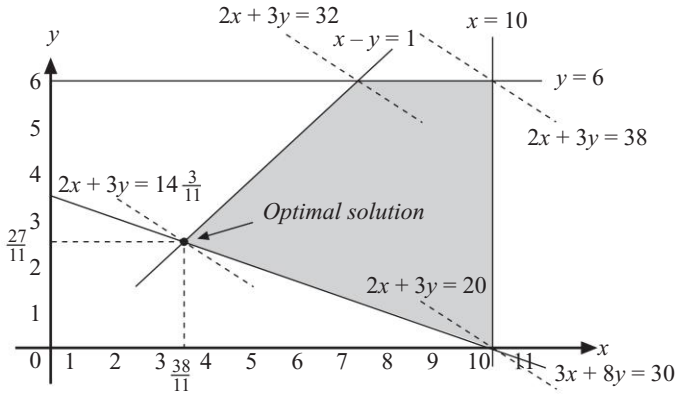
Minimize $Z = 2x + 3y$
Subject to $x \leq 10$
$y \leq 6$
$x - y \geq 1$
$3x + 8y \geq 30$.

The constraints form a feasible region, shown in Figure 6–7.
In Figure 6–7, each dashed-line is represented as

$Z = 2x + 3y$.

We can imagine that these straight lines form a sequence of parallel lines. The optimum solution is located at (38/11, 27/11) where the first dashed-line intersects with the feasible region.

FIGURE 6–7 A general 2-variable linear programming problem.



We may transform the above general 2-variable linear programming problem into a problem of minimizing the y-value only. Our original problem is

Minimize $Z = ax + by$

Subject to $a_i x + b_i y \geq c_i$ $(i = 1, 2, \ldots, n)$.

We now let $x' = x$

$y' = ax + by$.

Then this problem becomes

Minimize y'

Subject to $a_i' x' + b_i' y' \geq c_i'$ $(i = 1, 2, \ldots, n)$

where $a_i' = a_i - b_i a/b$

$b_i' = b_i/b$

and $c_i' = c_i$.

Therefore, we may say that a general 2-variable problem can be transformed, after n steps, to the following problem.

Minimize y

Subject to $a_i x + b_i y \geq c_i$ $(i = 1, 2, \ldots, n)$.

The reader should note that there is a difference between the above problem and the special 2-variable linear programming problem. In the special 2-variable linear programming problem, we defined the problem to be

Minimize y
Subject to $y \geq a_i x + b_i$ $(i = 1, 2, \ldots, n)$.

But, in the general case, there are three kinds of constraints:

$$y \geq a_i x + b_i$$
$$y \leq a_i x + b_i$$
and $a \leq x \leq b$.

Consider the four constraints in Figure 6–7. They are

$x \leq 10$
$y \leq 6$
$x - y \geq 1$
$3x + 8y \geq 30$.

We may rewrite the above equations as

$x \leq 10$
$y \leq 6$
$y \leq x - 1$
$y \geq \frac{-3}{8} x + \frac{30}{8}$.

We may group the constraints with positive (negative) coefficients for variable y in set $I_1(I_2)$. Then the original problem becomes

Minimize y
Subject to $y \geq a_i x + b_i$ $(i \in I_1)$
$y \leq a_i x + b_i$ $(i \in I_2)$
$a \leq x \leq b$.

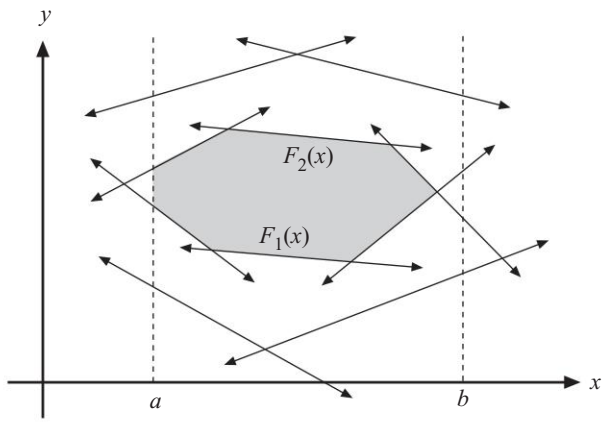
Let us define two functions:

$F_1(x) = \max\{a_i x + b_i : i \in I_1\}$
$F_2(x) = \min\{a_i x + b_i : i \in I_2\}$.

$F_1(x)$ is a convex piecewise linear function, $F_2(x)$ is a concave piecewise linear function and these two functions together with the constraint $a \leq x \leq b$

define the feasible region of the linear programming problem, shown in Figure 6–8.

FIGURE 6–8 A feasible region of the 2-variable linear programming problem.



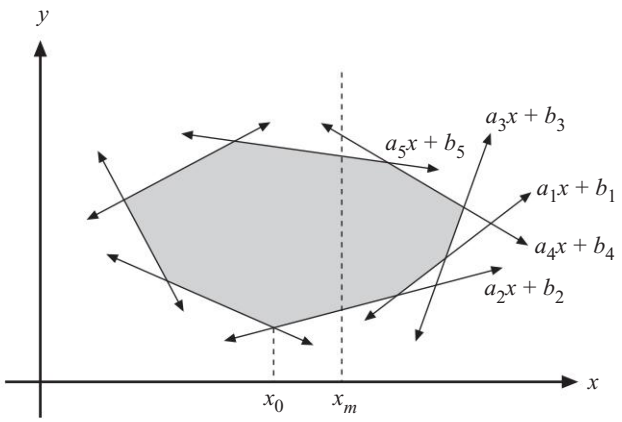
Thus, the original 2-variable linear programming problem can be further transformed to

$$\begin{aligned} &\text{Minimize} && F_1(x) \\ &\text{Subject to} && F_1(x) \le F_2(x) \\ & && a \le x \le b. \end{aligned}$$

Again, we would like to point out here that some constraints may be pruned away if we know the direction of searching. This reason is exactly the same as that used in the discussion of the special 2-variable linear programming problem. Consider Figure 6–9. If we know that the optimal solution x_0 lies to the left of x_m, then $a_1x + b_1$ can be eliminated without affecting the solution because $a_1x + b_1 < a_2x + b_2$ for $x < x_m$. We know this fact because the intersection of $a_1x + b_1$ and $a_2x + b_2$ lies to the right of x_m. Similarly, $a_4x + b_4$ can be eliminated because $a_4x + b_4 > a_5x + b_5$ for $x < x_m$.

The reader will now note that the most important issue in the prune-and-search algorithm to solve the 2-variable linear programming problem is to decide whether x_0 (the optimum solution) lies to the left, or right, of x_m. The solution of

FIGURE 6–9 The pruning of constraints for the general 2-variable linear programming problem.



x_m can be found in a way similar to that in the special two-variable linear programming problem.

In general, given a point x_m, $a \le x_m \le b$, we have to decide the following:

(1) Is x_m feasible?
(2) If x_m is feasible, we must decide whether the optimum solution x_0 lies to the left, or right, of x_m It may also happen that x_m itself is the optimum solution.
(3) If x_m is infeasible, we must decide whether any feasible solution exists. If a feasible solution exists, we must decide which side of x_m this optimum solution exists.

Next, we shall describe the decision process. Consider $F(x) = F_1(x) - F_2(x)$. Obviously, x_m is feasible if and only if $F(x_m) \le 0$. In order to decide which side the optimum solution lies, let us define the following:

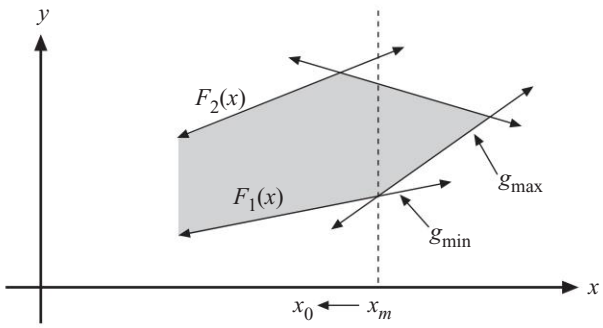
$$g_{\min} = \min\{a_i : i \in I_1, a_i x_m + b_i = F_1(x_m)\}$$
$$g_{\max} = \max\{a_i : i \in I_1, a_i x_m + b_i = F_1(x_m)\}$$
$$h_{\min} = \min\{a_i : i \in I_2, a_i x_m + b_i = F_2(x_m)\}$$
$$h_{\max} = \max\{a_i : i \in I_2, a_i x_m + b_i = F_2(x_m)\}.$$

We consider the following cases:

Case 1. $F(x_m) \le 0$.
This means that x_m is feasible.

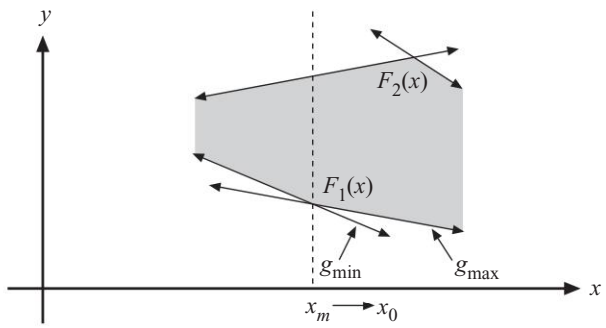
(1) If $g_{\min} > 0$ and $g_{\max} > 0$, then $x_0 < x_m$, shown in Figure 6–10.

FIGURE 6–10 The case where $g_{\min} > 0$ and $g_{\max} > 0$.



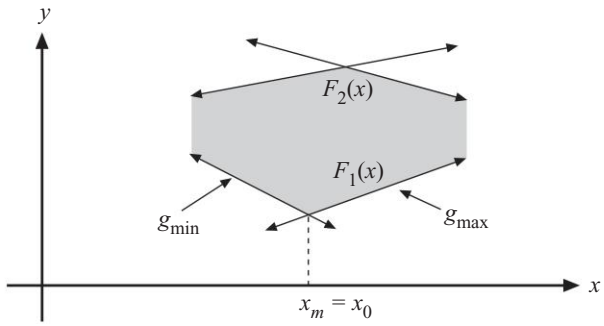
(2) If $g_{\max} < 0$ and $g_{\min} < 0$, then $x_0 > x_m$, shown in Figure 6–11.

FIGURE 6–11 The case where $g_{\max} < 0$ and $g_{\min} < 0$.



(3) If $g_{\min} < 0$ and $g_{\max} > 0$, then x_m is the optimum solution, shown in Figure 6–12.

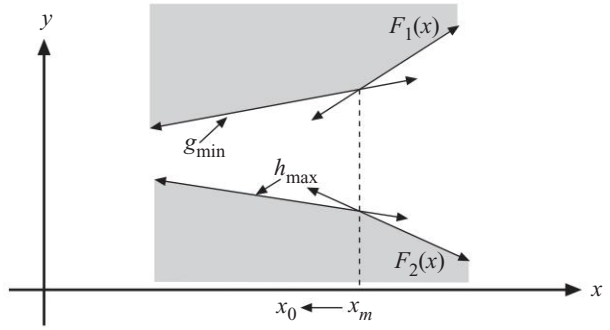
FIGURE 6–12 The case where $g_{\min} < 0$ and $g_{\max} > 0$.



Case 2. $F(x_m) > 0$.
This means that x_m is infeasible.

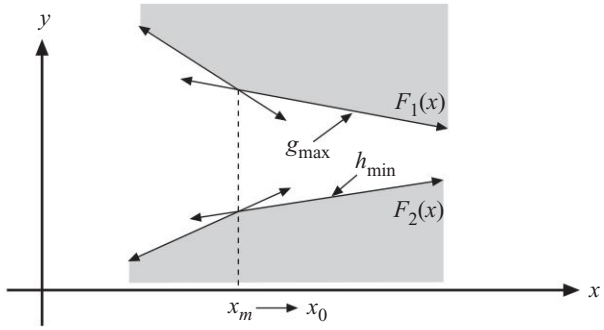
(1) If $g_{\min} > h_{\max}$, then $x_0 < x_m$, shown in Figure 6–13.

FIGURE 6–13 The case where $g_{\min} > h_{\max}$.



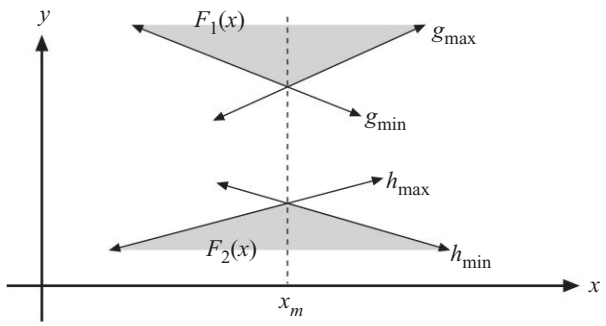
(2) If $g_{\max} < h_{\min}$, then $x_0 > x_m$, shown in Figure 6–14.

FIGURE 6–14 The case where $g_{\max} < h_{\min}$.



(3) If $(g_{\min} \le h_{\max})$ and $(g_{\max} \ge h_{\min})$, then no feasible solution exists because $F(x)$ attains its minimum at x_m. This is shown in Figure 6–15.

FIGURE 6–15 The case where $(g_{\min} \le h_{\max})$ and $(g_{\max} \ge h_{\min})$.



The above decision procedure is summarized in the following procedure.

Procedure 6–1 □ A procedure used in Algorithm 6–3

Input: x_m, the value of the x-coordinate of a point.

I_1: $y \ge a_i x + b_i$, $i = 1, 2, \ldots, n'_1$

I_2: $y \le a_i x + b_i$, $i = n'_1 + 1, n'_1 + 2, \ldots, n'$

$a \le x \le b$.

Output: Whether it is meaningful to continue searching from x_m and if yes, the direction of the searching.

Step 1. $F_1(x) = \max\{a_i x + b_i : i \in I_1\}$

$F_2(x) = \min\{a_i x + b_i : i \in I_2\}$

$F(x) = F_1(x) - F_2(x)$.

Step 2. $g_{\min} = \min\{a_i : i \in I_1,\ a_i x_m + b_i = F_1(x_m)\}$

$g_{\max} = \max\{a_i : i \in I_1,\ a_i x_m + b_i = F_1(x_m)\}$

$h_{\min} = \min\{a_i : i \in I_2,\ a_i x_m + b_i = F_2(x_m)\}$

$h_{\max} = \max\{a_i : i \in I_2,\ a_i x_m + b_i = F_2(x_m)\}$.

Step 3. **Case 1:** $F(x_m) \le 0$.

(a) If $g_{\min} > 0$ and $g_{\max} > 0$, report "$x_0 < x_m$" and exit.

(b) If $g_{\min} < 0$ and $g_{\max} < 0$, report "$x_0 > x_m$" and exit.

(c) If $g_{\min} < 0$ and $g_{\max} > 0$, report "x_m is the optimal solution" and exit.

Case 2: $F(x_m) > 0$.

(a) If $g_{\min} > h_{\max}$, report "$x_0 < x_m$" and exit.

(b) If $g_{\max} < h_{\min}$, report "$x_0 > x_m$" and exit.

(c) If $(g_{\min} \le h_{\max})$ and $(g_{\max} \ge h_{\min})$, report "no feasible solution exists" and exit.

The prune-and-search algorithm to solve the 2-variable linear programming problem follows.

Algorithm 6–3 □ A prune-and-search algorithm to solve the 2-variable linear programming problem

Input: I_1: $y \ge a_i x + b_i$, $i = 1, 2, \ldots, n_1$.

I_2: $y \le a_i x + b_i$, $i = n_1 + 1, n_1 + 2, \ldots, n$.

$a \le x \le b$.

Output: The value x_0 such that y is minimized at x_0 subject to

$y \ge a_i x + b_i$, $i = 1, 2, \ldots, n_1$.

$y \le a_i x + b_i$, $i = n_1 + 1, n_1 + 2, \ldots, n$.

$a \le x \le b$.

Step 1. If there are no more than two constraints in I_1 and I_2, then solve this problem by a brute force method.

Step 2. Arrange the constraints in I_1 into disjoint pairs and, similarly, arrange constraints in I_2 into disjoint pairs. For each pair, if $a_i x + b_i$ is parallel to $a_j x + b_j$, delete $a_i x + b_i$ if $b_i < b_j$ for $i, j \in I_1$ or $b_i > b_j$ for $i, j \in I_2$. Otherwise, find the intersection p_{ij} of $y = a_i x + b_i$ and $y = a_j x + b_j$. Let the x-coordinate of p_{ij} be x_{ij}.

Step 3. Find the median x_m of x_{ij}'s.

Step 4. Apply Procedure 6–1 to x_m.
If x_m is the optimal, report x_m and exit.
If no feasible solution exists, report this and exit.

Step 5. If $x_0 > x_m$
For each $x_{ij} < x_m$ and $i, j \in I_1$, prune constraint $y \geq a_i x + b_i$ if $a_i < a_j$;
otherwise, prune constraint $y \geq a_j x + b_j$.
For each $x_{ij} < x_m$ and $i, j \in I_2$, prune constraint $y \leq a_i x + b_i$ if $a_i > a_j$;
otherwise, prune constraint $y \leq a_j x + b_j$.
If $x_0 < x_m$
For each $x_{ij} > x_m$ and $i, j \in I_1$, prune constraint $y \geq a_i x + b_i$ if $a_i > a_j$;
otherwise, prune constraint $y \geq a_j x + b_j$.
For each $x_{ij} > x_m$ and $i, j \in I_2$, prune constraint $y \leq a_i x + b_i$ if $a_i < a_j$;
otherwise, prune constraint $y \leq a_j x + b_j$.

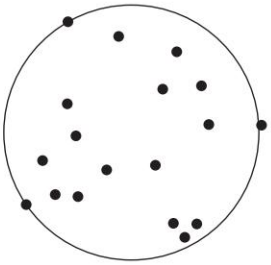
Step 6. Go to Step 1.

Since we can prune $\lfloor n/4 \rfloor$ constraints for each iteration, and each step in Algorithm 6–3 takes $O(n)$ time, it is easy to see that this algorithm is of the order $O(n)$.

6–4 THE 1-CENTER PROBLEM

The 1-center problem is defined as follows: We are given a set of n planar points and our job is to find a smallest circle to cover all of these n points. A typical example is shown in Figure 6–16.

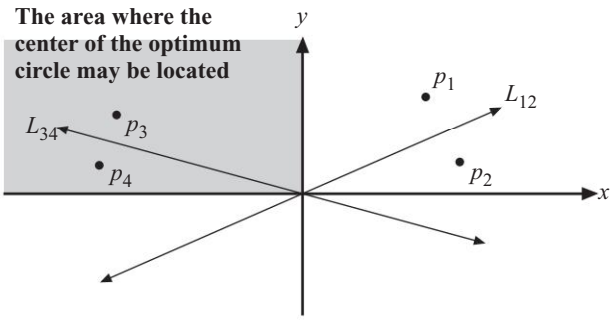
FIGURE 6–16 The 1-center problem.



In this section, we shall show a strategy to solve the 1-center problem which is based upon the prune-and-search strategy. For this problem, the input data are these n points. During each stage, we prune away $1/16$ of the points which will not affect our solution. The prune-and-search algorithm for the 1-center problem takes only linear time as it requires linear time for each stage.

Let us illustrate our prune-and-search algorithm by considering Figure 6–17. In Figure 6–17, L_{12} and L_{34} are bisectors of segments connecting p_1 and p_2, and p_3 and p_4 respectively. Suppose we also know that the center of the optimum circle must be in the shaded area. Then we consider L_{12}. Since L_{12} does not intersect with this shaded area, there must be one point which is closer to this optimum center than the other. In our case, p_1 is closer to the optimum center than p_2. Thus, p_1 may be eliminated because it will not affect our solution.

FIGURE 6–17 A possible pruning of points in the 1-center problem.



Essentially, we need to know the region where the optimum center is located and we must be certain that some points are closer to this optimum center than the others. These closer points may therefore be eliminated.

Before we present the general 1-center algorithm, we first introduce an algorithm for *the constrained 1-center problem where the center is restricted to lying on a straight line*. This will then be used as a subroutine called by the general 1-center algorithm. Let us assume, without losing generality, that the straight line is $y = y'$.

Algorithm 6–4 □ An algorithm to solve the constrained 1-center problem

Input: n points and a straight line $y = y'$.

Output: The constrained 1-center on the straight line $y = y'$.

Step 1. If n is no more than 2, solve this problem by a brute force method.

Step 2. Form disjoint pairs of points (p_1, p_2), (p_3, p_4), … , (p_{n-1}, p_n). If there are odd number of points, then just let the final pair be (p_n, p_1).

Step 3. For each pair of points, (p_i, p_{i+1}), find the point $x_{i,i+1}$ on the line $y = y'$ such that $d(p_i, x_{i,i+1}) = d(p_{i+1}, x_{i,i+1})$.

Step 4. Find the median of the $\lceil n/2 \rceil$ $x_{i,i+1}$'s. Denote it by x_m.

Step 5. Calculate the distance between p_i and x_m for all i. Let p_j be the point which is the farthest from x_m. Let x_j denote the projection of p_j onto $y = y'$. If x_j is to the left (right) of x_m, then the optimal solution, x^*, must be to the left (right) of x_m.

Step 6. If $x^* < x_m$ (illustrated in Figure 6–18)

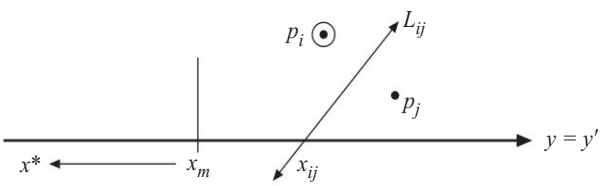
For each $x_{i,i+1} > x_m$, prune away the point p_i if p_i is closer to x_m than p_{i+1}; otherwise, prune away the point p_{i+1}.

If $x^* > x_m$

For each $x_{i,i+1} < x_m$, prune away the point p_i if p_i is closer to x_m than p_{i+1}; otherwise, prune away the point p_{i+1}.

Step 7. Go to Step 1.

FIGURE 6–18 The pruning of points in the constrained 1-center problem.



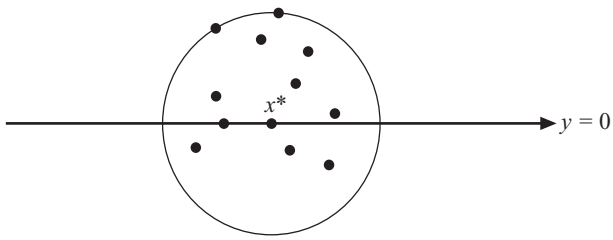
Since there are $\lfloor n/4 \rfloor$ $x_{i,i+1}$'s lying in the left (right) side of x_m, we can prune away $\lfloor n/4 \rfloor$ points for each iteration and algorithm. Each iteration takes $O(n)$ time. Hence, the time complexity of this algorithm is

$$T(n) = T(3n/4) + O(n)$$
$$= O(n).$$

It should be emphasized here that the constrained 1-center algorithm will be used in the main algorithm to find an optimal solution. The points pruned away in the process of executing this constrained 1-center problem will be still used by the main algorithm.

Let us now consider a more complicated problem. Imagine that we have a set of points and a line $y = 0$, as shown in Figure 6–19. Using the constrained 1-center algorithm, we can determine the exact location of x^* on this line. Actually, using this information, we can do more. Let (x_s, y_s) be the center of the optimum circle containing all points. We can now determine whether $y_s > 0$, $y_s < 0$ or $y_s = 0$. By the same token, we can also determine whether $x_s > 0$, $x_s < 0$ or $x_s = 0$.

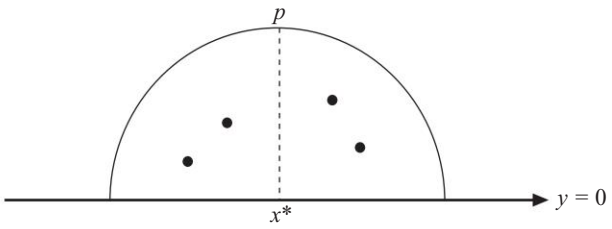
FIGURE 6–19 Solving a constrained 1-center problem for the 1-center problem.



Let I be the set of points which are farthest from $(x^*, 0)$. There are two possible cases.

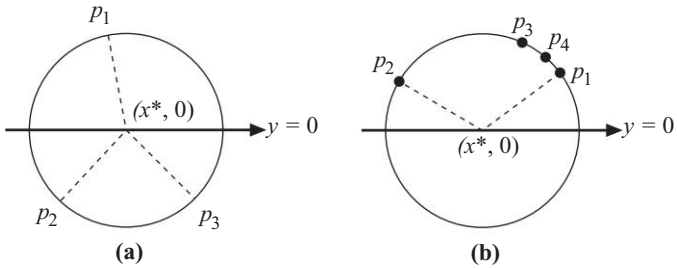
Case 1. I contains one point, say p.
In such a case, the x-value of p must be equal to x^*. Assume otherwise. Then we can move x^* towards p along the line $y = 0$. This contradicts with our assumption that $(x^*, 0)$ is an optimal solution on the line $y = 0$. Therefore, if p is the only farthest point of $(x^*, 0)$, its x-value must be equal to x^*, shown in Figure 6–20. *We conclude that* y_s *has the same sign as that of the y-value of p.*

FIGURE 6–20 The case where *I* contains only one point.



Case 2. *I* contains more than one point.
Among all the points in Set I, find the smallest arc spanning all the points in I. Let the two end points of this arc be p_1 *and* p_2. *If this arc is of degree greater than or equal to 180°, then* $y_s = 0$. *Otherwise, let* y_i *be the y-value of* p_i *and* $y_c = (y_1 + y_2)/2$. *Then the sign of* y_s *can be shown to be equal to that of* y_c. These situations are illustrated in Figure 6–21(a) and Figure 6–21(b) respectively and discussed below.

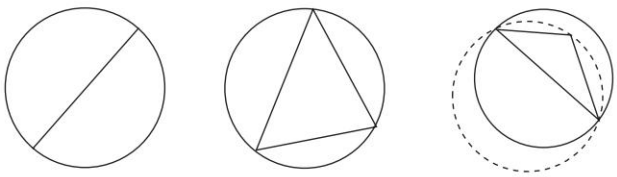
FIGURE 6–21 Cases where *I* contains more than one point.



Let us now consider the first case, namely the case where the smallest arc spanning all farthest points is of degree greater than or equal to 180°. Note that a smallest circle containing a set of points is defined by either two points or three points of this set, shown in Figure 6–22(a) and Figure 6–22(b). Three points define the boundary of a smallest circle enclosing all three of them, if and only if they do not form an obtuse triangle (see Figure 6–22(b)); otherwise, we can replace this circle by the circle with the longest edge of this triangle as the diameter (see Figure 6–22(c)). In our case, since the arc spanning all farthest points is of degree greater than or equal to 180°, there must be at least three such farthest points and furthermore, these three points do not form an obtuse triangle.

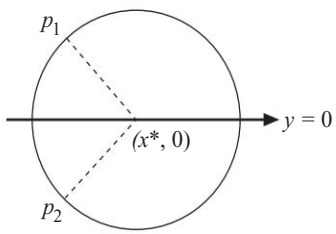
In other words, the present smallest circle is already optimal. We may consequently conclude that $y_s = 0$.

FIGURE 6–22 Two or three points defining the smallest circle covering all points.



For the second case the arc spanning all farthest points is of degree less than 180°, we shall first show that the x-values of end points p_1 *and* p_2 *must be of opposite signs*. Assume otherwise, as shown in Figure 6–23. We can then move x^* towards the direction where p_1 and p_2 are located. This is impossible because x^* is the center of the optimum circle on $y = 0$.

FIGURE 6–23 The direction of x^* where the degree is less than 180°.

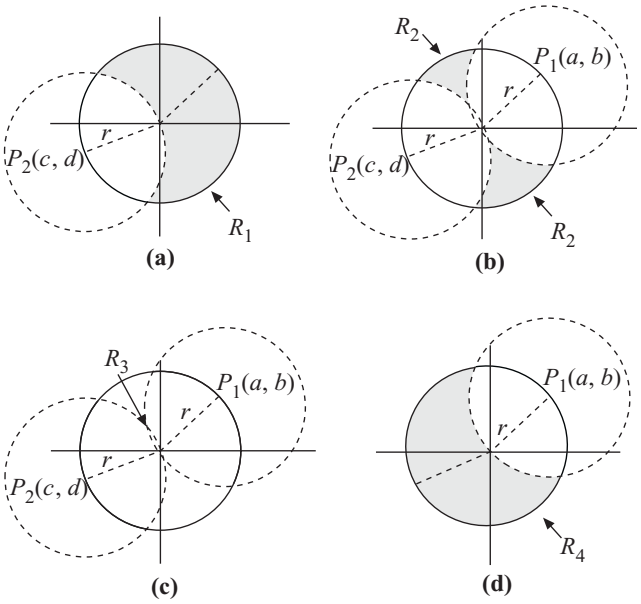


Let $p_1 = (a, b)$ and $p_2 = (c, d)$. Without losing generality, we may assume that

$$a > x^*,\ b > 0$$
$$\text{and} \quad c < x^*,\ d < 0$$

as shown in Figure 6–24. In Figure 6–24, there are three circles, centered at $(x^*, 0)$, (a, b) and (c, d) respectively. These three circles form four regions: R_1, R_2, R_3 and R_4. Let the radius of the present circle be r. We note the following three remarks:

FIGURE 6–24 The direction of x^* where the degree is larger than 180°.



(1) In R_2, the distance between any point x in this region and p_1, or p_2, is greater than r. Therefore, the center of the optimum circle cannot be located in R_2.
(2) In R_1, the distance between any point x in this region and p_2 is larger than r. Therefore, the center of the optimum circle cannot be located in R_1.
(3) Using similar reasoning, we can easily see that the center of the optimum circle cannot be located in R_4.

Based upon the above remarks, we conclude that the optimum center must be located in Region 3. Consequently, the sign of y_s must be the sign of $(b + d)/2 = (y_1 + y_2)/2$.

The above procedure can be summarized as follows.

Procedure 6–2 □ A procedure used in Algorithm 6–5

Input: A set S of points.
A line $y = y^*$.
(x^*, y^*): The solutions of the constrained 1-center problem for S sequencing that the solution is on the line $y = y^*$.

Output: Whether $y_s > y^*$, $y_s < y^*$ or $y_s = y^*$ where (x_s, y_s) is the optimal solution of the 1-center problem for S.

Step 1. Find I to be the set of points which are farthest from (x^*, y^*).

Step 2. **Case 1:** I contains only one point $p = (x_p, y_p)$.

If $y_p > y^*$, report "$y_s > y^*$" and exit.

If $y_p < y^*$, report "$y_s < y^*$" and exit.

Case 2: I contains more than one point. In I, find $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ which are the two end points forming the smallest arc spanning all of the points in I.

Case 2.1: The degree of the arc formed by p_1 and p_2 is greater than or equal to180°.

Report $y_s = y^*$ and exit.

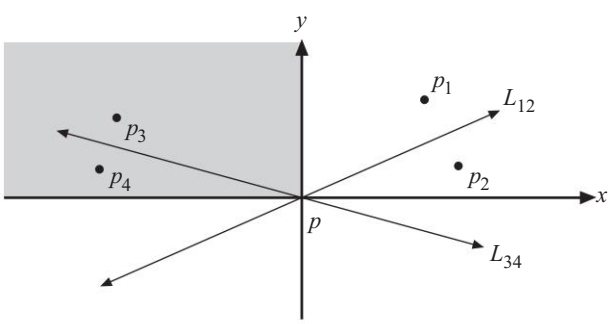
Case 2.2: The degree of arc formed by p_1 and p_2 is smaller than 180°. Let $y_c = (y_1 + y_2)/2$.

If $y_c > y^*$, report $y_s > y^*$ and exit.

If $y_c < y^*$, report $y_s < y^*$ and exit.

The constrained 1-center algorithm can now be used to prune away points. Before giving the precise algorithm, let us consider Figure 6–25. For pairs (p_1, p_2) and (p_3, p_4), we draw bisectors L_{12} and L_{34}, of line segments $\overline{p_1p_2}$ and $\overline{p_3p_4}$, respectively. Let L_{12} and L_{34} intersect at a point, say p. Rotate the x-axis such that L_{34} has a negative slope and L_{12} has a positive slope. Move the origin of the coordinate system to p. We first apply the constrained 1-center algorithm requiring that the center be located on $y = 0$. After finding this constrained

FIGURE 6–25 The appropriate rotation of the coordinates.



1-center, we then use Procedure 6–2 to find out that for the optimal solutions, where the y-direction is concerned, we should move upward. We repeat this process by applying the constrained 1-center algorithm again to $x = 0$. We then will find out that where the x-direction is concerned, we should move to the left. Therefore, the optimal location must be located in the shaded region, illustrated in Figure 6–25. Since there is one bisector which does not intersect with the shaded region, we may always use this information to eliminate one point from consideration.

In our case, L_{12} does not intersect with the shaded region. Point p_1 is at the same side of the shaded region, which means that as long as the optimal center is restricted in the shaded region, point p_1 is always nearer to the optimal center than p_2. Consequently, we may eliminate point p_1, because it is "dominated" by point p_2. In other words, only point p_2 should be considered.

The prune-and-search algorithm to solve the 1-center problem follows.

Algorithm 6–5 □ A prune-and-search algorithm to solve the 1-center problem

Input: A set $S = \{p_1, p_2, \ldots, p_n\}$ of n points.

Output: The smallest enclosing circle for S.

Step 1. If S contains no more than 16 points, solve the problem by a brute force method.

Step 2. Form disjoint pairs of points, (p_1, p_2), (p_3, p_4), $\ldots$, (p_{n-1}, p_n). For each pair of points, (p_i, p_{i+1}), find the perpendicular bisector of line segment $\overline{p_i p_{i+1}}$. Denote them as $L_{i/2}$, for $i = 2, 4, \ldots, n$ and compute their slopes. Let the slope of L_k be denoted as s_k, for $k = 1, 2, \ldots, n/2$.

Step 3. Compute the median of s_k's, and denote it by s_m.

Step 4. Rotate the coordinate system so that the x-axis coincides with $y = s_m x$. Let the set of L_k's with positive (negative) slopes be $I^+(I^-)$. (Both of them are of size $n/4$.)

Step 5. Construct disjoint pairs of the lines, (L_{i+}, L_{i-}) for $i = 1, 2, \ldots, n/4$, where $L_{i+} \in I^+$ and $L_{i-} \in I^-$. Find the intersection of each pair of them and denote it by (a_i, b_i), for $i = 1, 2, \ldots, n/4$.

Step 6. Find the median of b_i's. Denote it as y^*. Apply the constrained 1-center subroutine to S, requiring that the center of circle be located on $y = y^*$. Let the solution of this constrained 1-center problem be (x', y^*).

Step 7. Apply Procedure 6–2, using S and (x', y^*) as the parameters.
If $y_s = y^*$, report "the circle found in Step 6 with (x', y^*) as the center is the optimal solution" and exit.
Otherwise, report $y_s > y^*$ or $y_s < y^*$.

Step 8. Find the median of a_i's. Denote it as x^*. Apply the constrained 1-center algorithm to S, requiring that the center of circle be located on $x = x^*$. Let the solution of this contained 1-center problem be (x^*, y').

Step 9. Apply Procedure 6–2, using S and (x^*, y') as the parameters. If $x_s = x^*$, report "the circle found in Step 8 with (x^*, y') as the center is the optimal solution" and exit.
Otherwise, report $x_s > x^*$ or $x_s < x^*$.

Step 10. **Case 1:** $x_s > x^*$ and $y_s > y^*$.
Find all (a_i, b_i)'s such that $a_i < x^*$ and $b_i < y^*$. Let (a_i, b_i) be the intersection of L_{i+} and L_{i-} (see Step 5.). Let L_{i-} be the bisector of p_j and p_k. Prune away $p_j(p_k)$ if $p_j(p_k)$ is closer to (x^*, y^*) than $p_k(p_j)$.

Case 2: $x_s < x^*$ and $y_s > y^*$.
Find all (a_i, b_i)'s such that $a_i > x^*$ and $b_i < y^*$. Let (a_i, b_i) be the intersection of L_{i+} and L_{i-}. Let L_{i+} be the bisector of p_j and p_k. Prune away $p_j(p_k)$ if $p_j(p_k)$ is closer to (x^*, y^*) than $p_k(p_j)$.

Case 3: $x_s < x^*$ and $y_s < y^*$.
Find all (a_i, b_i)'s such that $a_i > x^*$ and $b_i > y^*$. Let (a_i, b_i) be the intersection of L_{i+} and L_{i-}. Let L_{i-} be the bisector of p_j and p_k. Prune away $p_j(p_k)$ if $p_j(p_k)$ is closer to (x^*, y^*) than $p_k(p_j)$.

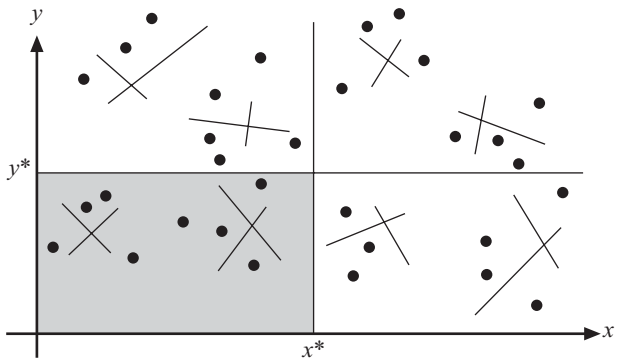
Case 4: $x_s > x^*$ and $y_s < y^*$.
Find all (a_i, b_i)'s such that $a_i < x^*$ and $b_i > y^*$. Let (a_i, b_i) be the intersection of L_{i+} and L_{i-}. Let L_{i+} be the bisector of p_j and p_k. Prune away $p_j(p_k)$ if $p_j(p_k)$ is closer to (x^*, y^*) than $p_k(p_j)$.

Step 11. Let S be the remaining points. Go to Step 1.

The analysis of Algorithm 6–5 is as follows. First, we assume that there are $n = (16)^k$ points for some k. There are $n/2$ bisectors formed in Step 2. After Step 4, $n/4$ of them have positive slopes and $n/4$ of them have negative slopes. Thus, there are a total $n/4$ intersections formed in Step 5. Since $x^*(y^*)$ is the median

of a_i's (b_i's), there are $(n/4)/4 = n/16$ (a_i, b_i)'s for each case in Step 10, shown in Figure 6–26 where it is assumed that the optimal solution lies in the shaded area. Then for each pair of intersection in the region where $x > x^*$ and $y > y^*$, the point below the line with negative slope can be pruned. For each such (a_i, b_i), exactly one point is pruned away. Therefore, in each intersection, $n/16$ points are pruned away.

FIGURE 6–26 The disjoint pairs of points and their slopes.



It is easy to see that each step in Algorithm 6–5 takes $O(n)$ time. Thus, the total time complexity of Algorithm 6–5 is $O(n)$.

6–5 THE EXPERIMENTAL RESULTS

To demonstrate the power of the prune-and-search approach, we implemented the prune-and-search algorithm to solve the 1-center problem. There is another way of solving this problem. Note that every solution of the 1-center problem must pass through at least three points. Therefore, we may try every possible combination of three points. That is, we select three points, construct a circle and see if it covers all points. If it does not, ignore it. If it does, record its radius. After all such circles have been examined, we can find the smallest one. Thus, this is an $O(n^3)$ approach, while the prune-and-search approach is an $O(n)$ algorithm.

We implemented both approaches on an IBM PC. The straightforward approach is labeled as the exhaustive search. All data were randomly generated. Table 6–1 summarizes the test results.

TABLE 6–1 Test results.

Sample points number	Exhaustive search (seconds)	Prune-and-search (seconds)
17	1	1
30	8	11
50	28	32
100	296	83
150	1080	130
200	2408	185
250	2713	232
300	–	257

It is obvious that the prune-and-search strategy is very efficient in this case. When n is 150, the exhaustive search is already much worse than the prune-and-search approach. When n is 300, the exhaustive search will take so long to solve the problem that this problem will never be solved by using that approach. On the other hand, it only takes 257 seconds to solve the 1-center problem if the prune-and-search algorithm is used. The results are not surprising as one method has $O(n^3)$ complexity and the other has $O(n)$ complexity.

6–6 NOTES AND REFERENCES

The approach of solving the selection problem has been traditionally classified as a divide-and-conquer strategy. We believe that it is more appropriate to classify it as a prune-and-search approach. See Blum, Floyd, Pratt, Rivest and Tarjan (1972) and Floyd and Rivest (1975) for more detailed discussion. A discussion of lower bound for selection problem can be found in Hyafil (1976). Many textbooks also presented the clever method for the selection problem (Brassard and Bratley, 1988; Horowitz and Sahni, 1978; Kronsjo, 1987).

The clever method presented in Section 6–3 was independently discovered by Dyer (1984) and Megiddo (1983). The results were extended to higher dimensions by Megiddo (1984). That the 1-center problem can be solved by the prune-and-search approach was also shown in Megiddo (1983). This disproved the conjecture by Shamos and Hoey (1975).

6–7 FURTHER READING MATERIALS

The prune-and-search approach is relatively new, and fewer papers have been published in this field compared to other fields. For further research, we recommend the following papers: Avis, Bose, Shermer, Snoeyink, Toussaint and Zhu (1996); Bhattacharya, Jadhav, Mukhopadhyay and Robert (1994); Chou and Chung (1994); Imai (1993); Imai, Kato and Yamamoto (1989); Imai, Lee and Yang (1992); Jadhav and Mukhopadhyay (1993); Megiddo (1983); Megiddo (1984); Megiddo (1985); and Shreesh, Asish and Binay (1996).

For some very interesting newly published papers, consult Eiter and Veith (2002); ElGindy, Everett and Toussaint (1993); Kirpatrick and Snoeyink (1993); Kirpatrick and Snoeyink (1995); and Santis and Persiano (1994).

Exercises

6.1 Make sure that you understand the differences and similarities between divide-and-conquer and prune-and-search. Note that the recurrence formulas may look quite similar.

6.2 Write a program to implement the selection algorithm introduced in this chapter. Another method to find the kth largest number is to perform the quick sort and pick out the kth largest number from the sorted array. Implement this approach also. Compare the two programs. Explain your testing results.

6.3 Two sets of points A and B in R^d are said to be linearly separable if there exists a $(d-1)$-dimensional hyperplane such that A and B lie on opposite sides of it. Show that the linear separability problem is a linear programming problem.

6.4 By the results in Exercise 6.3, show that the linear separability problem can be solved in $O(n)$ time in two and three dimensions.

6.5 Read Theorem 3.3 of Horowitz and Sahni (1978) for the average case analysis of the selection algorithm based on the prune-and-search approach.

chapter

7

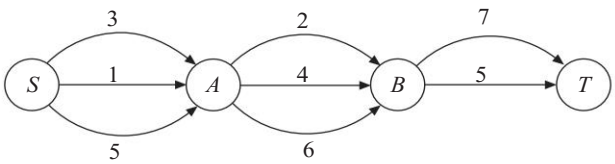
Dynamic Programming

The dynamic programming strategy is a very useful technique to solve many combinatorial optimization problems. Before introducing the dynamic programming approach, perhaps it is appropriate to reconsider a typical case where we can use the greedy method. We shall then show a case for which the greedy method will not work while the dynamic programming approach will work.

Let us consider Figure 7–1, which comprises a multi-stage graph. Suppose that we want to find the shortest path from S to T. In this case, the shortest route can be found through the following sequence of reasoning:

(1) Since we know that the shortest route must pass through vertex A, we should find a shortest route from S to A. The cost of this route is 1.
(2) Since we know that the shortest route must go through B, we find a shortest route from A to B. The cost of this route is 2.
(3) Similarly, we find a shortest route from B to T whose cost is 5.

FIGURE 7–1 A case where the greedy method works.



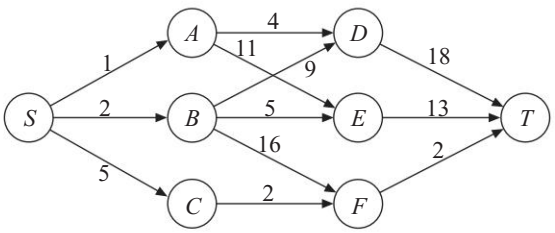
Thus, the total cost of the shortest route from S to T is $1 + 2 + 5 = 8$. In addition, we actually solved this shortest path problem using a greedy method.

Why can we use the greedy method to solve this problem? We can do so because we know definitely that our solution must consist of a subpath from S to A, a subpath from

A to B and so on. Therefore, our problem-solving strategy will first find a shortest path from S to A, later a shortest path from A to B and so on.

We now show a case where the greedy method will not work. Consider Figure 7–2. Again, we want to find a shortest route from S to T. However, this time, we do not know, among the three vertices A, B and C, which vertex the shortest path will go through. If we use the greedy method to solve the problem, we choose vertex A because the cost associated with the edge from S to A is the smallest. After A is chosen, we then choose D. The cost of this path will be $1 + 4 + 18 = 23$. This is not the shortest path because the shortest path is $S \rightarrow C \rightarrow F \rightarrow T$ whose total cost is $5 + 2 + 2 = 9$.

FIGURE 7–2 A case where the greedy method will not work.

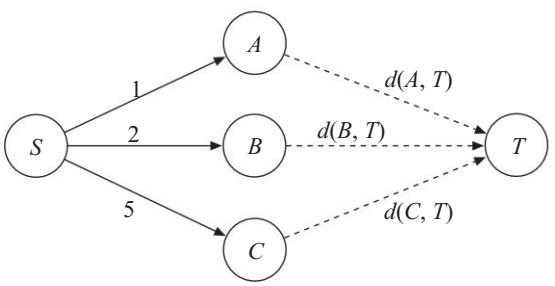


$$S \xrightarrow{5} C \xrightarrow{2} F \xrightarrow{2} T$$

As indicated before, we should select C, instead of A. The dynamic programming approach will produce this solution through the following way of reasoning.

(1) We know that we must start from S and go through A, B or C. This is illustrated in Figure 7–3.

FIGURE 7–3 A step in the process of using the dynamic programming approach.



The length of the shortest path from S to T is therefore determined by the following formula:

$$d(S, T) = \min\{1 + d(A, T),\ 2 + d(B, T),\ 5 + d(C, T)\} \qquad \textbf{(7.1)}$$

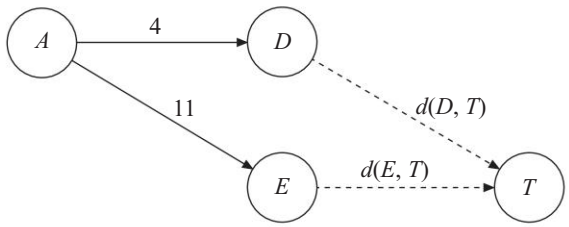
where $d(X, T)$ denotes the length of the shortest path from X to T. The shortest path from S to T will be found as soon as the shortest paths from A, B and C to T respectively are found.

(2) To find the shortest paths from A, B and C to T, we can again use the above approach. For vertex A, we have a subgraph shown in Figure 7–4. That is, $d(A, T) = \min\{4 + d(D, T),\ 11 + d(E, T)\}$. Since $d(D, T) = 18$ and $d(E, T) = 13$, we have

$$\begin{aligned} d(A, T) &= \min\{4 + 18,\ 11 + 13\} \\ &= \min\{22, 24\} \\ &= 22. \end{aligned}$$

We, of course, must record the fact that this shortest path is from A to D and then to T.

FIGURE 7–4 A step in the process of the dynamic programming approach.

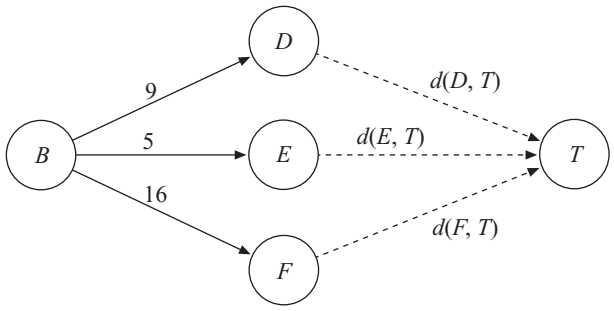


Similarly, for B, the step is shown in Figure 7–5.

$$\begin{aligned} d(B, T) &= \min\{9 + d(D, T),\ 5 + d(E, T),\ 16 + d(F, T)\} \\ &= \min\{9 + 18,\ 5 + 13,\ 16 + 2\} \\ &= \min\{27, 18, 18\} \\ &= 18. \end{aligned}$$

Finally, $d(C, T)$ can be easily found to be 4.

FIGURE 7–5 A step in the process of the dynamic programming approach.



(3) Having found $d(A, T)$, $d(B, T)$ and $d(C, T)$, we can find $d(S, T)$ by using formula (7.1).

$$\begin{aligned} d(S, T) &= \min\{1 + 22, 2 + 18, 5 + 4\} \\ &= \min\{23, 20, 9\} \\ &= 9. \end{aligned}$$

The basic principle of dynamic programming is to decompose a problem into subproblems, and each subproblem will be solved by the same approach recursively. We can summarize our approach as follows:

(1) To find the shortest path from S to T, we find shortest paths from A, B and C to T.
(2) (a) To find the shortest path from A to T, we find shortest paths from D and E to T.
    (b) To find the shortest path from B to T, we find shortest paths from D, E and F to T.
    (c) To find the shortest path from C to T, we find the shortest path from F to T.
(3) Shortest paths from D, E and F to T can be found easily.
(4) Having found shortest paths from D, E and F to T, we can find shortest paths from A, B and C to T.
(5) Having found shortest paths from A, B and C to T, we can find the shortest path from S to T.

The above way of reasoning is actually a backward reasoning. Next, we shall show that we can also solve the problem by forward reasoning. The shortest path will be found as follows:

(1) We first find $d(S, A)$, $d(S, B)$ and $d(S, C)$.

$$d(S, A) = 1$$
$$d(S, B) = 2$$
$$d(S, C) = 5.$$

(2) We then determine $d(S, D)$, $d(S, E)$ and $d(S, F)$ as follows:

$$\begin{aligned} d(S, D) &= \min\{d(A, D) + d(S, A), d(B, D) + d(S, B)\} \\ &= \min\{4 + 1, 9 + 2\} \\ &= \min\{5, 11\} \\ &= 5 \\ d(S, E) &= \min\{d(A, E) + d(S, A), d(B, E) + d(S, B)\} \\ &= \min\{11 + 1, 5 + 2\} \\ &= \min\{12, 7\} \\ &= 7 \\ d(S, F) &= \min\{d(B, F) + d(S, B), d(C, F) + d(S, C)\} \\ &= \min\{16 + 2, 2 + 5\} \\ &= \min\{18, 7\} \\ &= 7. \end{aligned}$$

(3) The shortest distance from S to T can now be determined as:

$$\begin{aligned} d(S, T) &= \min\{d(D, T) + d(S, D), d(E, T) + d(S, E), d(F, T) + d(S, F)\} \\ &= \min\{18 + 5, 13 + 7, 2 + 7\} \\ &= \min\{23, 20, 9\} \\ &= 9. \end{aligned}$$

That the dynamic programming approach saves computations can be explained by considering Figure 7–2 again. In Figure 7–2, there is one solution as follows:

$S \rightarrow B \rightarrow D \rightarrow T.$

The length of this solution, namely

$d(S, B) + d(B, D) + d(D, T)$

is never calculated if the backward reasoning dynamic programming approach is used. Using the backward reasoning approach, we will find

$$d(B, E) + d(E, T) < d(B, D) + d(D, T).$$

That is, we need not consider the solution:

$$S \rightarrow B \rightarrow D \rightarrow T$$

because we know that the length of

$$B \rightarrow E \rightarrow T$$

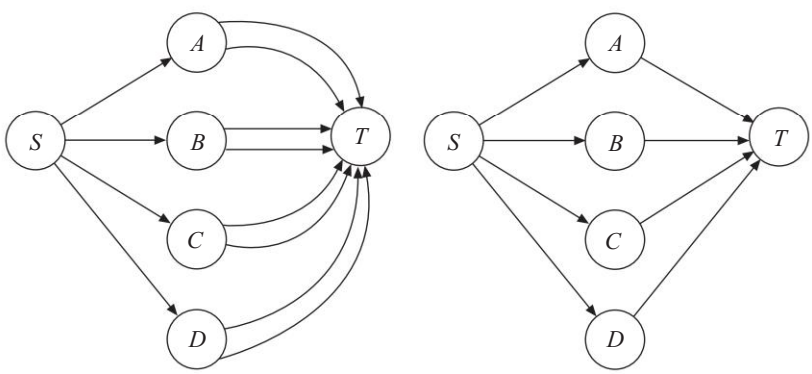
is shorter than the length of

$$B \rightarrow D \rightarrow T.$$

Thus, the dynamic programming approach, like the branch-and-bound approach, helps us avoid exhaustively searching the solution space.

We may say that the dynamic programming approach is an elimination by stage approach because after a stage is considered, many subsolutions are eliminated. For instance, consider Figure 7–6(a). There are originally eight solutions. If the dynamic programming approach is used, the number of solutions is reduced to four, shown in Figure 7–6(b).

FIGURE 7–6 An example illustrating the elimination of solution in the dynamic programming approach.



(a) **(b)**

The dynamic programming is based on a concept called the principle of optimality. Suppose that in solving a problem, we need to make a sequence of decisions $D_1, D_2, \ldots, D_n$. *If this sequence is optimal, then the last k decisions,* $1 \le k \le n$, *must be optimal*. We shall illustrate this principle several times in the following sections.

There are two advantages of applying the dynamic programming approach. The first advantage, as explained before, is that we may eliminate solutions and also save computations. Another advantage of dynamic programming is that it helps us solve the problem stage by stage systematically.

If a problem solver does not have any knowledge of dynamic programming, he may have to solve a problem by examining all possible combinatorial solutions. This may be exceedingly time consuming. If the dynamic programming approach is used, the problem may suddenly become a multi-stage problem and thus can be solved in a very systematic way. This will become clear later.

7-1 THE RESOURCE ALLOCATION PROBLEM

The resource allocation problem is defined as follows: We are given m resources and n projects. A profit $P(i, j)$ will be obtained if j, $0 \le j \le m$, resources are allocated to project i. The problem is to find an allocation of resources to maximize the total profit.

Assume that there are four projects and three resources, and the profit matrix P is shown in Table 7–1 with $P(i, 0) = 0$ for each i.

TABLE 7–1 Profit matrix.

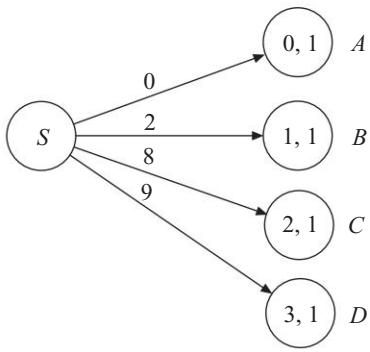
Project \ Resource	1	2	3
1	2	8	9
2	5	6	7
3	4	4	4
4	2	4	5

To solve this problem, we shall make a sequence of decisions. In each decision, we determine the number of resources to be allocated to project i. That is, without losing generality, we must decide:

(1) How many resources should we allocate to project 1?
(2) How many resources should we allocate to project 2?
⋮

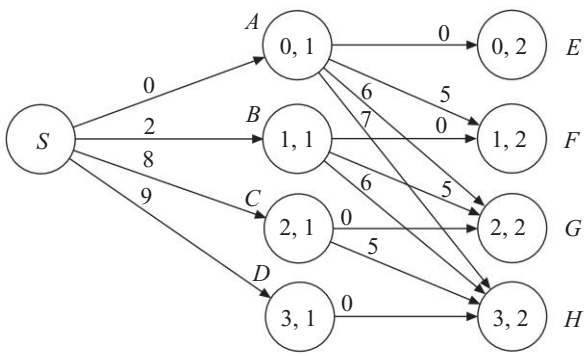
It is obvious that the greedy method approach does not work in this case. We can, however, solve the problem by using the dynamic programming approach. If the dynamic programming approach is used, we can imagine that each decision will create a new state. Let (i, j) represent the state attained where i resources have been allocated to projects $1, 2, \ldots, j$. Then, initially, we have only four states described, which are shown in Figure 7–7.

FIGURE 7–7 The first stage decisions of a resource allocation problem.



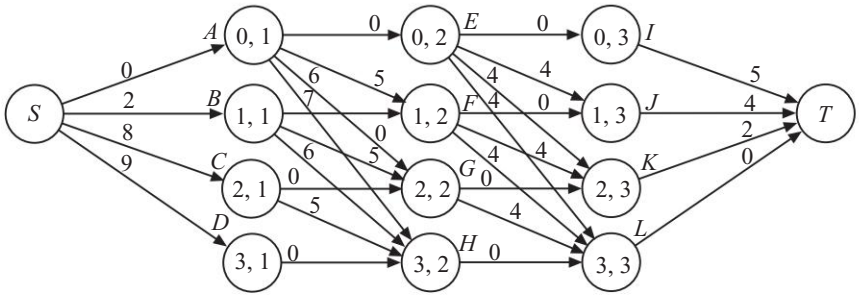
After allocating i resources to project 1, we can allocate at most $(3 - i)$ resources to project 2. Thus, the second stage decisions are related to the first stage decisions, which are shown in Figure 7–8.

FIGURE 7–8 The first two stage decisions of a resource allocation problem.



Finally, Figure 7–9 shows how the entire problem can be described.

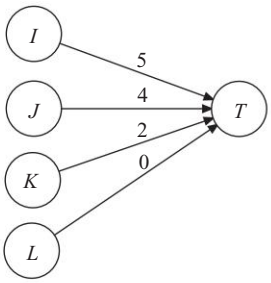
FIGURE 7–9 The resource allocation problem described as a multi-stage graph.



To solve the resource allocation problem, we merely have to find the longest path from S to T. We may use the backward reasoning approach as follows:

(1) The longest paths from I, J, K and L to T are shown in Figure 7–10.

FIGURE 7–10 The longest paths from I, J, K and L to T.

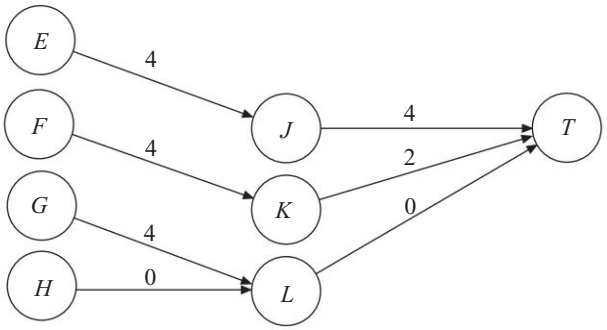


(2) Having obtained the longest paths from I, J, K and L to T, we can obtain the longest paths from E, F, G and H to T easily. For instance, the longest path from E to T is determined as follows:

$$\begin{aligned} d(E, T) &= \max\{d(E, I) + d(I, T), d(E, J) + d(J, T), \\ &\qquad d(E, K) + d(K, T), d(E, L) + d(L, T)\} \\ &= \max\{0 + 5, 4 + 4, 4 + 2, 4 + 0\} \\ &= \max\{5, 8, 6, 4\} \\ &= 8. \end{aligned}$$

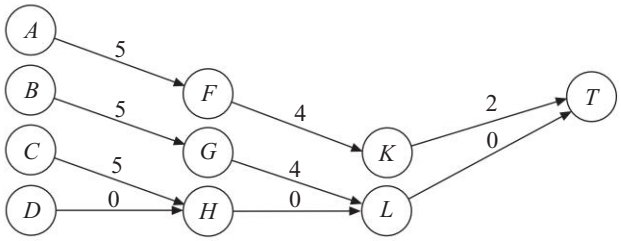
Figure 7–11 summarizes the results.

FIGURE 7–11 The longest paths from E, F, G and H to T.



(3) The longest paths from A, B, C and D to T respectively are found by the same method and shown in Figure 7–12.

FIGURE 7–12 The longest paths from A, B, C and D to T.



(4) Finally, the longest path from S to T is obtained as follows:

$$\begin{aligned} d(S, T) &= \max\{d(S, A) + d(A, T),\ d(S, B) + d(B, T),\\ &\qquad d(S, C) + d(C, T),\ d(S, D) + d(D, T)\}\\ &= \max\{0 + 11,\ 2 + 9,\ 8 + 5,\ 9 + 0\}\\ &= \max\{11, 11, 13, 9\}\\ &= 13. \end{aligned}$$

The longest path is

$$S \to C \to H \to L \to T.$$

This corresponds to

2 resources allocated to project 1,
1 resource allocated to project 2,
0 resource allocated to project 3,
and 0 resource allocated to project 4.

7–2 THE LONGEST COMMON SUBSEQUENCE PROBLEM

Consider a string $A = a\ b\ a\ a\ d\ e$. A subsequence of A is obtained by deleting 0 or more (not necessarily consecutive) symbols from A. For instance, the following strings are all subsequences of A:

a
b
d
a b
a a
b d
a e
a b a
a a a
a d e
b a d e
a a a d
a b a d e

A common subsequence between A and B is defined to be a subsequence of both strings. For instance, consider $A = a\ b\ a\ a\ d\ e\ c$ and $B = c\ a\ a\ c\ e\ d\ c$. The following strings are all common subsequences of A and B:

a
d
c
a d
d c
a a
a c
a a c
a a d
a a e c

The longest common subsequence problem is to find a longest common subsequence between two strings. Many problems are variations of the longest common subsequence problem. For instance, the speech recognition problem can be viewed as a longest common subsequence problem.

To a novice problem solver, the longest common subsequence problem is by no means easy to solve. A straightforward method is to find all common subsequences by an exhaustive search. Of course, we should start with the longest possible one. The length of the longest common subsequence between two strings cannot be longer than the length of the shorter string. Therefore, we should start with the shorter string.

For instance, let $A = a\ b\ a\ a\ d\ e\ c$ and $B = b\ a\ f\ c$. We may try to determine whether $b\ a\ f\ c$ is a common subsequence. It can be seen that it is not. We then try subsequences chosen from B with length three, namely $a\ f\ c$, $b\ f\ c$, $b\ a\ c$ and $b\ a\ f$. Since $b\ a\ c$ is a common subsequence, it must also be a longest common subsequence because we started from the longest possible one.

This straightforward approach is exceedingly time consuming because a large number of subsequences of B will be matched with a large number of subsequences of A. Thus, it is an exponential procedure.

Fortunately, we can use the dynamic programming approach to solve this longest common subsequence problem. Let us modify our original problem slightly. Instead of finding the longest common subsequence, let us try to determine the *length* of the longest common subsequence. Actually, by tracing back the procedure of finding the length of the longest common subsequence, we can easily find the longest common subsequence.

Consider two strings $A = a_1a_2 \ldots a_m$ and $B = b_1b_2 \ldots b_n$. Let us pay attention to the last two symbols: a_m and b_n. There are two possibilities:

Case 1: $a_m = b_n$. In this case, the longest common subsequence must contain a_m. We merely have to find the longest common subsequence of $a_1a_2 \ldots a_{m-1}$ and $b_1b_2 \ldots b_{n-1}$.

Case 2: $a_m \neq b_n$. In this case, we may match $a_1a_2 \ldots a_m$ with $b_1b_2 \ldots b_{n-1}$ and also $a_1a_2 \ldots a_{m-1}$ with $b_1b_2 \ldots b_n$. Whatever produces a longer longest common subsequence, this will be our longest common subsequence.

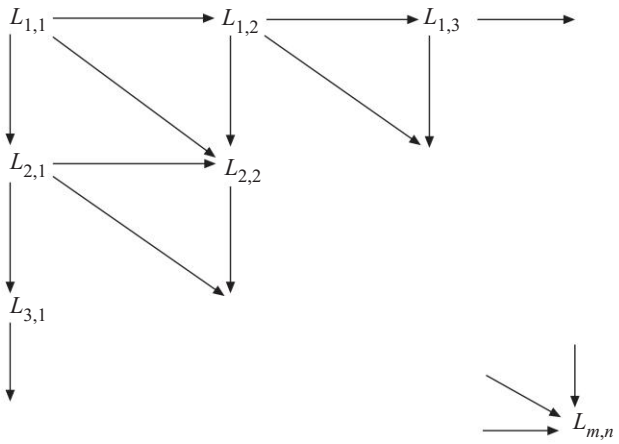
Let $L_{i,j}$ denote the length of the longest common subsequence of $a_1a_2 \ldots a_i$ and $b_1b_2 \ldots b_j$. $L_{i,j}$ can be found by the following recursive formula:

$$L_{i,j} = \begin{cases} L_{i-1,j-1} + 1 & \text{if } a_i = b_j \\ \max\{L_{i-1,j}, L_{i,j-1}\} & \text{if } a_i \neq b_j \end{cases}$$

$$L_{0,0} = L_{0,j} = L_{i,0} = 0, \qquad \text{for } 1 \leq i \leq m \text{ and } 1 \leq j \leq n.$$

The above formula shows that we may first find $L_{1,1}$. After finding $L_{1,1}$, we may find $L_{1,2}$ and $L_{2,1}$. After obtaining $L_{1,1}$, $L_{1,2}$ and $L_{2,1}$, we may find $L_{1,3}$, $L_{2,2}$ and $L_{3,1}$. The entire procedure can be depicted in Figure 7–13.

FIGURE 7–13 The dynamic programming approach to solve the longest common subsequence problem.



For implementing the algorithm sequentially, we can compute the value of each $L_{i,j}$ by the sequence: $L_{1,1}, L_{1,2}, L_{1,3}, \ldots, L_{1,n}, L_{2,1}, \ldots, L_{2,n}, L_{3,1}, \ldots, L_{3,n}, \ldots, L_{m,n}$. Since to compute each $L_{i,j}$ requires constant time, the time complexity of the algorithm is $O(mn)$.

We now illustrate our dynamic programming approach by an example. Let $A = a\ b\ c\ d$ and $B = c\ b\ d$. In this case, we have

$$a_1 = a$$
$$a_2 = b$$
$$a_3 = c$$
$$a_4 = d$$
$$b_1 = c$$
$$b_2 = b$$
and $$b_3 = d.$$

The length of the longest common subsequence is found step by step, illustrated in Table 7–2.

TABLE 7–2 The values of $L_{i,j}$'s.

b_j \ a_i			*a*	*b*	*c*	*d*
		0	**1**	**2**	**3**	**4**
	0	0	0	0	0	0
c	1	0	0	0	1	1
b	2	0	0	1	1	1
d	3	0	0	1	1	2

$L_{i,j}$

7–3 THE 2-SEQUENCE ALIGNMENT PROBLEM

Let $A = a_1a_2 \ldots a_m$ and $B = b_1b_2 \ldots b_n$ be two sequences over an alphabet set Σ. A sequence alignment of A and B is a $2 \times k$ matrix M ($k \geq m, n$) of characters over $\Sigma \bigcup \{-\}$ such that no column of M consists entirely of dashes, and the result sequences by removing all dashes in the first row and second row of M are equal to A and B respectively. For example, if $A = a\ b\ c\ d$ and $B = c\ b\ d$, a possible alignment of them would be

$a\ b\ c\ -\ d$
$-\ -\ c\ b\ d$

Another possible alignment of the same two sequences is

$a\ b\ c\ d$
$c\ b\ -\ d.$

We can see that the second alignment seems to be better then the first one. To be precise, we need to define a scoring function which can be used to measure the performance of an alignment. In the following, we shall present such a function. It must be understood that we can define other kinds of scoring functions.

Let $f(x, y)$ denote the score for aligning x with y. Suppose that both x and y are characters. Then $f(x, y) = 2$ if x and y are the same and $f(x, y) = 1$ if x and y are not the same. If x or y is "–", $f(x, y) = -1$.

The score of an alignment is the total sum of scores of columns. Then the score for the following alignment is $-1 - 1 + 2 - 1 + 2 = 1$.

$a - b\ c\ d$
$-\ c\ b - d$

The score of the following alignment is $1 + 2 - 1 + 2 = 4$.

$a\ b\ c\ d$
$c\ b - d$

The two-sequence alignment problem for A and B is to find an optimal alignment with the maximum score. A recursive formula similar to that used for finding the longest common sequence can be easily formulated. Let $A_{i,j}$ denote the optimal alignment score between $a_1a_2 \ldots a_i$ and $b_1b_2 \ldots b_j$, where $1 \le i \le m$ and $1 \le j \le n$. Then, $A_{i,j}$ can be expressed as follows:

$$A_{0,0} = 0$$
$$A_{i,0} = i \cdot f(a_i, -)$$
$$A_{0,j} = j \cdot f(-, b_j)$$
$$A_{i,j} = \max \begin{cases} A_{i-1,j} + f(a_i, -) \\ A_{i-1,j-1} + f(a_i, b_j) \\ A_{i,j-1} + f(-, b_j) \end{cases}$$

Perhaps it is necessary to explain the meaning of the above recursive formula:

$A_{0,0}$ is for the initial condition.

$A_{i,0}$ means that $a_1, a_2, \ldots, a_i$ are all aligned with "–".

$A_{0,j}$ means that $b_1, b_2, \ldots, b_j$ are all aligned with "–".

$A_{i-1,j} + f(a_i, -)$ means that a_i is aligned with "–" and we need to find an optimal alignment between $a_1, a_2, \ldots, a_{i-1}$ and $b_1, b_2, \ldots, b_j$.

$A_{i-1,j-1} + f(a_i, b_j)$ means that a_i is aligned with b_j and we need to find an optimal alignment between $a_1, a_2, \ldots, a_{i-1}$ and $b_1, b_2, \ldots, b_{j-1}$.

$A_{i,j-1} + f(-, b_j)$ means that b_j is aligned with "–" and we need to find an optimal alignment between $a_1, a_2, \ldots, a_i$ and $b_1, b_2, \ldots, b_{j-1}$.

The $A_{i,j}$'s for $A = a\ b\ d\ a\ d$ and $B = b\ a\ c\ d$ using the above recursive formula are listed in Table 7–3.

TABLE 7–3 The values of $A_{i,j}$'s for $A = a\ b\ d\ a\ d$ and $B = b\ a\ c\ d$.

a_i / b_j			**a**	**b**	**d**	**a**	**d**
		0	**1**	**2**	**3**	**4**	**5**
	0	0	−1	−2	−3	−4	−5
b	1	−1	1	1	0	−1	−2
a	2	−2	1	2	2	2	1
c	3	−3	0	2	3	3	3
d	4	−4	−1	1	4	4	5

In Table 7–3, we recorded how each $A_{i,j}$ is obtained. An arrow from (a_i, b_j) to (a_{i-1}, b_{j-1}) means that a_i is matched with b_j. An arrow from (a_i, b_j) to (a_{i-1}, b_j) means that a_i is matched with "–", and an arrow from (a_i, b_j) to (a_i, b_{j-1}) means that b_j is matched with "–". Based on the arrows in the table, we can trace back and find that the optimal alignment is:

a b d a d
– b a c d

Let us consider another example. Let the two sequences be $A = a\ b\ c\ d$ and $B = c\ b\ d$. Table 7–4 shows how $A_{i,j}$'s are found.

TABLE 7–4 The values of $A_{i,j}$'s for $A = a\ b\ c\ d$ and $B = c\ b\ d$.

			1	**2**	**3**	**4**
		–	**a**	**b**	**c**	**d**
	–	0	−1	−2	−3	−4
1	c	−1	1	0	0	−1
2	b	−2	0	3	2	1
3	d	−3	−1	2	4	4

Based on the arrows in Table 7–4, we can trace back and find that the optimal alignment is

$a\ b\ c\ d$
$c\ b\ -\ d$

Sequence alignment may be viewed as a method to measure the similarity of two sequences. Next, we shall introduce a concept, called the edit distance, which is also used quite often to measure the similarity between two sequences.

Let us consider two sequences $A = a_1a_2 \ldots a_m$ and $B = b_1b_2 \ldots b_n$. We may transform A to B by the following three edit operations: deletion of a character from A, insertion of a character into A and substitution of a character in A with another character. For example, let $A = GTAAHTY$ and $B = TAHHYC$. A can be transformed to B by the following operations:

(1) Delete the first character G of A. Sequence A becomes $A = TAAHTY$.
(2) Substitute the third character of A, namely A, by H. Sequence A becomes $A = TAHHTY$.
(3) Delete the fifth character of A, namely T, from A. Sequence A becomes $A = TAHHY$.
(4) Insert C after the last character of A. Sequence A becomes $A = TAHHYC$ which is identical to B.

We can associate a cost with each operation. The edit distance is the minimum cost associated with the edit operations needed to transform sequence A to sequence B. If the cost is one for each operation, the edit distance becomes the minimum number of edit operations needed to transform A to B. In the above example, if the cost of each operation is one and the edit distance between A and B is 4, at least four edit operations are needed.

It is obvious that the edit distance can be found by the dynamic programming approach. A recursive formula similar to that used for finding the longest common sequence or an optimal alignment between two sequences can be easily formulated. Let α, β and γ denote the costs of insertion, deletion and substitution respectively. Let $A_{i,j}$ denote the edit distance between $a_1a_2 \ldots a_i$ and $b_1b_2 \ldots b_j$. Then, $A_{i,j}$ can be expressed as follows:

$A_{0,0} = 0$

$$A_{i,0} = i\beta$$
$$A_{0,j} = j\alpha$$
$$A_{i,j} = \begin{cases} A_{i-1,j-1} & \text{if } a_i = b_j \\ \min \begin{cases} A_{i-1,j} + \beta \\ A_{i-1,j-1} + \gamma \\ A_{i,j-1} + \alpha \end{cases} & \text{otherwise} \end{cases}$$

Actually, it is easy to see that the edit distance finding problem is equivalent to the optimal alignment problem. We do not intend to present a formal proof here as it can be easily seen from the similarity of respective recursive formulas. Instead, we shall use the example presented above to illustrate our point.

Consider $A = GTAAHTY$ and $B = TAHHYC$ again. An optimal alignment would produce the following:

$G\ T\ A\ A\ H\ T\ Y\ -$
$-\ T\ A\ H\ H\ -\ Y\ C$

An examination of the above alignment shows the equivalence between edit operations and alignment operations as follows:

(1) (a_i, b_j) in the alignment finding is equivalent to the substitution operation in the edit distance finding. We substitute a_i by b_j in this case.
(2) $(a_i, -)$ in the alignment finding is equivalent to deleting a_i in A in the edit distance finding.
(3) $(-, b_j)$ in the alignment finding is equivalent to inserting b_j into A in the edit distance finding.

The reader can use the rules and the optimal alignment found above to produce the four edit operations. For two given sequences $A = a_1a_2 \ldots a_m$ and $B = b_1b_2 \ldots b_n$, a table with $(n + 1)(m + 1)$ entries is needed to record $A_{i,j}$'s. That is, it takes $O(nm)$ time to find an optimal alignment for two sequences $A = a_1a_2 \ldots a_m$ and $B = b_1b_2 \ldots b_n$.

7–4 THE RNA MAXIMUM BASE PAIR MATCHING PROBLEM

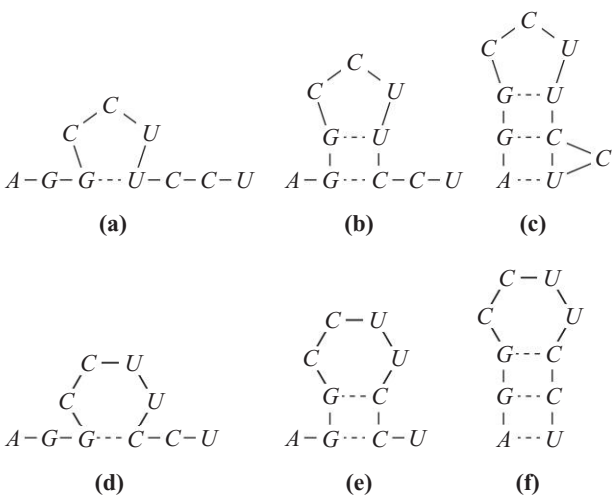
Ribonucleic acid (RNA) is a single strand of nucleotides (bases) adenine (A), guanine (G), cytosine (C) and uracil (U). The sequence of the bases A, G, C and U is called the *primary structure* of an RNA. In RNA, G and C can form a base

pair $G \equiv C$ by a triple-hydrogen bond, A and U can form a base pair $A=U$ by a double-hydrogen bond, and G and U can form a base pair $G–U$ by a single hydrogen bond. Due to these hydrogen bonds, the primary structure of an RNA can fold back on itself to form its secondary structure. For example, suppose that we have the following RNA sequence.

$A–G–G–C–C–U–U–C–C–U$

Then, this sequence can fold back on itself to form many possible secondary structures. In Figure 7–14, we show six possible secondary structures of this sequence. In nature, however, there is only one secondary structure corresponding to an RNA sequence. What is the actual secondary structure of an RNA sequence?

FIGURE 7–14 Six possible secondary structures of RNA sequence $A–G–G–C–C–U–U–C–C–U$ (the dashed lines denote the hydrogen bonds).



According to the thermodynamic hypothesis, the actual secondary structure of an RNA sequence is the one with the minimum free energy, which will be explained later. In nature, only the stable structure can exist and the stable structure must be the one with the minimum free energy. In a secondary structure of an RNA, the base pairs will increase the structural stability, and the unpaired bases will decrease the structural stability. The base pairs of the types $G \equiv C$ and

A=U (called *Watson-Crick* base pairs) are more stable than that of the type *G–U* (called *wobble* base pairs). According to these factors, we can find that the secondary structure of Figure 7–14(f) is the actual secondary structure of sequence *A–G–G–C–C–U–U–C–C–U*.

An RNA sequence will be represented as a string of n characters $R = r_1 r_2 \ldots r_n$, where $r_i \in \{A, C, G, U\}$. Typically, n can range from 20 to 2,000. A *secondary structure* of R is a set S of base pairs (r_i, r_j), where $1 \le i < j \le n$, such that the following conditions are satisfied.

(1) $j - i > t$, where t is a small positive constant. Typically, $t = 3$.
(2) If (r_i, r_j) and (r_k, r_l) are two base pairs in S, then either
   (a) $i = k$ and $j = l$, i.e., (r_i, r_j) and (r_k, r_l) are the same base pair,
   (b) $i < j < k < l$, i.e., (r_i, r_j) precedes (r_k, r_l), or
   (c) $i < k < l < j$, i.e., (r_i, r_j) includes (r_k, r_l).

The first condition implies that the RNA sequence does not fold too sharply on itself. The second condition means that each nucleotide can take part in at most one base pair, and guarantees that the secondary structure contains no pseudoknot. Two base pairs (r_i, r_j) and (r_k, r_l) are called a *pseudoknot* if $i < k < j < l$. Pseudoknots do occur in RNA molecules, but their exclusion simplifies the problem.

Recall that the goal of the secondary structure prediction is to find a secondary structure with the minimum free energy. Hence, we must have a method to calculate the free energy of a secondary structure S. Since the formations of base pairs give stabilizing effects to the structural free energy, the simplest method of measuring the free energy of S is to assign energy to each base pair of S and then the free energy of S is the sum of the energies of all base pairs. Due to different hydrogen bonds, the energies of base pairs are usually assigned different values. For example, the reasonable values for *A≡U*, *G=C* and *G–U* are -3, -2 and -1, respectively. Other possible values might be that the energies of base pairs are all equal. In this case, the problem becomes one of finding a secondary structure with the maximum number of base pairs. This version of the secondary structure prediction problem is also called *RNA maximum base pair matching problem* since we can view a secondary structure as a matching. Next, we will introduce a dynamic programming algorithm to solve the RNA maximum base pair matching problem, which is defined as follows: Given an RNA sequence $R = r_1 r_2 \ldots r_n$, find a secondary structure of RNA with the maximum number of base pairs.

Let $S_{i,j}$ denote the secondary structure of the maximum number of base pairs on the substring $R_{i,j} = r_i r_{i+1} \ldots r_j$. Denote the number of matched base pairs in $S_{i,j}$ by $M_{i,j}$. Notice that not any two bases r_i and r_j, where $1 \le i < j \le n$, can be paired with each other. The admissible base pairs we consider here are Watson-Crick base pairs (i.e., $A \equiv U$ and $G \equiv C$) and wobble base pairs (i.e., G–U). Let $WW = \{(A, U), (U, A), (G, C), (C, G), (G, U), (U, G)\}$. Then, we use a function $\rho(r_i, r_j)$ to indicate whether any two bases r_i and r_j can be a legal base pair:

$$\rho(r_i, r_j) = \begin{cases} 1 & \text{if } (r_i, r_j) \in WW \\ 0 & \text{otherwise} \end{cases}$$

By definition, we know that RNA sequence does not fold too sharply on itself. That is, if $j - i \le 3$, then r_i and r_j cannot be a base pair of $S_{i,j}$. Hence, we let $M_{i,j} = 0$ if $j - i \le 3$.

To compute $M_{i,j}$, where $j - i > 3$, we consider the following cases from r_j's point of view.

Case 1: In the optimal solution, r_j is not paired with any other base. In this case, find an optimal solution for $r_i r_{i+1} \ldots r_{j-1}$ and $M_{i,j} = M_{i,j-1}$ (see Figure 7–15).

Case 2: In the optimal solution, r_j is paired with r_i. In this case, find an optimal solution for $r_{i+1} r_{i+2} \ldots r_{j-1}$ and $M_{i,j} = 1 + M_{i+1,j-1}$ (see Figure 7–16).

Case 3: In the optimal solution, r_j is paired with some r_k, where $i + 1 \le k \le j - 4$. In this case, find the optimal solutions for $r_i r_{i+1} \ldots r_{k-1}$ and $r_{k+1} r_{k+1} \ldots r_{j-1}$ and $M_{i,j} = 1 + M_{i,k-1} + M_{k+1,j-1}$ (see Figure 7–17).

Since we want to find the k between $i + 1$ and $j + 4$ such that $M_{i,j}$ is the maximum, we have

FIGURE 7–15 Illustration of Case 1.

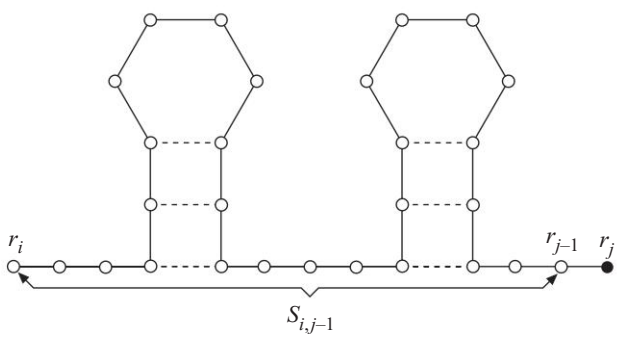


FIGURE 7–16 Illustration of Case 2.

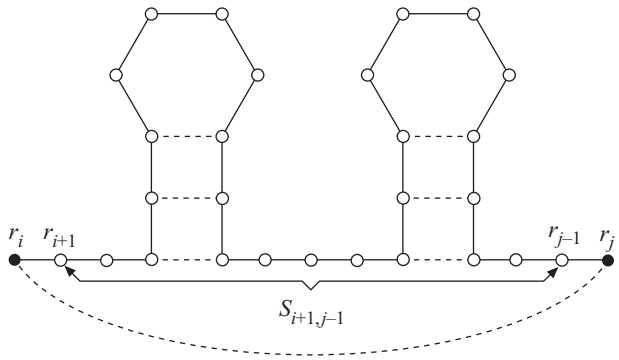
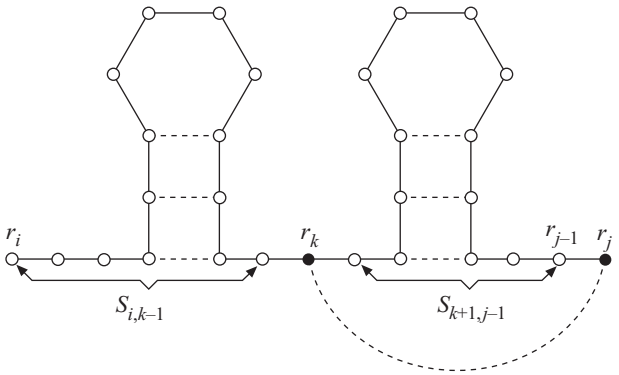


FIGURE 7–17 Illustration of Case 3.



$$M_{i,j} = \max_{i+1 \le k \le j-4} \{1 + M_{i,k-1} + M_{k+1,j-1}\}$$

In summary, we have the following recursive formula to compute $M_{i,j}$.

If $j - i \leqq 3$, then $M_{i,j} = 0$.

If $j - i > 3$, then $M_{i,j} = \max \begin{cases} M_{i,j-1} \\ (1 + M_{i+1,j-1}) \times \rho(r_i, r_j) \\ \max\limits_{i+1 \le k \le j-4} \{1 + M_{i,k-1} + M_{k+1,j-1}\} \times \rho(r_k, r_j) \end{cases}$

According to the formula on page 274, we can design Algorithm 7–1 to compute $M_{1,n}$ using the dynamic programming technique. Table 7–5 illustrates the computation of $M_{i,j}$, where $1 \le i < j \le 10$, for an RNA sequence $R_{1,10} = A$–G–G–C–C–U–U–C–C–U. As a result, we can find that the maximum number of base pairs in $S_{1,10}$ is 3 since $M_{1,10} = 3$.

TABLE 7–5 The computation of the maximum number of base pairs of an RNA sequence *A*–*G*–*G*–*C*–*C*–*U*–*U*–*C*–*C*–*U*.

i \ *j*	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	0	0	0	0	0	1	2	2	2	3
2	–	0	0	0	0	1	1	2	2	2
3	–	–	0	0	0	0	1	1	1	1
4	–	–	–	0	0	0	0	0	0	0
5	–	–	–	–	0	0	0	0	0	0
6	–	–	–	–	–	0	0	0	0	0
7	–	–	–	–	–	–	0	0	0	0
8	–	–	–	–	–	–	–	0	0	0
9	–	–	–	–	–	–	–	–	0	0
10	–	–	–	–	–	–	–	–	–	0

Algorithm 7–1 □ An RNA maximum base pair matching algorithm

Input: An RNA sequence $R = r_1 r_2 \ldots r_n$.

Output: Find a secondary structure of RNA with the maximum number of base pairs.

Step 1: /* Computation of $\rho(r_i, r_j)$ function for $1 \le i < j \le n$ */
$WW = \{(A, U), (U, A), (G, C), (C, G), (G, U), (U, G)\}$;
for $i = 1$ to n **do**
for $j = i$ to n **do**
if $(r_i, r_j) \in WW$ **then** $\rho(r_i, r_j) = 1$; **else** $\rho(r_i, r_j) = 0$;

```
        end for
    end for
Step 2: /* Initialization of M_{i,j} for j − i ≤ 3 */
    for i = 1 to n do
        for j = i to i + 3 do
            if j ≦ n then M_{i,j} = 0;
        end for
    end for
Step 3: /* Calculation of M_{i,j} for j − i > 3 */
    for h = 4 to n − 1 do
        for i = 1 to n − h do
            j = i + h;
            case1 = M_{i,j−1};
            case2 = (1 + M_{i+1,j−1}) × ρ(r_i, r_j);
            case3 = M_{i,j} max_{i+1≤k≤j−4} {(1 + M_{i,k−1} + M_{k+1,j−1}) × ρ(r_k, r_j)};
            M_{i,j} = max{case1, case2, case3};
        end for
    end for
```

We now illustrate the whole procedure in detail.

r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8	r_9	r_{10}
A	G	G	C	C	U	U	C	C	U

(1) $i = 1, j = 5, \rho(r_1, r_5) = \rho(A, C) = 0$

$$M_{1,5} = \max\begin{cases} M_{1,4} \\ (1 + M_{2,4}) \times \rho(r_1, r_5) \end{cases}$$
$$= \max\{0, 0\} = 0.$$

(2) $i = 2, j = 6, \rho(r_2, r_6) = \rho(G, U) = 1$

$$M_{2,6} = \max\begin{cases} M_{2,5} \\ (1 + M_{3,5}) \times \rho(r_2, r_6) \end{cases}$$
$$= \max\{0, (1 + 0) \times 1\} = \max\{0, 1\} = 1.$$

r_2 matches with r_6.

(3) $i = 3, j = 7, \rho(r_3, r_7) = \rho(G, U) = 1$

$$M_{3,7} = \max\begin{cases} M_{3,6} \\ (1 + M_{4,6}) \times \rho(r_3, r_7) \end{cases}$$
$$= \max\{0, (1 + 0) \times 1\} = \max\{0, 1\} = 1.$$

r_3 matches with r_7.

(4) $i = 4, j = 8, \rho(r_4, r_8) = \rho(C, C) = 0$

$$M_{4,8} = \max\begin{cases} M_{4,7} \\ (1 + M_{5,7}) \times \rho(r_4, r_8) \end{cases}$$
$$= \max\{0, (1 + 0) \times 0\} = 0.$$

(5) $i = 5, j = 9, \rho(r_5, r_9) = \rho(C, C) = 0$

$$M_{5,9} = \max\begin{cases} M_{5,8} \\ (1 + M_{6,8}) \times \rho(r_5, r_9) \end{cases}$$
$$= \max\{0, (1 + 0) \times 0\} = \max\{0, 0\} = 0.$$

(6) $i = 6, j = 10, \rho(r_6, r_{10}) = \rho(U, U) = 0$

$$M_{6,10} = \max\begin{cases} M_{6,9} \\ (1 + M_{7,9}) \times \rho(r_6, r_{10}) \end{cases}$$
$$= \max\{0, (1 + 0) \times 0\} = \max\{0, 0\} = 0.$$

(7) $i = 1, j = 6, \rho(r_1, r_6) = \rho(A, U) = 1$

$$M_{1,6} = \max\begin{cases} M_{1,5} \\ (1 + M_{2,5}) \times \rho(r_1, r_6) \\ (1 + M_{1,1} + M_{3,5}) \times \rho(r_2, r_6) \end{cases}$$
$$= \max\{0, (1 + 0) \times 1, (1 + 0 + 0) \times 1\}$$
$$= \max\{0, 1, 1\} = 1.$$

r_1 matches with r_6.

(8) $i = 2, j = 7, \rho(r_2, r_7) = \rho(G, U) = 1$

$$M_{2,7} = \max \begin{cases} M_{2,6} \\ (1 + M_{3,6}) \times \rho(r_2, r_7) \\ (1 + M_{2,2} + M_{4,6}) \times \rho(r_3, r_7) \end{cases}$$
$$= \max\{1, (1 + 0) \times 1, (1 + 0 + 0) \times 1\}$$
$$= \max\{1, 1, 1\} = 1.$$

(9) $i = 3, j = 8, \rho(r_3, r_8) = \rho(G, C) = 1$

$$M_{3,8} = \max \begin{cases} M_{3,7} \\ (1 + M_{4,7}) \times \rho(r_3, r_8) \\ (1 + M_{3,3} + M_{5,7}) \times \rho(r_4, r_8) \end{cases}$$
$$= \max\{1, (1 + 0) \times 1, (1 + 0 + 0) \times 0\}$$
$$= \max\{1, 1, 0\} = 1.$$

r_3 matches with r_8.

(10) $i = 4, j = 9, \rho(r_4, r_9) = \rho(C, C) = 0$

$$M_{4,9} = \max \begin{cases} M_{4,8} \\ (1 + M_{5,8}) \times \rho(r_4, r_9) \\ (1 + M_{4,4} + M_{6,8}) \times \rho(r_5, r_9) \end{cases}$$
$$= \max\{0, (1 + 0) \times 0, (1 + 0 + 0) \times 0\}$$
$$= \max\{0, 0, 0\} = 0.$$

(11) $i = 5, j = 10, \rho(r_5, r_{10}) = \rho(C, U) = 0$

$$M_{5,10} = \max \begin{cases} M_{5,9} \\ (1 + M_{6,9}) \times \rho(r_5, r_{10}) \\ (1 + M_{5,5} + M_{7,9}) \times \rho(r_6, r_{10}) \end{cases}$$
$$= \max\{0, (1 + 0) \times 0, (1 + 0 + 0) \times 0\}$$
$$= \max\{0, 0, 0\} = 0.$$

(12) $i = 1, j = 7, \rho(r_1, r_7) = \rho(A, U) = 1$

$$M_{1,7} = \max \begin{cases} M_{1,6} \\ (1 + M_{2,6}) \times \rho(r_1, r_7) \\ (1 + M_{1,1} + M_{3,6}) \times \rho(r_2, r_7) \\ (1 + M_{1,2} + M_{4,6}) \times \rho(r_3, r_7) \end{cases}$$

$$= \max\{1, (1 + 1) \times 1, (1 + 0 + 0) \times 1, (1 + 0 + 0) \times 1\}$$
$$= \max\{1, 2, 1, 1\} = 2.$$

(13) $i = 2, j = 8, \rho(r_2, r_8) = \rho(G, C) = 1$

$$M_{2,8} = \max \begin{cases} M_{2,7} \\ (1 + M_{3,7}) \times \rho(r_2, r_8) \\ (1 + M_{2,2} + M_{4,7}) \times \rho(r_3, r_8) \\ (1 + M_{2,3} + M_{5,7}) \times \rho(r_4, r_8) \end{cases}$$

$$= \max\{1, (1 + 1) \times 1, (1 + 0 + 0) \times 1, (1 + 0 + 0) \times 0\}$$
$$= \max\{1, 2, 1, 0\} = 2.$$

r_2 matches with r_8; r_3 matches with r_7.

(14) $i = 3, j = 9, \rho(r_3, r_9) = \rho(G, C) = 1$

$$M_{3,9} = \max \begin{cases} M_{3,8} \\ (1 + M_{4,8}) \times \rho(r_3, r_9) \\ (1 + M_{3,3} + M_{5,8}) \times \rho(r_4, r_9) \\ (1 + M_{3,4} + M_{6,8}) \times \rho(r_5, r_9) \end{cases}$$

$$= \max\{1, (1 + 0) \times 1, (1 + 0 + 0) \times 0, (1 + 0 + 0) \times 0\}$$
$$= \max\{1, 1, 0, 0\} = 1.$$

r_3 matches with r_9.

(15) $i = 4, j = 10, \rho(r_4, r_{10}) = \rho(C, U) = 0$

$$M_{4,10} = \max \begin{cases} M_{4,9} \\ (1 + M_{5,9}) \times \rho(r_4, r_{10}) \\ (1 + M_{4,4} + M_{6,9}) \times \rho(r_5, r_{10}) \\ (1 + M_{4,5} + M_{7,9}) \times \rho(r_6, r_{10}) \end{cases}$$

$$= \max\{0, (1 + 0) \times 0, (1 + 0 + 0) \times 0, (1 + 0 + 0) \times 0\}$$
$$= \max\{0, 0, 0, 0\} = 0.$$

(16) $i = 1, j = 8, \rho(r_1, r_8) = \rho(A, C) = 0$

$$M_{1,8} = \max \begin{cases} M_{1,7} \\ (1 + M_{2,7}) \times \rho(r_1, r_8) \\ (1 + M_{1,1} + M_{3,7}) \times \rho(r_2, r_8) \\ (1 + M_{1,2} + M_{4,7}) \times \rho(r_3, r_8) \\ (1 + M_{1,3} + M_{5,7}) \times \rho(r_4, r_8) \end{cases}$$

$$= \max\{2, (1 + 1) \times 0, (1 + 0 + 1) \times 1, (1 + 0 + 0) \times 1, (1 + 0 + 0) \times 0\}$$

$$= \max\{2, 0, 2, 1, 0\} = 2.$$

r_1 matches with r_7; r_2 matches with r_6.

(17) $i = 2, j = 9, \rho(r_2, r_9) = \rho(G, C) = 1$

$$M_{2,9} = \max \begin{cases} M_{2,8} \\ (1 + M_{3,8}) \times \rho(r_2, r_9) \\ (1 + M_{2,2} + M_{4,8}) \times \rho(r_3, r_9) \\ (1 + M_{2,3} + M_{5,8}) \times \rho(r_4, r_9) \\ (1 + M_{2,4} + M_{6,8}) \times \rho(r_5, r_9) \end{cases}$$

$$= \max\{2, (1 + 1) \times 1, (1 + 0 + 0) \times 1, (1 + 0 + 0) \times 0, (1 + 0 + 0) \times 0\}$$

$$= \max\{2, 2, 1, 0, 0\} = 2.$$

r_2 matches with r_9; r_3 matches with r_8.

(18) $i = 3, j = 10, \rho(r_3, r_{10}) = \rho(G, U) = 1$

$$M_{3,10} = \max \begin{cases} M_{3,9} \\ (1 + M_{4,9}) \times \rho(r_3, r_{10}) \\ (1 + M_{3,3} + M_{5,9}) \times \rho(r_4, r_{10}) \\ (1 + M_{3,4} + M_{6,9}) \times \rho(r_5, r_{10}) \\ (1 + M_{3,5} + M_{7,9}) \times \rho(r_6, r_{10}) \end{cases}$$

$$= \max\{1, (1 + 0) \times 1, (1 + 0 + 0) \times 0, (1 + 0 + 0) \times 0, (1 + 0 + 0) \times 0\}$$

$$= \max\{1, 1, 0, 0, 0\} = 1.$$

r_3 matches with r_{10}.

(19) $i = 1, j = 9, \rho(r_1, r_9) = \rho(A, C) = 0$

$$M_{1,9} = \max \begin{cases} M_{1,8} \\ (1 + M_{2,8}) \times \rho(r_1, r_9) \\ (1 + M_{1,1} + M_{3,8}) \times \rho(r_2, r_9) \\ (1 + M_{1,2} + M_{4,8}) \times \rho(r_3, r_9) \\ (1 + M_{1,3} + M_{5,8}) \times \rho(r_4, r_9) \\ (1 + M_{1,4} + M_{6,8}) \times \rho(r_5, r_9) \end{cases}$$

$$= \max\{2, (1 + 2) \times 0, (1 + 0 + 1) \times 1, (1 + 0 + 0) \times 1, (1 + 0 + 0) \times 0, (1 + 0 + 0) \times 0\}$$

$$= \max\{2, 0, 2, 1, 0, 0\} = 2.$$

r_1 matches with r_7; r_2 matches with r_6.

(20) $i = 2, j = 10, \rho(r_2, r_{10}) = \rho(G, U) = 1$

$$M_{2,10} = \max \begin{cases} M_{2,9} \\ (1 + M_{3,9}) \times \rho(r_2, r_{10}) \\ (1 + M_{2,2} + M_{4,9}) \times \rho(r_3, r_{10}) \\ (1 + M_{2,3} + M_{5,9}) \times \rho(r_4, r_{10}) \\ (1 + M_{2,4} + M_{6,9}) \times \rho(r_5, r_{10}) \\ (1 + M_{2,5} + M_{7,9}) \times \rho(r_6, r_{10}) \end{cases}$$

$$= \max\{2, 2, 1, 0, 0, 0\} = 2.$$

r_2 matches with r_{10}; r_3 matches with r_9.

(21) $i = 1, j = 10, \rho(r_1, r_{10}) = \rho(A, U) = 1$

$$M_{1,10} = \max \begin{cases} M_{1,9} \\ (1 + M_{2,9}) \times \rho(r_1, r_{10}) \\ (1 + M_{1,1} + M_{3,9}) \times \rho(r_2, r_{10}) \\ (1 + M_{1,2} + M_{4,9}) \times \rho(r_3, r_{10}) \\ (1 + M_{1,3} + M_{5,9}) \times \rho(r_4, r_{10}) \\ (1 + M_{1,4} + M_{6,9}) \times \rho(r_5, r_{10}) \\ (1 + M_{1,5} + M_{7,9}) \times \rho(r_6, r_{10}) \end{cases}$$

$$= \max\{2, 3, 2, 1, 0, 0, 0\} = 3.$$

r_1 matches with r_{10}; r_2 matches with r_9; r_3 matches with r_8.

Next, we analyze the time complexity of Algorithm 7–1. Clearly, the cost of Step 1 is $O(n)$. For Step 2, there are at most $\sum_{i=1}^{n} \sum_{j=i+4}^{n}$ iterations and each iteration costs $O(j - i)$ time. Hence, the cost of Step 2 is

$$\sum_{i=1}^{n} \sum_{j=i+4}^{n} O(j-i) = O(n^3)$$

Hence, the total time complexity of Algorithm 7–1 is $O(n^3)$.

7–5 0/1 KNAPSACK PROBLEM

The 0/1 knapsack problem was discussed in Chapter 6. In this section, we shall show that this problem can be easily solved by using the dynamic programming approach. The 0/1 knapsack problem is defined as follows: We are given n objects and a knapsack. Object i has a weight W_i and the knapsack has a capacity M. If object i is placed into the knapsack, we will obtain a profit P_i. The 0/1 knapsack problem is to maximize the total profit under the constraint that the total weight of all chosen objects is at most M.

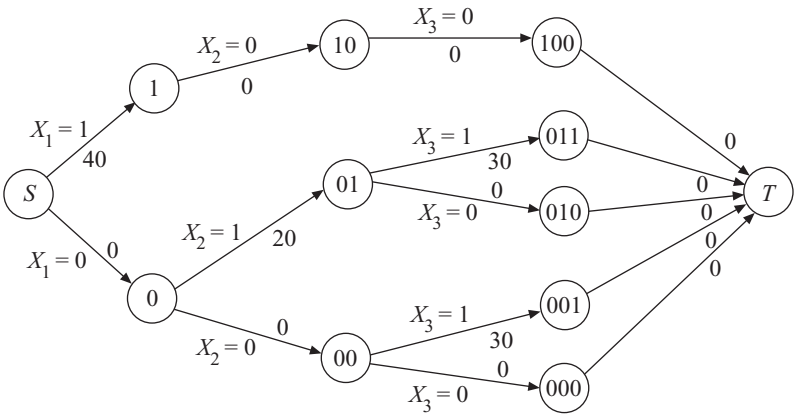
There are a sequence of actions to be taken. Let X_i be the variable denoting whether object i is chosen or not. That is, we let $X_i = 1$ if object i is chosen and 0 if it is not. If X_1 is assigned 1 (object 1 is chosen), then the remaining problem becomes a modified 0/1 knapsack problem where M becomes $M - W_1$. In general, after a sequence of decisions represented by $X_1, X_2, \ldots, X_i$ are made, the problem will be reduced to a problem involving decisions $X_{i+1}, X_{i+2}, \ldots, X_n$ and $M' = M - \sum_{j=1}^{i} X_j W_j$. Thus, whatever the decisions $X_1, X_2, \ldots, X_i$ are, the rest of decisions $X_{i+1}, X_{i+2}, \ldots, X_n$ must be optimal with respect to the new knapsack problem. In the following, we shall show that the 0/1 knapsack problem can be represented by a multi-stage graph problem. Let us suppose that we are given three objects and a knapsack with capacity 10. The weights and profits of these objects are shown in Table 7–6.

TABLE 7–6 Weights and profits of three objects.

i	W_i	P_i
1	10	40
2	3	20
3	5	30

The dynamic programming approach to solve this 0/1 knapsack problem can be illustrated in Figure 7–18.

FIGURE 7–18 The dynamic programming approach to solve the 0/1 knapsack problem.



In each node, we have a label specifying the decisions already made up to this node. For example, 011 means $X_1 = 0$, $X_2 = 1$ and $X_3 = 1$. In this case, we are interested in the longest path and it can be seen easily that the longest path is

$$S \rightarrow 0 \rightarrow 01 \rightarrow 011 \rightarrow T$$

which corresponds to

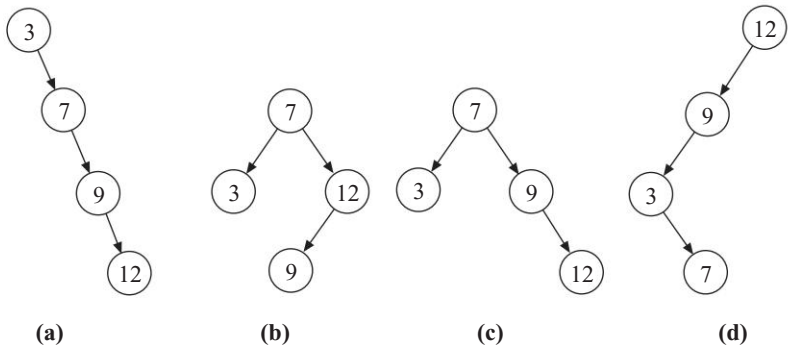
$$X_1 = 0,$$
$$X_2 = 1,$$
and $$X_3 = 1$$

with the total cost equal to $20 + 30 = 50$.

7–6 THE OPTIMAL BINARY TREE PROBLEM

The binary tree is probably one of the most popular data structures. Given n identifiers $a_1 < a_2 < a_3 \cdots < a_n$, we can have many different binary trees. For instance, let us assume that we are given four identifiers 3, 7, 9 and 12. Figure 7–19 lists four distinct binary trees for this set of data.

FIGURE 7–19 Four distinct binary trees for the same set of data.



For a given binary tree, the identifiers stored close to the root of the tree can be searched rather quickly while it takes many steps to retrieve data stored far away from the root. For each identifier a_i, we shall associate with it a probability P_i which is the probability that this identifier will be searched. For an identifier not stored in the tree, we shall also assume that there is a probability associated with it. We partition the data into $n + 1$ equivalence classes. We denote Q_i to be the probability that X will be searched where $a_i < X < a_{i+1}$, assuming that a_0 is $-\infty$ and a_{n+1} is $+\infty$. The probabilities satisfy the following equation:

$$\sum_{i=1}^{n} P_i + \sum_{i=0}^{n} Q_i = 1.$$

It is easy to determine the number of steps needed for a successful search. Let $L(a_i)$ denote the level of the binary tree where a_i is stored. Then the retrieval of a_i needs $L(a_i)$ steps if we let the root of the tree be in level 1. For unsuccessful searches, the best way to understand them is to add external nodes to the binary tree. Let us consider the case where we have identifiers 4, 5, 8, 10, 11, 12 and 14. For this set of data, we shall partition the entire region as follows:

$$\begin{array}{cccccccccccccc} & 4, & & 5, & & 8, & & 10, & & 11, & & 12, & & 14 & \\ \uparrow & & \uparrow & & \uparrow & & \uparrow & & \uparrow & & \uparrow & & \uparrow & & \uparrow \\ E_0 & & E_1 & & E_2 & & E_3 & & E_4 & & E_5 & & E_6 & & E_7 \end{array}$$

Suppose we have a binary tree as shown in Figure 7–20. In this binary tree, there are some nodes which do not have two descendent nodes. Unsuccessful searches always occur in these nodes. We may add fictitious nodes, drawn as squares, to all of these nodes, shown in Figure 7–21.

FIGURE 7–20 A binary tree.

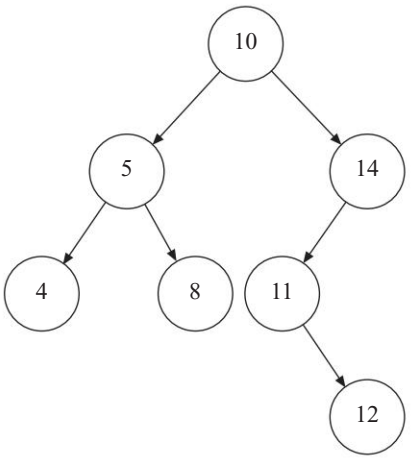
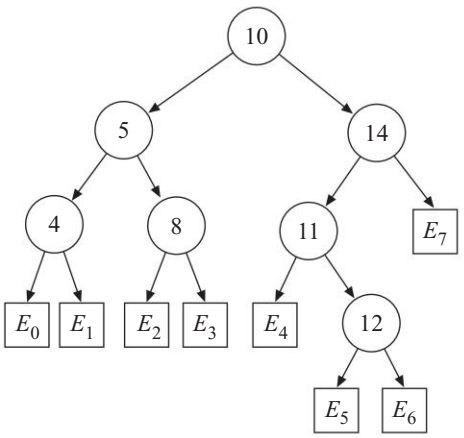


FIGURE 7–21 A binary tree with added external nodes.



All other nodes are called internal nodes. These added nodes are called external nodes. Unsuccessful searches always terminate at these external nodes, while successful searches terminate at internal nodes.

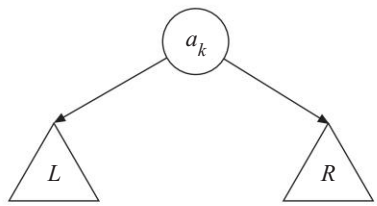
After external nodes are added, the expected cost of a binary tree can be expressed as follows:

$$\sum_{i=1}^{n} P_i L(a_i) + \sum_{i=0}^{n} Q_i (L(E_i) - 1).$$

An optimal binary tree is a binary tree with the above cost being minimized. It should be noted that it will be exceedingly time consuming to solve the optimal binary tree problem by exhaustively examining all possible binary trees. Given n identifiers, the number of all distinct binary trees that can be constructed out of these n identifiers is $\frac{1}{n+1}\binom{2n}{n}$, which is approximately $O(4^n/n^{3/2})$.

Why is it possible to apply the dynamic programming approach to solve the optimal binary tree problem? A critical first step to find an optimal binary tree is to select an identifier, say, a_k, to be the root of the tree. After a_k is selected, all identifiers smaller than a_k will constitute the left descendant of a_k and all identifiers greater than a_k will constitute the right descendant, illustrated in Figure 7–22.

FIGURE 7–22 A binary tree after a_k is selected as the root.

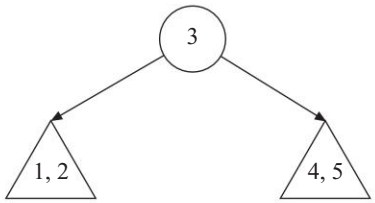


We cannot easily determine which a_k should be selected, and we need to examine all possible a_k's. However, once a_k is selected, we note that both subtrees L and R must also be optimal. That is, after a_k is selected, our job reduces to finding optimal binary trees for identifiers smaller than a_k and identifiers larger than a_k. That we can solve this problem recursively indicates that we can use the dynamic programming approach.

Why is the dynamic programming approach appropriate to find an optimal binary tree? First of all, it gives us a systematic way of thinking. To illustrate this, suppose that there are five identifiers: 1, 2, 3, 4 and 5. Suppose that 3 is selected as the root of the tree. Then, as shown is Figure 7–23, the left subtree will contain 1 and 2 and the right subtree will contain 4 and 5.

Our next step is to find optimal binary trees for 1 and 2 and for 4 and 5. For the binary tree containing 1 and 2, there are only two choices as the root, either 1 or 2. Similar situations occur for the binary tree containing 4 and 5. In other words, the optimal binary tree with 3 as its root is determined by the following four subtrees:

FIGURE 7–23 A binary tree with a certain identifier selected as the root.



(a) A subtree containing 1 and 2 with 1 as its root.
(b) A subtree containing 1 and 2 with 2 as its root.
(c) A subtree containing 4 and 5 with 4 as its root.
(d) A subtree containing 4 and 5 with 5 as its root.

The left subtree will be either (a) or (b) and the right subtree will be either (c) or (d).

Let us now consider the case of selecting 2 as the root. In this case, the left subtree will contain only one identifier, say 1. The right subtree will contain three identifiers: 3, 4 and 5. For this right subtree, we must consider three subcases: selecting 3 as the root, selecting 4 as the root and selecting 5 as the root. Let us pay particular attention to the case of selecting 3 as the root. If 3 is selected, then its right subtree will contain 4 and 5. That is, we again must examine, so far as a subtree containing 4 and 5 is concerned, whether we should select 4 or 5, as its root. We thus conclude that in order to find an optimal binary tree with 2 as its root, we should examine, among other subtrees, the following two trees:

(e) A subtree containing 4 and 5 with 4 as its root.
(f) A subtree containing 4 and 5 with 5 as its root.

It is obvious that (c) is equivalent to (e) and (d) is equivalent to (f). This means that the work done in finding an optimal binary tree with 3 as its root is also useful in finding an optimal binary tree with 2 as its root. By a similar reasoning, we can see that in finding an optimal binary tree with 4 as its root, we do need to find an optimal binary tree containing 1 and 2, with either 1 or 2 as its root. Again, as pointed out before, this work is needed in finding an optimal binary tree with 3 as its root.

In summary, we note that given $a_1, a_2, \ldots, a_n$, the work needed to find an optimal binary tree with, say a_i, as its root, may also be needed in finding an optimal binary tree with, say a_j, as its root. This principle can be applied to all subtrees at all levels. Dynamic programming would therefore solve the optimal binary tree problem by working bottom up. That is, we start by building small optimal binary trees. Using these small optimal binary trees, we build larger and larger optimal binary trees. We reach our goal when an optimal binary tree containing all identifiers has been found.

Let us be more specific. Suppose that we must find an optimal binary tree for four identifiers: 1, 2, 3 and 4. In the following, we shall use the notation $(a_k, a_i \rightarrow a_j)$ to denote an optimal binary tree containing identifiers a_i to a_j and with a_k as its root. We shall also use $(a_i \rightarrow a_j)$ to denote an optimal binary tree containing identifiers a_i to a_j. Our dynamic programming process of finding an optimal binary tree containing 1, 2, 3 and 4 is now described as follows:

(1) We start by finding
$(1, 1 \rightarrow 2)$
$(2, 1 \rightarrow 2)$
$(2, 2 \rightarrow 3)$
$(3, 2 \rightarrow 3)$
$(3, 3 \rightarrow 4)$
$(4, 3 \rightarrow 4)$.

(2) Using the above results, we can determine
$(1 \rightarrow 2)$ (Determined by $(1, 1 \rightarrow 2)$ and $(2, 1 \rightarrow 2)$)
$(2 \rightarrow 3)$
$(3 \rightarrow 4)$.

(3) We then find
$(1, 1 \rightarrow 3)$ (Determined by $(2 \rightarrow 3)$)
$(2, 1 \rightarrow 3)$
$(3, 1 \rightarrow 3)$
$(2, 2 \rightarrow 4)$
$(3, 2 \rightarrow 4)$
$(4, 2 \rightarrow 4)$.

(4) Using the above results, we can determine
$(1 \rightarrow 3)$ (Determined by $(1, 1 \rightarrow 3)$, $(2, 1 \rightarrow 3)$ and $(3, 1 \rightarrow 3)$)
$(2 \rightarrow 4)$.

(5) We then find
$(1, 1 \rightarrow 4)$ (Determined by $(2 \rightarrow 4)$)
$(2, 1 \rightarrow 4)$
$(3, 1 \rightarrow 4)$
$(4, 1 \rightarrow 4)$.

(6) Finally, we can determine
$(1 \rightarrow 4)$
because it is determined by
$(1, 1 \rightarrow 4)$
$(2, 1 \rightarrow 4)$
$(3, 1 \rightarrow 4)$
$(4, 1 \rightarrow 4)$.

Whichever of the above four binary trees with the minimum cost will be an optimal one.

The reader can see that dynamic programming is an efficient way to solve the optimal binary tree problem. It not only offers a systematic way of thinking, but also avoids redundant computations.

Having described the basic principles of solving the optimal binary tree problem by using the dynamic programming approach, we can now give the precise mechanism. Since the search for an optimal binary tree consists of finding many optimal binary subtrees, we simply consider the general case of finding an arbitrary subtree. Imagine that we are given a sequence of identifiers, $a_1, a_2, \ldots, a_n$ and a_k is one of them. If a_k is selected as the root of the binary tree, then the left (right) subtree will contain all identifiers $a_1, \ldots, a_{k-1}(a_{k+1}, \ldots, a_n)$. Besides, the retrieval of a_k will need one step and all other successful searches need one plus the number of steps needed to search either the left, or the right, subtree. This is also true for all the unsuccessful searches.

Let $C(i, j)$ denote the cost of an optimal binary tree containing a_i to a_j. Then, the optimal binary tree with a_k as its root will have the following cost

$$C(1, n) = \min_{1 \le k \le n} \left\{ P_k + \left[Q_0 + \sum_{i=1}^{k-1} (P_i + Q_i) + C(1, k-1) \right] + \left[Q_k + \sum_{i=k+1}^{n} (P_i + Q_i) + C(k+1, n) \right] \right\},$$

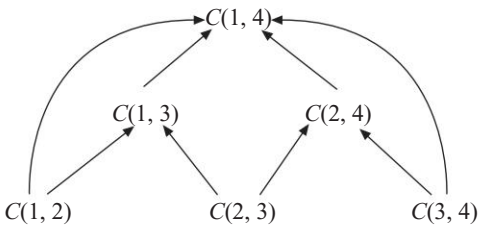
where P_k is the cost of searching for root and the second and third terms are the costs of searching for the left subtree and the right subtree respectively. The above formula can be generalized to obtain any $C(i, j)$ as follows:

$$C(i,j) = \min_{i \le k \le j} \left\{ P_k + \left[Q_{i-1} + \sum_{l=i}^{k-1} (P_l + Q_l) + C(i, k-1) \right] + \left[Q_k + \sum_{l=k+1}^{j} (P_l + Q_l) + C(k+1, j) \right] \right\}$$

$$= \min_{i \le k \le j} \left\{ C(i, k-1) + C(k+1, j) + \sum_{l=i}^{j} (P_l + Q_l) + Q_{i-1} \right\}.$$

For example, if we have four identifiers a_1, a_2, a_3 and a_4, the objective will be finding $C(1, 4)$. Since there are four possible choices for root, we need to compute the cost of four optimal subtrees, $C(2, 4)$ (a_1 as the root), $C(3, 4)$ (a_2 as the root), $C(1, 2)$ (a_3 as its root) and $C(1, 3)$ (a_4 as its root). To compute $C(2, 4)$, we need $C(3, 4)$ (a_2 as the root) and $C(2, 3)$ (a_4 as the root) and to compute $C(1, 3)$, we need $C(2, 3)$ (a_1 as the root) and $C(1, 2)$ (a_3 as the root). Their relationships can be illustrated as in Figure 7–24.

FIGURE 7–24 Computation relationships of subtrees.



In general, given n identifiers to compute $C(1, n)$, we can proceed by first computing all $C(i, j)$'s with $j - i = 1$. Next, we can compute all $C(i, j)$'s with $j - i = 2$, then all $C(i, j)$'s with $j - i = 3$, and so on. We now examine the complexity of this procedure. This evaluation procedure requires us to compute $C(i, j)$ for $j - i = 1, 2, \ldots, n - 1$. When $j - i = m$, there are $(n - m)$ $C(i, j)$'s to compute. The computation of each of these $C(i, j)$'s requires us to find the minimum of m quantities. Hence, each $C(i, j)$ with $j - i = m$ can be computed in time $O(m)$. The total time for computing all $C(i, j)$'s with $j - i = m$ is

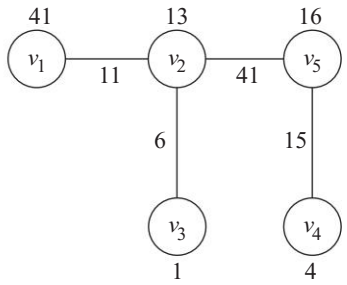
$O(mn - m^2)$. Therefore, the total time complexity of this dynamic programming approach to solve the optimal binary search tree problem is $O(\sum_{1 \le m \le n} (nm - m^2)) = O(n^3)$.

7–7 THE WEIGHTED PERFECT DOMINATION PROBLEM ON TREES

In the weighted perfect domination problem, we are given a graph $G = (V, E)$ where every vertex $v \in V$ has a cost $c(v)$ and every edge $e \in E$ also has a cost $c(e)$. A perfect dominating set of G is a subset D of vertices of V such that every vertex not in D is adjacent to exactly one vertex in D. The cost of a perfect dominating set D includes all the costs of the vertices in D and the cost of $c(u, v)$ if v does not belong to D, u belongs to D and (u, v) is an edge in E. The weighted perfect domination problem is to find a perfect dominating set with minimum cost.

Consider Figure 7–25. There are many perfect dominating sets.

FIGURE 7–25 A graph illustrating the weighted perfect domination problem.



For example, $D_1 = \{v_1, v_2, v_5\}$ is a perfect dominating set. The cost of D_1 is

$$c(v_1) + c(v_2) + c(v_5) + c(v_3, v_2) + c(v_4, v_5)$$
$$= 41 + 13 + 16 + 6 + 15$$
$$= 91.$$

Another example is $D_2 = \{v_2, v_5\}$. The cost of D_2 is

$$c(v_2) + c(v_5) + c(v_1, v_2) + c(v_3, v_2) + c(v_4, v_5)$$
$$= 13 + 16 + 11 + 6 + 15$$
$$= 61.$$

Finally, let D_3 be $\{v_2, v_3, v_4, v_5\}$. The cost of D_3 is

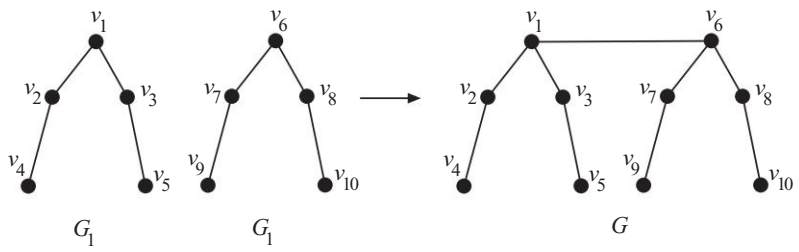
$$c(v_2) + c(v_3) + c(v_4) + c(v_5) + c(v_1, v_2)$$
$$= 13 + 1 + 4 + 16 + 11$$
$$= 45.$$

It can be proved that D_3 is a minimal cost perfect dominating set.

The weighted perfect domination problem is NP-hard for bipartite graphs and chordal graphs. But, we shall show that by applying the dynamic programming approach, we can solve the weighted perfect domination problem on trees.

The main job of applying the dynamic programming strategy is to merge two feasible solutions into a new feasible solution. We now illustrate our merging through an example. Consider Figure 7–26. In Figure 7–26, there are two graphs: G_1 and G_2 which will be merged into G by linking v_1 of G_1 and v_6 of G_2.

FIGURE 7–26 An example to illustrate the merging scheme in solving the weighted perfect domination problem.



Since we combine G_1 and G_2 by linking v_1 of G_1 and v_6 of G_2, we may consider G_1 as rooted at v_1 and G_2 as rooted at v_6. Let us consider six perfect dominating sets:

$$D_{11} = \{v_1, v_2, v_3\},\ D_{21} = \{v_6, v_7, v_8\}$$
$$D_{12} = \{v_3, v_4\},\ D_{22} = \{v_7, v_{10}\}$$
$$D_{13} = \{v_4, v_5\},\ D_{23} = \{v_9, v_{10}\}.$$

$D_{11}(D_{21})$ is a perfect dominating set of $G_1(G_2)$ under the condition that the root of $G_1(G_2)$, $v_1(v_6)$, is included in it.

$D_{12}(D_{22})$ is a perfect dominating set of $G_1(G_2)$ under the condition that the root of $G_1(G_2)$, $v_1(v_6)$, is not included in it.

$D_{13}(D_{23})$ is a perfect dominating set of $G_1 - \{v_1\}(G_2 - \{v_6\})$ under the condition that none of v_1's (v_6's) neighbors are included in it.

We can now produce perfect dominating sets for G by combining the above perfect dominating sets for G_1 and G_2.

(1) $D_{11} \bigcup D_{21} = \{v_1, v_2, v_3, v_6, v_7, v_8\}$ is a perfect dominating set of G. The cost of it is the sum of costs of D_{11} and D_{21}. This perfect dominating set includes both v_1 and v_6.
(2) $D_{11} \bigcup D_{23} = \{v_1, v_2, v_3, v_9, v_{10}\}$ is a perfect dominating set of G. Its cost is the sum of costs of D_{11} and D_{23} plus the cost of the edge linking v_1 and v_6. This perfect dominating set includes v_1 and does not include v_6.
(3) $D_{12} \bigcup D_{22} = \{v_3, v_4, v_7, v_{10}\}$ is a perfect dominating set of G. Its cost is the sum of costs of D_{12} and D_{22}. This perfect dominating set includes neither v_1 nor v_6.
(4) $D_{13} \bigcup D_{21} = \{v_4, v_5, v_6, v_7, v_8\}$ is a perfect dominating set of G. Its cost is the sum of costs of D_{13} and D_{21} plus the cost of the edge linking v_1 and v_6. This perfect dominating set does not include v_1 and includes v_6.

The above discussion describes the basis of our strategy of applying the dynamic programming approach to solve the weighted perfect domination problem.

In the following, we shall first assume that a graph is rooted at a certain vertex and when we combine two graphs, we link the two roots.

We define the following:

$D_1(G, u)$: an optimal perfect dominating set for graph G under the condition that the perfect dominating set includes u. The cost of $D_1(G, u)$ is denoted as $\delta_1(G, u)$.

$D_2(G, u)$: an optimal perfect dominating set for graph G under the condition that the perfect dominating set does not include u. The cost of $D_2(G, u)$ is denoted as $\delta_2(G, u)$.

$D_3(G, u)$: an optimal perfect dominating set for graph $G - \{u\}$ under the condition that the perfect dominating set of $G - \{u\}$ does not include any neighboring vertex of u. The cost of $D_3(G, u)$ is denoted as $\delta_3(G, u)$.

Given two graphs G_1 and G_2 rooted at u and v respectively, let G denote the graph obtained by linking u and v. Then, obviously, we shall have the following rules:

RULE 1: Both $D_1(G_1, u) \bigcup D_1(G_2, v)$ and $D_1(G_1, u) \bigcup D_3(G_2, v)$ are perfect dominating sets of G which include u.

RULE 1.1: The cost of $D_1(G_1, u) \bigcup D_1(G_2, v)$ is $\delta_1(G_1, u) + \delta_1(G_2, v)$.
RULE 1.2: The cost of $D_1(G_1, u) \bigcup D_3(G_2, v)$ is $\delta_1(G_1, u) + \delta_3(G_2, v) + c(u, v)$.

RULE 2: Both $D_2(G_1, u) \bigcup D_2(G_2, v)$ and $D_3(G_1, u) \bigcup D_1(G_2, v)$ are perfect dominating sets of G which do not include u.

RULE 2.1: The cost of $D_2(G_1, u) \bigcup D_2(G_2, v)$ is $\delta_2(G_1, u) + \delta_2(G_2, v)$.
RULE 2.2: The cost of $D_3(G_1, u) \bigcup D_1(G_2, v)$ is $\delta_3(G_1, u) + \delta_1(G_2, v) + c(u, v)$.

The basic principle of using dynamic programming strategy to find an optimal perfect dominating set of a graph can be summarized as follows:

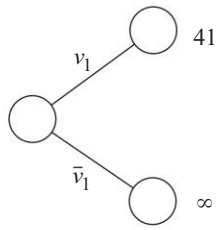
If a graph G can be decomposed into two subgraphs G_1 and G_2 in such a way that G can be reconstructed by linking u of G_1 and v of G_2, then an optimal perfect dominating set of G denoted as $D(G)$ can be found as follows:

(1) If $\delta_1(G_1, u) + \delta_1(G_2, v)$ is less than $\delta_1(G_1, u) + \delta_3(G_2, v) + c(u, v)$, then set $D_1(G, u)$ to be $D_1(G_1, u) \bigcup D_1(G_2, v)$ and $\delta_1(G, u)$ to be $\delta_1(G_1, u) + \delta_1(G_2, v)$. Otherwise, set $D_1(G, u)$ to be $D_1(G_1, u) \bigcup D_3(G_2, v)$ and $\delta_1(G, u)$ to be $\delta_1(G_1, u) + \delta_3(G_2, v) + c(u, v)$.
(2) If $\delta_2(G_1, u) + \delta_2(G_2, v)$ is less than $\delta_3(G_1, u) + \delta_1(G_2, v) + c(u, v)$, set $D_2(G, u)$ to be $D_2(G_1, u) \bigcup D_2(G_2, v)$ and $\delta_2(G, u)$ to be $\delta_2(G_1, u) + \delta_2(G_2, v)$. Otherwise, set $D_2(G, u)$ to be $D_3(G_1, u) \bigcup D_1(G_2, v)$ and $\delta_2(G, u)$ to be $\delta_3(G_1, u) + \delta_1(G_2, v) + c(u, v)$.
(3) $D_3(G, u) = D_3(G_1, u) \bigcup D_2(G_2, v)$.
$\delta_3(G, u) = \delta_3(G_1, u) + \delta_2(G_2, v)$.
(4) If $\delta_1(G, u)$ is less than $\delta_2(G, u)$, set $D(G)$ to be $D_1(G, u)$. Otherwise, set $D(G)$ to be $D_2(G, u)$.

The above rules are for general graphs. In the following, we shall focus on trees. For trees, we shall use a special algorithm which starts from the leaf nodes and works towards the inner part of the tree. Before introducing the special algorithm for trees, we illustrate this algorithm with a complete example. Consider Figure 7–25 again. Let us start from leaf node v_1 only. There are only

two cases: The perfect dominating set either contains v_1 or does not contains v_1, as shown in Figure 7–27.

FIGURE 7–27 The computation of the perfect dominating set involving v_1.



It is easy to see that

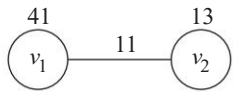
$$D_1(\{v_1\}, v_1) = \{v_1\},$$
$$D_2(\{v_1\}, v_1) \text{ does not exist},$$
$$D_3(\{v_1\}, v_1) = \phi,$$
$$\delta_1(\{v_1\}, v_1) = c(v_1) = 41,$$
$$\delta_2(\{v_1\}, v_1) = \infty,$$
and $$\delta_3(\{v_1\}, v_1) = 0.$$

Now we consider the subtree containing v_2 only. Again, we can see the following:

$$D_1(\{v_2\}, v_2) = \{v_2\},$$
$$D_2(\{v_2\}, v_2) \text{ does not exist},$$
$$D_3(\{v_2\}, v_2) = \phi,$$
$$\delta_1(\{v_2\}, v_2) = c(v_2) = 13,$$
$$\delta_2(\{v_2\}, v_2) = \infty,$$
and $$\delta_3(\{v_2\}, v_2) = 0.$$

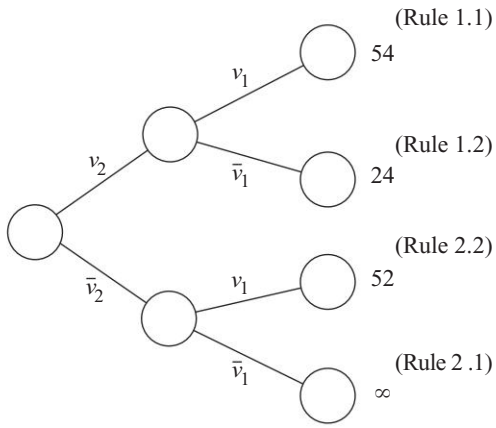
Consider the subtree containing v_1 and v_2, shown in Figure 7–28.

FIGURE 7–28 The subtree containing v_1 and v_2.



The computation of the perfect dominating set for $\{v_1, v_2\}$, through dynamic programming is illustrated in Figure 7–29.

FIGURE 7–29 The computation of the perfect dominating set of the subtree containing v_1 and v_2.



Since min(54, 24) = 24, we have

$D_1(\{v_1, v_2\}, v_2) = \{v_2\}$,
$\delta_1(\{v_1, v_2\}, v_2) = 24$.

Since min(∞, 52) = 52, we have

$D_2(\{v_2, v_1\}, v_2) = \{v_1\}$,
$\delta_2(\{v_2, v_1\}, v_2) = 52$.

Besides,

$D_3(\{v_1, v_2\}, v_2)$ does not exist,
$\delta_3(\{v_1, v_2\}, v_2) = \infty$.

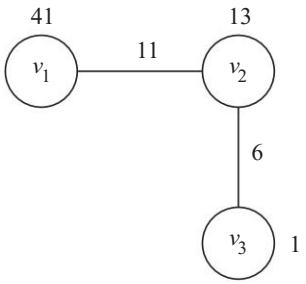
Consider the tree containing v_3 only. It is obvious that

$D_1(\{v_3\}, v_3) = \{v_3\}$,
$D_2(\{v_3\}, v_3)$ does not exist,

$D_3(\{v_3\}, v_3) = \phi$,
$\delta_1(\{v_3\}, v_3) = c(v_3) = 1$,
$\delta_2(\{v_3\}, v_3) = \infty$,
and $\delta_3(\{v_3\}, v_3) = 0$.

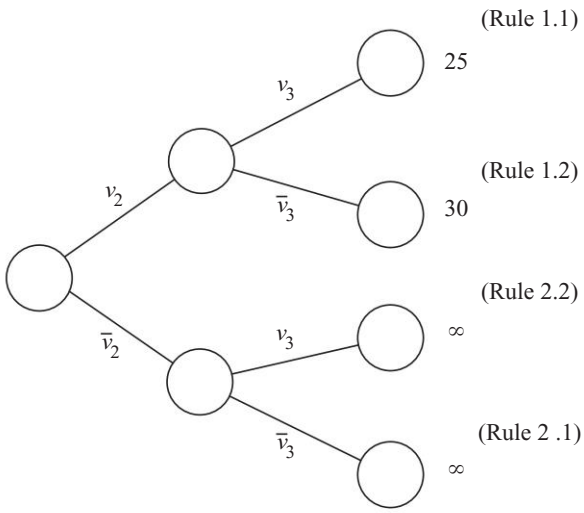
We add v_3 to the subtree containing v_1 and v_2, shown in Figure 7–30.

FIGURE 7–30 The subtree containing v_1, v_2 and v_3.



The computation of the perfect dominating set for this subtree is illustrated in Figure 7–31.

FIGURE 7–31 The computation of the perfect dominating set of the subtree containing v_1, v_2 and v_3.



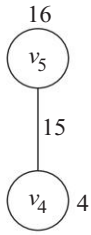
From the computation, we have

$$D_1(\{v_1, v_2, v_3\}, v_2) = \{v_2, v_3\},$$
$$D_2(\{v_1, v_2, v_3\}, v_2) \text{ does not exist,}$$
$$D_3(\{v_1, v_2, v_3\}, v_2) \text{ does not exist,}$$
$$\delta_1(\{v_1, v_2, v_3\}, v_2) = c(v_2) + c(v_3) + c(v_1, v_2) = 13 + 1 + 11 = 25,$$
$$\delta_2(\{v_1, v_2, v_3\}, v_2) = \infty,$$

and $\delta_3(\{v_1, v_2, v_3\}, v_2) = \infty.$

Consider the subtree containing v_5 and v_4 shown in Figure 7–32.

FIGURE 7–32 The subtree containing v_5 and v_4.



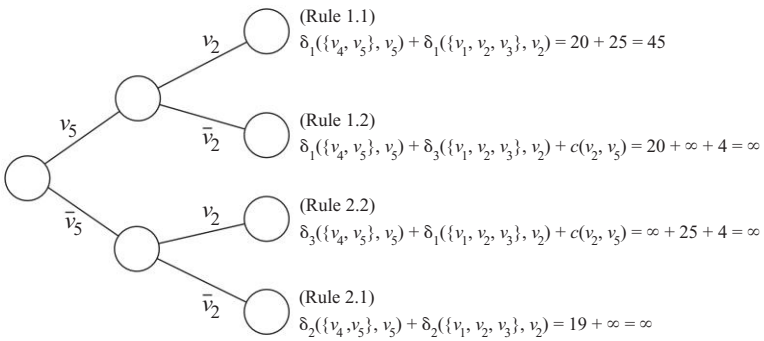
It can be easily seen that the following is true.

$$D_1(\{v_5, v_4\}, v_5) = \{v_5, v_4\},$$
$$D_2(\{v_5, v_4\}, v_5) = \{v_4\},$$
$$D_3(\{v_5, v_4\}, v_5) \text{ does not exist,}$$
$$\delta_1(\{v_5, v_4\}, v_5) = c(v_5) + c(v_4) = 16 + 4 = 20,$$
$$\delta_2(\{v_5, v_4\}, v_5) = c(v_4) + c(v_5, v_4) = 4 + 15 = 19,$$

and $\delta_3(\{v_5, v_4\}, v_5) = \infty.$

We now consider the subtree containing v_1, v_2 and v_3 and the subtree containing v_4 and v_5. That is, we now consider the entire tree. The computation of its perfect dominating set is shown in Figure 7–33.

FIGURE 7–33 The computation of the perfect dominating set of the entire tree shown in Figure 7–25.



We can conclude that the perfect dominating set with minimum cost is $\{v_2, v_3, v_4, v_5\}$ and its cost is 45.

In the following algorithm, let $TP(u)$ denote the tree partially constructed in the process which is rooted at u.

Algorithm 7–2 □ An algorithm to solve the weighted perfect domination problem on trees

Input: A tree $T = (V, E)$ in which every vertex $v \in V$ has a cost $c(v)$ and every edge $e \in E$ has a cost $c(e)$.

Output: A perfect dominating set $D(T)$ of T with minimal cost.

Step 1: For each vertex $v \in V$, do

$TP(v) = \{v\}$

$D_1(TP(v), v) = TP(v)$

$\delta_1(TP(v), v) = c(v)$

$D_2(TP(v), v)$ does not exist

$\delta_2(TP(v), v) = \infty$

$D_3(TP(v), v) = \phi$

$\delta_3(TP(v), v) = 0.$

Step 2: $T' = T.$

Step 3: While T' has more than one vertex, do

choose a leave v of T' which is adjacent to a unique vertex u in T'.

Set $TP'(u) = TP(u) \bigcup TP(v) \bigcup \{u, v\}$.

If $(\delta_1(TP(u), u) + \delta_1(TP(v), v)) < (\delta_1(TP(u), u) + \delta_3(TP(v), v) + c(u, v))$

$D_1(TP'(u), u) = D_1(TP(u), u) \bigcup D_1(TP(v), v)$

$\delta_1(TP'(u), u) = \delta_1(TP(u), u) + \delta_1(TP(v), v)$.

Otherwise,

$D_1(TP'(u), u) = D_1(TP(u), u) \bigcup D_3(TP(v), v)$

$\delta_1(TP'(u), u) = \delta_1(TP(u), u) + \delta_3(TP(v), v) + c(u, v)$.

If $(\delta_2(TP(u), u) + \delta_2(TP(v), v)) < (\delta_3(TP(u), u) + \delta_1(TP(v), v) + c(u, v))$

$D_2(TP'(u), u) = D_2(TP(u), u) \bigcup D_2(TP(v), v)$

$\delta_2(TP'(u), u) = \delta_2(TP(u), u) + \delta_2(TP(v), v)$.

Otherwise,

$D_2(TP'(u), u) = D_3(TP(u), u) \bigcup D_1(TP(v), v)$

$\delta_2(TP'(u), u) = \delta_3(TP(u), u) + \delta_1(TP(v), v) + c(u, v)$

$D_3(TP'(u), u) = D_3(TP(u), u) \bigcup D_2(TP(v), v)$

$\delta_3(TP'(u), u) = \delta_3(TP(u), u) + \delta_2(TP(v), v)$

$TP(u) = TP'(u)$

$T' = T' - v$;

end while.

Step 4: If $\delta_1(TP(u), u) < \delta_2(TP(u), u)$

Set $D(T) = D_1(TP(u), u)$.

Otherwise,

set $D(T) = D_2(TP(u), u)$.

Return $D(T)$ as the minimal perfect dominating set of $T(V, E)$.

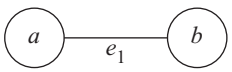
7–8 THE WEIGHTED SINGLE STEP GRAPH EDGE SEARCHING PROBLEM ON TREES

In the weighted single step graph edge searching problem, we are given a simple undirected graph $G = (V, E)$ where each vertex $v \in V$ is associated with a weight $wt(v)$. Assume that every edge of G is of the same length. A fugitive who can move in any speed is assumed to be hidden in some edge of G. For each edge of G, an edge searcher is assigned to search it. The edge searcher always starts from a vertex. The cost of clearing an edge (u, v) is defined as $wt(u)$ if the edge searcher starts from u and as $wt(v)$ if he searches from v. Assume that every edge searcher moves with the same speed. The weighted single step graph edge searching problem is to arrange the searching directions of edge searchers in such a way that the fugitive will be captured in one step and the number of searchers used is minimized.

It is clear that at least m edge searchers are required if there are m edges. Since the fugitive can move as fast as the edge searchers and every edge searcher moves forward, the team of searchers can capture the fugitive in a single step if the fugitive cannot sneak through any searched vertex and hide behind one of the edge searchers.

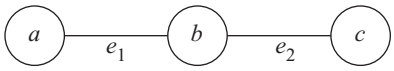
Let us consider Figure 7–34. In this case, only one searcher is needed. No matter where the searcher is initially located, the fugitive cannot escape. Assume that the searcher is located at a and searches towards b. The fugitive can at most reach b. But then the searcher will finally reach b and capture him. Similarly, if a searcher is initially placed at b, the same purpose can be achieved. That is, the fugitive can be captured in one step.

FIGURE 7–34 A case where no extra searchers are needed.



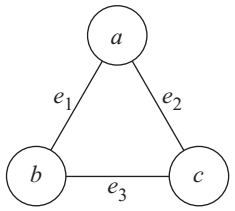
Now, consider Figure 7–35. By placing searchers at a and c, we can also capture the fugitive in one step.

FIGURE 7–35 Another case where no extra searchers are needed.



Now consider Figure 7–36.

FIGURE 7–36 A case where extra searchers are needed.



In this case, if only three searchers are used, then no matter how we arrange the searching directions, it is possible that a fugitive can avoid capture. For instance, consider the following searching plan:

(1) The edge searcher of e_1 searches from a to b.
(2) The edge searcher of e_2 searches from a to c.
(3) The edge searcher of e_3 searches from b to c.

Then, if the fugitive is originally in e_1, he may stay undetected by moving to e_3.

Suppose that we place an extra searcher at vertex b and apply the same searching described above, then no matter where the fugitive was originally, he will be detected. This shows that in some cases, extra searchers are needed.

A single step searching plan is to arrange the searching directions of edge searchers and determine the minimum number of extra searchers needed. The cost of a single step searching plan is the sum of the costs of all searchers. The weighted single step graph edge searching problem is to find a feasible single step searching plan with minimum cost. For the above example, the cost of this searching plan is $2wt(a) + wt(b) + wt(b)$.

The weighted single step graph edge searching problem is NP-hard for general graphs. But we shall show that by applying the dynamic programming strategy, we can solve the weighted single step graph edge searching problem in trees in polynomial time.

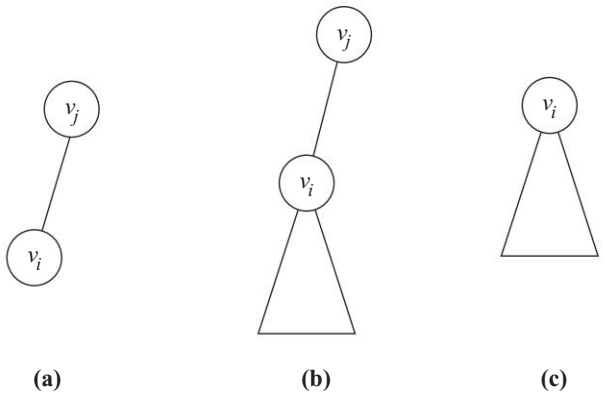
Our dynamic programming approach to solve this problem is based upon several basic rules which will be explained below.

We first define some notations.

Let v_i be a vertex of a tree.

Case 1: v_i is a leaf node. Then $T(v_i)$ denotes v_i and its parent node, shown in Figure 7–37(a).

FIGURE 7–37 Definition of $T(v_i)$.



Case 2: v_i is an internal node, but not the root node. Then $T(v_i)$ denotes the tree involving v_i, the parent node v_j and all of the descendant nodes of v_i, shown in Figure 7–37(b).

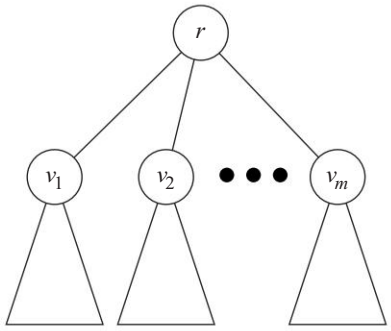
Case 3: v_i is the root of a tree T. Then $T(v_i)$ denotes T, as shown in Figure 7–37(c).

Furthermore, let v_j be the parent node of v_i. Then $C(T(v_i), v_i, v_j)$ ($C(T(v_i), v_j, v_i)$) denotes the cost of an optimal single step searching plan where the searching direction of (v_j, v_i) is from v_i to v_j (v_j to v_i).

RULE 1: Let r be the root of a tree. Let $C(T(r), r)(C(T(r), \bar{r}))$ be the cost of an optimal single step searching plan for T with (without) an extra searcher stationed at r. Thus, the cost of an optimal single step searching plan for T, denoted as $C(T(r))$, is $\min\{C(T(r), r), C(T(r), \bar{r})\}$.

RULE 2: Let r be the root of a tree where $v_1, v_2, \ldots, v_m$ are descendants of r (see Figure 7–38). If there is no extra guard stationed at r, then the searching directions of $(r, v_1), (r, v_2), \ldots, (r, v_m)$ are either all from r to $v_1, v_2, \ldots, v_m$ or all from $v_1, v_2, \ldots, v_m$ to r. If an extra guard is stationed at r, then the searching direction of each (r, v_i) can be determined independently.

FIGURE 7–38 An illustration of Rule 2.



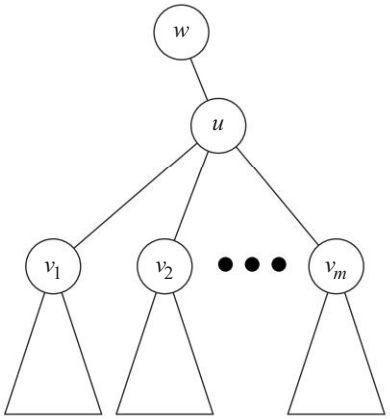
That is,

$$C(T(r), \bar{r}) = \min\left\{\sum_{i=1}^{m} C(T(v_i), r, v_i), \sum_{i=1}^{m} C(T(v_i), v_i, r)\right\}$$

$$C(T(r), r) = wt(r) + \sum_{i=1}^{m} \min\{C(T(v_i), r, v_i), C(T(v_i), v_i, r)\} \ .$$

RULE 3: Let u be an internal node, w be its parent node and $v_1, v_2, \ldots, v_m$ be its descendant nodes (Figure 7–39).

FIGURE 7–39 An illustration of Rule 3.



If there is no extra guard stationed at u, then if the searching direction of (w, u) is from w to u (from u to w), all searching directions of (u, v_1), (u, v_2), … , (u, v_m) are from $v_1, v_2, \ldots, v_m$ to u (from u to $v_1, v_2, \ldots, v_m$). If an extra guard is stationed at u, then the searching directions of (u, v_j) can be determined independently. Let $C(T(u), w, u, \bar{u})$ ($C(T(u), w, u, u)$) denote the cost of an optimal single step searching plan where the searching direction of edge (u, w) is from w to u and no extra guard is placed at vertex u (an extra guard is stationed at vertex u). Similarly, let $C(T(u), u, w, \bar{u})$ ($C(T(u), u, w, u)$) denote the cost of an optimal single step searching plan where the searching direction of edge (u, w) is from u to w and no extra guard is placed at vertex u (an extra guard is stationed at vertex u). Then we have the following formulas.

$$C(T(u), w, u, \bar{u}) = wt(w) + \sum_{i=1}^{m} C(T(v_i), v_i, u),$$

$$C(T(u), w, u, u) = wt(w) + wt(u) + \sum_{i=1}^{m} \min\{C(T(v_i), v_i, u), C(T(v_i), u, v_i)\},$$

$$C(T(u), u, w, \bar{u}) = wt(u) + \sum_{i=1}^{m} C(T(v_i), u, v_i),$$

and $$C(T(u), u, w, u) = 2wt(u) + \sum_{i=1}^{m} \min\{C(T(v_i), v_i, u), C(T(v_i), u, v_i)\}.$$

Then,

$$C(T(u), w, u) = \min\{C(T(u), w, u, \bar{u}), C(T(u), w, u, u)\},$$
$$C(T(u), u, w) = \min\{C(T(u), u, w, \bar{u}), C(T(u), u, w, u)\}.$$

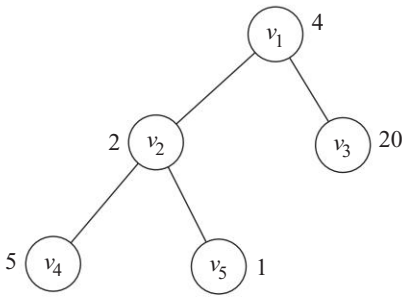
Finally, we have a rule concerning the boundary conditions.

RULE 4: If u is a leaf node and w is its parent node, then $C(T(u), w, u) = wt(w)$ and $C(T(u), u, w) = wt(u)$.

Our dynamic programming approach solves the weighted single step graph edge searching problem by working from the leaf nodes and gradually towards the root of the tree. If a node is a leaf, then use Rule 4. If it is an internal node, but not the root, then use Rule 3. If it is the root node, then use Rule 2 first and Rule 1 finally.

We illustrate our approach through an example. Consider the weighted tree in Figure 7–40.

FIGURE 7–40 A tree illustrating the dynamic programming approach.



Our dynamic programming approach may work as follows:

Step 1. We choose v_3, which is a leaf node. Apply Rule 4. The parent node of v_3 is v_1.

$C(T(v_3), v_1, v_3) = wt(v_1) = 4$

$C(T(v_3), v_3, v_1) = wt(v_3) = 20.$

Step 2. We choose v_4, which is a leaf node. Apply Rule 4. The parent node of v_4 is v_2.

$C(T(v_4), v_2, v_4) = wt(v_2) = 2$

$C(T(v_4), v_4, v_2) = wt(v_4) = 5.$

Step 3. We choose v_5, which is a leaf node. Apply Rule 4. The parent node of v_5 is v_2.

$C(T(v_5), v_2, v_5) = wt(v_2) = 2$

$C(T(v_5), v_5, v_2) = wt(v_5) = 1.$

Step 4. We choose v_2, which is an internal node, but not the root. Apply Rule 3. The parent node of v_2 is v_1. The descendant nodes of v_2 are v_4 and v_5.

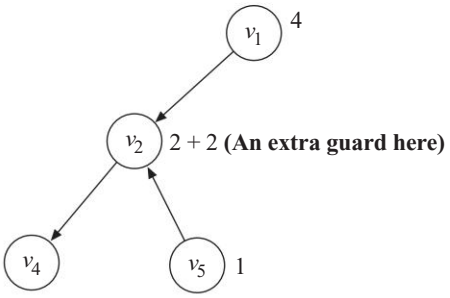
$$C(T(v_2), v_1, v_2) = \min\{C(T(v_2), v_1, v_2, \overline{v_2}), C(T(v_2), v_1, v_2, v_2)\}$$

$$\begin{aligned} C(T(v_2), v_1, v_2, \overline{v_2}) &= wt(v_1) + C(T(v_4), v_4, v_2) + C(T(v_5), v_5, v_2) \\ &= 4 + 5 + 1 \\ &= 10 \end{aligned}$$

$$
\begin{aligned}
&C(T(v_2),\ v_1,\ v_2,\ \overline{v_2})\\
&= wt(v_1) + wt(v_2) + \min\{C(T(v_4),\ v_2,\ v_4),\ C(T(v_4),\ v_4,\ v_2)\}\\
&\quad + \min\{C(T(v_5),\ v_2,\ v_5),\ C(T(v_5),\ v_5,\ v_2)\}\\
&= 4 + 2 + \min\{2, 5\} + \min\{2, 1\}\\
&= 6 + 2 + 1\\
&= 9\\
&C(T(v_2),\ v_1,\ v_2) = \min\{10, 9\} = 9.
\end{aligned}
$$

Note that the above calculation indicates that if the searching direction of (v_1, v_2) is from v_1 to v_2, then the optimal searching plan is that shown in Figure 7–41. An extra guard is stationed at v_2.

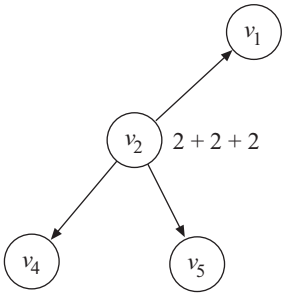
FIGURE 7–41 A single step searching plan involving v_2.



$$
\begin{aligned}
&C(T(v_2),\ v_2,\ v_1) = \min\{C(T(v_2),\ v_2,\ v_1,\ \overline{v_2}),\ C(T(v_2),\ v_2,\ v_1,\ v_2)\}\\
&C(T(v_2),\ v_2,\ v_1,\ \overline{v_2}) = wt(v_2) + C(T(v_4),\ v_2,\ v_4) + C(T(v_5),\ v_2,\ v_5)\\
&\qquad\qquad = 2 + 2 + 2\\
&\qquad\qquad = 6\\
&C(T(v_2),\ v_2,\ v_1,\ v_2)\\
&= wt(v_2) + wt(v_2) + \min\{C(T(v_4),\ v_2,\ v_4),\ C(T(v_4),\ v_4,\ v_2)\}\\
&\quad + \min\{C(T(v_5),\ v_2,\ v_5),\ C(T(v_5),\ v_5,\ v_2)\}\\
&= 2 + 2 + \min\{2, 5\} + \min\{2, 1\}\\
&= 4 + 2 + 1\\
&= 7\\
&C(T(v_2),\ v_2,\ v_1) = \min\{6, 7\} = 6.
\end{aligned}
$$

The above calculation indicates that if the searching plan of (v_1, v_2) is from v_2 to v_1, then the optimum single step searching plan is that shown in Figure 7–42.

FIGURE 7–42 Another single step searching plan involving v_2.

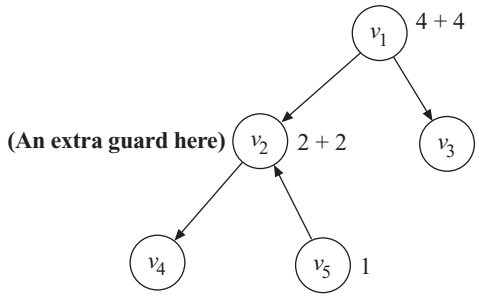


Step 5. We choose v_1. Since v_1 is the root, we apply Rule 2.

$$\begin{aligned} C(T(v_1), \overline{v_1}) &= \min\{C(T(v_2), v_1, v_2) + C(T(v_3), v_1, v_3), \\ &\qquad C(T(v_2), v_2, v_1) + C(T(v_3), v_3, v_1)\} \\ &= \min\{9 + 4, 6 + 20\} \\ &= \min\{13, 26\} \\ &= 13. \end{aligned}$$

By tracing back, we understand that if no extra guard is stationed at v_1, then the optimum single step searching plan is that shown in Figure 7–43.

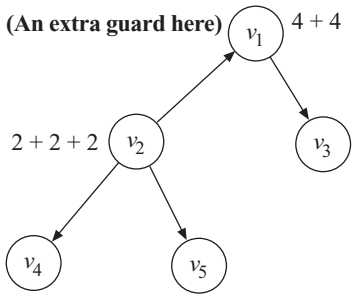
FIGURE 7–43 A single step searching plan involving v_1.



$$
\begin{aligned}
C(T(v_1), v_1) &= wt(v_1) + \min\{C(T(v_2), v_1, v_2), C(T(v_2), v_2, v_1)\} \\
&\quad + \min\{C(T(v_3), v_1, v_3), C(T(v_3), v_3, v_1)\} \\
&= 4 + \min\{9, 6\} + \min\{4, 20\} \\
&= 4 + 6 + 4 \\
&= 14.
\end{aligned}
$$

By tracing back, we understand that if an extra guard is stationed at v_1, then the optimum single step searching plan is that shown in Figure 7–44.

FIGURE 7–44 Another single step searching plan involving v_1.



Finally, we apply Rule 1.

$$
\begin{aligned}
C(T(v_1)) &= \min\{C(T(v_1), v_1), C(T(v_1), \overline{v_1})\} \\
&= \min\{14, 13\} \\
&= 13.
\end{aligned}
$$

This means that we should have no extra guard stationed at v_1 and the optimal searching plan is that shown in Figure 7–43.

The number of operations on each vertex is two: one for computing minimum searching cost and one for determining edge searching directions. Thus, the total number of operations is $O(n)$, where n is the number of nodes of the tree. Since each operation takes constant time, this is a linear algorithm.

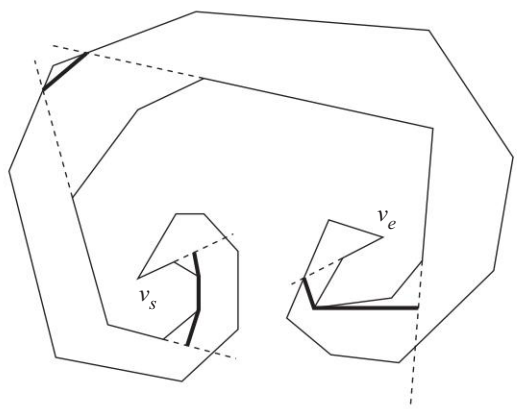
7–9 THE *m*-WATCHMEN ROUTES PROBLEM FOR 1-SPIRAL POLYGONS SOLVED BY THE DYNAMIC PROGRAMMING APPROACH

The m-watchmen routes problem is defined as follows. We are given a simple polygon and an integer $m \geq 1$, and we are required to find the routes for m

watchmen, called the *m*-watchmen routes, such that every point of the polygon is seen by at least one watchman from some position on his route. The objective is to minimize the sum of the lengths of the routes. The *m*-watchmen routes problem was proved to be NP-hard.

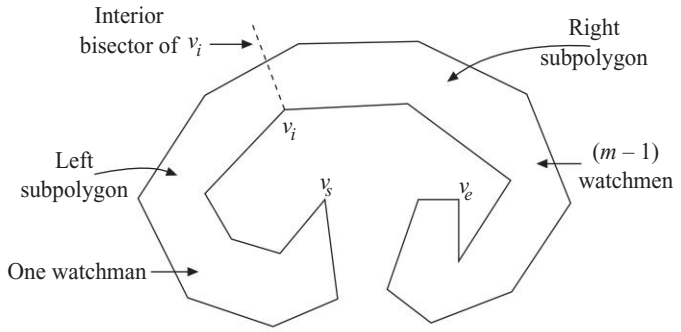
In this section, we shall show that the *m*-watchmen routes problem for 1-spiral polygons can be solved by the dynamic programming approach. Recall that in Chapter 3 we defined the 1-spiral polygons as a simple polygon whose boundary can be partitioned into a reflex chain and a convex chain. By traversing the boundary of a 1-spiral polygon, we call the starting and the ending vertices of the reflex chain v_s and v_e, respectively. Figure 7–45 shows a solution of the 3-watchmen routes problem for the 1-spiral polygon.

FIGURE 7–45 A solution of the 3-watchmen routes problem for a 1-spiral polygon.



The basic idea is that we can divide a 1-spiral polygon into two parts by an interior bisector from a vertex of the reflex chain. Consider Figure 7–46. An interior bisector passing through v_i intersects the convex chain and partitions the 1-spiral polygon into two parts: either both 1-spiral's or one 1-spiral and one convex. We call the subpolygon including v_s the left subpolygon and the other one the right subpolygon with respect to v_i. Vertex v_i is called a cut point. For the convenience of later discussion, we define the interior bisector of the first vertex v_s (the last vertex v_e) in the reflex chain as the first (last) edge of the convex chain.

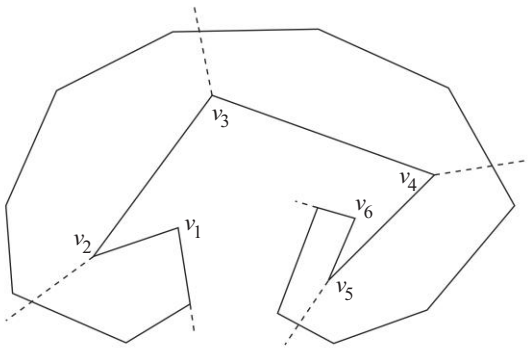
FIGURE 7–46 The basic idea for solving the *m*-watchmen routes problem.



We can distribute the m watchmen into the two subpolygons as follows: one watchman for the left subpolygon and $(m - 1)$ watchmen for the right subpolygon. Let us assume that we know how to solve the 1-watchman route problem. Then the $(m - 1)$ watchmen routes problem can be solved recursively. That is, we again partition the right subpolygon into two parts and distribute one watchman for the left and $(m - 2)$ watchmen for the right. The m-watchmen routes problem can be solved by exhaustively trying all possible cut points and choosing the one with the least sum of the lengths of the routes. In the following, we shall show why this approach is a dynamic programming approach.

Consider Figure 7–47 and assume that we are going to solve a 3-watchmen routes problem. Our dynamic programming approach solves this 3-watchmen routes problem as follows. Let $SP(v_i, v_j)$ denote the subpolygon bounded between interior bisectors of v_i and v_j.

FIGURE 7–47 A 1-spiral polygon with six vertices in the reflex chain.



(1) Find the following 1-watchman route solutions.
Solution 1: The 1-watchman route solution for $SP(v_1, v_2)$.
Solution 2: The 1-watchman route solution for $SP(v_1, v_3)$.
Solution 3: The 1-watchman route solution for $SP(v_1, v_4)$.
Solution 4: The 1-watchman route solution for $SP(v_2, v_3)$.
Solution 5: The 1-watchman route solution for $SP(v_2, v_4)$.
Solution 6: The 1-watchman route solution for $SP(v_2, v_5)$.
Solution 7: The 1-watchman route solution for $SP(v_3, v_4)$.
Solution 8: The 1-watchman route solution for $SP(v_3, v_5)$.
Solution 9: The 1-watchman route solution for $SP(v_3, v_6)$.
Solution 10: The 1-watchman route solution for $SP(v_4, v_5)$.
Solution 11: The 1-watchman route solution for $SP(v_4, v_6)$.
Solution 12: The 1-watchman route solution for $SP(v_5, v_6)$.
(2) Find the following 2-watchmen routes solutions.
Solution 13: The 2-watchmen routes solution for $SP(v_2, v_6)$.
We first obtain the following solutions:
Solution 13–1: Combining Solution 4 and Solution 9.
Solution 13–2: Combining Solution 5 and Solution 11.
Solution 13–3: Combining Solution 6 and Solution 12.
We select the solution with the least length among Solution 13–1, Solution 13–2, and Solution 13–3 as Solution 13.
Solution 14: The 2-watchmen routes solution for $SP(v_3, v_6)$.
We first obtain the following solutions:
Solution 14–1: Combining Solution 7 and Solution 11.
Solution 14–2: Combining Solution 8 and Solution 12.
We select the solution with the shorter length between Solution 14–1 and Solution 14–2 as Solution 14.
Solution 15: The 2-watchmen routes solution for $SP(v_4, v_6)$.
This can be obtained by combining Solution 10 and Solution 12.
(3) Find the 3-watchmen routes solution of the original problem by finding the following solutions:
Solution 16–1: Combining Solution 1 and Solution 13.
Solution 16–2: Combining Solution 2 and Solution 14.
Solution 16–3: Combining Solution 3 and Solution 15.

We select Solution 16–1, Solution 16–2, or Solution 16–3, whichever has the shortest length.

The discussion shows that the m-watchmen routes problem can be solved by the dynamic programming approach. Note that the spirit of the dynamic programming approach is that we start by solving basic problems and then gradually solve increasingly complicated problems by combining those subproblems already solved. We first solve all relevant 1-watchman route problems, then 2-watchmen routes problems and so on.

Let $OWR_k(v_i, v_j)$ denote the length of the optimal k-watchmen routes for the subpolygon $SP(v_i, v_j)$. The optimal m-watchmen route $OWR_m(v_s, v_e)$ can be obtained by the following formulas:

$$OWR_m(v_s, v_e) = \min_{s+1 \le i \le e-1} \{OWR_1(v_s, v_i) + OWR_{m-1}(v_i, v_e)\},$$

$$OWR_k(v_i, v_e) = \min_{i+1 \le j \le e-1} \{OWR_1(v_i, v_j) + OWR_{k-1}(v_j, v_e)\},$$

for $2 \le k \le m - 1, s + 1 \le i \le e - 1.$

As for the 1-watchman route problem, we shall not elaborate here because our main purpose is only to show that this problem can be solved by the dynamic programming approach. Figure 7–48 shows a typical solution of a 1-watchman route problem and Figure 7–49 shows special cases.

FIGURE 7–48 A typical 1-watchman route p, v_a, $C[v_a, v_b]$, v_b, r_1 in a 1-spiral polygon.

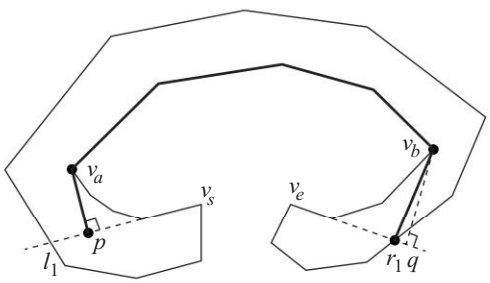
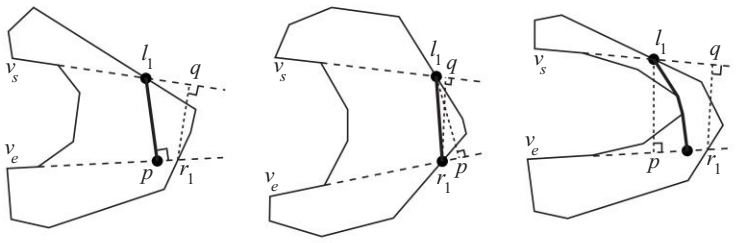


FIGURE 7–49 Special cases of the 1-watchman route problem in a 1-spiral polygon.



(a) Either $\overline{l_1 p}$ or $\overline{r_1 q}$ is inside the polygon.

(b) Both p or q are outside the polygon.

(c) One of $\overline{l_1 p}$ or $\overline{r_1 q}$ intersects the reflex chain.

7–10 THE EXPERIMENTAL RESULTS

To demonstrate the power of dynamic programming, we implemented the dynamic programming approach to solve the longest common subsequence problem on a computer. We also programmed the straightforward method on the same computer. Table 7–7 describes the experimental results. The results clearly indicate the superiority of the dynamic programming approach.

TABLE 7–7 Experimental results.

Length of string	CPU time in milliseconds	
	Dynamic programming	Exhaustive method
4	< 1	20
6	< 2	172
8	< 2	2,204
10	< 10	32,952
12	< 14	493,456

7-11 NOTES AND REFERENCES

The term dynamic programming may be said to be invented by Bellman (1962). Many authors have written books on this topic: Nemhauser (1966); Dreyfus and Law (1977); and Denardo (1982).

In 1962, Bellman gave a review of the application of dynamic programming to solve combinatorial problems (1962). The formulation of the dynamic programming approach for the resource allocation and scheduling problem appeared in Lawler and Moore (1969); Sahni (1976); and Horowitz and Sahni (1978). The dynamic programming formulation for the traveling salesperson problem is due to Held and Karp (1962) and Bellman (1962). The application of dynamic programming to the longest common subsequence problem was proposed by Hirschberg (1975). The dynamic programming approach for solving the 0/1 knapsack problem may be found in Nemhauser and Ullman (1969) and Horowitz and Sahni (1974). The construction of optimal binary search trees using dynamic programming approach can be found in Gilbert and Moore (1959), Knuth (1971) and Knuth (1973). The application of dynamic programming approach to the weighted perfect domination set problem on trees can be found in Yen and Lee (1990) and that to solve the single step searching problem can be found in Hsiao, Tang and Chang (1993). The 2-sequence alignment algorithm can be found in Neddleman and Wunsch (1970), while that for RNA secondary structure prediction can be found in Waterman and Smith (1978).

Other important applications of dynamic programming, which we did not mention in this book, include chained matrix multiplication: Godbole (1973); Hu and Shing (1982); and Hu and Shing (1984); all shortest paths: Floyd (1962); syntactic analysis of context-free languages: Younger (1967), and/or series-parallel graphs: Simon and Lee (1971) and Viterbi decoding: Viterbi (1967) and Omura (1969).

Dynamic programming can be used together with branch-and-bound. See Morin and Marsten (1976). It can also be used to solve a special kind of partition problem. See Section 4.2 of Garey and Johnson (1979). This concept was used by Hsu and Nemhauser (1979).

7-12 FURTHER READING MATERIALS

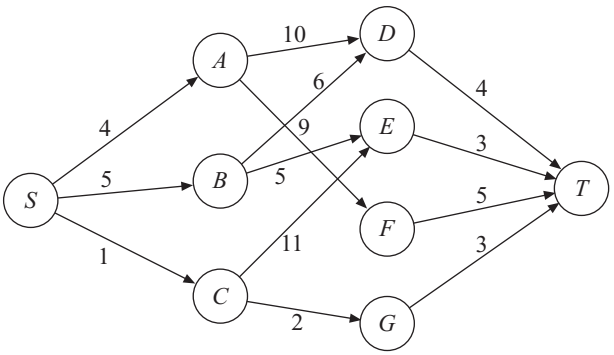
Since dynamic programming is so elegant, it has been a research topic for both theoreticians and practitioners in computer science for a long time. For further research, we recommend the following papers: Akiyoshi and Takeaki (1997);

Auletta, Parente and Persiano (1996); Baker (1994); Bodlaender (1993); Chen, Kuo and Sheu (1988); Chung, Makedon, Sudborough and Turner (1985); Even, Itai and Shamir (1976); Farber and Keil (1985); Fonlupt and Nachef (1993); Gotlieb (1981); Gotlieb and Wood (1981); Hirschberg and Larmore (1987); Horowitz and Sahni (1974); Huo and Chang (1994); Johnson and Burrus (1983); Kantabutra (1994); Kao and Queyranne (1982); Kilpelainen and Mannila (1995); Kryazhimskiy and Savinov (1995); Liang (1994); Meijer and Rappaport (1992); Morin and Marsten (1976); Ozden (1988); Park (1991); Peng, Stephens and Yesha (1993); Perl (1984); Pevzner (1992); Rosenthal (1982); Sekhon (1982); Tidball and Atman (1996); Tsai and Hsu (1993); Tsai and Lee (1997); Yannakakis (1985); Yen and Lee (1990); Yen and Lee (1994) and Yen and Tang (1995).

The following is a list of interesting and new results: Aho, Ganapathi and Tjang (1989); Akutsu (1996); Alpert and Kahng (1995); Amini, Weymouth and Jain (1990); Baker and Giancarlo (2002); Bandelloni, Tucci and Rinaldi (1994); Barbu (1991); Brown and Whitney (1994); Charalambous (1997); Chen, Chern and Jang (1990); Cormen (1999); Culberson and Rudnicki (1989); Delcoigne and Hansen (1975); Eppstein, Galil, Giancarlo and Italiano (1990); Eppstein, Galil, Giancarlo and Italiano (1992(a)); Eppstein, Galil, Giancarlo and Italiano (1992(b)); Erdmann (1993); Farach and Thorup (1997); Fischel-Ghodsian, Mathiowitz and Smith (1990); Geiger, Gupta, Costa and Vlontzos (1995); Gelfand and Roytberg (1993); Galil and Park (1992); Hanson (1991); Haussmann and Suo (1995); Hein (1989); Hell, Shamir and Sharan (2001); Hirosawa, Hoshida, Ishikawa and Toya (1993); Holmes and Durbin (1998); Huang, Liu and Viswanathan (1994); Huang and Waterman (1992); Ibaraki and Nakamura (1994); Karoui and Quenez (1995); Klein (1995); Kostreva and Wiecek (1993); Lewandowski, Condon and Bach (1996); Liao and Shoemaker (1991); Lin, Fan and Lee (1993); Lin, Chen, Jiang and Wen (2002); Littman, Cassandra and Kaelbling (1996); Martin and Talley (1995); Merlet and Zerubia (1996); Miller and Teng (1999); Mohamed and Gader (1996); Moor (1994); Motta and Rampazzo (1996); Myoupo (1992); Ney (1984); Ney (1991); Nuyts, Suetens, Oosterlinck, Roo and Mortelmans (1991); Ohta and Kanade (1985); Ouyang and Shahidehpour (1992); Pearson and Miller (1992); Rivas and Eddy (1999), Sakoe and Chiba (1978); Schmidt (1998); Snyder and Stormo (1993); Sutton (1990); Tataru (1992); Tatman and Shachter (1990); Tatsuya (2000); Vintsyuk (1968); Von Haeselerm, Blum, Simpson, Strum and Waterman (1992); Waterman and Smith (1986); Wu (1996); Xu (1990) and Zuker (1989).

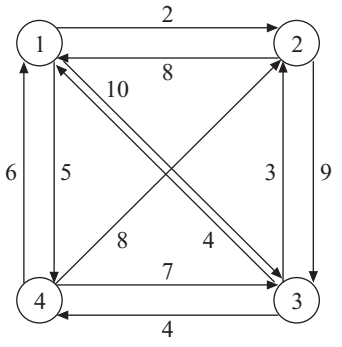
Exercises

7.1 Consider the following graph. Find the shortest route from S to T by the dynamic programming approach.



7.2 For the graph shown in Figure 7–1, solve the same problem by using the branch-and-bound approach. For this problem, which approach (dynamic programming versus branch-and-bound) is better? Why?

7.3 For the graph shown as follows, solve the traveling salesperson problem by the branch-and-bound approach. Compare it with the dynamic programming approach.



	1	2	3	4
1	∞	2	10	5
2	8	∞	9	∞
3	4	3	∞	4
4	6	8	7	∞

7.4 For the following table, find an optimal allocation of resources to maximize the total profit for those three projects and four resources.

project \ resource	1	2	3	4
1	3	7	10	12
2	1	2	6	9
3	2	4	8	9

7.5 Solve the following linear programming problem by dynamic programming.

Maximize $x_0 = 8x_1 + 7x_2$

subject to

$2x_1 + x_2 \leq 8$

$5x_1 + 2x_2 \leq 15$

where x_1 and x_2 are non-negative integers.

7.6 Find a longest common subsequence of

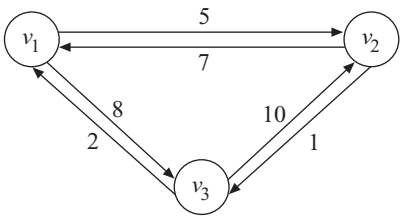
$S_1 = a\ a\ b\ c\ d\ a\ e\ f$

and $S_2 = b\ e\ a\ d\ f.$

7.7 In general, the partition problem is NP-complete. However, under some constraints, a special kind of the partition problem is a polynomial problem because it can be solved by dynamic programming. Read Section 4–2 of Garey and Johnson (1979).

7.8 Find an optimal binary tree for $a_1, a_2, \ldots, a_6$, if the identifiers, in order, have probabilities 0.2, 0.1, 0.15, 0.2, 0.3, 0.05 respectively and all other identifiers have zero probability.

7.9 Consider the following graph: Solve the all-pairs shortest paths problem of the graph. The all-pairs shortest paths problem is to determine the shortest path between every pair of vertices. Consult Section 5–3 of Horowitz and Sahni (1978), or Section 5–4 of Brassard and Bratley (1988).



7.10 Let f be a real function of x and $y = (y_1, y_2, \ldots, y_k)$. We say that f is decomposable into f_1 and f_2 if f is separable $(f(x, y) = f_1(x), f_2(y))$ and if, moreover, the function is monotone non-decreasing relative to its second argument. Prove that if f is decomposable with $f(x, y) = (f_1(x), f_2(y))$, then

$$\underset{(x,y)}{Opt}\{f(x,y)\} = Opt\{f_1(x, Opt\{f_2(y)\})\} \quad (Opt = \text{min or max})$$

(Consult Section 9–2 of [Minoux 1986].)

7.11 Floyd's algorithm, which can be easily found in many textbooks, is to find all-pairs shortest paths in a weighted graph. Give an example to explain the algorithm.

7.12 Write a dynamic programming algorithm to solve longest increasing subsequence problem.

7.13 Given two sequences S_1 and S_2 on a alphabet set Σ, and a scoring function f: $\Sigma \times \Sigma \rightarrow \Re$, the local alignment problem is to find a subsequence S_1' from S_1 and a subsequence S_2' from S_2 such that the score obtained by aligning S_1' and S_2' is the highest, among all possible subsequences of S_1 and S_2. Use the dynamic programming strategy to design an algorithm of $O(nm)$ time or better for this problem, where n and m denote the lengths of S_1 and S_2 respectively.

chapter

8

The Theory of NP-Completeness

The theory of NP-completeness is perhaps one of the most interesting topics in computer science. The major investigator of this field, Professor S. A. Cook of Toronto University received a Turing Award for his contribution in this field of research. There is no doubt at all, among many interesting research results in computer science, the theory of NP-completeness is one of the most exciting, and also puzzling, theories. The main theorem, now called Cook's theorem, is perhaps the most widely cited theorem.

In this book, we shall not only introduce the application of Cook's theorem, but also try to explain the real meaning of Cook's theorem.

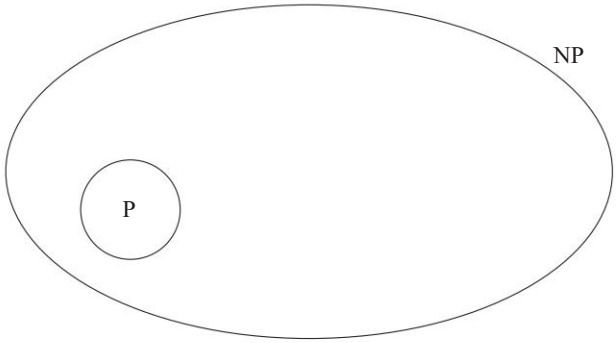
8–1 An Informal Discussion of the Theory of NP-Completeness

The theory of NP-completeness is important because it identifies a large class of difficult problems. Here, by a difficult problem, we mean a problem whose lower bound seems to be in the order of an exponential function. In other words, the theory of NP-completeness has identified a large class of problems which do not seem to have any polynomial time algorithms to solve them.

Roughly speaking, we may say that the theory of NP-completeness first points out that many problems are called NP (non-deterministic polynomial) problems. (The formal definition of NP will be given in Section 8–4.) Not all NP problems are difficult; many are easy. For example, the searching problem is an NP problem. It can be solved by algorithms with polynomial time complexities. Another example is the minimum spanning tree problem. Again, this problem can be solved by a polynomial algorithm. We shall call all of these problems P (polynomial) problems.

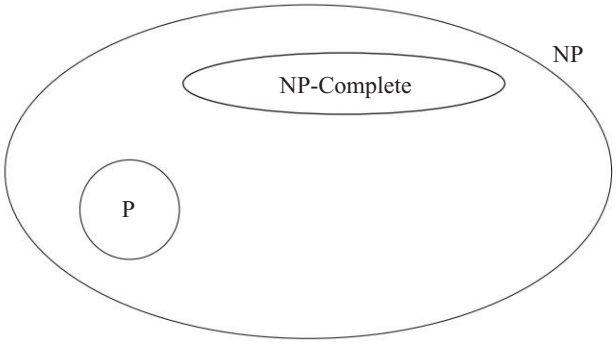
Since the set of NP problems contains many P problems, we may draw a figure depicting their relation, shown in Figure 8–1.

FIGURE 8–1 The set of NP problems.



Moreover, it has been shown that there is also a large class of problems, which are NP-complete problems, contained in the set of NP problems, shown in Figure 8–2.

FIGURE 8–2 NP problems including both P and NP-complete problems.



The set of known NP-complete problems is very large and keeps growing all the time. They include many famous problems, such as the satisfiability problem, the traveling salesperson problem and the bin packing problem. All of these problems have a common characteristic: *Up to now, none of the NP-complete problems can be solved by any polynomial algorithm, in worst cases. In other words, up to now, the best algorithm to solve any NP-complete problem has exponential time complexity in the worst case*. It is important to emphasize here that the theory of NP-completeness always talks about worst cases.

It is possible that an NP-complete problem can already be solved by an algorithm with polynomial average case time complexity. In the rest of this

chapter, unless specially explained, whenever we discuss time complexity, we mean worst-case time complexity.

Identifying a set of problems which, up to now, cannot be solved by any polynomial algorithm is not interesting enough. By the definition of NP-completeness, the following is true:

> *If any NP-complete problem can be solved in polynomial time, then all NP problems can be solved in polynomial time. Or, if any NP-complete problem can be solved in polynomial time, then NP = P.*

Thus, the theory of NP-completeness indicates that every NP-complete problem is like an important pillar. If it falls, the entire building would collapse. Or, to put it in another way, an NP-complete problem is like an important general; if he surrenders to the enemy, the entire army would also surrender.

Since it is very unlikely that all NP problems can be solved by polynomial algorithms, it is therefore quite unlikely that any NP-complete problem can be solved by any polynomial algorithm.

We would like to emphasize here that the theory of NP-completeness has not claimed that NP-complete problems can never be solved by polynomial algorithms. It merely says that it is quite unlikely that any NP-complete problem can be solved in polynomial number of steps. It somehow discourages us from trying to find polynomial algorithms to solve these NP-complete problems.

8–2 THE DECISION PROBLEMS

We may divide most of the problems considered in this book into two categories: optimization problems and decision problems.

Consider the traveling salesperson problem. The problem is defined as follows: Given a set of points, find a shortest tour which starts at any point v_0. This traveling salesperson problem is obviously an optimization problem.

A decision problem is a problem whose solution is simply "yes" or "no". For the traveling salesperson problem, there is a corresponding decision problem which is defined as follows: Given a set of points, is there a tour, starting from any point v_0, whose total length is less than a given constant c?

Note that the traveling salesperson problem is more difficult than the traveling salesperson decision problem. For if we can solve the traveling salesperson problem, we shall know that the shortest tour is equal to some value, say a. If $a < c$, then the answer to the traveling salesperson decision problem is "yes"; otherwise "no". Thus, we may say that *if we can solve the traveling salesperson problem, then we can solve the traveling salesperson decision*

problem but not vice versa. Therefore, we conclude that the traveling salesperson problem is more difficult than the traveling salesperson decision problem.

Let us now take a look at another example, the 0/1 knapsack problem which was introduced in Chapter 5, is defined as follows:

Given M, W_i's and P_i's, $W_i > 0$, $P_i > 0$, $1 \le i \le n$, $M > 0$, find x_i's such that $x_i = 1$ or 0, and $\sum_{i=1}^{n} P_i x_i$ is maximized subject to $\sum_{i=1}^{n} W_i x_i \le M$.

This 0/1 knapsack problem is clearly an optimization problem. It also has a corresponding decision problem which is defined as follows:

Given M, R, W_i's and P_i's, $M > 0$, $R > 0$, $W_i > 0$, $P_i > 0$, $1 \le i \le n$, determine whether there exist x_i's, $x_i = 1$ or 0, such that $\sum_{i=1}^{n} P_i x_i \ge R$ and $\sum_{i=1}^{n} W_i x_i \le M$.

It can be easily shown that the 0/1 knapsack problem is more difficult than the 0/1 knapsack decision problem.

In general, optimization problems are more difficult to solve than their corresponding decision problems. Therefore, so far as the discussion of whether a problem can be solved by a polynomial algorithm is concerned, we may simply consider decision problems only. If the traveling salesperson decision problem cannot be solved by polynomial algorithms, we can conclude that the traveling salesperson problem cannot be solved by polynomial algorithms either. *In discussing NP problems, we shall only discuss decision problems*.

In the following section, we shall discuss the satisfiability problem which is one of the most famous decision problems.

8–3 THE SATISFIABILITY PROBLEM

The satisfiability problem is important because this problem was the first NP-complete problem ever found.

Let us consider the following logical formula:

$$\begin{aligned} & x_1 \vee x_2 \vee x_3 \\ \& \quad & -x_1 \\ \& \quad & -x_2. \end{aligned}$$

The following assignment will make the formula true.

$x_1 \leftarrow F$
$x_2 \leftarrow F$
$x_3 \leftarrow T.$

In the following, we shall use the notation $(-x_1, -x_2, x_3)$ to represent $\{x_1 \leftarrow F, x_2 \leftarrow F, x_3 \leftarrow T\}$. *If an assignment makes a formula true, we shall say that this assignment satisfies the formula; otherwise, it falsifies the formula.*

If there is at least one assignment which satisfies a formula, then we say that this formula is satisfiable; otherwise, it is unsatisfiable.

A typical unsatisfiable formula is

$$\begin{array}{lr} & x_1 \\ \& & -x_1. \end{array}$$

Another unsatisfiable formula is

$$\begin{array}{lrcl} & x_1 & \vee & x_2 \\ \& & x_1 & \vee & -x_2 \\ \& & -x_1 & \vee & x_2 \\ \& & -x_1 & \vee & -x_2. \end{array}$$

The satisfiability problem is defined as follows: Given a Boolean formula, determine whether this formula is satisfiable or not.

In the rest of this section, we shall discuss some methods to solve the satisfiability problem. We need some definitions first.

Definition

A literal is x_i or $-x_i$ where x_i is a Boolean variable.

Definition

A clause is a disjunction of literals. It is understood that no clause contains a literal and its negation simultaneously.

Definition

A formula is in its conjunctive normal form if it is in the form of $c_1 \,\&\, c_2 \,\&\, \ldots \,\&\, c_m$ where each c_i, $1 \le i \le m$, is a clause.

It is well known that every Boolean formula can be transformed into the conjunctive normal form. Therefore, we shall assume that all our formulas are already in conjunctive normal forms.

Definition

A formula G is a logical consequence of a formula F if and only if whenever F is true, G is true. In other words, every assignment satisfying F also satisfies G.

For example,

$$\begin{aligned} & -x_1 \vee x_2 && \textbf{(1)} \\ \& \quad & x_1 && \textbf{(2)} \\ \& \quad & x_3 && \textbf{(3)} \end{aligned}$$

is a formula in conjunctive normal form. The only assignment which satisfies the above formula is (x_1, x_2, x_3). The reader can easily see that the formula x_2 and x_3 is a logical consequence of the above formula. Given two clauses

$$c_1\colon\ L_1 \vee L_2 \ldots \vee L_j$$

and $\quad c_2\colon\ -L_1 \vee L_2' \ldots \vee L_k'$,

we can deduce a clause

$$L_2 \vee \ldots \vee L_j \vee L_2' \vee \ldots \vee L_k'$$

as a logical consequence of $c_1 \,\&\, c_2$, if the clause

$$L_2 \vee \ldots \vee L_j \vee L_2' \vee \ldots \vee L_k'$$

does not contain any pair of literals which are complementary to each other.

For example, consider the following clauses:

c_1: $-x_1 \vee x_2$
c_2: $x_1 \vee x_3$.

Then

c_3: $x_2 \vee x_3$

is a logical consequence of c_1 & c_2.

The above inference rule is called the *resolution principle*, and the clause c_3 generated by applying this resolution principle to c_1 and c_2 is called a *resolvent* of c_1 and c_2.

Let us consider another example:

c_1: $-x_1 \vee -x_2 \vee x_3$
c_2: $x_1 \vee x_4$.

Then

c_3: $-x_2 \vee x_3 \vee x_4$

is a resolvent of c_1 and c_2. It is, of course, also a logical consequence of c_1 & c_2.

Consider the following two clauses:

c_1: x_1
c_2: $-x_1$.

Then the resolvent is a special clause because it contains no literal and is denoted as

$c_3 = \square$

which is an empty clause.

If an empty clause can be deduced from a set of clauses, then this set of clauses must be unsatisfiable. Consider the following set of clauses:

$$x_1 \vee x_2 \quad (1)$$
$$x_1 \vee -x_2 \quad (2)$$
$$-x_1 \vee x_2 \quad (3)$$
$$-x_1 \vee -x_2. \quad (4)$$

We can deduce an empty clause as follows:

(1) & (2)	x_1	**(5)**
(3) & (4)	$-x_1$	**(6)**
(5) & (6)	$\square$.	**(7)**

Since (7) is an empty clause, we can conclude that (1) & (2) & (3) & (4) is unsatisfiable.

Given a set of clauses, we may repeatedly apply the resolution principle to deduce new clauses. These new clauses are added to the original set of clauses and the resolution principle is applied to them again. This process is terminated if an empty clause is generated or no new clauses can further be deduced. *If an empty clause is deduced, this set of clauses is unsatisfiable. If no new clauses can be deduced when the process is terminated, this set of clauses is satisfiable.*

Let us consider a satisfiable set of clauses

	$-x_1 \vee -x_2 \vee x_3$	**(1)**
	x_1	**(2)**
	x_2	**(3)**
(1) & (2)	$-x_2 \vee x_3$	**(4)**
(4) & (3)	x_3	**(5)**
(1) & (3)	$-x_1 \vee x_3$.	**(6)**

We notice now that no new clause can be deduced from clauses (1) to (6) any further. Therefore, we can conclude that this set of clauses is satisfiable.

If we modify the above set of clauses by adding $-x_3$ into it, we have an unsatisfiable set of clauses:

	$-x_1 \vee -x_2 \vee x_3$	**(1)**
	x_1	**(2)**
	x_2	**(3)**
	$-x_3$	**(4)**
(1) & (2)	$-x_2 \vee x_3$	**(5)**
(5) & (4)	$-x_2$	**(6)**
(6) & (3)	$\square$.	**(7)**

The property of being unsatisfiable can now be established because an empty clause is deduced.

For more theoretical discussions of the resolution principle, consult any book on mechanical theorem proving.

In the above discussion, we showed that the satisfiability problem can be viewed as a deduction problem. In other words, we are constantly looking for inconsistency. We conclude that the set of clauses is unsatisfiable if inconsistency can be derived, and satisfiable if otherwise. Our approach seems to have nothing to do with finding assignments satisfying the formula. Actually, we shall immediately show that the deduction (or inference) approach is equivalent to the assignment finding approach. That is, the deduction of an empty clause is actually equivalent to the failure of finding any assignment satisfying all clauses. Conversely, the failure to deduce an empty clause is equivalent to finding at least one assignment satisfying all clauses.

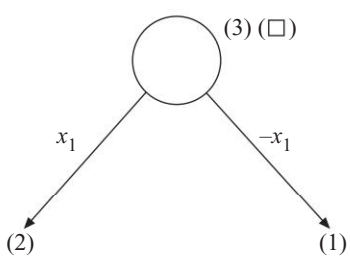
Let us start with a simplest example:

$$x_1 \quad \textbf{(1)}$$

$$-x_1. \quad \textbf{(2)}$$

Since the above set of clauses contains only one variable, namely x_1, we may start to construct a semantic tree, shown in Figure 8–3.

FIGURE 8–3 A semantic tree.



The left-side branch means that the assignment will contain x_1 (meaning $x_1 \leftarrow T$) and the right-side branch means that the assignment will contain $-x_1$ (meaning $x_1 \leftarrow F$). It can be seen that the left-side assignment, namely x_1, falsifies clause (2). We therefore terminate this branch with (2). Similarly, we terminate the right-side branch with (1). Figure 8–3 indicates that clause (1) must contain x_1 and clause (2) must contain $-x_1$. We apply the resolution principle to these clauses and deduce a new clause, which is empty, as follows:

(1) & (2) $\square$. **(3)**

We may put (3) beside the parent node, shown in Figure 8–3. Consider the following set of clauses:

$-x_1 \vee -x_2 \vee x_3$ **(1)**

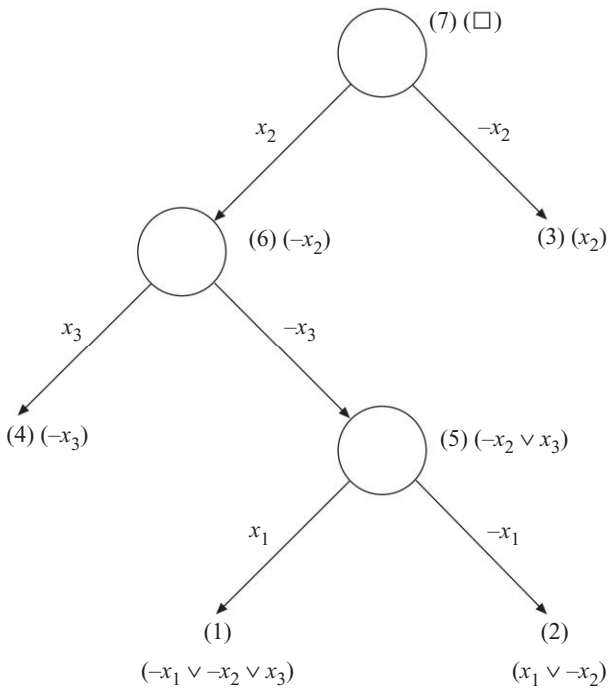
$x_1 \vee -x_2$ **(2)**

x_2 **(3)**

$-x_3$. **(4)**

We can construct a semantic tree shown in Figure 8–4. In Figure 8–4, each path from the root of the tree to a terminal node represents a class of assignments. For instance, each assignment must contain x_2 or $-x_2$. The first right-side branch is labeled $-x_2$. This represents all assignments containing $-x_2$ (there are four such assignments). Since clause (3) contains x_2 only, it is falsified by every assignment containing $-x_2$. Thus, the first right-side branch is terminated by (3).

FIGURE 8–4 A semantic tree.



Consider the path containing x_2, $-x_3$ and x_1. This assignment falsifies clause (1). Similarly, the path containing x_2, $-x_3$ and $-x_1$ represents an assignment which falsifies clause (2).

Consider the terminal nodes marked with (1) and (2) respectively. Since the branches leading to them are labeled with x_1 and $-x_1$ respectively, clause (1) must contain $-x_1$ and clause (2) must contain x_1. Applying resolution principle to clauses (1) and (2), we can deduce a clause $-x_2 \vee x_3$. This clause may be labeled as clause (5) and attached to the parent node, shown in Figure 8–4. Using the same reasoning, we may apply the resolution principle to clauses (5) and (4) and obtain clause (6). Clauses (6) and (3) are contradictory to each other and an empty clause is deduced. All these are shown in Figure 8–4. The entire deduction process is detailed as follows:

(1) & (2) $-x_2 \vee x_3$ **(5)**

(4) & (5) $-x_2$ **(6)**

(6) & (3) $\square$. **(7)**

In general, given a set of clauses representing a Boolean formula, we can construct a semantic tree according to the following rules:

(1) From each internal node of the semantic tree, there are two branches branching out of it. One of them is labeled with x_i and the other one is labeled with $-x_i$ where x_i is a variable occurring in the set of clauses.
(2) A node is terminated as soon as the assignment corresponding to the literals occurring in the path from the root of the tree to this node falsifies a clause (j) in the set. Mark this node as a terminal node and attach clause (j) to this terminal node.
(3) No path in the semantic tree may contain complementary pair so that each assignment is consistent.

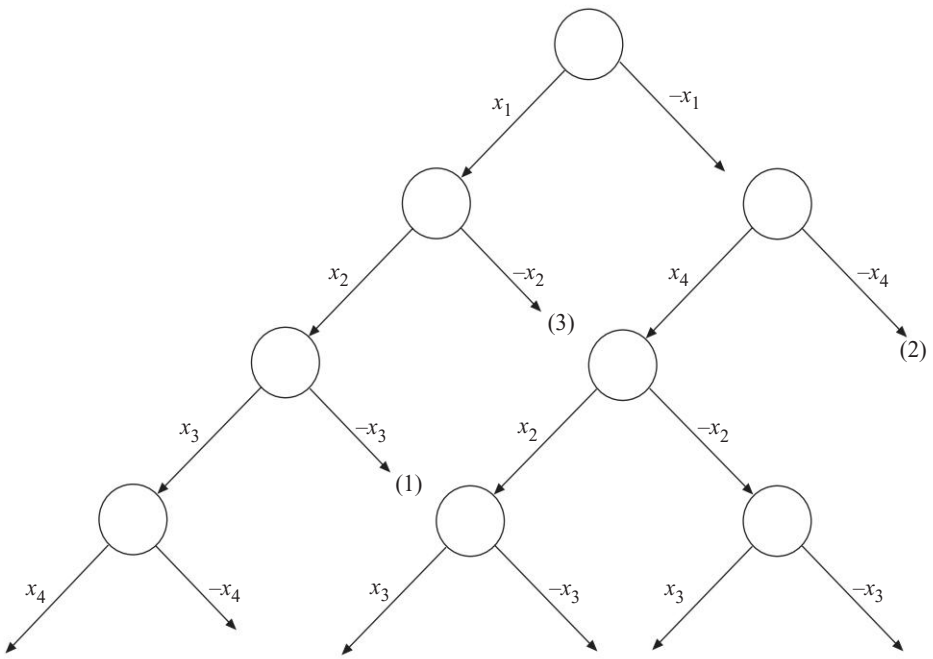
It is obvious that each semantic tree is finite. *If each terminal is attached with a clause, then no assignment satisfying all clauses exists. This means that this set of clauses is unsatisfiable. Otherwise, there exists at least one assignment satisfying all clauses and this set of clauses is satisfiable.*

Consider the following set of clauses:

$$-x_1 \vee -x_2 \vee x_3 \quad (1)$$
$$x_1 \vee x_4 \quad (2)$$
$$x_2 \vee -x_1. \quad (3)$$

We may construct a semantic tree as shown in Figure 8–5.

FIGURE 8–5 A semantic tree.



From the above semantic tree, we conclude that this set of clauses is satisfiable. The following assignments all satisfy the formula:

(x_1, x_2, x_3, x_4),
$(x_1, x_2, x_3, -x_4)$,
$(-x_1, -x_2, x_3, x_4)$,
$(-x_1, -x_2, -x_3, x_4)$,
$(-x_1, x_2, x_3, x_4)$,
and $(-x_1, x_2, -x_3, x_4)$.

If a set of clauses is unsatisfiable, then every semantic tree corresponds to the deduction of an empty clause using the resolution principle. This deduction is extracted from the semantic tree as follows:

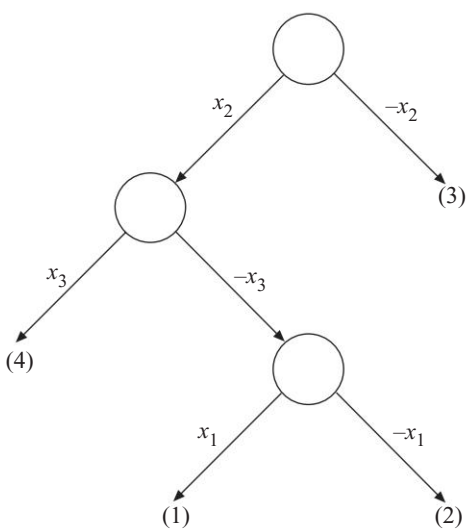
(1) Consider an internal node whose descendants are terminal nodes. Let the clauses attached to these two terminal nodes be c_i and c_j respectively. Apply the resolution principle to these two clauses and attach the resolvent to this parent node. Delete the descendant nodes altogether. The original internal node now becomes a terminal node.
(2) Repeat the above step until the tree becomes empty and an empty clause is deduced.

Let us explain the above idea with another example. Consider the following set of clauses:

$$-x_1 \vee -x_2 \vee x_3 \quad \textbf{(1)}$$
$$x_1 \vee x_3 \quad \textbf{(2)}$$
$$x_2 \quad \textbf{(3)}$$
$$-x_3. \quad \textbf{(4)}$$

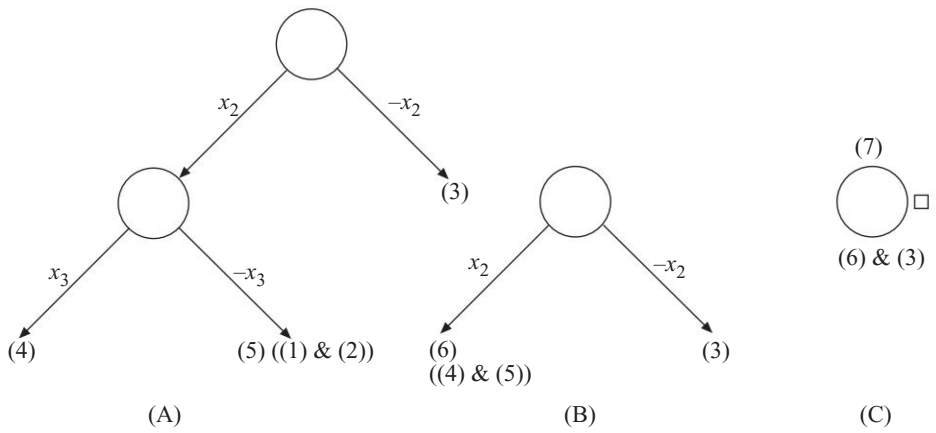
A semantic tree is now constructed, shown in Figure 8–6.

FIGURE 8–6 A semantic tree.



The semantic tree in Figure 8–6 can be gradually collapsed, shown in Figure 8–7.

FIGURE 8–7 The collapsing of the semantic tree in Figure 8–6.



The deduction is as follows:

(1) & (2) $-x_2 \vee x_3$ **(5)**

(5) & (4) $-x_2$ **(6)**

(6) & (3) $\square$. **(7)**

What we have shown is that even as we use the deduction approach, we are actually finding assignments satisfying all clauses. If there are n variables, then there are 2^n possible assignments. Up to now, for the best available algorithm, in worst cases, we must examine an exponential number of possible assignments before we can make any conclusion.

Is there any possibility that the satisfiability problem can be solved in polynomial time? The theory of NP-completeness does not rule out this possibility. However, it does make the following claim. *If the satisfiability problem can be solved in polynomial number of steps, then all NP problems can be solved in polynomial number of steps*.

We have not defined the NP problem yet. They will be discussed in the next section.

8–4 THE NP PROBLEMS

The notation NP denotes non-deterministic polynomial. Let us first define a non-deterministic algorithm as follows: *A non-deterministic algorithm is an algorithm consisting of two phases: guessing and checking. Furthermore, it is assumed that a non-deterministic algorithm always makes a correct guessing*.

For instance, given the satisfiability problem with a particular Boolean formula, a non-deterministic algorithm at first guesses an assignment and then checks whether this assignment satisfies the formula or not. An important concept to note here is that a correct solution is always obtained by guessing. In other words, if the formula is satisfiable, then a non-deterministic algorithm always guesses correctly and obtains an assignment satisfying this formula.

Consider the traveling salesperson decision problem. A non-deterministic algorithm will always guess a tour and check whether this tour is shorter than the constant c.

The reader may be outraged by this concept of non-deterministic algorithms because it is physically impossible to have such an algorithm. How can we always make a correct guess?

Actually, non-deterministic algorithms do not exist and they will never exist. The concept of non-deterministic algorithms is useful only because it will help us later define a class of problems, called NP problems.

If the checking stage of a non-deterministic algorithm is of polynomial time complexity, then this non-deterministic algorithm is called a non-deterministic polynomial algorithm. If a decision problem can be solved by a non-deterministic polynomial algorithm, this problem is called a non-deterministic polynomial (NP for short) problem.

From the above definition, we may immediately conclude that *every problem which can be solved in polynomial time (by deterministic algorithms, of course) must be a non-deterministic polynomial problem*. Typical cases are searching, merging, sorting and minimum spanning tree problems. The reader is reminded here that we are talking about decision problems. Searching is a decision problem; sorting is obviously not. But we can always create a decision problem out of the sorting problem. The original sorting problem is to sort $a_1, a_2, \ldots, a_n$ into an ascending or descending sequence. We may construct a decision problem as follows: Given $a_1, a_2, \ldots, a_n$ and C, determine whether there exists a permutation of a_i's $(a'_1, a'_2, \ldots, a'_n)$ such that $|a'_2 - a'_1| + |a'_3 - a'_2| + \cdots + |a'_n - a'_{n-1}| < C$. All of the problems which can be solved in polynomial time are called P problems.

The satisfiability problem and the traveling salesperson decision problem are both NP problems because the checking stage for both problems is of polynomial time complexity. In fact, most solvable problems that one can think of are NP problems.

A famous decision problem which is not an NP problem is the halting problem. The halting problem is defined as follows: Given an arbitrary program with an arbitrary input data, will the program terminate or not? Another problem is the first-order predicate calculus satisfiability problem. Both problems are so called undecidable problems.

Undecidable problems cannot be solved by guessing and checking. Although they are decision problems, somehow they cannot be solved by exhaustively examining the solution space. The reader should notice that in Boolean logic (or also called propositional logic), an assignment is characterized by an n-tuple. But, for first-order predicate calculus, an assignment is not bounded. It may be of infinite length. This is why the first-order predicate calculus is not an NP problem. It suffices to remind the reader that undecidable problems are even more difficult than NP problems.

Let us be more explicit. For the satisfiability problem and the traveling salesperson decision problem, the number of solutions is finite. There are 2^n possible assignments for the satisfiability problem and $(n - 1)!$ possible tours for the traveling salesperson decision problems. Therefore, although these problems are difficult, they at least have some upper bounds for them. For instance, for the satisfiability problem, we can at least use an algorithm with $O(2^n)$ time complexity to solve it.

There is, however, no such upper bound for the undecidable problems. It can be shown that the upper bounds never exist. Intuitively, we may say that we can let the program run, say 1 million years, and still cannot make any conclusion because it is still possible that in the next step, the program halts. Similarly, for the first-order predicate calculus satisfiability problem, we have the same situation. Suppose after running the program for a very long time, we still have not produced an empty clause. But, it is actually possible that the next clause being generated is an empty clause.

8–5 COOK'S THEOREM

In this section, we shall introduce Cook's theorem. We shall only give an informal proof because a formal proof is very complicated. Cook's theorem can be stated as follows.

Cook's Theorem

NP = P if and only if the satisfiability problem is a P problem.

The proof of the above theorem consists of two parts. The first part is "If NP = P, then the satisfiability problem is a P problem". This part is obvious because the satisfiability problem is an NP problem. The second part is "If the satisfiability problem is a P problem, then NP = P". This is a crucial part of Cook's theorem and we shall elaborate this in the rest of this section.

Let us now explain the main spirit of Cook's theorem. Suppose that we have an NP problem A which is quite difficult to solve. Instead of solving this problem A directly, we shall create another problem A' and by solving that problem A', we shall obtain the solution of A. It is important to note here that every problem is a decision problem. Our approach is as follows:

(1) *Since problem **A** is an NP problem, there must exist an NP algorithm **B** which solves this problem. An NP algorithm is a non-deterministic polynomial algorithm. It is physically impossible and therefore we cannot use it. However, as we shall see, we can still use **B** conceptually in the following steps.*

(2) *We shall construct a Boolean formula **C** corresponding to **B** such that **C** is satisfiable if and only if the non-deterministic algorithm **B** terminates successfully and returns an answer "yes". If **C** is unsatisfiable, then algorithm **B** would terminate unsuccessfully and return the answer "no".*

We shall note at this point that when we mention a problem, we mean an instance of a problem. That is, we mean a problem with a particular input. Otherwise, we cannot say that the algorithm terminates.

We shall also delay the discussion about how C is constructed. This part is the crucial part of Cook's theorem.

(3) *After constructing formula **C**, we shall temporarily forget about our original problem **A** and the non-deterministic algorithm **B**. We shall try to see whether **C** is satisfiable or not. If it is satisfiable, then we say that the answer of problem **A** is "yes"; otherwise, the answer is "no". That we can do so is due to the property of formula **C** stated in Step (2). That is, **C** is satisfiable if and only if **B** terminates successfully.*

All things are beautiful. The above approach seems to suggest that we only have to pay attention to the satisfiability problem. For instance, we never have

to know how to solve the traveling salesperson decision problem; we merely have to know how to determine whether the Boolean formula corresponding to the traveling salesperson problem is satisfiable or not. Yet, there is a big and serious problem here. If the satisfiability problem is hard to solve, the original traveling salesperson problem is still hard to solve. This is the heart of Cook's theorem. It indicates that if the satisfiability problem can be solved in polynomial number of steps, then every NP problem can be solved in polynomial number of steps, essentially because of the above approach.

The reader can see that the above approach is valid if and only if we can always construct a Boolean formula C out of a non-deterministic algorithm B such that C is satisfiable if and only if B terminates successfully. That we can do so will now be illustrated by some examples.

Example 8–1 A Boolean Formula for the Searching Problem (Case 1)

Consider the searching problem. We are given a set $S = \{x(1), x(2), \ldots, x(n)\}$ of n numbers and we are to determine whether there exists a number in S, which is equal to, say 7. To simplify the discussion, we assume that $n = 2$, $x(1) = 7$ and $x(2) \neq 7$.

Our non-deterministic algorithm is as follows:

$i = choice(1, 2)$

If $x(i) = 7$, then *SUCCESS*

else *FAILURE*.

The Boolean formula corresponding to the above non-deterministic algorithm is as follows:

$$
\begin{array}{llll}
 & i = 1 & \vee & i = 2 \\
\& & i = 1 & \rightarrow & i \neq 2 \\
\& & i = 2 & \rightarrow & i \neq 1 \\
\& & x(1) = 7 \ \& \ i = 1 & \rightarrow & SUCCESS \\
\& & x(2) = 7 \ \& \ i = 2 & \rightarrow & SUCCESS \\
\& & x(1) \neq 7 \ \& \ i = 1 & \rightarrow & FAILURE \\
\& & x(2) \neq 7 \ \& \ i = 2 & \rightarrow & FAILURE \\
\& & \text{FAILURE} & \rightarrow & -SUCCESS
\end{array}
$$

& SUCCESS (guarantees a successful termination)
& $x(1) = 7$ (input data)
& $x(2) \neq 7$. (input data)

To facilitate our discussion, let us transform the above formula into its conjunctive normal form:

$$
\begin{array}{lllll}
i = 1 & \vee & i = 2 & & \textbf{(1)} \\
i \neq 1 & \vee & i \neq 2 & & \textbf{(2)} \\
x(1) \neq 7 & \vee & i \neq 1 & \vee \ \mathit{SUCCESS} & \textbf{(3)} \\
x(2) \neq 7 & \vee & i \neq 2 & \vee \ \mathit{SUCCESS} & \textbf{(4)} \\
x(1) = 7 & \vee & i \neq 1 & \text{v} \ \mathit{FAILURE} & \textbf{(5)} \\
x(2) = 7 & \vee & i \neq 2 & \vee \ \mathit{FAILURE} & \textbf{(6)} \\
-\mathit{FAILURE} & \vee & -\mathit{SUCCESS} & & \textbf{(7)} \\
\mathit{SUCCESS} & & & & \textbf{(8)} \\
x(1) = 7 & & & & \textbf{(9)} \\
x(2) \neq 7. & & & & \textbf{(10)}
\end{array}
$$

The above set of clauses are connected by "&" which is omitted here. They are satisfiable as the following assignment will satisfy all of the clauses:

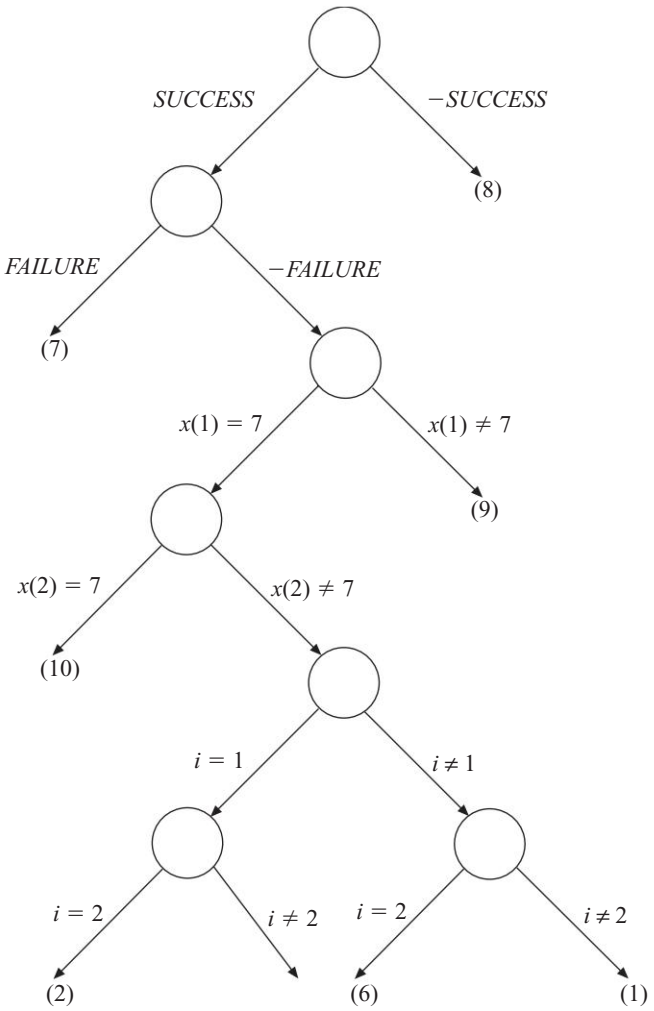
$i = 1$	satisfying	(1)
$i \neq 2$	satisfying	(2), (4) and (6)
SUCCESS	satisfying	(3), (4) and (8)
$-$*FAILURE*	satisfying	(7)
$x(1) = 7$	satisfying	(5) and (9)
$x(2) \neq 7$	satisfying	(4) and (10).

As can be seen, all the clauses are now satisfied. The above assignment which satisfies all the clauses can be found by constructing a semantic tree, shown in Figure 8–8.

We can see from the semantic tree that the only assignment satisfying all clauses is what we gave before.

We have proved that the formula is satisfiable. Why can we claim that the non-deterministic searching algorithm will terminate successfully? This is due to

FIGURE 8–8 A semantic tree.



the fact that this formula describes the execution of the non-deterministic algorithm and there is a special clause, namely *SUCCESS*, which insists that we want the algorithm to terminate successfully.

Proving the satisfiability of the above set of clauses not only tells us that our algorithm will terminate successfully and return "yes", but also the reason why the answer is "yes", which can be found in the assignment. In the assignment, there is one literal,

$$i = 1$$

which constitutes our solution. That is, we not only know that the searching will succeed with answer "yes", we shall also know that the searching will succeed because $x(1) = 7$.

Let us emphasize here that we have successfully transformed a searching problem into a satisfiability problem. Nevertheless one should not be excited by this transformation because the satisfiability problem is still, up to now, hard to solve.

Example 8–2 A Boolean Formula for the Searching Problem (Case 2)

In Example 8–1, we showed a case where a non-deterministic algorithm will terminate successfully. In this case, we shall show a case where a non-deterministic algorithm will terminate unsuccessfully. In such a situation, the corresponding Boolean formula will be unsatisfiable.

We shall still use the searching problem as our example. To simplify our discussion, we assume that $n = 2$ and none of the two numbers is equal to 7. Our Boolean formula will contain the following set of clauses:

$$i = 1 \quad \vee \quad i = 2 \qquad (1)$$

$$i \neq 1 \quad \vee \quad i \neq 2 \qquad (2)$$

$$x(1) \neq 7 \quad \vee \quad i \neq 1 \quad \vee \quad SUCCESS \qquad (3)$$

$$x(2) \neq 7 \quad \vee \quad i \neq 2 \quad \vee \quad SUCCESS \qquad (4)$$

$$x(1) = 7 \quad \vee \quad i \neq 1 \quad \vee \quad FAILURE \qquad (5)$$

$$x(2) = 7 \quad \vee \quad i \neq 2 \quad \vee \quad FAILURE \qquad (6)$$

$$SUCCESS \qquad (7)$$

$$-SUCCESS \quad \vee \quad -FAILURE \qquad (8)$$

$$x(1) \neq 7 \qquad (9)$$

$$x(2) \neq 7. \qquad (10)$$

The above set of clauses are unsatisfiable and this can be proved easily by using the resolution principle.

$$(9)\ \&\ (5) \qquad i \neq 1 \quad \vee \quad FAILURE \qquad (11)$$

$$(10)\ \&\ (6) \qquad i \neq 2 \quad \vee \quad FAILURE \qquad (12)$$

$$(7)\ \&\ (8) \qquad -FAILURE \qquad (13)$$

$$(13)\ \&\ (11) \qquad i \neq 1 \qquad (14)$$

(13) & (12)	$i \neq 2$	**(15)**
(14) & (1)	$i = 2$	**(16)**
(15) & (16)	$\square$.	**(17)**

Perhaps it is interesting to see that the above deduction of an empty clause can be translated into an English-like proof:

(1) Since $x(1) \neq 7$ and $i = 1$ imply *FAILURE* and $x(1)$ is indeed not equal to 7, we have
$i = 1$ implies *FAILURE*. **(11)**

(2) Similarly, we have
$i = 2$ implies *FAILURE*. **(12)**

(3) Since we insist on *SUCCESS*, we have
$-FAILURE$. **(13)**

(4) Therefore i can be neither 1 nor 2. **(14) & (15)**

(5) However, i is either 1 or 2. If i is not 1, i must be 2. **(16)**

(6) A contradiction is derived.

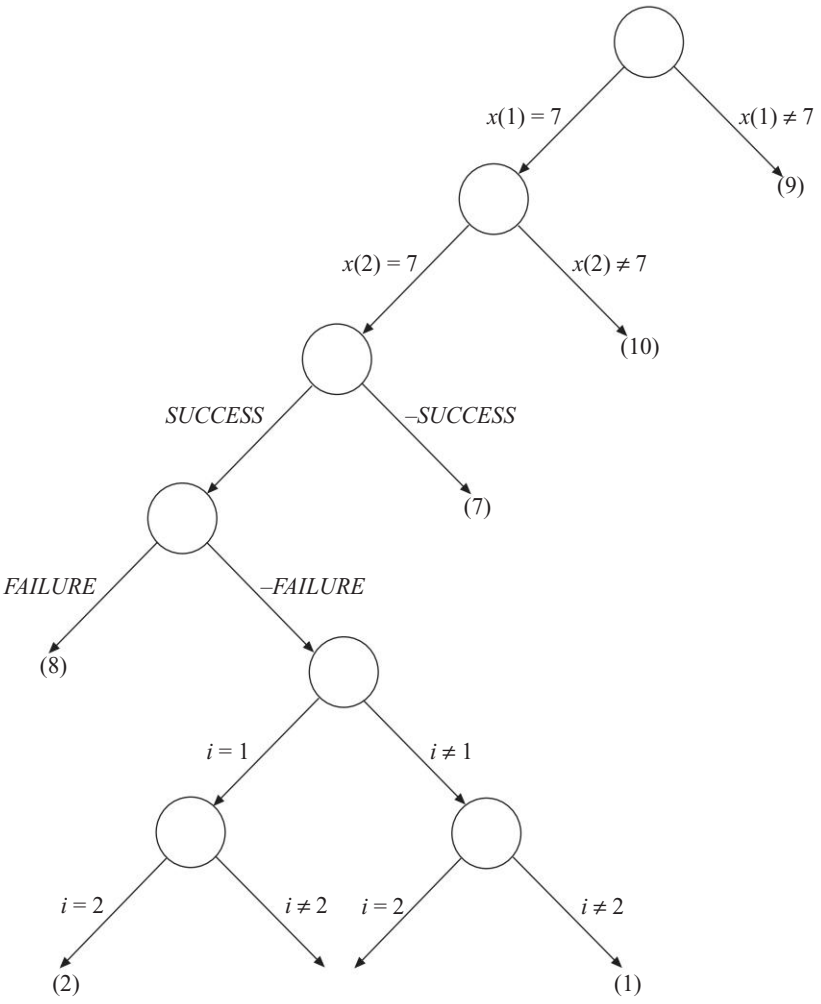
Example 8–3 A Boolean Formula for the Searching Problem (Case 3)

We shall modify Example 8–1 again so that both are equal to 7. In this case, we shall have the following set of clauses:

$i = 1$	$\vee$	$i = 2$			**(1)**
$i \neq 1$	$\vee$	$i \neq 2$			**(2)**
$x(1) \neq 7$	$\vee$	$i \neq 1$	$\vee$	*SUCCESS*	**(3)**
$x(2) \neq 7$	$\vee$	$i \neq 2$	$\vee$	*SUCCESS*	**(4)**
$x(1) = 7$	$\vee$	$i \neq 1$	$\vee$	*FAILURE*	**(5)**
$x(2) = 7$	$\vee$	$i \neq 2$	$\vee$	*FAILURE*	**(6)**
SUCCESS					**(7)**
$-SUCCESS$	$\vee$	$-FAILURE$			**(8)**
$x(1) = 7$					**(9)**
$x(2) = 7$.					**(10)**

A semantic tree is now constructed, shown in Figure 8–9.

FIGURE 8–9 A semantic tree.



In the above semantic tree, we can see that there are two assignments satisfying all of the above set of clauses. In one assignment, $i = 1$ and in the other assignment, $i = 2$.

Example 8–4 A Boolean Formula for the Satisfiability Problem (Case 1)

In this example, we shall show that we can apply our idea to the satisfiability problem. That is, for a satisfiability problem, we can construct a Boolean formula

such that the original satisfiability problem is solved with an answer "yes" if and only if the constructed Boolean formula is satisfiable.

Let us consider the following set of clauses:

$$x_1 \quad \textbf{(1)}$$
$$-x_2. \quad \textbf{(2)}$$

We shall try to determine whether the above set of clauses is satisfiable or not. A non-deterministic algorithm to solve this problem is as follows:

Do $i = 1, 2$
$x_i = choice(T, F)$

If x_1 and x_2 satisfy clauses 1 and 2, then *SUCCESS*, else *FAILURE*.

We shall show how the algorithm can be transformed into a Boolean formula. First of all, we know that in order for the non-deterministic algorithm to terminate with *SUCCESS*, we must have clauses 1 and 2 being true. Thus,

$$\left.\begin{array}{lcll} -SUCCESS & \vee & c_1 = T & \textbf{(1)} \\ -SUCCESS & \vee & c_2 = T & \textbf{(2)} \end{array}\right\} (SUCCESS \rightarrow c_1 = T \,\&\, c_2 = T)$$
$$-c_1 = T \vee x_1 = T \quad \textbf{(3)} \quad (c_1 = T \rightarrow x_1 = T)$$
$$-c_2 = T \vee x_2 = F \quad \textbf{(4)} \quad (c_2 = T \rightarrow x_2 = F)$$
$$x_1 = T \vee x_1 = F \quad \textbf{(5)}$$
$$x_2 = T \vee x_2 = F \quad \textbf{(6)}$$
$$x_1 \neq T \vee x_1 \neq F \quad \textbf{(7)}$$
$$x_2 \neq T \vee x_2 \neq F \quad \textbf{(8)}$$
$$SUCCESS. \quad \textbf{(9)}$$

It is easy to see that the following assignment satisfies all the clauses.

$c_1 = T$	satisfying	(1)
$c_2 = T$	satisfying	(2)
$x_1 = T$	satisfying	(3) and (5)
$x_2 = F$	satisfying	(4) and (6)
$x_1 \neq F$	satisfying	(7)
$x_2 \neq T$	satisfying	(8)
SUCCESS	satisfying	(9).

Thus, the set of clauses is satisfiable.

Example 8–5 A Boolean Formula for the SatisfiabilityProblem (Case 2)

In Example 8–4, we showed that the constructed formula is satisfiable because the original formula is satisfiable. If we start with an unsatisfiable set of clauses, the corresponding formula must also be unsatisfiable. This fact will be demonstrated in this example.

Consider the following set of clauses:

$$x_1 \quad \textbf{(1)}$$
$$-x_1. \quad \textbf{(2)}$$

For the above set of clauses, we may construct the following Boolean formula:

$$-SUCCESS \vee c_1 = T \quad \textbf{(1)}$$
$$-SUCCESS \vee c_2 = T \quad \textbf{(2)}$$
$$-c_1 = T \vee x_1 = T \quad \textbf{(3)}$$
$$-c_2 = T \vee x_1 = F \quad \textbf{(4)}$$
$$x_1 = T \vee x_1 = F \quad \textbf{(5)}$$
$$x_1 \neq T \vee x_1 \neq F \quad \textbf{(6)}$$
$$SUCCESS. \quad \textbf{(7)}$$

That the above set of clauses is unsatisfiable can be proved by using the resolution principle:

$$\text{(1) \& (7)} \quad c_1 = T \quad \textbf{(8)}$$
$$\text{(2) \& (7)} \quad c_2 = T \quad \textbf{(9)}$$
$$\text{(8) \& (3)} \quad x_1 = T \quad \textbf{(10)}$$
$$\text{(9) \& (4)} \quad x_1 = F \quad \textbf{(11)}$$
$$\text{(10) \& (6)} \quad x_1 \neq F \quad \textbf{(12)}$$
$$\text{(11) \& (12)} \quad \square. \quad \textbf{(13)}$$

When we transform a non-deterministic algorithm to a Boolean formula, we must be careful that the corresponding Boolean formula does not contain an exponential number of clauses; otherwise, this transformation would be

meaningless. For instance, suppose we transform a satisfiability problem containing n variables into a set of clauses containing 2^n clauses. Then the transformation procedure itself is an exponential process.

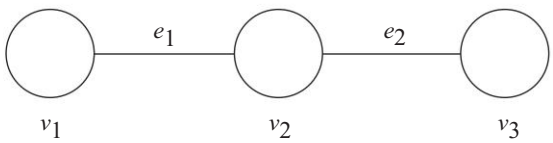
To emphasize this point, let us consider another problem (see Example 8–6).

Example 8–6 The Node Cover Decision Problem

Given a graph $G = (V, E)$, a set S of nodes in V is called a node cover of G if every edge is incident to some node in S.

Consider the graph in Figure 8–10. For this graph, $S = \{v_2\}$ is a node cover as every edge is incident to node v_2.

FIGURE 8–10 A graph.



The node cover decision problem is: Given a graph $G = (V, E)$ and a positive integer k, determine whether there exists a node cover S of G such that $|S| \leq k$.

The above problem can be solved by a non-deterministic algorithm. However, this non-deterministic algorithm cannot just try all subsets of V as the number of such subsets is exponential. But we can use the following non-deterministic polynomial algorithm: Let $|V| = n$ and $|E| = m$.

```
Begin
    i_1 = choice ({1, 2, ... , n})
    i_2 = choice ({1, 2, ... , n} − {i_1})
      ⋮
    i_k = choice ({1, 2, ... , n} − {i_1, i_2, ... , i_{k−1}}).
    For j ≔ 1 to m do
        Begin
            If e_j is not incident to one of v_{i_t}(1 ≤ t ≤ k)
            then FAILURE; stop
        End
    SUCCESS
End
```

Let us consider Figure 8–10 and assume that $k = 1$. In this case, we shall have the following set of clauses, where $v_i \in e_j$ means that e_j is incident to v_i.

$$i_1 = 1 \quad \vee \quad i_1 = 2 \quad \vee \quad i_1 = 3 \qquad \textbf{(1)}$$
$$i_1 \neq 1 \quad \vee \quad v_1 \in e_1 \quad \vee \quad \textit{FAILURE} \qquad \textbf{(2)}$$
$$i_1 \neq 1 \quad \vee \quad v_1 \in e_2 \quad \vee \quad \textit{FAILURE} \qquad \textbf{(3)}$$
$$i_1 \neq 2 \quad \vee \quad v_2 \in e_1 \quad \vee \quad \textit{FAILURE} \qquad \textbf{(4)}$$
$$i_1 \neq 2 \quad \vee \quad v_2 \in e_2 \quad \vee \quad \textit{FAILURE} \qquad \textbf{(5)}$$
$$i_1 \neq 3 \quad \vee \quad v_3 \in e_1 \quad \vee \quad \textit{FAILURE} \qquad \textbf{(6)}$$
$$i_1 \neq 3 \quad \vee \quad v_3 \in e_2 \quad \vee \quad \textit{FAILURE} \qquad \textbf{(7)}$$
$$v_1 \in e_1 \qquad \textbf{(8)}$$
$$v_2 \in e_1 \qquad \textbf{(9)}$$
$$v_2 \in e_2 \qquad \textbf{(10)}$$
$$v_3 \in e_2 \qquad \textbf{(11)}$$
$$\textit{SUCCESS} \qquad \textbf{(12)}$$
$$-\textit{SUCCESS} \quad \vee \quad -\textit{FAILURE}. \qquad \textbf{(13)}$$

We can see that the following assignment satisfies the above set of clauses:

$i_1 = 2$	satisfying	(1)
$v_1 \in e_1$	satisfying	(2) and (8)
$v_2 \in e_1$	satisfying	(4) and (9)
$v_2 \in e_2$	satisfying	(5) and (10)
$v_3 \in e_2$	satisfying	(7) and (11)
SUCCESS	satisfying	(12)
$-$*FAILURE*	satisfying	(13)
$i_1 \neq 1$	satisfying	(3)
$i_1 \neq 3$	satisfying.	(6)

Since the set of clauses is satisfiable, we conclude the answer to the node cover decision problem is YES. The solution is to select node v_2.

From the above informal description, we can now understand the meaning of Cook's theorem. *For every NP problem **A**, we can transform the NP algorithm **B** corresponding to this NP problem to a Boolean formula **C** such that **C** is satisfiable if and only if **B** terminates successfully and returns answer "yes".*

Furthermore, it takes polynomial number of steps to complete this transformation. Thus, if we can determine the satisfiability of a Boolean formula ***C*** *in polynomial number of steps, we can say definitely that the answer to problem* ***A*** *is "yes" or "no". Or, equivalently, we may say that if the satisfiability problem can be solved in polynomial number of steps, then every NP problem can be solved in polynomial number of steps. To put it in another way: If the satisfiability problem is in P, then NP = P.*

It is important to note that Cook's theorem is valid under one constraint: It takes polynomial number of steps to transform an NP problem into a corresponding Boolean formula. If it takes exponential number of steps to construct the corresponding Boolean formula, Cook's theorem cannot be established.

Another important point to note: Although we can construct a Boolean formula describing the original problem, we are still unable to easily solve the original problem because the satisfiability of the Boolean formula cannot be determined easily. Note that when we prove that a formula is satisfiable, we are finding an assignment satisfying this formula. This work is equivalent to finding a solution of the original problem. The non-deterministic algorithm irresponsibly ignores the time needed to find this solution as it claims that it always makes a correct guess. A deterministic algorithm to solve the satisfiability problem cannot ignore this time needed to find an assignment. Cook's theorem indicates that if we can find an assignment satisfying a Boolean formula in polynomial time, then we can really correctly guess a solution in polynomial time. Unfortunately, up to now, we cannot find an assignment in polynomial time. Therefore, we cannot guess correctly in polynomial time.

Cook's theorem informs us that the satisfiability problem is a very difficult problem among all NP problems because if it can be solved in polynomial time, then all NP problems can be solved in polynomial time. But, is the satisfiability problem the only problem in NP with this kind of property? We shall see that there is a class of problems which are equivalent to one another in the sense that if any of them can be solved in polynomial time, then all NP problems can be solved in polynomial time. They are called the class of NP-complete problems and we shall discuss these problems in the next section.

8–6 NP-COMPLETE PROBLEMS

Definition

Let A_1 and A_2 be two problems. A_1 reduces to A_2 (written as $A_1 \propto A_2$) if and only if A_1 can be solved in polynomial time, by using a polynomial time algorithm which solves A_2.

From the above definition, we can say that if $A_1 \propto A_2$, and there is a polynomial time algorithm solving A_2, then there is a polynomial time algorithm to solve A_1.

Using Cook's theorem, we can say that *every NP problem reduces to the satisfiability problem*, because we can always solve this NP problem by first solving the satisfiability problem of the corresponding Boolean formula.

Example 8–7 The *n*-Tuple Optimization Problem

Let us consider the following problem: We are given a positive integer C, $C > 1$ and a positive integer n. Our problem is to determine whether there exist positive integers $c_1, c_2, \ldots, c_n$ such that $\prod_{i=1}^{n} c_i = C$ and $\sum_{i=1}^{n} c_i$ is minimized. We shall call this problem the n-tuple optimization problem.

Let us also consider the famous prime number problem which is to determine whether a positive integer C is a prime number or not. Obviously, the following relation holds: Prime number problem $\propto$ n-tuple optimization problem. The reason is obvious. After solving the n-tuple optimization problem, we examine the solution $c_1, c_2, \ldots, c_n$; C is a prime number if and only if there is exactly one c_i not equal to 1, and all other c_i's are equal to 1. This examination process takes only n steps and is therefore a polynomial process. In summary, if the n-tuple optimization problem can be solved in polynomial time, then the prime number problem can be solved in polynomial time.

Up to now, the n-tuple optimization problem still cannot be solved by any polynomial time algorithm.

Example 8–8 The Bin Packing Problem and Bucket Assignment Problem

Consider the bin packing decision problem and the bucket assignment decision problem.

The bin packing decision problem is defined as follows: We are given a set of n objects which will be put into B bins. Each bin has capacity C and each object requires c_i units of capacity. The bin packing decision problem is to determine whether we can divide these n objects into k, $1 \le k \le B$, groups such that each group of objects can be put into a bin.

For instance, let $(c_1, c_2, c_3, c_4) = (1, 4, 7, 4)$, $C = 8$ and $B = 2$. Then we can divide the objects into two groups: objects 1 and 3 in one group and objects 2 and 4 in one group.

If $(c_1, c_2, c_3, c_4) = (1, 4, 8, 4)$, $C = 8$ and $B = 2$, then there is no way to divide the objects into two groups or one group such that each group of objects can be put into a bin without exceeding the capacity of that bin.

The bucket assignment decision problem is defined as follows: We are given n records which are all characterized by one key. This key assumes h distinct values: $v_1, v_2, \ldots, v_h$ and there are n_i records corresponding to v_i. That is, $n_1 + n_2 + \cdots + n_h = n$. The bucket assignment decision problem is to determine whether we can put these n records into k buckets such that records with the same v_i are within one bucket and no bucket contains more than C records.

For instance, let the key assume values a, b, c and d, $(n_a, n_b, n_c, n_d) = (1, 4, 2, 3)$, $k = 2$ and $C = 5$. Then we can put the records into two buckets as follows:

Bucket 1	**Bucket 2**
a	c
b	c
b	d
b	d
b	d

If $(n_a, n_b, n_c, n_d) = (2, 4, 2, 2)$, then there is no way for us to assign the records into buckets without exceeding the capacity of each bucket and keeping the records with the same key value in the same bucket.

The bucket assignment decision problem is an interesting problem. If the records are stored on disks, the bucket assignment decision problem is related to the problem of minimizing the number of disk accesses. Certainly, if we want to minimize the total number of disk accesses, we should put records with the same key value into the same bucket as much as possible.

It can be easily shown that the bin packing problem reduces to the bucket assignment decision problem. For each bin packing problem, we can create, in polynomial number of steps, a corresponding bucket assignment decision problem. Therefore, if there is a polynomial algorithm to solve the bucket assignment decision problem, we can solve the bin packing problem in polynomial time.

From the definition of "reduce to", we can easily see the following: *If* $A_1 \propto A_2$ *and* $A_2 \propto A_3$, *then* $A_1 \propto A_3$.

Having defined "reduce to", we can now define NP-complete problems.

Definition

A problem A is NP-complete if $A \in$ NP and every NP problem reduces to A.

From the above definition, we know that if ***A*** *is an NP-complete problem and* ***A*** *can be solved in polynomial time, then every NP problem can be solved in polynomial time. Clearly, the satisfiability problem is an NP-complete problem because of Cook's theorem.*

By definition, if any NP-complete problem can be solved in polynomial time, then $NP = P$.

The satisfiability problem was the first found NP-complete problem. Later, R. Karp showed 21 NP-complete problems. These NP-complete problems include node cover, feedback arc set, Hamiltonian cycle, etc. Karp received a Turing Award in 1985.

To show that a problem A is NP-complete, we do not have to prove that all NP problems reduce to A. This is what Cook did when he showed the NP-completeness of the satisfiability problem. Nowadays, we merely have to use the transitive property of "reduce to". *If* A_1 *is an NP-complete problem,* A_2 *is an NP problem and we can prove that* $A_1 \propto A_2$, *then* A_2 *is an NP-complete problem*. The reasoning is rather straightforward. If A is an NP-complete problem, then all NP problems reduce to A. If $A \propto B$, then all NP problems reduce to B because of the transitive property of "reduce to". Therefore B must be NP-complete.

In the previous discussion, we made the following statements:

(1) The satisfiability problem is the most difficult problem among all NP problems.

(2) When we prove the NP-completeness of a problem A, we often try to prove that the satisfiability problem reduces to A. Thus, it appears that A is more difficult than the satisfiability problem.

To see that there is no inconsistency in these statements, let us note that every NP problem reduces to the satisfiability problem. Thus, if we are interested in an NP problem A, then certainly A reduces to the satisfiability problem. However, we must emphasize here that saying a problem A reduces to the satisfiability problem is not significant at all because it only means that the satisfiability problem is more difficult than A, which is a well-established fact. If we successfully proved that the satisfiability problem reduces to A, then A is even more difficult than the satisfiability problem, which is a highly significant statement. Note that $A \propto$ the satisfiability problem and the satisfiability problem $\propto A$. Therefore, so far as the degree of difficulty is concerned, A is equivalent to the satisfiability problem.

We may extend the above arguments to all NP-complete problems. *If **A** is an NP-complete problem, then by definition every NP problem, say **B**, reduces to **A**. If we further prove that **B** is NP-complete by proving **A** $\propto$ **B**, then **A** and **B** are equivalent to each other. In summary, all NP-complete problems form an equivalent class*.

Note that we have restricted NP problems to be decision problems. We may now extend the concept of NP-completeness to optimization problems by defining "NP-hardness". *A problem **A** is NP-hard if every NP problem reduces to **A**. (Note that **A** is not necessarily an NP-problem. In fact, **A** may be an optimization problem.) Thus, a problem is NP-complete if **A** is NP-hard and A is an NP*. Through this way, an optimization problem is NP-hard if its corresponding decision problem is NP-complete. For instance, the traveling salesperson problem is NP-hard.

8–7 EXAMPLES OF PROVING NP-COMPLETENESS

In this section, we shall show that many problems are NP-complete. We would like to remind the reader that when we want to prove that a problem A is NP-complete, we usually do this in two steps:

(1) We first prove that A is an NP problem.
(2) We then prove that some NP-complete problem reduces to A.

Many readers make the mistake by showing that A reduces to an NP-complete problem. This is totally meaningless because by definition, every NP problem reduces to every NP-complete problem.

We note that up to now, we have only accepted that the satisfiability problem is NP-complete. To produce more NP-complete problems, we must start from the satisfiability problem. That is, we should try to prove that the satisfiability problem reduces to the problem which we are interested.

Example 8–9 The 3-Satisfiability Problem

The 3-satisfiability problem is similar to the satisfiability problem with more restriction: Every clause contains exactly three literals.

It is obvious that the 3-satisfiability problem is an NP problem. To show that it is an NP-complete problem, we need to show that the satisfiability problem reduces to the 3-satisfiability problem. We shall show that *for every ordinary Boolean formula F_1, we can create another Boolean formula F_2, in which every clause contains exactly three literals, such that F_1 is satisfiable if and only if F_2 is satisfiable*.

Let us start with one example. Consider the following set of clauses:

$$x_1 \vee x_2 \quad \textbf{(1)}$$
$$-x_1. \quad \textbf{(2)}$$

We may extend the above set of clauses so that every clause now contains three literals:

$$x_1 \vee x_2 \vee y_1 \quad \textbf{(1)}'$$
$$-x_1 \vee y_2 \vee y_3. \quad \textbf{(2)}'$$

We can see that (1) & (2) is satisfiable and (1)′ & (2)′ is also satisfiable. But the above method of adding some new literals may create problems as can be seen in the following case:

$$x_1 \quad \textbf{(1)}$$
$$-x_1. \quad \textbf{(2)}$$

(1) & (2) is unsatisfiable. If we add arbitrarily some new literals to these two clauses:

$$x_1 \vee y_1 \vee y_2 \qquad (1)'$$
$$-x_1 \vee y_3 \vee y_4. \qquad (2)'$$

$(1)'$ & $(2)'$ becomes a satisfiable formula.

The above discussion shows that we cannot arbitrarily add new literals to a formula without affecting its satisfiability. What we can do is to append new clauses which themselves are unsatisfiable to the original set of clauses. If the original set of clauses is satisfiable, the new set is of course still satisfiable. If the original set of clauses is unsatisfiable, the new set of clauses will be unsatisfiable.

If the original clause contains only one literal, we may add the following set of clauses:

$$y_1 \vee y_2$$
$$-y_1 \vee y_2$$
$$y_1 \vee -y_2$$
$$-y_1 \vee -y_2.$$

For instance, suppose the original clause is x_1, the newly created clauses will be

$$x_1 \vee y_1 \vee y_2$$
$$x_1 \vee -y_1 \vee y_2$$
$$x_1 \vee y_1 \vee -y_2$$
$$x_1 \vee -y_1 \vee -y_2.$$

If the original clause contains two literals, we may add

$$y_1$$
$$-y_1.$$

For instance, suppose the original clause is

$$x_1 \vee x_2.$$

The newly created clauses will be

$$x_1 \vee x_2 \vee y_1$$
$$x_1 \vee x_2 \vee -y_1.$$

Consider the following unsatisfiable set of clauses:

$$x_1 \quad \textbf{(1)}$$
$$-x_1. \quad \textbf{(2)}$$

We shall now have

$$\begin{array}{lllll}
x_1 & \vee & y_1 & \vee & y_2 \\
x_1 & \vee & -y_1 & \vee & y_2 \\
x_1 & \vee & y_1 & \vee & -y_2 \\
x_1 & \vee & -y_1 & \vee & -y_2 \\
-x_1 & \vee & y_3 & \vee & y_4 \\
-x_1 & \vee & -y_3 & \vee & y_4 \\
-x_1 & \vee & y_3 & \vee & -y_4 \\
-x_1 & \vee & -y_3 & \vee & -y_4.
\end{array}$$

This new set of clauses is still unsatisfiable.

If a clause contains more than three literals, we can break this clause into a set of new clauses by adding the following set of unsatisfiable clauses:

$$\begin{array}{lll}
y_1 & & \\
-y_1 & \vee & y_2 \\
-y_2 & \vee & y_3 \\
 & \vdots & \\
-y_{i-1} & \vee & y_i \\
-y_i. & &
\end{array}$$

Consider the following clause:

$$x_1 \vee -x_2 \vee x_3 \vee x_4 \vee -x_5.$$

We may add new variables to create new clauses containing exactly three literals:

$$\begin{array}{lllll}
x_1 & \vee & -x_2 & \vee & y_1 \\
x_3 & \vee & -y_1 & \vee & y_2 \\
x_4 & \vee & -x_5 & \vee & -y_2.
\end{array}$$

We may summarize the rules of transforming a clause into a set of clauses containing exactly three literals. (In the following, all y_i's represent new variables.)

(1) If the clause contains only one literal L_1, then create the following four clauses:

$$\begin{aligned} &L_1 \vee y_1 \vee y_2 \\ &L_1 \vee -y_1 \vee y_2 \\ &L_1 \vee y_1 \vee -y_2 \\ &L_1 \vee -y_1 \vee -y_2. \end{aligned}$$

(2) If the clause contains two literals L_1 and L_2, then create the following two clauses:

$$\begin{aligned} &L_1 \vee L_2 \vee y_1 \\ &L_1 \vee L_2 \vee -y_1. \end{aligned}$$

(3) If the clause contains three literals, do nothing.

(4) If the clause contains more than three literals, create new clauses as follows: Let there be literals $L_1, L_2, \ldots, L_k$. The new clauses are

$$\begin{aligned} &L_1 \vee L_2 \vee y_1 \\ &L_3 \vee -y_1 \vee y_2 \\ &\vdots \\ &L_{k-1} \vee L_k \vee -y_{k-3}. \end{aligned}$$

Consider the following set of clauses:

$$\begin{aligned} &x_1 \vee x_2 \\ &-x_3 \\ &x_1 \vee -x_2 \vee x_3 \vee -x_4 \vee x_5. \end{aligned}$$

We shall now create the following set of clauses:

$$\begin{aligned} &x_1 \vee x_2 \vee y_1 \\ &x_1 \vee x_2 \vee -y_1 \\ &-x_3 \vee y_2 \vee y_3 \\ &-x_3 \vee -y_2 \vee y_3 \\ &-x_3 \vee y_2 \vee -y_3 \\ &-x_3 \vee -y_2 \vee -y_3 \\ &x_1 \vee -x_2 \vee y_4 \\ &x_3 \vee -y_4 \vee y_5 \\ &-x_4 \vee x_5 \vee -y_5. \end{aligned}$$

As we indicated before, the above transformation must preserve the satisfiability property of the original set of clauses. That is, let S denote the original set of clauses. Let S' denote the set of transformed clauses where each clause contains three literals. Then S' is satisfiable if and only if S is satisfiable. This will be proved next.

(1) Part 1: If S is satisfiable, then S' is satisfiable. Let I denote an assignment satisfying S. Then, obviously I satisfies all clauses created from clauses containing no more than three literals. Let C be a clause in S which contains more than three literals and $T(C)$ be the set of clauses in S' related to C. For instance, for

$$C = x_1 \vee x_2 \vee -x_3 \vee x_4 \vee -x_5$$

$$T(C) = \begin{cases} x_1 \vee x_2 \vee y_1 \\ -x_3 \vee -y_1 \vee y_2 \\ x_4 \vee -x_5 \vee -y_2. \end{cases}$$

I satisfies C, as assumed. If I also satisfies $T(C)$, we are done. Otherwise, I must satisfy at least a subset of $T(C)$. We shall now explain how I can be expanded into I' so that I' satisfies all clauses in $T(C)$. This can be explained as follows: Let C_i be a clause in $T(C)$ satisfied by I. Assign the last literal of clause C_i false. This assignment will satisfy another clause C_j in $T(C)$. If every clause in $T(C)$ is satisfied, we are done. Otherwise, assign the last literal of clause C_j false. This process can be repeated until every clause is satisfied.

Consider $C = x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5$.

$T(C)$ is:

$$x_1 \vee x_2 \vee y_1 \quad \textbf{(1)}$$
$$x_3 \vee -y_1 \vee y_2 \quad \textbf{(2)}$$
$$x_4 \vee x_5 \vee -y_2. \quad \textbf{(3)}$$

Suppose $I = \{x_1\}$. I satisfies (1). We assign y_1 of (1) false. Thus, $I' = \{x_1, -y_1\}$ and (2) is also satisfied. We further assign y_2 of (2) false. This finally makes (3) satisfied. Thus $I' = \{x_1, -y_1, -y_2\}$ satisfies all clauses.

(2) Part 2: If S' is satisfiable, S is satisfiable. Note that for S', the newly appended clauses are unsatisfiable themselves. Therefore, if S' is satisfiable, the assignment satisfying S' cannot contain y_i's only. It must assign truth-values to some x_i's, the original variables. Consider a set S_j of clauses in S' which are created corresponding to a clause C_j in S. Since any assignment satisfying S_j must satisfy at least one literal of C_j, this assignment satisfies C_j. Therefore, if S' is satisfiable, S is satisfiable.

Since it takes polynomial time to transform an arbitrary set of clauses S to a set of clauses S' where each clause contains three literals, and S' is satisfiable if and only if S is satisfiable, we conclude that if we can solve the 3-satisfiability problem in polynomial time, then we can solve the satisfiability problem in polynomial time. Thus, the satisfiability problem reduces to the 3-satisfiability problem and the 3-satisfiability problem is NP-complete.

In the above example, we showed that the 3-satisfiability problem is NP-complete. Perhaps many readers may be a little bit puzzled at this point by the following wrong reasoning: The 3-satisfiability problem is a special case of the satisfiability problem. Since the satisfiability problem is an NP-complete problem, the 3-satisfiability problem must automatically be an NP-complete problem.

Note that the degree of difficulty of a special case of a general problem cannot be deduced by examining the general problem. Consider the satisfiability problem. It is NP-complete. But it is entirely possible that a special version of the satisfiability problem is not NP-complete. Consider the case where each clause contains positive literals only. That is, no negative sign occurs. A typical example is as follows:

$$
\begin{aligned}
&x_1 \vee x_2 \vee x_3\\
&x_1 \vee x_4\\
&x_4 \vee x_5\\
&x_6.
\end{aligned}
$$

In this case, a satisfying assignment can be easily obtained by assigning each variable T. Thus, this special class of the satisfiability problems is not NP-complete.

In general, if a problem is NP-complete, its special cases may or may not be NP-complete. On the other hand, if a special case of a problem is NP-complete, then this problem is NP-complete.

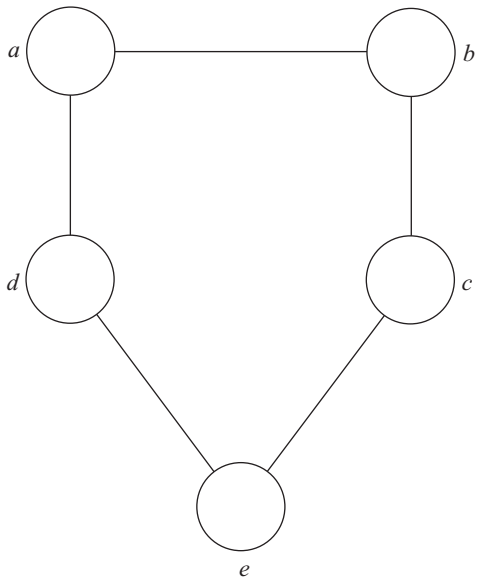
Example 8–10 The Chromatic Number Decision Problem

In this example, we shall show that the chromatic number decision problem is NP-complete. The chromatic number decision problem is defined as follows: We are given a graph $G = (V, E)$. For each vertex, associate with it a color in such a way that if two vertices are connected by an edge, then these two vertices must be associated with different colors. The chromatic number decision problem is to determine whether it is possible to use k colors to color the vertices.

Consider Figure 8–11. For this graph, we may use three colors to color the graph as follows:

$a \leftarrow 1,\ b \leftarrow 2,\ c \leftarrow 1,\ d \leftarrow 2,\ e \leftarrow 3.$

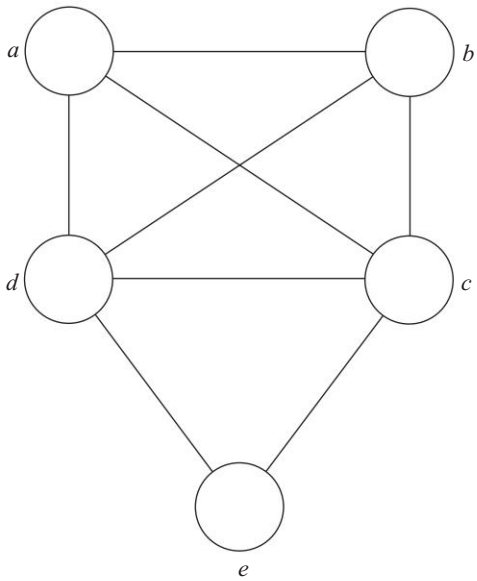
FIGURE 8–11 A graph which is 3-colorable.



Consider Figure 8–12. It is easy to see that we have to use four colors to color the graph.

To prove that the chromatic number decision problem is NP-complete, we need an NP-complete problem and show that this NP-complete problem reduces to the chromatic number decision problem. In this case, we shall use a problem which is similar to the 3-satisfiability problem discussed in Example 8–9. This problem is

FIGURE 8–12 A graph which is 4-colorable.



a satisfiability problem where each clause contains at most three literals. We can easily see that this satisfiability problem with at most three literals per clause is also NP-complete because of the discussion in Example 8–9. For every ordinary satisfiability problem, we can always transform it into a satisfiability problem with at most three literals per clause without affecting the original satisfiability; we merely have to break clauses with more than three literals to clauses with exactly three literals.

We shall now prove that the satisfiability problem with at most three literals per clause reduces to the chromatic number decision problem. Essentially, we shall show that for every satisfiability problem with at most three literals per clause, we can construct a corresponding graph such that the original Boolean formula is satisfiable if and only if the constructed graph can be colored by using $n + 1$ colors where n is the number of variables occurring in the Boolean formula.

Let $x_1, x_2, \ldots, x_n$ denote the variables in the Boolean formula F where $n \geq 4$. If $n < 4$, then n is a constant and the satisfiability problem can be determined easily. Let $C_1, C_2, \ldots, C_r$ be the clauses where each clause contains at most three literals.

The graph G corresponding to the Boolean formula is constructed according to the following rules:

(1) The vertices of the graph G are $x_1, x_2, \ldots, x_n, -x_1, -x_2, \ldots, -x_n, y_1, y_2, \ldots, y_n, C_1, C_2, \ldots, C_r$.
(2) The edges of the graph G are formed by the following rules:
   (a) There is an edge between every x_i and every $-x_i$, for $1 \le i \le n$.
   (b) There is an edge between every y_i and every y_j if $i \ne j$, $1 \le i, j \le n$.
   (c) There is an edge between every y_i and every x_j if $i \ne j$, $1 \le i, j \le n$.
   (d) There is an edge between every y_i and every $-x_j$ if $i \ne j$, $1 \le i, j \le n$.
   (e) There is an edge between every x_i and every C_j if $x_i \notin C_j$, $1 \le i \le n$, $1 \le j \le r$.
   (f) There is an edge between every $-x_i$ and every C_j if $-x_i \notin C_j$, $1 \le i \le n$, $1 \le j \le r$.

We shall now prove that F is satisfiable if and only if G is $n + 1$ colorable. The proof consists of two parts:

(1) If F is satisfiable, then G is $n + 1$ colorable.
(2) If G is $n + 1$ colorable, then F is satisfiable.

Let us prove the first part. We now assume that F is satisfiable. In this case, we may pick any assignment A satisfying F and color the vertices as follows:

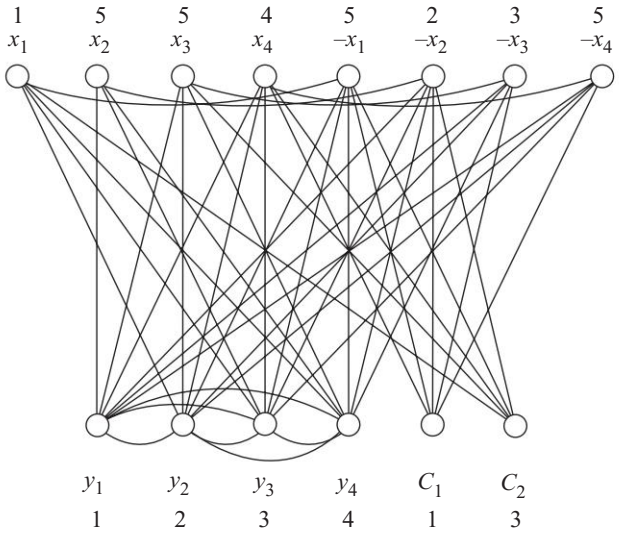
(1) For all y_i's, y_i is colored with color i.
(2) For all x_i's and $-x_i$'s, assign x_i and $-x_i$ with colors as follows: If x_i is assigned T in A, then x_i is colored with color i and $-x_i$ is colored with $n + 1$; otherwise, x_i is colored with color $n + 1$ and $-x_i$ is colored with i.
(3) For each C_j, find a literal L_i in C_j which is true in A. Since A satisfies every clause, such L_i exists for all clauses. Assign C_j with the same color as this literal. That is, if L_i is x_i, assign C_j with the same color as x_i; otherwise, assign C_j with the same color as $-x_i$.

Consider the following set of clauses:

$$x_1 \vee x_2 \vee x_3 \quad \textbf{(1)}$$
$$-x_3 \vee -x_4 \vee x_2. \quad \textbf{(2)}$$

The graph is shown in Figure 8–13.

FIGURE 8–13 A graph constructed for the chromatic number decision problem.



Let $A = (x_1, -x_2, -x_3, x_4)$. Then the x_i's are assigned as follows:

$x_1 \leftarrow 1, \qquad -x_1 \leftarrow 5$
$x_2 \leftarrow 5, \qquad -x_2 \leftarrow 2$
$x_3 \leftarrow 5, \qquad -x_3 \leftarrow 3$
$x_4 \leftarrow 4, \qquad -x_4 \leftarrow 5.$

The C_i's are colored as follows:

$C_1 \leftarrow 1$ (x_1 in A satisfies C_1.)
$C_2 \leftarrow 3.$ ($-x_3$ in A satisfies C_2.)

To show that this coloring is legal, we may use the following reasoning:

(1) Each y_i is connected to each y_j if $i \neq j$. Therefore, no two y_i's can be assigned with the same color. This is done as we assign y_i with color i.
(2) Each x_i is connected to $-x_i$. Therefore, x_i and $-x_i$ cannot be colored the same. This is done because no x_i and $-x_i$ are assigned with the same color. Besides, no y_j will have the same color as x_i or $-x_i$ according to our rules.

(3) Consider C_j's. Suppose L_i appears in C_j, L_i is true in A and C_j is colored the same as L_i. Among all x_i's and $-x_i$'s, only that particular x_i, or $-x_i$, which is equal to L_i, will have the same color as C_j. But L_i is not connected to C_j because L_i appears in C_j. Therefore, no C_j will have the same color as any x_i, or $-x_i$, which is connected to C_j.

From the above discussion, we may conclude that if F is satisfiable, G is $n + 1$ colorable.

We now prove the other part. If G is $n + 1$ colorable, F must be satisfiable. The reasoning is as follows:

(1) Without losing generality, we may assume that y_i is colored with color i.
(2) Since x_i is connected to $-x_i$, x_i and $-x_i$ cannot be assigned with the same color. Since x_i and $-x_i$ are connected to y_j if $i \neq j$, either x_i is assigned with color i and $-x_i$ with color $n + 1$, or $-x_i$ is assigned with color i and x_i with color $n + 1$.
(3) Since every C_j contains at most three literals and $n \geq 4$, there is at least one i such that neither x_i nor $-x_i$ appears in C_j. Therefore, every C_j must be connected to at least one vertex colored with color $n + 1$. Consequently, no C_j is colored with color $n + 1$.
(4) For each C_j, $1 \leq j \leq r$, if C_j is assigned with color i and x_i is assigned with color i, then in the assignment A assign x_i to be true. If C_j is assigned with color i and $-x_i$ is assigned with color i, then in the assignment A assign x_i to be false.
(5) Note that if C_j is assigned with color i, it must not be linked to x_i or $-x_i$, whichever is assigned with color i. This means that the literal which is assigned with color i must appear in C_j. In A, this particular literal is assigned true and therefore satisfies C_j. Consequently, A satisfies all clauses. F must be satisfiable because there exists at least one assignment satisfying all clauses.

In the above discussion, we have shown that for every set of clauses where each clause contains at most three literals, we may construct a graph such that the original set of clauses with n variables is satisfiable if and only if the corresponding constructed graph is $n + 1$ colorable. Besides, it is easy to show that the construction of the graph takes polynomial number of steps. Therefore, *the satisfiability problem with at most three literals per clause reduces to the chromatic number decision problem and the chromatic number decision problem is NP-complete*.

In the rest of this section, we shall show that a VLSI discrete layout problem is NP-complete. To do this, we must prove that some other problems are NP-complete.

Example 8–11 The Exact Cover Problem

Let there be a family of sets $F = \{S_1, S_2, \ldots, S_k\}$ and a set S of elements $\{u_1, u_2, \ldots, u_n\}$, which is $\bigcup_{S_i \in F} S_i$. The exact cover problem is to determine whether there is a subset $T \subseteq F$ of pairwise disjoint sets such that

$$\bigcup_{S_i \in T} S_i = \{u_1, u_2, \ldots, u_n\} = \bigcup_{S_i \in F} S_i.$$

For instance, suppose that $F = \{(a_3, a_1), (a_2, a_4), (a_2, a_3)\}$. Then $T = \{(a_3, a_1), (a_2, a_4)\}$ is an exact cover of F. Note that every pair of sets in T must be disjoint. If $F = \{(a_3, a_1), (a_4, a_3), (a_2, a_3)\}$, then there is no exact cover.

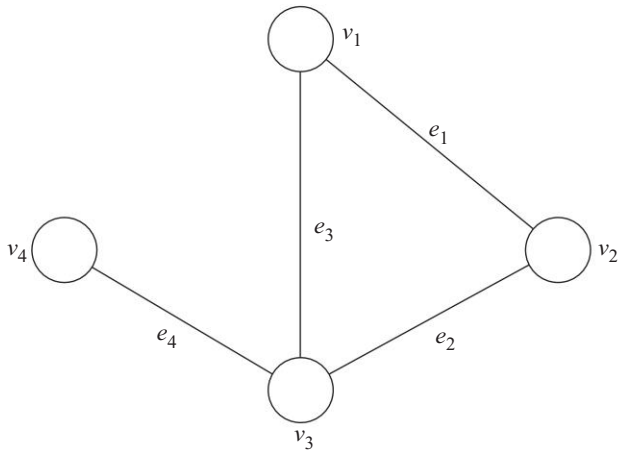
We now try to demonstrate that this exact cover problem is NP-complete by reducing the chromatic coloring problem to this exact cover problem. The chromatic coloring problem was introduced in Example 8–10.

Let the set of vertices of the given graph in the chromatic coloring problem be $V = \{v_1, v_2, \ldots, v_n\}$ and the set of edges be $E = \{e_1, e_2, \ldots, e_m\}$. Together with the integer k, we transform this chromatic coloring problem instance into an exact cover problem instance $S = \{v_1, v_2, \ldots, v_n, E_{11}, E_{12}, \ldots, E_{1k}, E_{21}, E_{22}, \ldots, E_{2k}, \ldots, E_{m1}, E_{m2}, \ldots, E_{mk}\}$, where $E_{i1}, E_{i2}, \ldots, E_{ik}$ correspond to e_i, $1 \le i \le m$, and a family F of subsets $F = \{C_{11}, C_{12}, \ldots, C_{1k}, C_{21}, C_{22}, \ldots, C_{2k}, \ldots, C_{n1}, C_{n2}, \ldots, C_{nk}, D_{11}, D_{12}, \ldots, D_{1k}, D_{21}, D_{22}, \ldots, D_{2k}, \ldots, D_{m1}, D_{m2}, \ldots, D_{mk}\}$. Each C_{ij} and D_{ij} are determined according to the following rule:

(1) If edge e_i has vertices v_a and v_b as its ending terminals, then C_{ad} and C_{bd} will both contain E_{id} for $d = 1, 2, \ldots, k$.
(2) $D_{ij} = \{E_{ij}\}$ for all i and j.
(3) C_{ij} contains v_i for $j = 1, 2, \ldots, k$.

Let us give an example. Consider Figure 8–14 which shows a graph.

FIGURE 8–14 A graph illustrating the transformation of a chromatic coloring problem to an exact cover problem.



In this case, $n = 4$ and $m = 4$. Suppose $k = 3$. We therefore have $S = \{v_1, v_2, v_3, v_4, E_{11}, E_{12}, E_{13}, E_{21}, E_{22}, E_{23}, E_{31}, E_{32}, E_{33}, E_{41}, E_{42}, E_{43}\}$ and $F = \{C_{11}, C_{12}, C_{13}, C_{21}, C_{22}, C_{23}, C_{31}, C_{32}, C_{33}, C_{41}, C_{42}, C_{43}, D_{11}, D_{12}, D_{13}, D_{21}, D_{22}, D_{23}, D_{31}, D_{32}, D_{33}, D_{41}, D_{42}, D_{43}\}$. Each D_{ij} contains exactly one E_{ij}. For C_{ij}, we shall illustrate its contents by one example. Consider e_1, which is connected by v_1 and v_2. This means that C_{11} and C_{21} will both contain E_{11}. Similarly, C_{12} and C_{22} will both contain E_{12}. C_{13} and C_{23} will also both contain E_{13}.

The entire family F of sets will be constructed as follows:

$C_{11} = \{E_{11}, E_{31}, v_1\}$,
$C_{12} = \{E_{12}, E_{32}, v_1\}$,
$C_{13} = \{E_{13}, E_{33}, v_1\}$,
$C_{21} = \{E_{11}, E_{21}, v_2\}$,
$C_{22} = \{E_{12}, E_{22}, v_2\}$,
$C_{23} = \{E_{13}, E_{23}, v_2\}$,
$C_{31} = \{E_{21}, E_{31}, E_{41}, v_3\}$,
$C_{32} = \{E_{22}, E_{32}, E_{42}, v_3\}$,
$C_{33} = \{E_{23}, E_{33}, E_{43}, v_3\}$,
$C_{41} = \{E_{41}, v_4\}$,
$C_{42} = \{E_{42}, v_4\}$,
$C_{43} = \{E_{43}, v_4\}$,

$$D_{11} = \{E_{11}\}, D_{12} = \{E_{12}\}, D_{13} = \{E_{13}\},$$
$$D_{21} = \{E_{21}\}, D_{22} = \{E_{22}\}, D_{23} = \{E_{23}\},$$
$$D_{31} = \{E_{31}\}, D_{32} = \{E_{32}\}, D_{33} = \{E_{33}\},$$
$$D_{41} = \{E_{41}\}, D_{42} = \{E_{42}\}, D_{43} = \{E_{43}\}.$$

We shall omit a formal proof that the chromatic coloring problem has a k coloring if and only if the constructed exact cover problem has a solution. Meanwhile, we shall just demonstrate the validity of this transformation through an example.

For the graph in Figure 8–14, there is a 3-coloring. We may let v_1, v_2, v_3 and v_4 be colored as 1, 2, 3 and 1 respectively. In this case, let us choose C_{11}, C_{22}, C_{33}, C_{41}, D_{13}, D_{21}, D_{32} and D_{42} as the cover. We first can easily see that they are pairwise disjoint. Furthermore, set S is exactly covered by those sets. For instance, v_1, v_2, v_3 and v_4 are covered by C_{11}, C_{22}, C_{33} and C_{41} respectively. E_{11} is covered by C_{11}, E_{12} is covered by C_{22} and E_{13} is covered by D_{13}.

The formal proof of this reducibility is left as an exercise.

Example 8–12 The Sum of Subsets Problem

The sum of subsets problem is defined as follows: Let there be a set of numbers $A = \{a_1, a_2, \ldots, a_n\}$ and a constant C. Determine whether there exists a subset A' of A such that the elements of A' add up to C.

For instance, let $A = \{7, 5, 19, 1, 12, 8, 14\}$ and C be 21. Then this sum of subsets problem has a solution, namely $A' = \{7, 14\}$. If C is equal to 11, the sum of subsets problem has no solution.

We can prove the NP-completeness of this problem by reducing the exact cover problem to this sum of subsets problem. Given an exact cover problem instance $F = \{S_1, S_2, \ldots, S_n\}$ and a set $S = \bigcup_{S_i \in F} S_i = \{u_1, u_2, \ldots, u_m\}$, we construct a corresponding sum of subsets problem instance according to the following rule: The sum of subsets problem instance contains a set $A = \{a_1, a_2, \ldots, a_n\}$ where

$$a_j = \sum_{1 \le i \le m} e_{ji}(n+1)^{i-1} \text{ where } e_{ji} = 1 \text{ if } u_i \in S_j \text{ and } e_{ji} = 0 \text{ otherwise.}$$
$$C = \sum_{0 \le i \le m-1} (n+1)^i = ((n+1)^m - 1)/n.$$

Again, we shall leave the formal proof of the validity of this transformation as an exercise.

Example 8–13 The Partition Decision Problem

The partition problem is defined as follows: We are given $A = \{a_1, a_2, \ldots, a_n\}$ where each a_i is a positive integer. The partition problem is to determine whether there is a partition $A = \{A_1, A_2\}$ such that

$$\sum_{a_i \in A_1} a_i = \sum_{a_i \in A_2} a_i .$$

For instance, let $A = \{1, 3, 8, 4, 10\}$. We can partition A into two subsets $\{1, 8, 4\}$ and $\{3, 10\}$ and it is easy to check that the sum of elements in the first subset is equal to the sum of elements in the second subset.

The NP-completeness can be proved by reducing the sum of subsets problem into this problem. How this can be done will be left as an exercise.

Example 8–14 The Bin Packing Decision Problem

The bin packing decision problem, introduced before, can be described as follows: We are given a set of n items, each of size c_i which is a positive integer. We are also given positive integers B and C which are the number of bins and the bin capacity respectively. We are asked to determine whether we can assign items into k bins, $1 \le k \le B$, such that the sum of c_i's over all items assigned to each bin does not exceed C.

We can establish the NP-completeness of this problem by reducing the partition problem to it. Suppose that a partition problem instance has $A = \{a_1, a_2, \ldots, a_n\}$. We can define the corresponding bin packing decision problem by setting $B = 2$, $c_i = a_i$, $1 \le i \le n$ and $C = \sum_{1 \le i \le n} a_i/2$. Clearly, the bin packing decision problem has a solution with k equal to 2 if and only if there exists a partition for A.

Example 8–15 VLSI Discrete Layout Problem

In this problem, we are given a set S of n rectangles and our job is to determine whether it is possible to place these rectangles, under some constraints, into a large rectangle with a specified area. Formally speaking, we are given a set of

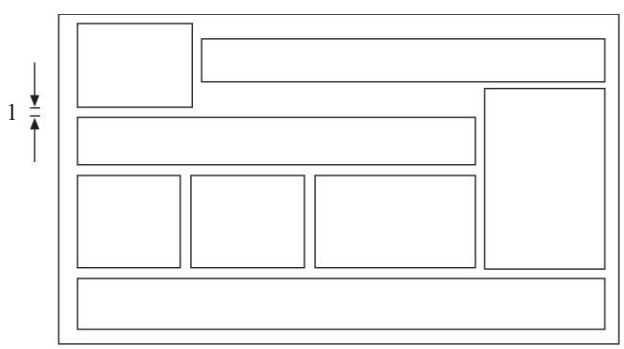
n rectangles and an integer A. For $1 \le i \le n$, each rectangle r_i has dimensions h_i and w_i which are positive integers. The VLSI discrete layout problem is to determine whether there is a placement of rectangles on the plane such that:

(1) Each boundary is parallel to one of the coordinate system axes.
(2) Corners of the rectangles lie on integer points in the plane.
(3) No two rectangles overlap.
(4) The boundaries of two rectangles are separated by at least a unit distance.
(5) There is a rectangle in the plane which circumscribes the placed rectangles, has boundaries parallel to the axes and is of area at most A. The boundary of the circumscribing rectangle is allowed to contain boundaries of placed rectangles.

Consider Figure 8–15, which shows a successful placement. The NP-completeness of this VLSI discrete layout problem can be established by showing that the bin packing decision problem can be reduced to the VLSI discrete layout problem. In a bin packing decision problem, we are given n items where each item is of size c_i. The number of bins is B and the size capacity for each bin is C. For such a bin packing decision problem, we construct a VLSI discrete layout problem as follows:

(1) Corresponding to each c_i we shall have a rectangle r_i with height $h_i = 1$ and width $w_i = (2B + 1)c_i - 1$.
(2) In addition, we shall have another rectangle with width $w = (2B + 1)C - 1$ and height $h = 2Bw + 1$.

FIGURE 8–15 A successful placement.



(3) The area of the circumscribing rectangle is $A = w(h + 2B)$.

We first show that if the bin packing decision problem has a solution, then the constructed VLSI discrete layout problem also has a solution. Suppose that for bin i, the sizes of items stored in it are $d_{i1}, d_{i2}, \ldots, d_{ij_i}$. Our placement will arrange the corresponding rectangles in a row in such a way that the height of this row is kept to be 1, shown in Figure 8–16. If k bins are used, there will be k rows of rectangles where each rectangle is of height 1. We then place the rectangle with width w and height h exactly below these rows, shown in Figure 8–17.

FIGURE 8–16 A particular row of placement.

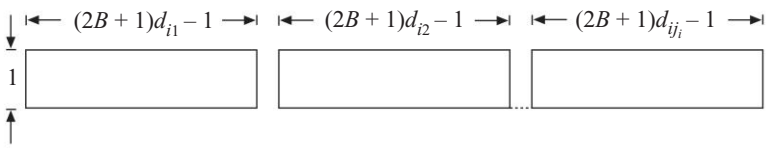
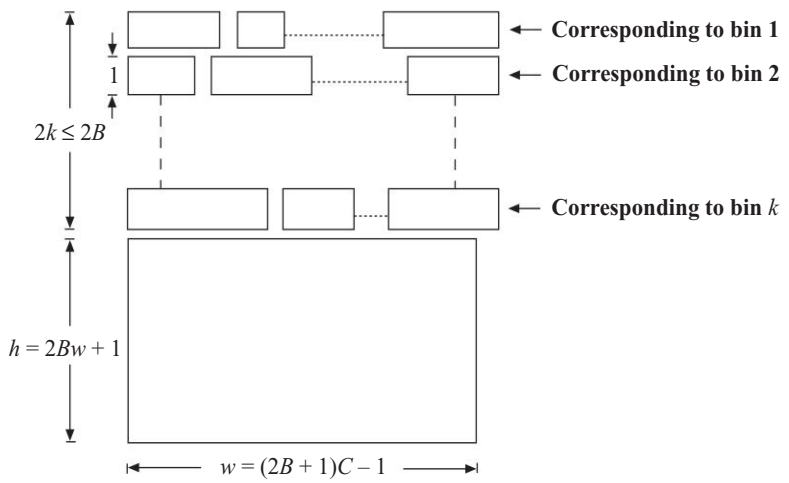


FIGURE 8–17 A placement of $n + 1$ rectangles.



For each row i with j_i rectangles, the width can be found as follows:

$$\sum_{a=1}^{j_i} ((2B + 1)d_{i_a} - 1) + (j_i - 1)$$
$$= (2B + 1)\left(\sum_{a=1}^{j_i} d_{i_a}\right) - j_i + j_i - 1$$
$$\leq (2B + 1)C - 1$$
$$= w.$$

Thus, the width of these $n + 1$ rectangles is less than or equal to w and the corresponding height is less than $(2k + h) \leq (2B + h)$. The total area of the placement of these $n + 1$ rectangles is thus less than

$$(2B + h)w = A.$$

This means that if the bin packing decision problem has a solution, the corresponding $n + 1$ rectangles can be placed in a rectangle with size A.

We now prove the other way round. Suppose that we have successfully placed the $n + 1$ rectangles into a rectangle with size $A = w(h + 2B)$. We shall now show that the original bin packing decision problem has a solution.

Our main reasoning is that this successful placement cannot be arbitrary; it must be under some constraints. The constraints are as follows:

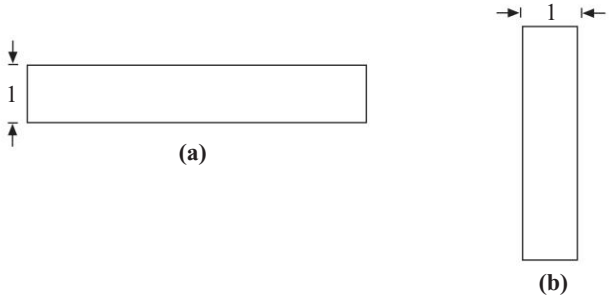
(1) The width of the placement must be less than $w + 1$. Suppose otherwise. Since we have one rectangle with height h, the total area will be larger than

$$h(w + 1)$$
$$= hw + h$$
$$= hw + 2Bw + 1$$
$$= w(h + 2B) + 1$$
$$= A + 1 > A,$$

which is impossible.

(2) Each rectangle r_i must be laid out in such a way that its contribution to the total height is 1. In other words, rectangle r_i must be placed as shown in Figure 8–18(a); it cannot be placed as shown in Figure 8–18(b).

FIGURE 8–18 Possible ways to place r_i.



Suppose otherwise. Then the total height must be larger than

$$h + (2B + 1)c_i - 1 + 1$$
$$= h + (2B + 1)c_i$$

for some i. In this case, the total area will become larger than

$$(h + (2B + 1)c_i)w$$
$$= hw + 2Bc_iw + wc_i$$
$$= (2Bc_i + h)w + wc_i$$
$$\geq A + wc_i$$
$$> A$$

which is impossible.

(3) The total number of rows occupied by the n rectangles cannot be larger than B. If it is larger, then the area will be larger than

$$w(h + 2B) = A,$$

which is impossible.

Based on the above arguments, we can easily prove that

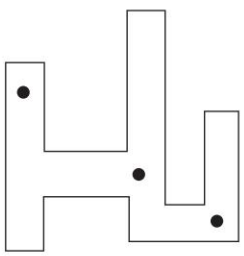
$$\sum_{k=1}^{j_i} d_{i_k} \leq C.$$

In other words, we can put the items corresponding to rectangles in row i into bin i without exceeding the bin capacity. Therefore, we complete the proof.

Example 8–16 The Art Gallery Problem for Simple Polygons

The art gallery problem, mentioned in Chapter 3, is defined as follows: We are given an art gallery, we are asked to station a minimum number of guards in it such that every point of the art gallery is visible to at least one guard. We assume that the art gallery is represented by a simple polygon. Thus, the art gallery problem can be also phrased as follows: *We are given a simple polygon, station the minimum number of guards in it such that each point of the simple polygon is visible to at least one guard*. For instance, consider the simple polygon in Figure 8–19. We need at least three guards in this case.

FIGURE 8–19 A simple polygon and the minimum number of guards for i_t.



If we stipulate that every guard can be stationed only in a vertex of the simple polygon, then we call this special version of the art gallery problem the minimum vertex guard problem. We shall now show that the minimum vertex guard problem is NP-hard. The NP-hardness of the art gallery problem can be proved similarly. We first define the *vertex guard decision problem as follows: We are given a simple polygon* $\boldsymbol{P}$ *with* $\boldsymbol{n}$ *vertices and a positive integer* $\boldsymbol{K} < \boldsymbol{n}$*, we are asked to determine whether there exists a subset* $\boldsymbol{T} \subseteq \boldsymbol{V}$ *with* $|\boldsymbol{T}| \leq \boldsymbol{K}$ *such that placing one guard per each vertex in* $\boldsymbol{T}$ *will cause every point in* $\boldsymbol{P}$ *to be visible to at least one guard stationed in* $\boldsymbol{T}$.

It is easy to see that the vertex guard decision problem is NP since a nondeterministic algorithm needs only to guess a subset $V' \subseteq V$ of K vertices and check in polynomial time whether stationing one guard at each vertex in V' will allow each point in the polygon to be visible to that guard. To show the NP-completeness of this decision problem, the 3-satisfiability (3SAT) problem, which is NP-complete, will be used.

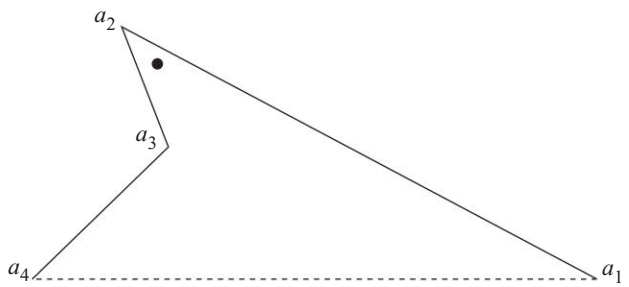
We shall show that the 3SAT problem is polynomially reducible to the vertex guard decision problem. We first define distinguished points as follows: *Two*

distinct points in a polygon are distinguished if they cannot be visible from any point in this polygon. This definition is needed for our later discussions. In order to construct a simple polygon from a given instance of the 3SAT problem, we shall first define basic polygons corresponding to literals, clauses and variables respectively. These basic polygons will be combined into the final transformed simple polygon.

Literal Polygons

For each literal, we shall construct a polygon, shown in Figure 8–20. In Figure 8–20, the dot sign "•" indicates a distinguished point because, as we shall show later, these literal polygons will be arranged in such a way that in the final simple polygon, no point in it can see both of them. It is easy to see that only a_1 and a_3 can see this entire literal polygon. In a 3SAT problem instance, for each clause, there are three literals. Therefore, in each polygon corresponding to a clause, there will be three such literal polygons.

FIGURE 8–20 Literal pattern subpolygon.

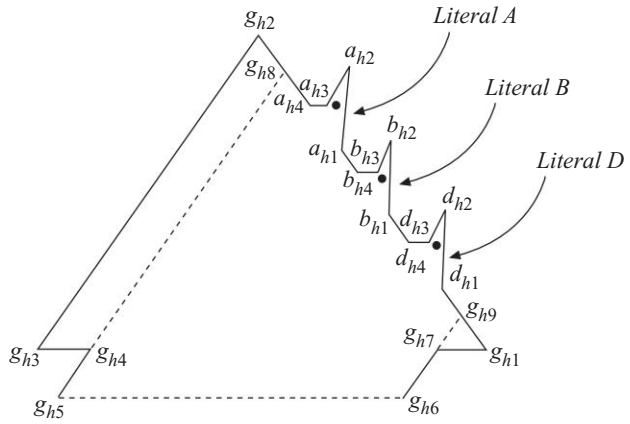


Clause Polygons

For each clause $C_i = A \vee B \vee D$, we shall construct a clause polygon, shown in Figure 8–21. Let $(a_1, a_2, \ldots, a_n)$ denote the fact that $a_1, a_2, \ldots, a_n$ are collinear. Then, in Figure 8–21, we have (g_{h8}, g_{h4}, g_{h5}), $(g_{h3}, g_{h4}, g_{h7}, g_{h1})$, $(g_{h2}, g_{h8}, a_{h4}, a_{h1}, b_{h4}, b_{h1}, d_{h4}, d_{h1}, g_{h9}, g_{h1})$ and (g_{h9}, g_{h7}, g_{h6}). Moreover, $|(g_{h2}, g_{h8})| = |(g_{h8}, a_{h4})|$ and $|(d_{h1}, g_{h9})| = |(g_{h9}, g_{h1})|$ where $|(u, v)|$ denotes the length of the line segment (u, v).

There are some important properties in a clause polygon. First of all, we can easily see that no g_{hi}, $i = 1, 2, \ldots, 7$, can see any entire literal polygon. Thus, we must station guards inside the literal polygons. Note that no guard stationed in any

FIGURE 8–21 Clause junction $C_h = A \vee B \vee D$.



literal polygon can see the entire other two literal polygons. Therefore, we need three guards, one stationed in each literal polygon. But there are certain vertices in these literal polygons where no guard can be stationed. In the following, we shall show that only seven combinations chosen from $\{a_{h1}, a_{h3}, b_{h1}, b_{h3}, d_{h1}, d_{h3}\}$ are candidates for stationing guards. The reasoning is as follows:

(1) None of a_{h4}, b_{h4} and d_{h4} can see any of the three entire literal polygons. Thus, they cannot be candidates.

(2) Consider the literal polygon for literal A. If a_{h2} is chosen, then no matter how we station guards in literal polygons for literals B and D, $\Delta a_{h1}a_{h3}a_{h4}$ will not be visible from any guard. Thus, a_{h2} should be ruled out. That is, we only have to choose these guards from $\{a_{h1}, a_{h3}, b_{h1}, b_{h3}, d_{h1}, d_{h3}\}$.

(3) No two vertices should be chosen from the same literal polygon. Thus, there are only eight possible combinations of the vertices chosen from $\{a_{h1}, a_{h3}, b_{h1}, b_{h3}, d_{h1}, d_{h3}\}$. But $\{a_{h3}, b_{h3}, d_{h3}\}$ should be ruled out because it cannot see the entire $\Delta g_{h1}g_{h2}g_{h3}$. This means that we have only seven possible combinations as summarized below.

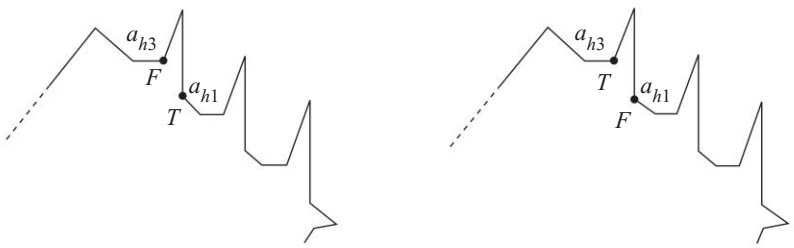
Property 1: Only seven combinations of vertices chosen from $\{a_{h1}, a_{h3}, b_{h1}, b_{h3}, d_{h1}, d_{h3}\}$ can see the entire clause polygon. These seven combinations of these vertices are $\{a_{h1}, b_{h1}, d_{h1}\}$, $\{a_{h1}, b_{h1}, d_{h3}\}$, $\{a_{h1}, b_{h3}, d_{h1}\}$, $\{a_{h1}, b_{h3}, d_{h3}\}$, $\{a_{h3}, b_{h1}, d_{h1}\}$, $\{a_{h3}, b_{h1}, d_{h3}\}$ and $\{a_{h3}, b_{h3}, d_{h1}\}$.

Labeling Mechanism 1 for Clause Polygons

Note that every literal polygon corresponds to a literal. A literal may appear positively or negatively in a clause. However, from the above discussion of the construction of these literal polygons, we do not seem to take the signs of literals into consideration. This is puzzling. Actually, as we shall see later, the sign of a literal will determine the labeling of some vertices of this literal polygon. Then the assignment of truth value to this variable will determine where guards are to be stationed by taking the labeling of vertices into consideration.

Consider literal A. The other cases are similar. Note that for literal A, we must station one guard at either a_{h1} or a_{h3}. If A is positive (negative), we label vertex $a_{h1}(a_{h3})$ as T and vertex $a_{h1}(a_{h1})$ as false. See Figure 8–22. If this variable is assigned true (false), we station a guard at the vertex which is labeled $T(F)$. This means that vertex $a_{h1}(a_{h3})$ represents a true (false) assignment for literal A.

FIGURE 8–22 Labeling mechanism 1 for clause junctions.



(a) literal $A = u_i$ **(b) literal $A = -u_i$**

Let us now use an example to explain the relationship between the labeling mechanism, the truth assignment and finally the stationing of guards. Consider the case where $C_h = u_1 \vee -u_2 \vee u_3$. Figure 8–23 shows the labeling for this case.

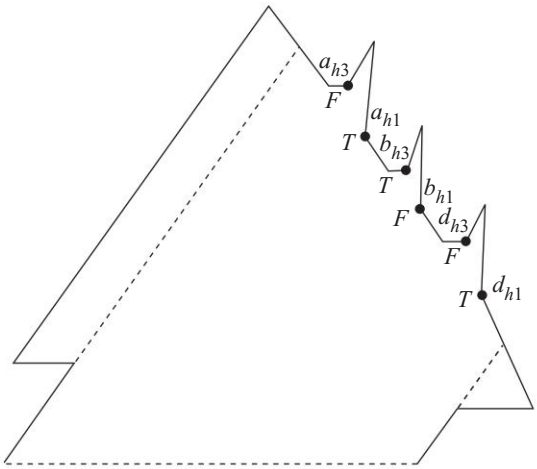
Consider a satisfying assignment, say

$$u_1 \leftarrow T, u_2 \leftarrow F, u_3 \leftarrow T.$$

In this case, guards will be stationed at a_{h1}, b_{h1} and d_{h1}. Guards stationed in such a way will be unable to see the entire clause polygon. Consider another satisfying assignment, say

$$u_1 \leftarrow T, u_2 \leftarrow T, u_3 \leftarrow F.$$

FIGURE 8–23 An example for labeling mechanism 1 for clause junction $C_h = u_1 \vee -u_2 \vee u_3$.



In this case, the guards will be stationed at a_{h1}, b_{h3} and d_{h3}. Again, the guards stationed in this way will be able to see the entire clause polygon. Finally, consider the following unsatisfiable assignment:

$u_1 \leftarrow F, u_2 \leftarrow T, u_3 \leftarrow F.$

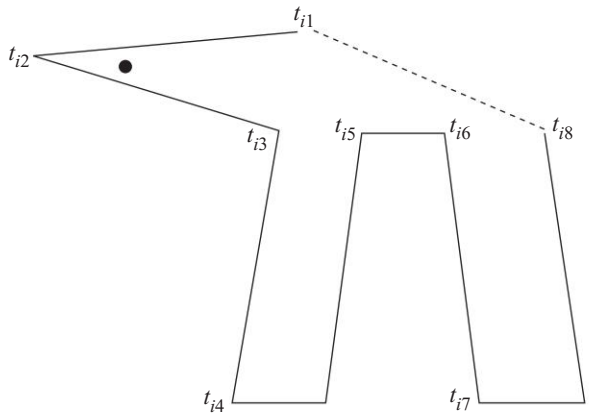
In this case, guards will be stationed at a_{h3}, b_{h3} and d_{h3}. Guards stationed in such a way will be unable to see the entire clause polygon. By carefully examining labeling mechanism 1, we can see that among the eight different combinations of vertices from $\{a_{h1}, a_{h3}, b_{h1}, b_{h3}, d_{h1}, d_{h3}\}$, only $\{a_{h3}, b_{h3}, d_{h3}\}$ corresponds to an unsatisfying assignment of truth values to variables appearing in the clause. Therefore, we have the following property:

Property 2: *The clause polygon of a clause C_h can be visible from three vertices in the literal polygons if and only if the truth values corresponding to the labels of the vertices constitute a satisfying assignment to clause C_h.*

Variable Polygons

For each variable, we construct a variable polygon, shown in Figure 8–24. We have $(t_{i3}, t_{i5}, t_{i6}, t_{i8})$. A distinguished point exists in $\Delta t_{i1}t_{i2}t_{i3}$. Note that $\Delta t_{i1}t_{i2}t_{i3}$ can only be visible from t_{i1}, t_{i2}, t_{i3}, t_{i5}, t_{i6}, and t_{i8}.

FIGURE 8–24 Variable pattern for u_i.



We have described clause polygons and variable polygons. We now describe how we can merge them into the problem instance polygon.

The Construction of a Problem Instance Polygon

Step 1. Putting variable polygons and clause polygons together.
We put variable polygons and clause polygons together, shown in Figure 8–25. In Figure 8–25, we have the following facts:

(1) Vertex w can visualize the n variable polygons except $\Delta t_{i1}t_{i2}t_{i3}$ where $i = 1, 2, 3, \ldots, n$.
(2) $(w, g_{15}, g_{16}, g_{25}, g_{26}, \ldots, g_{m5}, g_{m6})$ and $(t_{11}, g_{h5}, g_{h4}, g_{h8})$ are satisfied where $h = 1, 2, \ldots, m$.

Step 2. Augmenting variable polygons with spikes.
Suppose that variable u_i appears in clause C_h. If u_i appears in clause C_h positively, then the two spikes, $\overline{pq}$ and $\overline{rs}$, shown in Figure 8–26(a) are such that (a_{h1}, t_{i5}, p, q) and (a_{h3}, t_{i8}, r, s) are satisfied. The situation of u_i appearing in C_h negatively is shown in Figure 8–26(b).

Step 3. Replacing spikes by polygons.
Since the added spikes $\overline{pq}$ and $\overline{rs}$ are line segments, we would not have a simple polygon after Step 2 is completed. Therefore, we must replace spikes by polygons as follows. We shall explain the case in Figure 8–26(a) as that in Figure 8–26(b) is quite similar. In

FIGURE 8–25 Merging variable patterns and clause junctions together.

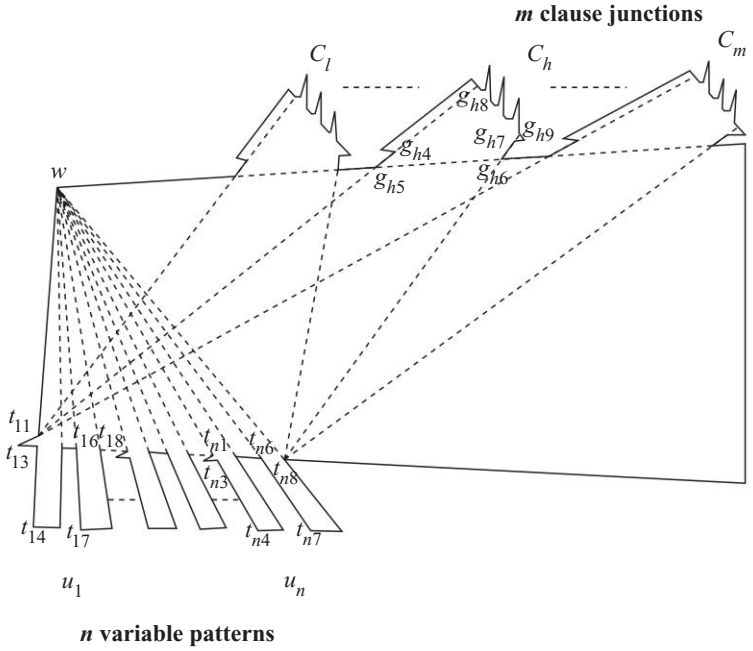


FIGURE 8–26 Augmenting spikes.

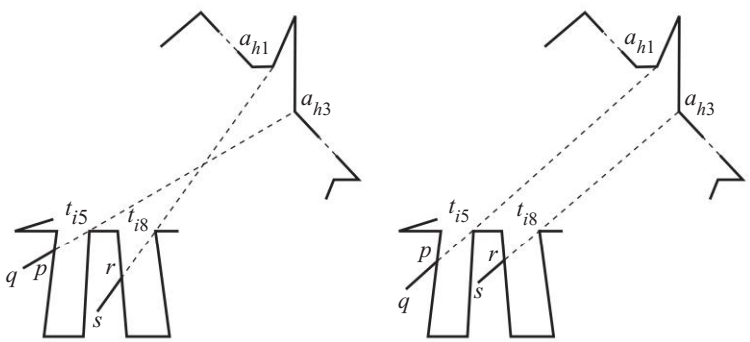
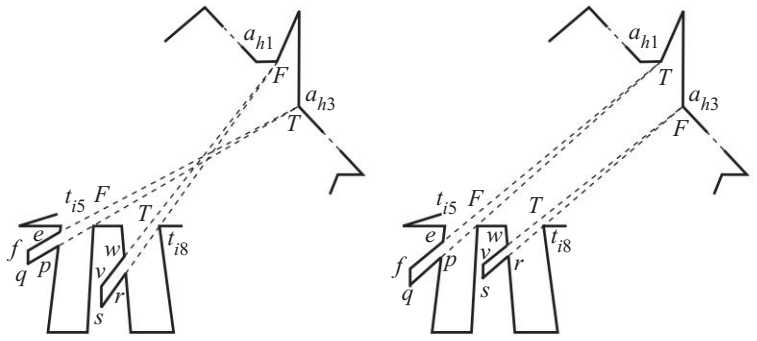


Figure 8–26(a), we replace $\overline{pq}$ and $\overline{rs}$ by the shaded areas shown in Figure 8–27(a). In Figure 8–27(b), we have (a_{h1}, t_{i5}, p, q), (a_{h1}, e, f), (a_{h3}, t_{i8}, r, s) and (a_{h3}, u, v). That is, we have $\Delta a_{h1}qf$ and $\Delta a_{h3}sv$. The polygons created in Figure 8–27 are called consistency-check polygons. The meaning of consistency check will be explained later.

FIGURE 8–27 Replacing each spike by a small region, called a consistency-check pattern.



Labeling Mechanism 2 for Variable Polygons

For variable polygon u_i, vertices t_{i5} and t_{i8} are always labeled F and T respectively, shown in Figure 8–27. If u_i is assigned true (false), we then station a guard at the vertex which is labeled $T(F)$.

Let us consider Figure 8–27(a) again. In this case, $u_i \in C_h$. Note that there are two ways of placing one guard in the literal polygon and one guard at the variable polygon.

$$F = (u_1 \vee u_2 \vee -u_3) \wedge (-u_1 \vee -u_2 \vee u_3).$$

They are $\{a_{h1}, t_{i8}\}$ and $\{a_{h3}, t_{i5}\}$. The first case corresponds to assigning true to u_i and the second case corresponds to assigning false to u_i.

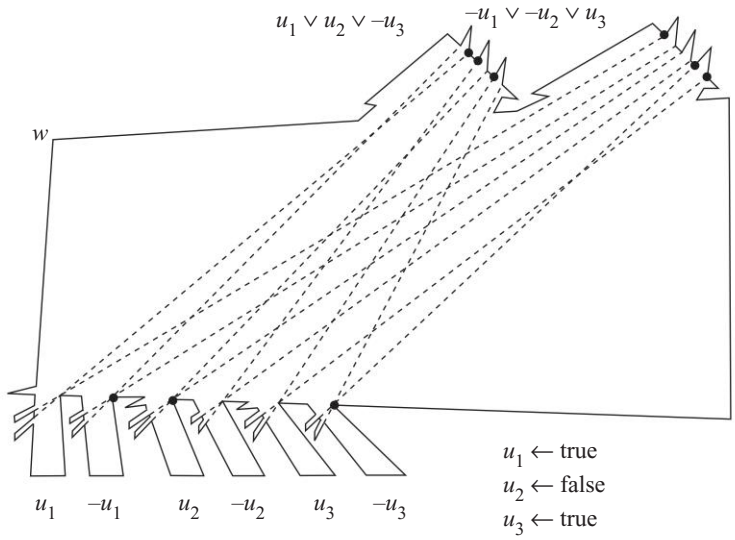
In Figure 8–28, a complete simple polygon is constructed from the 3SAT formula $F = (u_1 \vee u_2 \vee -u_3) \wedge (-u_1 \vee -u_2 \vee u_3)$. Guards are represented by dots. In this case, the minimum number of guards is 10. According to labeling mechanisms 1 and 2, this pattern of stationing of guards corresponds to

$$u_1 \leftarrow T, u_2 \leftarrow F, u_3 \leftarrow T.$$

We now show the following main results.

Claim 1: Given a set S of clauses, if S is satisfiable, the minimum number of vertices needed to see the entire problem instance polygon of S is $K = 3m + n + 1$.

FIGURE 8–28 An example of a simple polygon constructed from the 3SAT formula.



This can be reasoned as follows: If S is satisfiable, there is a truth value assignment to the variables appearing in S such that each clause C_h is true under this assignment. If u_i is assigned true, we station a guard at t_{i8} of the corresponding variable polygon for u_i and we station a guard at either a_{h1} or a_{h3} of the corresponding literal polygon depending on whether u_i is C_h, or $-u_i$ is in C_h respectively. If u_i is assigned false, we station a guard at t_{i5} of the corresponding variable polygon for u_i and we station a guard at either a_{h1} or a_{h3} of the corresponding literal polygon depending upon whether $-u_i$ or u_i is in C_h respectively. Thus, for the polygons defined by the consistency-check polygons and literal polygons, we need to place $3m + n$ guards to see them according to labeling mechanisms 1 and 2. For the remaining rectangles defined by the variable polygons, we only need one guard placed at vertex w to see them. Thus, we have $K = 3m + n + 1$.

Claim 2: Given a set S of clauses, if the minimum number of guards is $K = 3m + n + 1$, then S is satisfiable.

This part of the proof is much more complicated. First, we claim that w must be chosen; otherwise, at lease $3m + 2n$ vertices are needed since a guard must be stationed in each of the $2n$ rectangles in the n variable polygons Thus, in the following, we only consider the remaining $K - 1 = 3m + n$ vertices.

In the constructed simple polygon, each of the $3m$ literal polygons has a distinguished point and each of the n variable polygons also has one. Thus, there are $3m + n$ distinguished points in the simple polygon. We know that no vertex which sees a distinguished point can see any other distinguished points. Thus, at least $3m + n$ vertices are needed to see the $3m + n$ distinguished points.

We cannot just choose these $3m + n$ points arbitrarily; otherwise, they cannot see the rest of the simple polygon. In the following, we shall show that actually they must be chosen so that it corresponds to an assignment of truth values to variables satisfying S. We shall call such a pattern of stationing of guards consistent which is defined as follows:

Let u_i be a variable. Let u appear in $C_1, C_2, \ldots, C_a$, positively or negatively. Let $x_1, x_2, \ldots, x_a$ be vertices chosen from literal polygons corresponding to u appearing in $C_1, C_2, \ldots, C_a$. Let $\boldsymbol{X_i}$ be $\{x_1, x_2, \ldots, x_a\}$. Then the variable polygon of u_i is consistent with respect to $\boldsymbol{X_i}$ if the following conditions are satisfied:

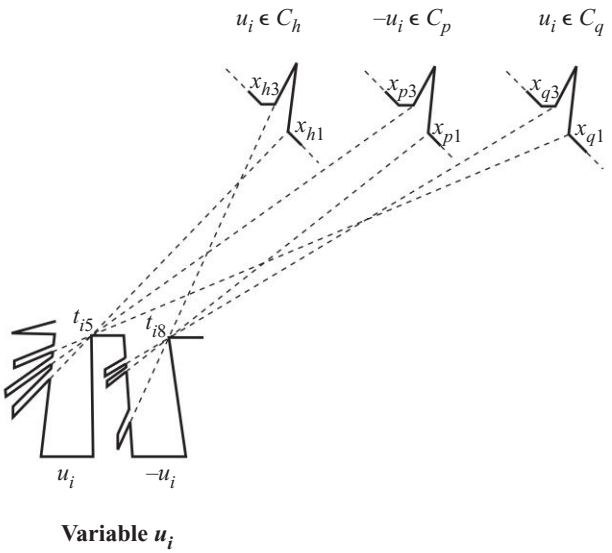
(1) *All the consistency-check polygons connected to one of its two rectangles can be visible from the vertices in $\boldsymbol{X_i}$.*
(2) *Those connected to the other rectangle cannot be visible at all from the same set in $\boldsymbol{X_i}$.*

The variable polygon of $\boldsymbol{u_i}$ is inconsistent with respect to $\boldsymbol{X_i}$ if otherwise.

For example, consider Figure 8–29. Assume that variable u_i appears in clauses C_h, C_p and C_q and appears positively (u_i), negatively ($-u_i$) and positively (u_i) respectively. If we choose vertices x_{h1}, y_{p3} and z_{q1}, then the variable polygon is consistent with respect to this set of vertices as the left rectangle can be seen from these three variables and the right rectangle cannot be seen from them at all. This set of vertices correspond to u_i assigned true. Similarly, it is consistent with respect to x_{h3}, y_{p1} and z_{p3}. It is inconsistent with respect to x_{h3}, y_{p1} and z_{q1}, which corresponds to assigning u_i false. *One may see that when a variable polygon for variable $\boldsymbol{u_i}$ is consistent with respect to $\boldsymbol{X_i}$, then $\boldsymbol{X_i}$ corresponds to a certain assignment of truth value to u_i and if it is inconsistent to $\boldsymbol{X_i}$, then $\boldsymbol{X_i}$ does not correspond to any truth value assignment to $\boldsymbol{u_i}$.*

We claim that all variable polygons must be consistent with respect to the set of vertices in the literal polygons if we only have $K = 3m + n$ vertices. In any variable polygon L_i for U_i, *the number of consistency-check polygons depends on the number of clauses in which u_i* appears. Let this number be q. Since the number of consistency-check polygons connected to the left rectangle and that to

FIGURE 8–29 Illustrating the concept of being consistent.



the right rectangle of L_i are the same, the total number of consistency-check polygons in L_i is $2q$. There are q of them connected to one rectangle, either left or right. If left (right), we put a guard at $t_{i8}(t_{i5})$. Thus, only $3m + n + 1$ guards are needed. If L_i is inconsistent, then some of the consistency-check polygons connected to the left rectangle and some of the consistency-check polygons connected to the right rectangle cannot be seen from any vertex. Thus, at least $3m + n + 2$ vertices are needed. Thus, $K = 3m + n + 1$ requires that all of the variable polygons are consistent with respect to the vertices chosen from the literal polygons.

Since every variable u_i is consistent to X_i, the consistency-check polygons connected to one of its rectangles can be seen from vertices in X_i. For the other rectangle, the consistency-check polygons can be seen by either t_{i8} or t_{i5}. Besides, t_{i8} or t_{i5} can see the distinguished point of the variable polygon also. This means that exactly one vertex is needed to see each variable polygon and n vertices are needed to see the n variable polygons. We thus have it that all the m entire clause polygons, which cannot be seen from w and any vertex chosen from variable polygons, must be seen from $3m + n + 1 - (n + 1) = 3m$ vertices chosen from the literal polygons. According to Property 2, three vertices per clause will correspond to a truth assignment to variables appearing in this clause. Consequently, C is satisfiable.

8–8 THE 2-SATISFIABILITY PROBLEM

In this chapter, we showed that the satisfiability problem is NP-complete. We also stated that while a problem is NP-complete, its variant is not necessarily NP-complete. To emphasize this point, we shall show that the 2-satisfiability problem is in P, which is quite surprising.

The 2-satisfiability problem (2SAT for short) is a special case of the satisfiability problem in which there are exactly two literals in each clause.

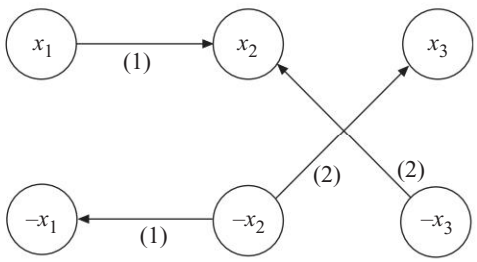
Given a 2SAT problem instance, we can construct a directed graph, called the assignment graph, of this problem instance. The assignment graph is constructed as follows: If $x_1, x_2, \ldots, x_n$ are variables in the problem instance, then the nodes of the assignment graph are $x_1, x_2, \ldots, x_n$, and $-x_1, -x_2, \ldots, -x_n$. If there is a clause in the form of $L_1 \vee L_2$ where L_1 and L_2 are literals, then there are an edge from $-L_1$ to L_2 and an edge from $-L_2$ to L_1 in the assignment graph. Since every edge is related to one and only one clause, we shall label each edge with the corresponding clause.

Consider the following 2SAT problem:

$$-x_1 \quad \vee \quad x_2 \qquad \textbf{(1)}$$
$$x_2 \quad \vee \quad x_3. \qquad \textbf{(2)}$$

The assignment graph of the above 2SAT problem is now constructed, shown in Figure 8–30.

FIGURE 8–30 An assignment graph.



Consider clause (1) which is

$$-x_1 \quad \vee \quad x_2.$$

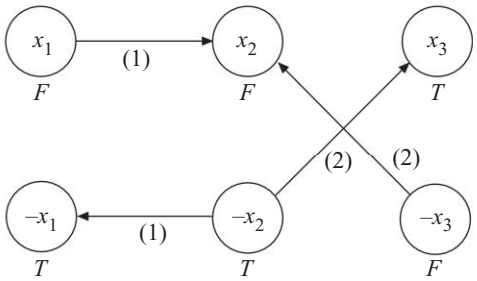
Suppose that we assign x_1T, then we must assign x_2T in order to make the clause T. This is indicated by the edge from x_1 to x_2 in the assignment graph. On the other hand, suppose that we assign $-x_2T$, we must assign $-x_1T$. Again, this is indicated by the edge from $-x_2$ to $-x_1$. This is why there are two edges for each clause. These two edges are symmetrical. That is, if there is an edge from node x to node y, then there will be an edge from $-y$ to $-x$. Each edge corresponds to a possible assignment which will satisfy this clause.

Based on the above reasoning, we shall have the following rules assigning truth values to the nodes of the assignment graph:

(1) If a node A is assigned T, then all of the nodes reachable from node A should be assigned T.
(2) If a node A is assigned T, then node $-A$ should be simultaneously assigned F.

For example, consider the graph Figure 8–30. If we assign $-x_2T$, we must assign both $-x_1$ and x_3T. Simultaneously, we must assign all other nodes F, illustrated in Figure 8–31. It is possible that there is more than one way of assigning truth values to an assignment graph to satisfy all clauses. For instance, for the graph in Figure 8–30, we may assign $-x_3$ and x_2T and x_1F. It can be easily seen that these two assignments, namely $(-x_1, -x_2, x_3)$ and $(-x_1, x_2, -x_3)$, satisfy the clauses.

FIGURE 8–31 Assigning truth values to an assignment graph.



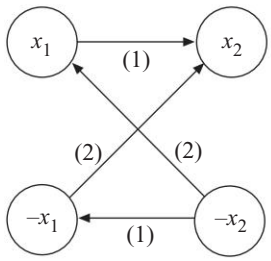
Since there is an edge incident to a node A if and only if variable A appears in a clause, *whenever node **A** is assigned **T**, all of the clauses corresponding to those edges incident to node **A** will be satisfied*. For instance, suppose that node x_2 in Figure 8–30 is assigned T. Since the edges incident to node x_2 correspond to clauses 1 and 2, both clauses 1 and 2 are now satisfied.

Let us consider another example. Suppose the clauses are

$$-x_1 \quad \vee \quad x_2 \qquad (1)$$
$$x_1 \quad \vee \quad x_2. \qquad (2)$$

The assignment graph corresponding to the above set of clauses is shown in Figure 8–32. In this case, we cannot assign $-x_2$ to T. If we do that, we shall have two problems:

FIGURE 8–32 An assignment graph.



(1) Both x_1 and $-x_1$ will be assigned T because both x_1 and $-x_1$ are reachable from $-x_2$. This is not allowed.
(2) Since $-x_2$ is assigned T, x_1 will be assigned T. This will finally cause x_2 to be assigned T. This again cannot be allowed because we cannot have both $-x_2$ and x_2 assigned T.

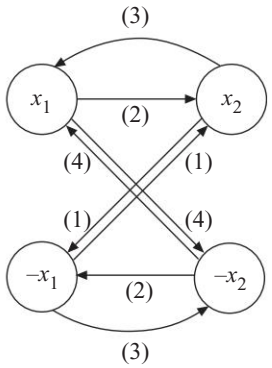
But we may assign x_2T. This assignment will satisfy both clauses 1 and 2 as indicated in Figure 8–32. As for x_1, it can be assigned either value. Where satisfiability is concerned, after assigning x_2T, the assignment of x_1 is no longer important.

Consider the following example:

$$x_1 \quad \vee \quad x_2 \qquad (1)$$
$$-x_1 \quad \vee \quad x_2 \qquad (2)$$
$$x_1 \quad \vee \quad -x_2 \qquad (3)$$
$$-x_1 \quad \vee \quad -x_2. \qquad (4)$$

The corresponding assignment graph is shown in Figure 8–33.

FIGURE 8–33 An assignment graph corresponding to an unsatisfiable set of clauses.



In this case, we can easily see that there is no way of assigning truth values to nodes in the graph. Every assignment will lead to a contradiction. Suppose we assign x_1T. Since there is a path from x_1 to $-x_1$, this will cause $-x_1$ also to be assigned T. If we assign $-x_1T$, this will force x_1 to be assigned T as there is also a path from $-x_1$ to x_1. Thus, we just cannot assign any truth value to x_1.

It is by no means accidental that there is no way to assign truth values to the nodes in this graph. We cannot do that because this set of clauses is unsatisfiable.

In general, *a set of 2SAT clauses is satisfiable if and only if its assignment graph does not simultaneously contain a path from x to $-x$ and a path from $-x$ to x, for some x*. The reader can easily see that there are such paths in the graph in Figure 8–33, and no such path in all other graphs in Figure 8–31 and Figure 8–32.

Suppose that there is a path from x to $-x$ and a path from $-x$ to x, for some x, in the assignment graph. It is obvious that we cannot assign any truth values to x and thus the input clauses must be unsatisfiable.

Suppose that in the assignment graph, for all x, no path from x to $-x$ and from $-x$ to x coexist. Then we shall show that the input set of clauses must be satisfiable. Our reasoning is that in such a situation, we can assign truth values to the nodes of the assignment graph successfully. In the following, we give an algorithm which can be used to find such an assignment.

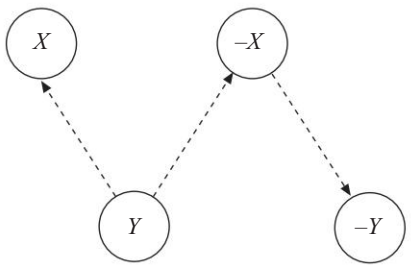
Step 1. Choose a node A in the assignment graph which has not been assigned any truth value and does not lead to $-A$.

Step 2. Assign truth value T to A and all nodes reachable from A. For every node x assigned T, assign F to node $-x$.

Step 3. If there is any node which has not been assigned yet, go to Step 1; otherwise, exit.

There is still one point which we must be careful about. In the above algorithm, will our assignment lead to a situation where both x and $-x$ are assigned T, for some x? This is impossible. If a node y leads to node x, then there must be an edge from $-x$ to $-y$ due to the symmetry of the assignment graph. Thus, y is such a node that will lead to $-y$, shown in Figure 8–34, and will not be chosen in Step 1.

FIGURE 8–34 The symmetry of edges in the assignment graph.



Because for all x, paths from x to $-x$ and from $-x$ to x do not coexist, there will always be a node for us to choose in our algorithm. In other words, our algorithm will always produce an assignment satisfying the set of input clauses. Thus, the set of clauses is satisfiable.

We have shown that to determine the satisfiability of a 2SAT problem instance, we only need to determine whether its corresponding assignment graph contains a path from x to $-x$ and a path from $-x$ to x, for some x. This is equivalent to determining whether there exists a cycle in the form of $x \rightarrow \cdots \rightarrow -x \rightarrow \cdots \rightarrow x$ in the assignment graph. Up to this point, we have not discussed how to design the algorithm. It will become very easy later for readers to design a polynomial algorithm to determine whether such a cycle exists in the assignment graph.

In conclusion, the 2SAT problem is a P problem.

8–9 NOTES AND REFERENCES

The concept of NP-completeness is perhaps one of the most difficult concepts to understand in computer science. The most important paper concerning with this

concept was written by Cook in 1971 which showed the significance of the satisfiability problem. In 1972, Karp proved many combinatorial problems are NP-complete (Karp, 1972). These two papers have been considered landmark papers on this subject. Since then, numerous problems have been proved to be NP-complete. There are so many of them that a database is maintained by David Johnson of AT&T. Johnson also writes regularly for *Journal of Algorithms* on NP-completeness. His articles are entitled "The NP-completeness Column: An Ongoing Guide". These articles are very important and interesting for computer scientists interested in algorithms.

The best book totally devoted to NP-completeness is Garey and Johnson (1979). It gives the history of the development of NP-completeness and can always be used as an encyclopedia of it. In this book, we did not give a formal proof of Cook's theorem. It was a painful decision for us. We found out that we could give a formal proof. However, it would be very hard for the reader to grasp its true meaning. Instead we gave examples to illustrate how a non-deterministic algorithm can be transformed to a Boolean formula such that the non-deterministic algorithm terminates with "yes" if and only if the transformed Boolean formula is satisfiable. Formal proofs of Cook's theorem can be found in Cook's original paper (Cook, 1971). One may also consult Horowitz and Sahni (1978); Garey and Johnson (1979) and Mandrioli and Ghezzi (1987).

Note that NP-completeness is in worst case analysis only. That a problem is NP-complete should not discourage anyone from developing efficient algorithms to solve it in average cases. For instance, there are many algorithms developed to solve the satisfiability problem. Many of them are based on the resolution principle, discussed in this chapter. The resolution principle was invented by Robinson (Robinson, 1965) and discussed in Chang and Lee (1973). Recently, several algorithms for the satisfiability problem have been found to be polynomial for average cases (Chao, 1985; Chao and Franco, 1986; Franco, 1984; Hu, Tang and Lee, 1992; Purdom and Brown, 1985).

Another famous NP-complete problem is the traveling salesperson problem whose NP-completeness can be established by reducing the Hamiltonian cycle problem to it (Karp, 1972). There is a whole book devoted to algorithms for the traveling salesperson problem (Lawler, Lenstra, Rinnooy Kan and Shmoys, 1985). The art gallery problem for simple polygons is also NP-complete (Lee and Lin, 1986). There is also a book totally devoted to theorems and algorithms on this problem (O'Rourke, 1987).

The 3-satisfiability problem was found to be NP-complete in (Cook, 1971). The NP-completeness of the chromatic number problem was established by

Karp (1972). For the NP-completeness of sum of subsets, partition and bin packing problems, consult Horowitz and Sahni (1978). The VLSI discrete layout problem was proved to be NP-complete by Lapaugh (1980).

The 2-satisfiability problem was proved to be a polynomial time problem by Cook (1971) and Even, Itai and Shamir (1976). This problem is also discussed in Papadimitiriou (1994).

There are still several famous problems which have been suspected to be NP-complete. Garey and Johnson (1978) gave a list of them. One of these problems, namely the linear programming problem, was later found to be a polynomial problem by Khachian (1979). Consult Papadimitriou and Steiglitz (1982).

Finally, there is another concept, called average complete. If a problem is average complete, then it is difficult even in average cases. The tiling problem (Berger, 1966) was proved to be average complete by Levin (1986). Levin (1986) is probably the shortest paper ever published, only one and a half pages long. But this is one of the hardest papers to be digested. If you fail to understand it, do not be upset. A large number of computer scientists do not understand it.

8–10 FURTHER READING MATERIALS

NP-completeness is one of the most exciting research topics. Many seemingly easy problems have been discovered to be NP-complete. We encourage the reader to read Johnson's articles in *Journal of Algorithms* for progress in this area. The following papers are recommended by us; most of them were published recently and cannot be found in any textbook: Ahuja (1988); Bodlaender (1988); Boppana, Hastad and Zachos (1987); Cai and Meyer (1987); Chang and Du (1988); Chin and Ntafos (1988); Chung and Krishnamoorthy (1994); Colbourn, Kocay and Stinson (1986); Du and Book (1989); Du and Leung (1988); Fellows and Langston (1988); Flajolet and Prodiuger (1986); Friesen and Langston (1986); Homer (1986); Homer and Long (1987); Huynh (1986); Johnson, Yanakakis and Papadimitriou (1988); Kirousis and Papadimitriou (1988); Krivanek and Moravek (1986); Levin (1986); Megido and Supowit (1984); Monien and Sudborough (1988); Murgolo (1987); Papadimitriou (1981); Ramanan, Deogun and Lin (1984); Wu and Sahni (1988); Sarrafzadeh (1987); Tang, Buehrer and Lee (1985); Valiant and Vazirani (1986); Wang and Kuo (1988) and Yannakakis (1989).

For newer results, consult Agrawal, Kayal and Saxena (2004); Berger and Leighton (1998); Bodlaender, Fellows and Hallet (1994); Boros, Crama, Hammer and Saks (1994); Caprara (1997a); Caprara (1997b); Caprara (1999); Feder and Motwani (2002); Ferreira, de Souza and Wakabayashi (2002); Foulds and

Graham (1982): Galil, Haber and Yung (1989); Goldberg, Goldberg and Paterson (2001); Goldstein and Waterman (1987); Grove (1995); Hochbaum (1997); Holyer (1981); Kannan and Warnow (1994); Kearney, Hayward and Meijer (1997); Lathrop (1994); Lent and Mahmoud (1996); Lipton (1995); Lyngso and Pedersen (2000); Ma, Li and Zhang (2000); Maier and Storer (1977); Pe'er and Shamir (1998); Pierce and Winfree (2002); Storer (1977); Thomassen (1997); Unger and Moult (1993) and Wareham (1995).

Exercises

8.1 Determine whether the following statements are correct or not

(1) If a problem is NP-complete, then it cannot be solved by any polynomial algorithm in worst cases.

(2) If a problem is NP-complete, then we have not found any polynomial algorithm to solve it in worst cases.

(3) If a problem is NP-complete, then it is unlikely that a polynomial algorithm can be found in the future to solve it in worst cases.

(4) If a problem is NP-complete, then it is unlikely that we can find a polynomial algorithm to solve it in average cases.

(5) If we can prove that the lower bound of an NP-complete problem is exponential, then we have proved that NP $\neq$ P.

8.2 Determine the satisfiability of each of the following sets of clauses.

(1)
$$\begin{array}{lllll} -X_1 & \vee & -X_2 & \vee & X_3 \\ X_1 & & & & \\ X_2 & \vee & X_3 & & \\ -X_3 & & & & \end{array}$$

(2)
$$\begin{array}{lllll} X_1 & \vee & X_2 & \vee & X_3 \\ -X_1 & \vee & X_2 & \vee & X_3 \\ X_1 & \vee & -X_2 & \vee & X_3 \\ X_1 & \vee & X_2 & \vee & -X_3 \\ -X_1 & \vee & -X_2 & \vee & X_3 \end{array}$$

$$
\begin{array}{lccccc}
 & X_1 & \vee & -X_2 & \vee & -X_3 \\
 & -X_1 & \vee & X_2 & \vee & -X_3 \\
 & -X_1 & \vee & -X_2 & \vee & -X_3 \\
(3) & -X_1 & \vee & X_2 & \vee & X_3 \\
 & X_1 & \vee & X_2 & & \\
 & X_3 & & & & \\
(4) & X_1 & \vee & X_2 & \vee & X_3 \\
 & X_1 & & & & \\
 & X_2 & & & & \\
(5) & -X_1 & \vee & X_2 & & \\
 & -X_2 & \vee & X_3 & & \\
 & -X_3 & & & &
\end{array}
$$

8.3 We all know how to prove that a problem is NP-complete. How can we prove that a problem is not NP-complete?

8.4 Complete the proof of the NP-completeness of the exact cover problem as described in this chapter.

8.5 Complete the proof of the NP-completeness of the sum of subset problem as described in this chapter.

8.6 Consider the following problem. Given two input variables a and b, return "**YES**" if $a > b$ and "**NO**" if otherwise. Design a nondeterministic polynomial algorithm to solve this problem. Transform it into a Boolean formula such that the algorithm returns "**YES**" if and only if the transformed Boolean formula is satisfiable.

8.7 Maximal clique decision problem: A maximal clique is a maximal complete subgraph of a graph. The size of a maximal clique is the number of vertices in it. The clique decision problem is to determine whether there is a maximal clique at least size k for some k in a graph or not. Show that the maximal clique decision problem is NP-complete by reducing the satisfiability problem to it.

8.8 Vertex cover decision problem: A set S of vertices of a graph is a vertex cover of this graph if and only if all edges of the graph are incident to at least one vertex in S. The vertex cover decision problem is to determine whether a graph has a vertex cover having at worst k vertices. Show that the vertex cover decision problem is NP-complete.

8.9 Traveling salesperson decision problem: Show that the traveling salesperson decision problem is NP-complete by proving that the Hamiltonian cycle decision problem reduces polynomially to it. The definition of Hamiltonian cycle decision problem can be found in almost any textbook on algorithms.

8.10 Independent set decision problem: Given a graph G and an integer k, the independent set decision problem is to determine whether there exists a set S of k vertices such that no two vertices in S are connected by an edge. Show that the independent set problem is NP-complete.

8.11 Bottleneck traveling salesperson decision problem: Given a graph and a number M, the bottleneck traveling salesperson decision problem is to determine whether there exists a Hamiltonian cycle in this graph such that the longest edge of this cycle is less than M. Show that the bottleneck traveling salesperson decision problem is NP-complete.

8.12 Show that 3-coloring $\propto$ 4-coloring $\propto$ k-coloring.

8.13 Clause-monotone satisfiability problem: A formula is monotone if each clause of it contains either only positive variables or only negative variables. For instance

$$F = (X_1 \vee X_2) \,\&\, (-X_3) \,\&\, (-X_2 \vee -X_3)$$

is a monotone formula. Show that the problem of deciding whether a monotone formula is satisfiable or not is NP-complete.

8.14 Read Theorem 15.7 of Papadimitriou and Steiglitz (1982) for the NP-completeness of the 3-dimensional matching problem.

chapter

9

Approximation Algorithms

In previous chapters, every algorithm which we introduced will give us optimal solutions. We obviously have to pay a price for this: the time consumed in producing those optimal solutions may be very long. If the problem is an NP-complete problem, our algorithms producing optimal solutions could be unrealistically time consuming.

One compromise is to use heuristic solutions. The word "heuristic" may be interpreted as "educated guess". Thus, a heuristic algorithm will be an algorithm based on educated guesses. Because a heuristic algorithm guesses, there is no guarantee that it will produce optimal solutions. In fact, usually it does not.

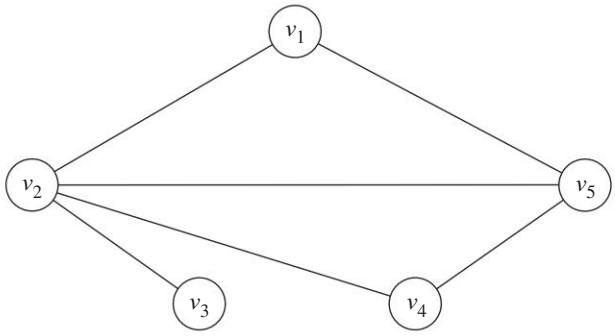
In this chapter, we shall introduce several heuristic algorithms. While these heuristic algorithms cannot guarantee optimal solutions, they do guarantee that the errors caused by their non-optimal solutions can be estimated beforehand. All these algorithms are called approximation algorithms.

9–1 An Approximation Algorithm for the Node Cover Problem

Given a graph $G = (V, E)$, a set S of nodes in V is called a node cover of G if every edge is incident to some node in S. The size of a node cover S is the number of nodes in S. The node cover problem is to find a node cover S of $G = (V, E)$ with the minimum size. For example, $\{v_1, v_2, v_4\}$ is a node cover of the graph shown in Figure 9–1, and $\{v_2, v_5\}$ is a node cover of minimum size.

The node cover problem was shown to be NP-hard. We introduce an approximation algorithm, Algorithm 9–1, for this problem.

FIGURE 9–1 An example graph.



Algorithm 9–1 □ An approximation algorithm for the node cover problem

Input: A graph $G = (V, E)$.
Output: A node cover S of G.
Step 1. Let $S = \phi$ and $E' = E$.
Step 2. While $E' \neq \phi$
Pick an arbitrary edge (a, b) in E'.
Let $S = S \bigcup \{a, b\}$ and E'' be the set of edges in E' incident to a or b. Let $E' = E' - E''$.
end while
Step 3. Output S.

Let us consider the graph G shown in Figure 9–1. In Step 1 of Algorithm 9–1, $S = \phi$ and $E' = \{(v_1, v_2), (v_1, v_5), (v_2, v_3), (v_2, v_4), (v_2, v_5), (v_4, v_5)\}$. Since $E' \neq \phi$, we perform the loop in Step 2. Suppose that we pick edge (v_2, v_3) from E'. Then we have $S = \{v_2, v_3\}$ and $E' = \{(v_1, v_5), (v_4, v_5)\}$ in the end of the loop. Next, we perform the second loop in Step 2 because $E' \neq \phi$. Suppose that we pick edge (v_1, v_5) from E'. So we have $S = \{v_1, v_2, v_3, v_5\}$ and $E' = \phi$ in the end of the second loop. Finally, we report $\{v_1, v_2, v_3, v_5\}$ as a node cover for G.

The time complexity of the above algorithm is $O(|E|)$. We are going to show that the size of a node cover of G found by Algorithm 9–1 is at most two times the minimum size of a node cover of G. Let M^* denote the minimum size of a node cover of G. Let M denote the size of a node cover of G found by Algorithm 9–1. Let L denote the total number of edges picked in the algorithm. Since one

edge is picked in each loop of Step 2 and there are two vertices incident by an edge, we have $M = 2L$. Since no two edges picked in Step 2 share any same end vertex, we need at least L nodes to cover these L edges. Then we have $L \leq M^*$. Finally, $M = 2L \leq 2M^*$.

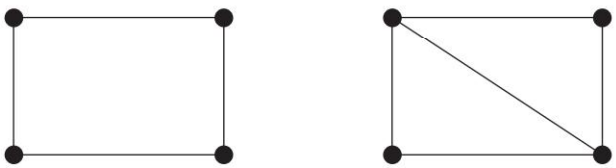
9–2 AN APPROXIMATION ALGORITHM FOR THE EUCLIDEAN TRAVELING SALESPERSON PROBLEM

The Euclidean traveling salesperson problem (ETSP) is to find a shortest closed path through a set S of n points in the plane. This was proved to be NP-hard. Hence, it is very unlikely that there exists an efficient worst case Euclidean traveling salesperson problem algorithm.

In this section, we shall describe an approximation algorithm for the Euclidean traveling salesperson problem which can find a tour within 3/2 of the optimal one. That is, if the optimal tour has tour length L, then this approximate tour length is not greater than $(3/2)L$.

The basic idea of this approximation scheme is to construct an Eulerian cycle of the set of points and then use it to find an approximate tour. *An Eulerian cycle of a graph is a cycle in the graph in which every vertex is visited not necessarily once, and each edge appears exactly once. A graph is called Eulerian if it has an Eulerian cycle. Euler proved that a graph G is Eulerian if and only if G is connected and all vertices in G have even degrees*. Consider Figure 9–2. It can be easily seen that the graph in Figure 9–2(a) is Eulerian and that in Figure 9–2(b) is not. We show later that after an Eulerian cycle is found, we can find a Hamiltonian cycle from it simply by traversing it and bypassing previously visited vertices.

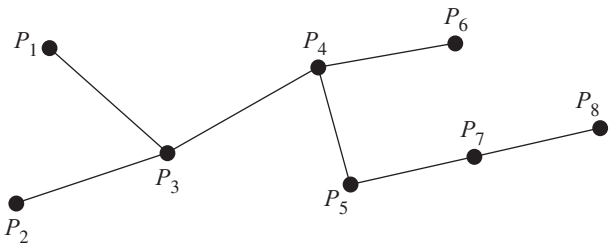
FIGURE 9–2 Graphs with and without Eulerian cycles.



Thus, if we can construct a connected graph G of the point set S (i.e. use the points in S as the vertices of G) such that the degree of each vertex of G is even, then we can construct an Eulerian cycle and then find a Hamiltonian cycle to approximate the optimal traveling salesperson tour.

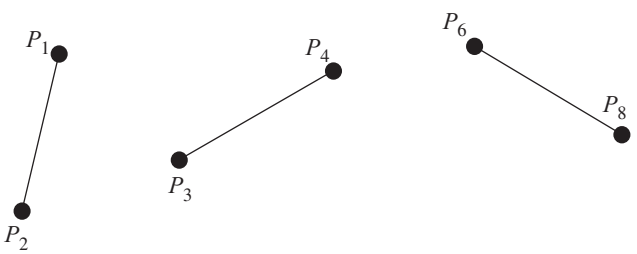
The approximation scheme consists of three steps: First, we construct a minimum spanning tree of S. For example, consider Figure 9–3. For point set $S_1 = \{P_1, P_2, \ldots, P_8\}$ with eight points, a minimum spanning tree of S_1 is shown in Figure 9–3.

FIGURE 9–3 A minimum spanning tree of eight points.



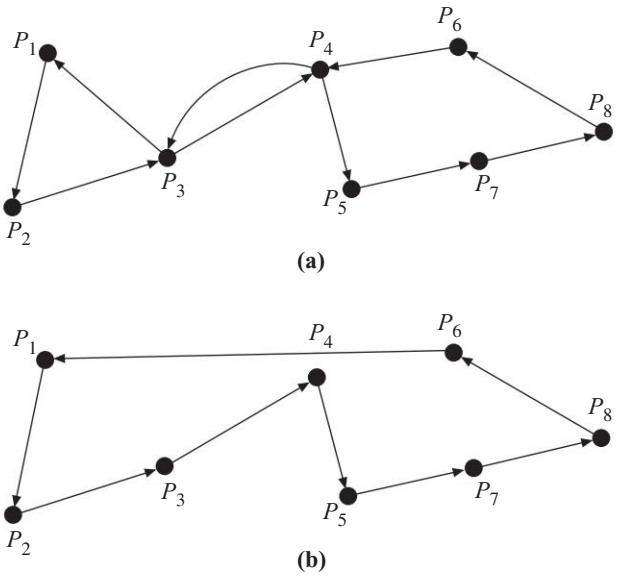
Although the minimum spanning tree is connected, it may contain odd-degree vertices. In fact, it can be shown that in any tree, there is always an even number of such vertices. For instance, the degree of each vertex in $\{P_1, P_2, P_3, P_4, P_6, P_8\}$ in Figure 9–3 is odd. The second step is to construct an Eulerian graph; that is, we must add some more edges to make all the vertices into even degrees. The problem becomes how to find this set of edges? In this algorithm, we make use of the minimum Euclidean weighted matching for the set of points with odd degrees. *Given a set of points in the plane, the minimum Euclidean weighted matching problem is to join the points in pairs by line segments such that the total length is minimum*. If we can find a minimum Euclidean weighted matching on the vertices with odd degrees and augment this set of matching edges to the edge set of the minimum spanning tree, then the resulting graph is an Eulerian graph. For example, the minimum Euclidean weighted matching for vertices P_1, P_2, P_3, P_4, P_6 and P_8 is illustrated in Figure 9–4.

FIGURE 9–4 A minimum weighted matching of six vertices.



After we add the edges in Figure 9–4 to the minimum spanning tree in Figure 9–3, all of the vertices are of even degrees. The third step in the algorithm is to construct an Eulerian cycle out of the resulting graph and then obtain a Hamiltonian cycle from this Eulerian cycle. The Eulerian cycle of the set of eight points is P_1–P_2–P_3–P_4–P_5–P_7–P_8–P_6–P_4–P_3–P_1 (note that P_3 and P_4 are visited twice) and is shown in Figure 9–5(a). By bypassing P_4 and P_3 and connecting P_6 directly to P_1, we obtain a Hamiltonian cycle. Hence, the resulting approximate tour is P_1–P_2–P_3–P_4–P_5–P_7–P_8–P_6–P_1 and is shown in Figure 9–5(b).

FIGURE 9–5 An Eulerian cycle and the resulting approximate tour.



Algorithm 9–2 □ An approximation algorithm for ETSP

Input: A set S of n points in the plane.
Output: An approximate traveling salesperson tour of S.
Step 1. Find a minimum spanning tree T of S.
Step 2. Find a minimum Euclidean weighted matching M on the set of vertices of odd degrees in T. Let $G = M \bigcup T$.
Step 3. Find an Eulerian cycle of G and then traverse it to find a Hamiltonian cycle as an approximate tour of ETSP by bypassing all previously visited vertices.

The analysis of the time complexity of this approximation algorithm is easy. Step 1 can be executed in $O(n \log n)$ time. The minimum Euclidean weighted matching can be found in $O(n^3)$ time, the Eulerian cycle can be constructed in linear time, and a Hamiltonian cycle can be obtained from it in linear time afterwards too. Hence, the time complexity of the above algorithm is $O(n^3)$. Since the main object of this discussion is to show approximation algorithms, we shall not give details of algorithms for finding minimum matching and Eulerian cycle.

Let L be an optimal traveling salesperson tour of S. If we remove one edge from L, we obtain a path L_p, which is also a spanning tree of S. Let T denote a minimum spanning tree of S. Then, length(T) ≤ length(L_p) ≤ length(L) because T is a minimum spanning tree of S. Let $\{i_1, i_2, \ldots, i_{2m}\}$ be the set of odd-degree vertices in T where indices of vertices in the set are arranged in the same order as they are in the optimal traveling salesperson tour L. Consider the two matchings of the odd-degree vertices $M_1 = \{[i_1, i_2], [i_3, i_4], \ldots, [i_{2m-1}, i_{2m}]\}$ and $M_2 = \{[i_2, i_3], [i_4, i_5], \ldots, [i_{2m}, i_1]\}$. It is easy to see that length(L) ≥ length(M_1) + length(M_2) since the edges satisfy triangular inequality. Let M be the optimal Euclidean weighted matching of $\{i_1, i_2, \ldots, i_{2m}\}$. Thus, the length of the shorter one of these two matchings is longer than length(M). Thus, length(M) ≤ length(L)/2. Since $G = T \bigcup M$, length(G) = length(T) + length(M) ≤ length(L) + length(L)/2 ≤ 3/2(length(L)).

Since our solution, which is a Hamiltonian cycle, is shorter than the length of G, we have an approximation algorithm for the Euclidean traveling salesperson problem which can produce an approximate tour within 3/2 of an optimal one in time $O(n^3)$.

9–3 AN APPROXIMATION ALGORITHM FOR A SPECIAL BOTTLENECK TRAVELING SALESPERSON PROBLEM

The traveling salesperson problem can also be defined on graphs: In such a case we are given a graph and we are to find a shortest closed tour, which starts from some vertex, visits all other vertices in the graph and returns to the starting vertex. This problem was proved to be NP-hard.

In this section, we consider another type of traveling salesperson problem. We are still interested in a closed tour. However, we do not want the length of the entire tour to be minimized; instead, we want the longest edge of this tour to be minimized. In other words, we want to find a tour whose longest edge is the shortest. This is thus a mini-max problem and we shall call this kind of traveling

salesperson problem a bottleneck traveling salesperson problem because it belongs to a class of bottleneck problems, often related to problems in transportation science. In fact, one can see that many problems which we discussed before can be modified to become bottleneck problems. For instance, the minimum spanning tree problem has a bottleneck version. In this version, we shall find a spanning tree whose longest edge is the shortest.

Again, as in the previous two sections, we shall give an approximation algorithm. Before giving this algorithm, let us state our assumption as follows:

(1) *Our graph is a complete graph. That is, there is an edge between every pair of vertices*.
(2) *All of the edges obey the triangular inequality rule*. Thus, our bottleneck traveling salesperson problem is a special one. It can be proved that this special bottleneck traveling salesperson problem is NP-hard.

To prepare the reader for our approximation algorithm, we shall first propose an algorithm which is not polynomial. This algorithm will produce an optimal solution for the bottleneck traveling salesperson problem. We then modify this algorithm to an approximation algorithm.

Let us first sort the edges of the graph $G = (V, E)$ into a non-decreasing sequence $|e_1| \leq |e_2| \ldots \leq |e_m|$. Let $G(e_i)$ denote the graph obtained from $G = (V, E)$ by deleting all edges longer than e_i. Consider $G(e_1)$. If $G(e_1)$ contains a Hamiltonian cycle, then this Hamiltonian cycle must be our desired optimal solution. Otherwise, consider $G(e_2)$. We repeat this process until some $G(e_i)$ contains a Hamiltonian cycle. Then return this Hamiltonian cycle as our optimal solution to the special bottleneck traveling salesperson problem.

We can use an example to illustrate the above idea. Consider Figure 9–6. To find a Hamiltonian cycle with minimum longest edge, we may try $G(AC)$ to begin with. The length of AC is 12. Therefore, $G(AC)$ contains only edges smaller than or equal to 12, shown in Figure 9–7. There is no Hamiltonian cycle in $G(AC)$. We then try $G(BD)$ because BD is the next longest edge. $G(BD)$ is shown in Figure 9–8. In $G(BD)$, there is a Hamiltonian cycle, namely A–B–D–C–E–F–G–A. We may now conclude that an optimal solution is found and the value of this optimal solution is 13.

Our algorithm cannot be a polynomial one because it is an NP-hard problem to find a Hamiltonian cycle. Next, we shall present an approximation algorithm. This approximation algorithm still has the spirit of the above algorithm. Before

FIGURE 9–6 A complete graph.

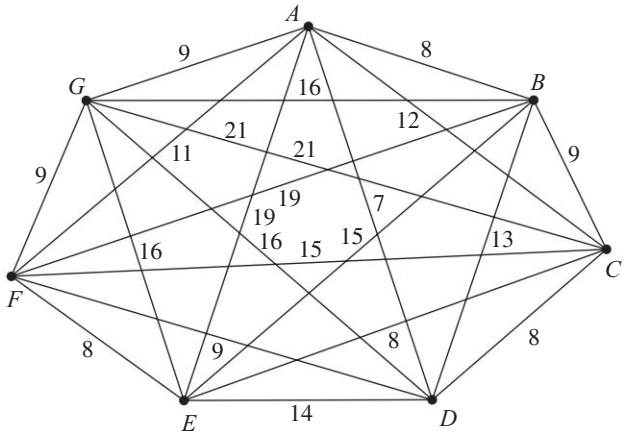
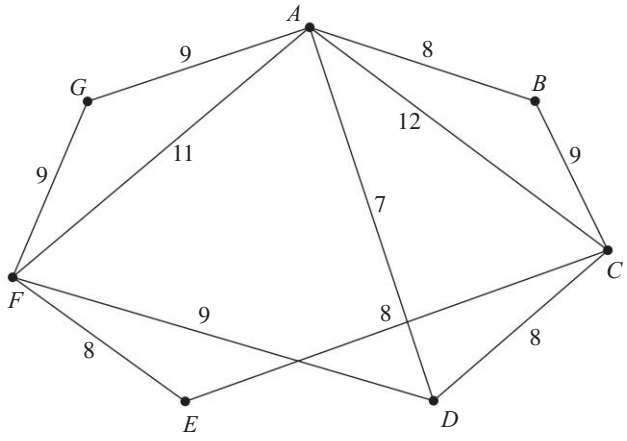


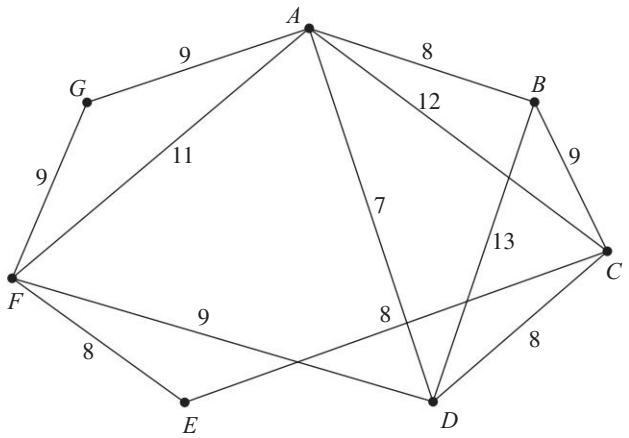
FIGURE 9–7 $G(AC)$ of the graph in Figure 9–6.



introducing this approximation algorithm, we shall first introduce the concept of biconnectedness. *A graph is biconnected if and only if every pair of its vertices belong to at least one common simple cycle.*

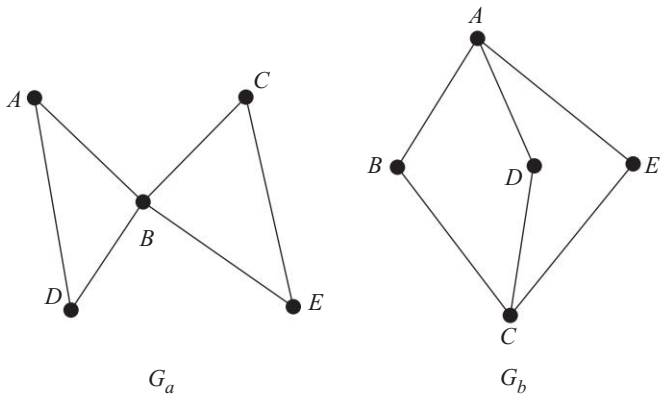
For example, in Figure 9–9, G_b is biconnected while G_a is not. G_a is not because vertices A and C are not in any simple cycle. On the other hand, in G_b, every pair of vertices belong to at least one common simple cycle. Next, we shall define the power of a graph. *If $G = (V, E)$ is an arbitrary graph and t is a positive*

FIGURE 9–8 $G(BD)$ of the graph in Figure 9–6.



integer, let the t-th power of G be $G^t = (V, E^t)$*, where there is an edge* (u, v) *in* G^t *whenever there is a path from u to v with at most t edges in G.*

FIGURE 9–9 Examples illustrating biconnectedness.



Biconnected graphs have a very important property. That is, *if a graph G is biconnected, then* G^2 *has a Hamiltonian cycle*. Consider Figure 9–10. Obviously, the graph G in Figure 9–10 is biconnected. For example, vertices D and F are on the simple cycle D–C–F–E–D. Let us now consider G^2 as shown in Figure 9–11. There is a Hamiltonian cycle in the graph in Figure 9–11, which is A–B–C–D–F–E–G–A.

FIGURE 9–10 A biconnected graph.

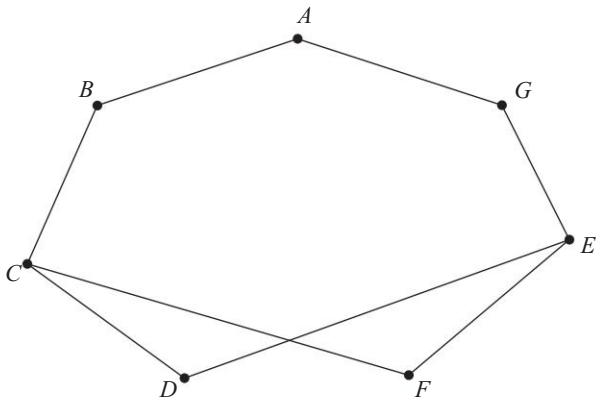
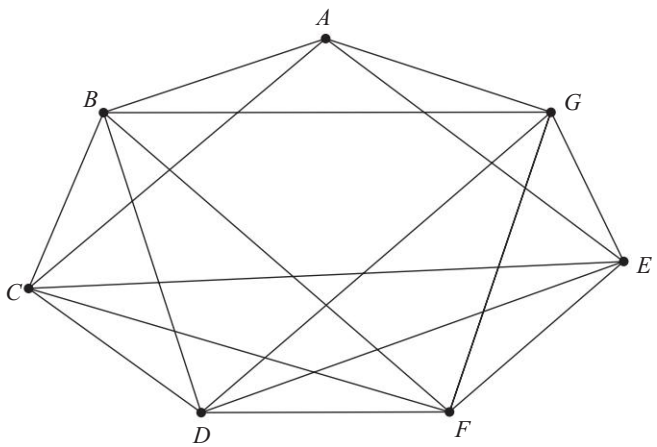


FIGURE 9–11 G^2 of the graph in Figure 9–10.



Based on the property that for each biconnected graph G, G^2 has a Hamiltonian cycle, we can use the following approach to give an approximate solution to the bottleneck traveling salesperson problem. *As we did before, we sort the edges of G into a non-decreasing sequence:* $|e_1| \le |e_2| \ldots \le |e_m|$. *Let* $G(e_i)$ *contain only the edges whose lengths are smaller than or equal to the length of* e_i. *We consider* $G(e_1)$ *to start with. If* $G(e_1)$ *is biconnected, construct* $G(e_1)^2$. *Otherwise, consider* $G(e_2)$. *We repeat this process until we hit the first i where* $G(e_i)$ *is biconnected. Then construct* $G(e_i)^2$. *This graph must contain a Hamiltonian cycle and we can use this Hamiltonian cycle as our approximate solution.*

The approximation algorithm is formally stated as follows:

Algorithm 9–3 □ An approximation algorithm to solve the special bottleneck traveling salesperson problem

Input: A complete graph $G = (V, E)$ where all edges satisfy the triangular inequality.

Output: A tour in G whose longest edge is not greater than twice of the value of an optimal solution to the special bottleneck traveling salesperson problem of G.

Step 1. Sort the edges into $|e_1| \leq |e_2| \ldots \leq |e_m|$.

Step 2. $i := 1$.

Step 3. If $G(e_i)$ is biconnected, construct $G(e_i)^2$, find a Hamiltonian cycle in $G(e_i)^2$ and return this as the output; otherwise go to Step 4.

Step 4. $i := i + 1$. Go to Step 3.

We now illustrate this algorithm by an example. Consider Figure 9–12, which shows $G(FE)$ ($|FE| = 8$) where G is the graph in Figure 9–6. $G(FE)$ is not biconnected. We now consider $G(FG)$ because $|FG| = 9$. $G(FG)$ is biconnected, shown in Figure 9–13. We then construct $G(FG)^2$ which is shown in Figure 9–14. There is a Hamiltonian cycle in this graph which is *A–G–E–F–D–C–B–A*. The longest edge of this cycle has length 16. We must note that this is not an optimal solution as the value of an optimal solution for the bottleneck salesperson problem of the graph in Figure 9–6 is 13, which is less than 16.

FIGURE 9–12 $G(FE)$ of the graph in Figure 9–6.

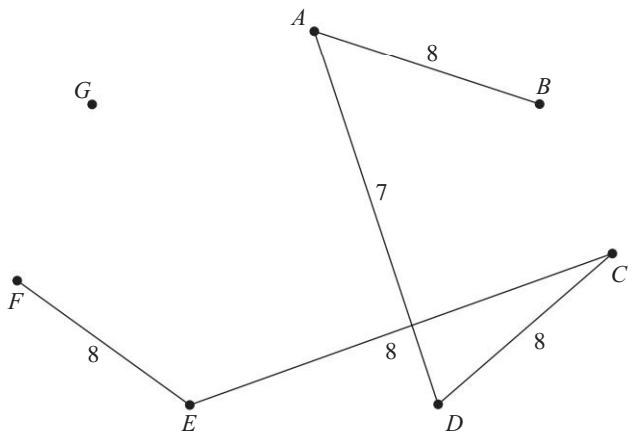


FIGURE 9–13 $G(FG)$ of the graph in Figure 9–6.

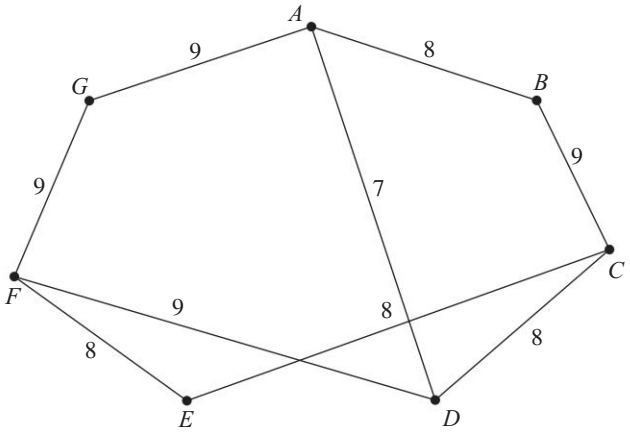
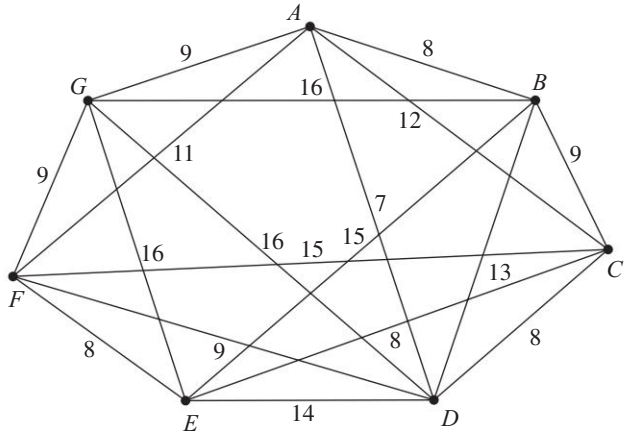


FIGURE 9–14 $G(FG)^2$.



Next, we discuss three questions related to the approximation algorithm:

(1) What is the time complexity of our approximation algorithm?
(2) What is the bound of the approximate solution?
(3) Is this bound optimal in a certain sense?

Let us answer the first question. We note that the determination of biconnectedness can be solved by polynomial algorithms. Furthermore, if G is bi-

connected, then there exists a polynomial algorithm to solve the Hamiltonian cycle problem of G^2. Thus, our approximation algorithm is polynomial.

We now try to answer the second question: What is the bound of the approximate solution? Consider the following problem: An edge subgraph of $G = (V, E)$ is a subgraph of G with vertex set V. Among all edge subgraphs of G, find an edge subgraph which is biconnected and the longest edge is minimized. Obviously, the first $G(e_i)$ found in the approximation algorithm mentioned above which is biconnected must be a solution of this problem. Let e_{op} denote the longest edge of an optimal solution of the special bottleneck traveling salesperson problem. Obviously, $|e_i| \leq |e_{op}|$ because any traveling salesperson solution is biconnected. After $G(e_i)$ is found, $G(e_i)^2$ is constructed. The length of the longest edge of $G(e_i)^2$ cannot be greater than $2|e_i|$ because all edges obey the triangular inequality. *The value of the solution produced by the approximation algorithm is therefore not greater than* $2|e_i| \leq 2|e_{op}|$. *That is, although our approximation algorithm does not produce an optimal solution, the value of its solution is still bounded by two times that of an optimal solution.*

Let us check our claim against the example which we gave. In the above example, $|e_{op}| = 13$. The solution produced by the approximation algorithm is 16. Since $16 \leq 2 \cdot 13 = 26$, our claim is correct in this case.

Finally, we answer the third question: Is the bound of the approximate solution optimal? That is, can we have another polynomial approximation algorithm which will produce an approximate solution whose bound is less than two? For instance, is it possible that we can have a polynomial approximation algorithm which produces an approximate solution whose longest edge is less than or equal to 1.5 times that of an optimal solution for all problem instances?

We shall now show that this is probably impossible because if there is such an approximation algorithm, which is polynomial, then we can use this algorithm to solve an NP-complete problem. That is, if there is a polynomial approximation algorithm which produces a bound less than two, then NP = P. The particular NP-complete problem which we shall use is the Hamiltonian cycle decision problem.

Let us assume that there exists a polynomial algorithm A for the special bottleneck traveling salesperson problem such that A is guaranteed to produce a solution less than $2v^*$ for every complete graph G_c whose edges satisfy triangular inequality where v^* is the value of an optimal solution of the special bottleneck traveling salesperson problem of G_c. For an arbitrary graph $G = (V, E)$, we can transform G to a complete graph G_c and define the edge values of G_c as

$c_{ij} = 1$ if $(i, j) \in E$

$c_{ij} = 2$ if otherwise.

Clearly, the above definition of c_{ij} satisfies the triangular inequality. We now try to solve the bottleneck traveling salesperson problem of G_c. Obviously

$v^* = 1$ if G is Hamiltonian

and $v^* = 2$ if otherwise.

Equivalently, $v^* = 1$ if and only if G is Hamiltonian.

In other words, if we can solve the bottleneck traveling salesperson problem of G_c, we can solve the Hamiltonian cycle decision problem of G. We simply test the value of v^*. G is Hamiltonian if and only if $v^* = 1$. Next, we shall show that we can use the approximation algorithm A to solve the Hamiltonian cycle problem, in spite of the fact that A is only an approximation algorithm of the bottleneck traveling salesperson problem.

By assumption, A can produce an approximate solution whose value is less than $2v^*$. Consider the case where G is Hamiltonian. Then the approximation algorithm A produces an approximate solution whose value V_A is less than $2v^* = 2$.

That is, *if G is Hamiltonian, then $V_A = 1$ because there are only two kinds of edge weights*, 1 *and* 2. *We therefore claim that Algorithm A can be used to solve the Hamiltonian cycle decision problem of G. If A is polynomial, then NP = P because the Hamiltonian cycle decision problem is NP-complete.*

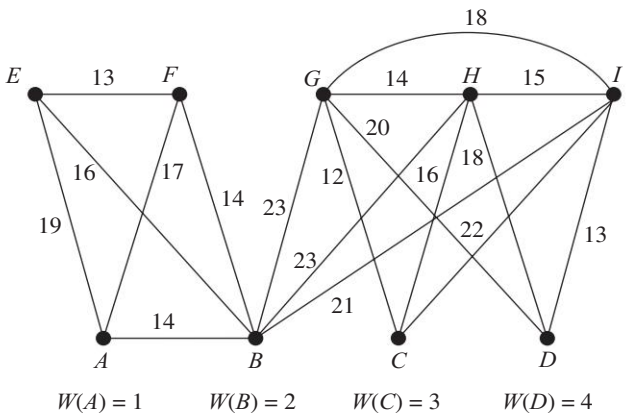
The above argument indicates that it is quite unlikely that such an algorithm exists.

9–4 AN APPROXIMATION ALGORITHM FOR A SPECIAL BOTTLENECK WEIGHTED *K*-SUPPLIER PROBLEM

A special bottleneck weighted k-supplier problem is another example for our approximation algorithm. In this problem, we are given a complete graph whose edges all satisfy the triangular inequality. The weight of an edge can be considered as a distance. All the vertices are separated into a supplier set V_{sup} and a customer set V_{cust}. Furthermore, each supplier vertex i has a weight w_i which indicates the cost to build a supply center at this vertex. Our problem is to choose a set of suppliers of total weight at most k, such that the longest distance between suppliers and customers is minimized.

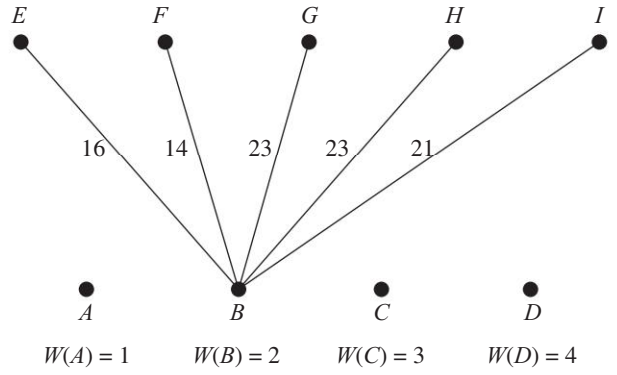
Let us consider Figure 9–15, which has nine vertices, $V = \{A, B, C, \ldots, I\}$, $V_{sup} = \{A, B, C, D\}$ and $V_{cust} = \{E, F, G, H, I\}$. The costs of suppliers A, B, C and D are 1, 2, 3 and 4 respectively. Note that the graph shown in Figure 9–15 should be a complete graph. All edges missing in Figure 9–15 are with costs greater than the maximum cost shown in Figure 9–15 but all edge costs still satisfy the triangular inequality.

FIGURE 9–15 A weighted k-supplier problem instance.



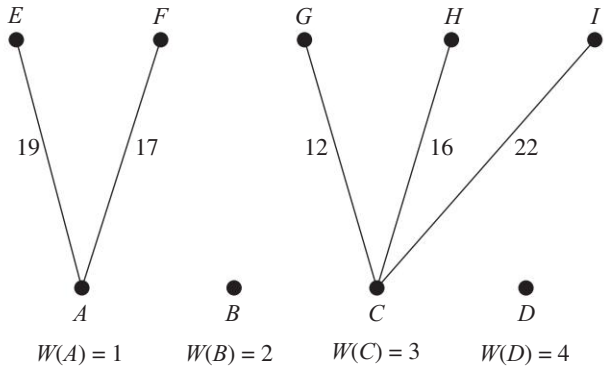
Assume that $k = 5$. Figure 9–16 shows several feasible solutions. The optimum solution is shown in Figure 9–16(d) and the value of this solution is 20.

FIGURE 9–16 Feasible solutions for the problem instance in Figure 9–15.

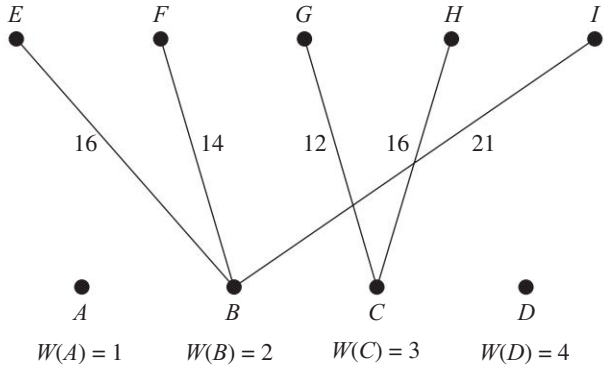


(a) Maximum = 23

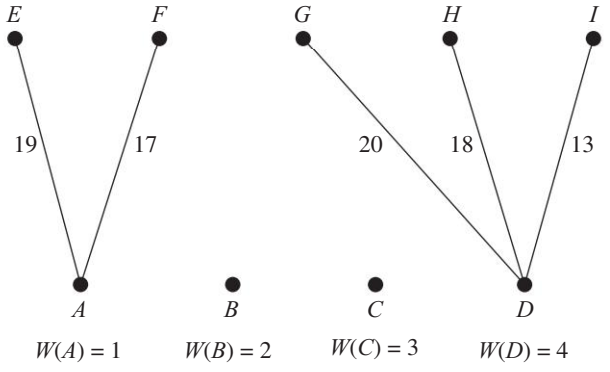
FIGURE 9–16 (*cont'd*)



(b) Maximum = 22



(c) Maximum = 21



(d) Maximum = 20

Our approximation algorithm is again similar to the approximation algorithm proposed in Section 9–3, which is used to solve the special bottleneck traveling salesperson problem. *To solve this special bottleneck weighted k-supplier problem, we sort the edges into a non-decreasing sequence:* $|e_1| \le |e_2| \ldots \le |e_m|$. *As we did before, let* $G(e_i)$ *denote the graph obtained from* $G = (V, E)$ *by deleting all edges which are longer than* e_i. *Ideally, our algorithm would test* $G(e_1)$, $G(e_2)$, … *and so on until we encounter the first* $G(e_i)$ *such that* $G(e_i)$ *contains a feasible solution for the weighted k-supplier problem. This feasible solution must be an optimal one. Unfortunately, as we can easily imagine, to determine whether a graph contains a feasible solution for the weighted k-supplier problem is NP-complete*. We consequently must modify this approach to obtain an approximation algorithm.

In our approximation algorithm, we shall employ a testing procedure. This testing procedure satisfies the following properties:

(1) If the testing returns "YES", then we output an induced solution as our solution for the approximation algorithm.
(2) If the testing returns "NO", then we are assured that $G(e_i)$ does not contain any feasible solution.

If the testing procedure does satisfy the above conditions, and the edges of the graph satisfy the triangular inequality relation, then we can easily show that our approximation algorithm will produce an approximate solution whose value is less than or equal to three times the value of an optimal solution.

Let V_{op} and V_{ap} denote the values of optimal solutions and approximation solutions respectively. Because of Property (2), $|e_i| \le V_{op}$.

Furthermore, because of Property (1) and reasons which will be explained later,

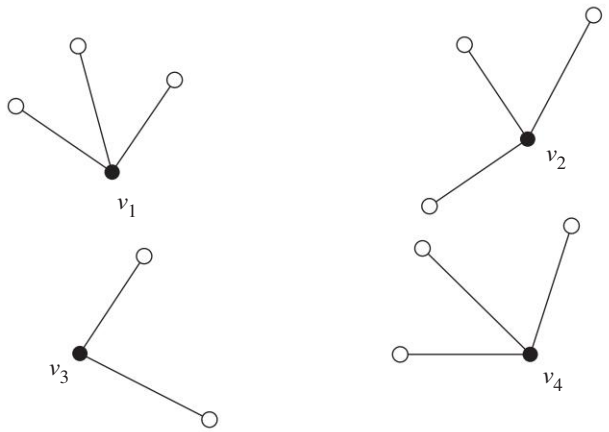
$$V_{ap} \le 3|e_i|.$$

Therefore, we have

$$V_{ap} \le 3V_{op}.$$

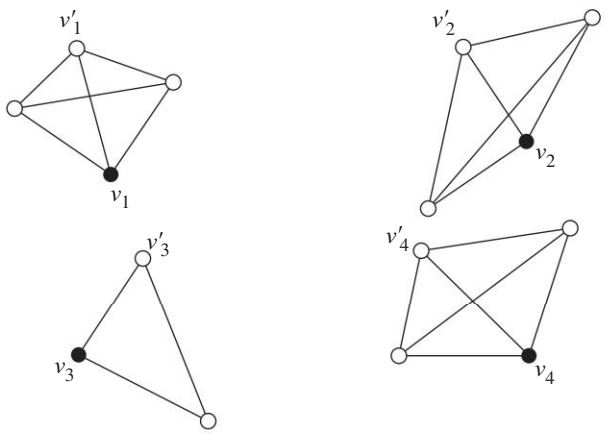
We now explain how the testing works. Imagine any $G(e_i)$ containing a feasible solution. Then every customer vertex must be connected to at least one supplier vertex, illustrated in Figure 9–17, where v_1, v_2, v_3 and v_4 are supplier vertices and all other vertices are customer vertices. For each v_i, there are several customer vertices connected to it. Consider $G(e_i)^2$ of the graph $G(e_i)$ in Figure 9–17.

FIGURE 9–17 A $G(e_i)$ containing a feasible solution.



$G(e_i)^2$ is shown in Figure 9–18. Since $G(e_i)$ contains a feasible solution and each customer vertex is connected to at least one supplier vertex, $G(e_i)^2$ will contain several cliques. In a clique, every pair of vertices are connected.

FIGURE 9–18 G^2 of the graph in Figure 9–17.



We may select a customer vertex from each clique. The result will be a maximal independent set. *An independent set of a graph $G = (V, E)$ is a subset $V' \subset V$ such that each edge e of E is incident on at most one vertex in V'. A maximal independent set of a graph is an independent set which is not properly*

contained in any independent set. As shown in Figure 9–18, $\{v'_1, v'_2, v'_3, v'_4\}$ is a maximal independent set. After a maximal independent set is found, for each customer vertex v'_i, find a supplier vertex connected to it with minimum weight. This reversely created supplier set can be proved to be a feasible solution in $G(e_i)^3$.

Our testing procedure (on $G(e_i)$ and $G(e_i)^2$) consists of the following steps:

(1) If there is one vertex v in V_{cust} which is not adjacent to any supplier vertex in $G(e_i)$, return "NO".
(2) Otherwise, find a maximal independent set S in V_{cust} of $G(e_i)^2$.
(3) For every vertex v' in S, find an adjacent supplier vertex v_i with minimum weight in $G(e_i)$. Let S' be the set of these vertices.
(4) If the total weight of $w(v_i)$ in S' is less than or equal to k, return "YES". Otherwise, return "NO".

If the testing result is "NO", obviously $G(e_i)$ cannot contain any feasible solution. If the testing result is "YES", we can use the induced supplier set as the solution of our approximation algorithm. Thus, our approximation algorithm can be stated as follows in Algorithm 9–4.

Algorithm 9–4 □ An approximation algorithm for solving the special bottleneck *k*-supplier problem

Input: A complete graph $G = (V, E)$, where $V = V_{cust} \bigcup V_{sup}$ and $V_{cust} \bigcap V_{sup} = \phi$. There is a weight $w(v_i)$ associated with each v_i in V_{sup}. All of the edges of G satisfy the triangular inequality.

Output: An approximate solution for the special bottleneck k-supplier problem. The value of this approximate solution is not greater than three times the value of an optimal solution.

Step 1. Sort the edges in E such that $|e_1| \leq |e_2| \ldots \leq |e_m|$ where $m = \binom{n}{2}$.

Step 2. $i := 0$.

Step 3. $i := i + 1$.

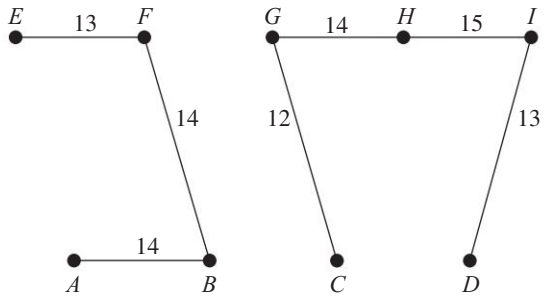
Step 4. If there is some vertex (which is a customer vertex) in V_{cust} that is not adjacent to any supplier vertex in $G(e_i)$, go to Step 3.

Step 5. Find a maximal independent set S in V_{cust} of $G(e_i)^2$.
For every vertex v'_j in S, find a supplier vertex v_j which is a neighbor of v'_j in $G(e_i)$ with minimum weight. Let $S' = \bigcup_j v_j$.

Step 6. If $\Sigma w(v_j) > k$, where $v_j \in S'$, then go to Step 3.
Step 7. Output S' as the supplier set.
Step 8. For every vertex v_j in V_{cust}, let DIST(j) denote the shortest distance between v_j and S'. Let $V_{ap} = \max_j(\text{DIST}(j))$. Output V_{ap}.

We illustrate our algorithm by an example. Consider the graph in Figure 9–15. Figure 9–19 shows $G(HI)$ of the graph in Figure 9–15. Since H is not adjacent to any supplier in $G(HI)$, the answer to the testing is "NO".

FIGURE 9–19 $G(H)$ of the graph in Figure 9–15.



In Figure 9–20, we show $G(HC)$ as HC is the next shortest edge. Figure 9–21 shows $G(HC)^2$. Suppose in V_{cust} of $G(HC)^2$, we select $S = \{E, G\}$. Then $S' = \{B, C\}$. $W(B) + W(C) = 5 \leq k = 5$. Hence, $S' = \{B, C\}$ is our solution. From $S' = \{B, C\}$, we shall find a nearest neighbor for each customer vertex. The result is shown in Figure 9–22. As shown in Figure 9–22, $DIST(E) = 16$, $DIST(F) = 14$, $DIST(G) = 12$, $DIST(H) = 16$ and $DIST(I) = 21$. Therefore, $V_{ap} = 21$.

FIGURE 9–20 $G(HC)$ of the graph in Figure 9–15.

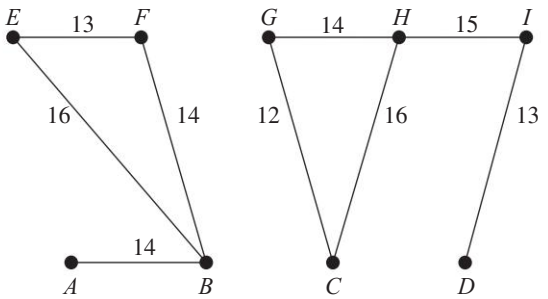


FIGURE 9–21 $G(HC)^2$.

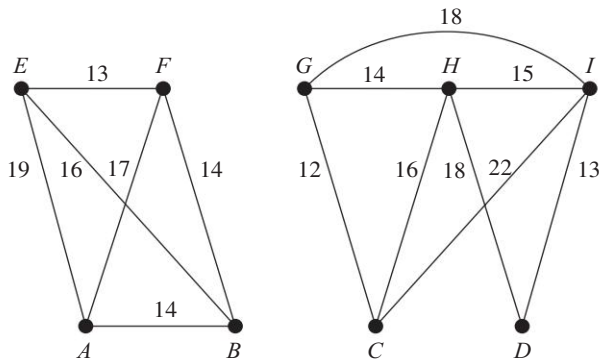
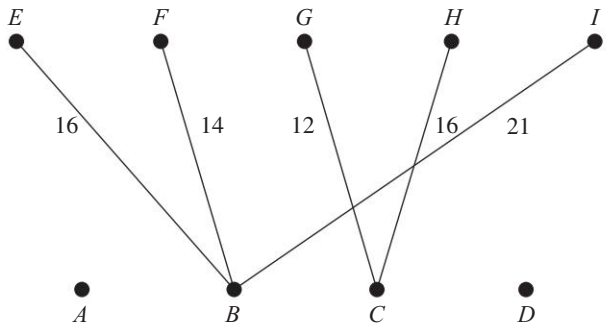


FIGURE 9–22 An induced solution obtained from $G(HC)^2$.



As we did before, we must answer the following three questions:

(1) What is the time complexity of the approximation algorithm?
(2) What is the bound of the approximation algorithm?
(3) Is this bound optimal?

To answer the first question, we simply state that the approximation algorithm is dominated by the steps needed to find a maximal independent set. This is a polynomial algorithm. Thus, our approximation algorithm is a polynomial algorithm.

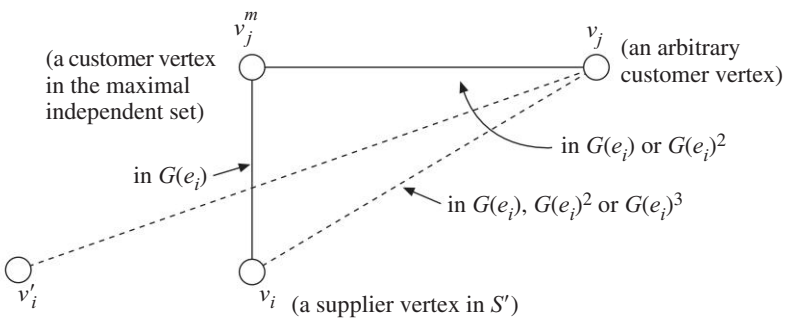
We now answer the second question. What is the bound of the value of the solution obtained by using our approximation algorithm? Let e_i denote the particular e_i for which our approximation algorithm outputs a solution. Let V_{op} denote the value of an optimal solution for the special bottleneck k-supplier

problem. It is easy to see that $|e_i| \le V_{op}$. We further note that V_{ap}, the value of our approximate solution, is obtained in Step 8 of the approximation algorithm. For every customer vertex v_j in S we find a nearest neighbor v_i in the original supplier vertex set. These nearest neighbors form S'. We must note that the edge linking v_j and its nearest neighbor, which is in S', is not necessarily an edge in $G(e_i)^2$. For example, consider Figure 9–22. BI is not an edge in $G(HC)^2$. Let us suppose that we selected C, instead of B, as the supplier vertex for I. We should note that CI may not be an edge of $G(HC)^2$ either. But, $V_{ap} = |BI|$ must be smaller or equal to $|CI|$. That is, $|CI|$ is an upper bound of $|BI|$. Therefore, it is important to investigate which $G(HC)^d$ CI belongs to. In the following, we shall show that CI belongs to $G(HC)$, $G(HC)^2$ or $G(HC)^3$.

The reasoning goes as follows: For each customer v_j which is not a member of the maximal independent set which we selected in our algorithm, let v_j^m be the vertex of the maximal independent set which is directly linked to v_j. Note that v_j^m must exist; otherwise, the maximal independent set will not be a maximal independent set. v_j^m must be a customer vertex and directly linked to a supplier vertex v_i in S', determined in Step 5 of the approximation algorithm, illustrated in Figure 9–23. Now, edge (v_j, v_j^m) must be either in $G(e_i)$ or in $G(e_i)^2$. Furthermore, (v_j^m, v_i) must be in $G(e_i)$ as stipulated in Step 5. Thus, edge (v_i, v_j) must be in $G(e_i)$, $G(e_i)^2$ or $G(e_i)^3$. In other words, edge (v_i, v_j) is in $G(e_i)^d$ if $d \ge 3$. At the end of the approximation algorithm, v_j may be linked to some v_i' which is not v_i. (v_i', v_j) must be shorter than (v_i, v_j); otherwise v_i' would not have been selected. Thus,

$$V_{ap} = |(v_i', v_j)| \le |(v_i, v_j)| \le 3|e_i| \le 3V_{op}.$$

FIGURE 9–23 An illustration explaining the bound of the approximation algorithm for the special bottleneck k-supplier problem.



Our final question is concerned with the optimality of this bounding factor 3. We shall show that it is quite unlikely that we can lower this bounding factor. Because if we can, then NP = P. Essentially, we shall prove that if there exists a polynomial approximation algorithm D which can produce an approximation solution of the special bottleneck k-supplier problem whose value is guaranteed to be smaller than $a \cdot V_{op}$ where $a < 3$ and V_{op} is the value of the optimal solution, then P = NP.

To prove this, we shall make use of the hitting set problem which is NP-complete. We can show that if the above approximation algorithm D exists, we can use algorithm D to solve the hitting set problem. This implies that P = NP.

The hitting set problem is defined as follows: We are given a finite set S and a collection C of subsets of S and a positive integer k. The question is: Does there exist a subset $S' \subseteq S$ where $|S'| \leq k$ and $S' \cap S_i \neq \phi$ for any S_i in C?

For example, let

$$S = \{1, 2, 3, 4, 5\}$$
$$C = \{\{1, 2, 3\}, \{4, 5\}, \{1, 2, 4\}, \{3, 5\}, \{4, 5\}, \{3, 4\}\}$$

and $k = 3$.

Then $S' = \{1, 3, 4\}$ has the property that $|S'| = 3 \leq k$ and $S' \cap S_i \neq \phi$ for each S_i in C.

For illustrative purposes, we shall call a hitting set problem a k-hitting set problem to emphasize the parameter k.

Given an instance of the k-hitting set problem, we can transform this instance into a special bottleneck k-supplier problem instance. Suppose $S = \{a_1, a_2, \ldots, a_n\}$. Then let $V_{sup} = \{v_1, v_2, \ldots, v_n\}$ where each v_i corresponds to a_i. Suppose $C = \{S_1, S_2, \ldots, S_m\}$. Then let $V_{cust} = \{v_{n+1}, v_{n+2}, \ldots, v_{n+m}\}$ where v_{n+i} corresponds to S_i. Define the weight of each supplier vertex to be 1. The distance between v_i and v_j is defined as follows:

$$\begin{aligned} d_{ij} &= 2 \text{ if } \{v_i, v_j\} \subseteq V_{sup} \text{ or } \{v_i, v_j\} \subseteq V_{cust} \\ &= 3 \text{ if } v_i \in V_{sup}, v_j \in V_{cust} \text{ and } a_i \notin S_j \\ &= 1 \text{ if } v_i \in V_{sup}, v_j \in V_{cust} \text{ and } a_i \in S_j. \end{aligned}$$

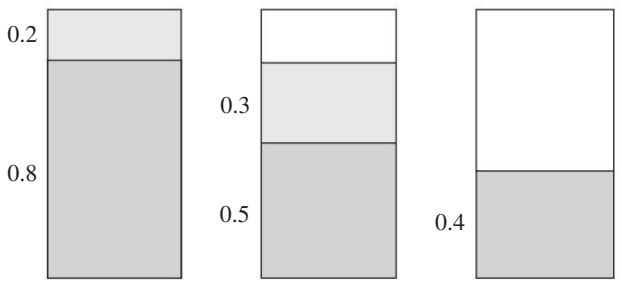
It is straightforward to prove that these distances satisfy the triangular inequality, and *the k-supplier problem has a solution of cost 1 if and only if there exists a k-hitting set S'*. Furthermore, if there is no k-hitting set, the cost of an

optimal k-supplier must be 3. Therefore, if an approximation algorithm guarantees an approximate solution V_{ap}, where $V_{ap} < 3V_{op}$, then $V_{ap} < 3$ if and only if $V_{op} = 1$ and there is a k-hitting set. That is, we can use this polynomial approximation algorithm to solve the k-hitting set problem which is NP-complete. This will imply that NP = P.

9–5 AN APPROXIMATION ALGORITHM FOR THE BIN PACKING PROBLEM

Given n items in the list $L = \{a_i \mid 1 \le i \le n$ and $0 < a_i \le 1\}$ which have to be placed in bins of unit capability, the bin packing problem is to determine the minimum number of bins to accommodate all n items. If we regard the items of different sizes to be the lengths of time of executing different jobs on a standard processor, then the problem becomes one of using the minimum number of processors which can finish all of the jobs within a fixed time. For example, let $L = \{0.3, 0.5, 0.8, 0.2, 0.4\}$. We need at least three bins to pack these five items, shown in Figure 9–24.

FIGURE 9–24 An example of the bin packing problem.



Since we can easily transform the partition problem to this problem, the bin packing problem is NP-hard. Next, we shall introduce an approximation algorithm for the bin packing problem.

The approximation algorithm is called a first-fit algorithm. Let the bins needed have indices 1, 2, … , m, and let B_1 be the first bin of unit capability. The first-fit algorithm places the items into the bins, one at a time, in order of increasing index. To pack item a_i, the first-fit algorithm always places it into the lowest-indexed bin for which the sum of the sizes of the items already in that bin does not exceed one minus the size of a_i.

The first-fit algorithm is very natural and easy to implement. The problem remaining is how this algorithm approximates an optimal solution of the bin packing problem.

Denote the size of an item a_i as $S(a_i)$. Since each bin has capability one, the size of an optimal solution of a problem instance I, denoted as $OPT(I)$, is greater than or equal to the ceiling of the summation of sizes of all the items. That is,

$$OPT(I) \geq \left\lceil \sum_{i=1}^{n} S(a_i) \right\rceil.$$

Let $FF(I)$ be the number of bins needed in the first-fit algorithm. $FF(I) = m$. An upper bound of $FF(I)$ can be derived as follows: Take any bin B_i and let $C(B_i)$ be the sum of the sizes of a_j's packed in bin B_i. Then we have

$$C(B_i) + C(B_{i+1}) > 1;$$

otherwise the items in B_{i+1} would be put in B_i. Summing up all the m non-empty bins, we have

$$C(B_1) + C(B_2) + \cdots + C(B_m) > m/2.$$

This means that

$$FF(I) = m < 2\left\lceil \sum_{i=1}^{m} C(B_i) \right\rceil = 2\left\lceil \sum_{i=1}^{n} S(a_i) \right\rceil.$$

We thus conclude that

$$FF(I) < 2OPT(I).$$

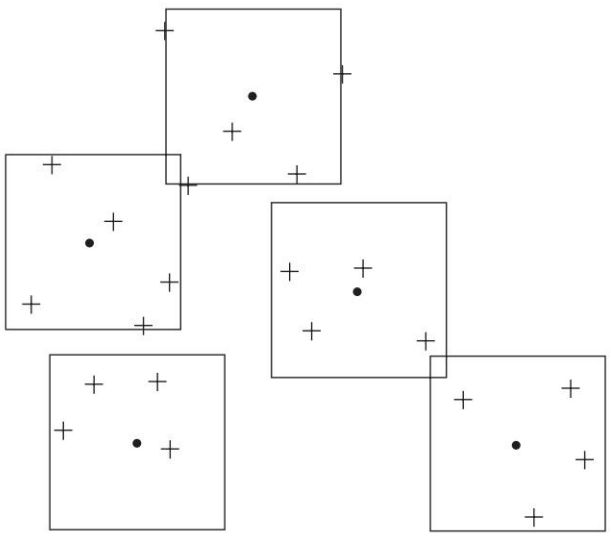
9-6 AN OPTIMAL APPROXIMATION ALGORITHM FOR THE RECTILINEAR *m*-CENTER PROBLEM

Given a set of n points on the plane, the rectilinear m-center problem is to find m rectilinear squares covering all these n points such that the maximum side length of these squares is minimized. By a rectilinear square, we mean a square with sides perpendicular or parallel to the x-axis of the Euclidean plane. Since we can enlarge smaller rectilinear squares in a solution for the rectilinear m-center

problem to the size of the maximum one without affecting our objective, we may assume that all m rectilinear squares in a solution are of the same side length. The side length is called the size of the solution.

In Figure 9–25, we are given an instance of the rectilinear 5-center problem. The points marked with "+" are to be covered by five rectilinear squares. The five rectilinear squares, shown in this figure, form an optimal solution.

FIGURE 9–25 A rectilinear 5-center problem instance.



The rectilinear m-center problem was proved to be NP-hard. Furthermore, it was proved that any polynomial time approximation algorithm solving the problem has error ratio ≥ 2, unless NP = P. In this section, we shall present an approximate solution with error ratio exactly equal to 2.

Note that for any optimal solution to the rectilinear m-center problem on $P = \{p_1, p_2, \ldots, p_n\}$, the size of the solution must be equal to one of the rectilinear distances among input points. That is, the size of an optimal solution must be equal to one of the $L_\infty(p_i, p_j)$'s, $1 \leq i < j \leq n$, where $L_\infty((x_1, y_1), (x_2, y_2)) = \max\{|x_1 - x_2|, |y_1 - y_2|\}$. Let us say that a number r is feasible if there exist m squares with side length r covering all the n points. A set of m squares covering all the input points is called a feasible solution. One straightforward method to solve the rectilinear m-center problem is as follows:

(1) Calculate all possible $L_\infty(p_i, p_j)$'s, $1 \le i < j \le n$.
(2) Sort the above found distances. Let them be denoted as $D[1] \le D[2] \le \cdots \le D[n(n-1)/2]$.
(3) Conduct a binary search on $D[i]$'s, $i = 1, 2, \ldots, n(n-1)/2$, to find the smallest index i_0, such that $D[i_0]$ is feasible, and a feasible solution of size $D[i_0]$. For each distance $D[i]$ being searched, we have to test whether $D[i]$ is feasible.

There are only $O(\log n)$ distances to be tested. However, for each $D[i]$, the testing of whether there exist m squares with side length $D[i]$ covering all n points is still an NP-complete problem. Therefore, as expected, the above straightforward algorithm is bound to be exponential, at least for the time being.

In this section, we shall show a "relaxed" test subroutine which will be used in our approximation algorithm. Given a set of n points on the plane, and two numbers m and r as input parameters, this relaxed test subroutine generates m squares with side length $2r$, instead of r. It then tests whether the m squares cover these n points. If it fails, it will return "failure". If it succeeds, it gives a feasible solution of size $2r$. We shall see later that "failure" guarantees that no m squares with side length r can cover these n points.

We shall give the details of this subroutine later. Let this relaxed subroutine be labeled as Test(m, P, r). With this relaxed test subroutine, our approximation algorithm is one which replaces the test subroutine in the straightforward optimal algorithm above with the relaxed one.

Algorithm 9–5 □ An approximation algorithm for the rectilinear *m*-center problem

Input: A set P of n points, number of centers: m.

Output: $SQ[1], \ldots, SQ[m]$: A feasible solution of the rectilinear m-center problem with size less than or equal to twice the size of an optimal solution.

Step 1. Compute rectilinear distances of all pairs of two points and sort them together with 0 into an ascending sequence $D[0] = 0$, $D[1], \ldots, D[n(n-1)/2]$.

Step 2. $LEFT := 1$, $RIGHT := n(n-1)/2$.

Step 3. $i := \lceil (RIGHT + LEFT)/2 \rceil$.

Step 4. If Test(m, P, $D[i]$) is not "failure" then
$RIGHT := i$
else
$LEFT := i$.
Step 5. If $RIGHT = LEFT + 1$ then
return Test(m, P, $D[RIGHT]$)
else
go to Step 3.

The relaxed test subroutine Test(m, P, r) is now described as follows:

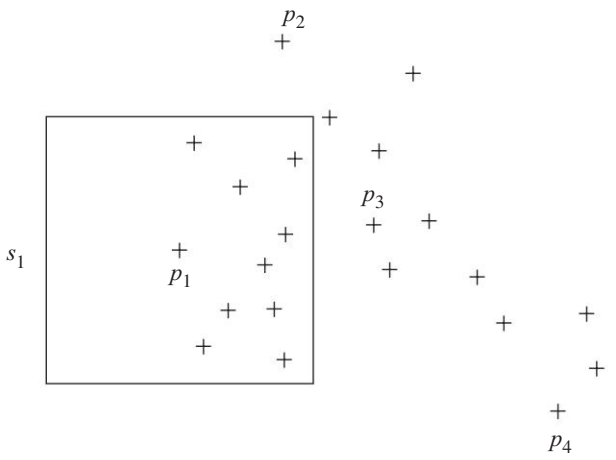
(1) Find the point with the smallest x-value in the remaining points which have not been covered yet and draw a square with center at this point with side length $2r$.
(2) Add the square to the solution and remove all of the points already covered by this square.
(3) Repeat m times Steps 1 and 2. If we succeed in finding m squares to cover all the input points in this way, we return these m squares as a feasible solution; otherwise, we return "failure".

Algorithm 9–6 □ Algorithm test(m, P, r)

Input: Point set: P, number of centers: m, size: r.
Output: "failure", or $SQ[1], \ldots, SQ[m]$ m squares of size $2r$ covering P.
Step 1. $PS := P$
Step 2. For $i := 1$ to m do
If $PS \neq \phi$ then
$p :=$ the point in PS with the smallest x-value
$SQ[i] :=$ the square of size $2r$ with center at p
$PS := PS - \{$points covered by $SQ[i]\}$
else
$SQ[i] := SQ[i-1]$.
Step 3. If $PS = \phi$ then
return $SQ[1], \ldots, SQ[m]$
else
return "failure".

Consider the instance of a rectilinear 5-center problem in Figure 9–25 again. Let r be the side length of squares in Figure 9–25. Applying the relaxed test subroutine to the instance with r, we first find that p_1 is the point with the smallest x-value, and then draw a square S_1 with center at p_1 of side length $2r$, shown in Figure 9–26. We then remove the points covered by square S_1. Among the remaining points, we find that point p_2 is the point with the smallest x-value. Again, we draw a square S_2 with center at p_2 of side length $2r$, shown in Figure 9–27, and remove the points covered by S_2. The above process is repeated five times. Finally, we have a feasible solution, shown in Figure 9–28. In this instance, four squares are enough.

FIGURE 9–26 The first application of the relaxed test subroutine.



Next, we shall prove that our approximation algorithm is of error ratio 2. The key statement is: *If r is feasible, then our relaxed test subroutine* Test(m, P, r) *always returns a feasible solution of size $2r$*. Assume that the above statement is true. Let r^* be the size of an optimal solution. Since r^* is feasible, by the claimed statement, the subroutine Test(m, P, r^*) returns a feasible solution. Because our approximation algorithm will terminate at the least r such that Test(m, P, r) returns a feasible solution of size $2r$, we have $r \le r^*$. Therefore, the feasible solution found is of size $2r$, which is less than or equal to $2r^*$. In other words, the error ratio is 2.

Next, we prove the claimed statement. Let $S_1, S_2, \ldots, S_m$ be a feasible solution of size r and $S'_1, \ldots, S'_m$ be the m squares with side length $2r$ generated

FIGURE 9–27 The second application of the test subroutine.

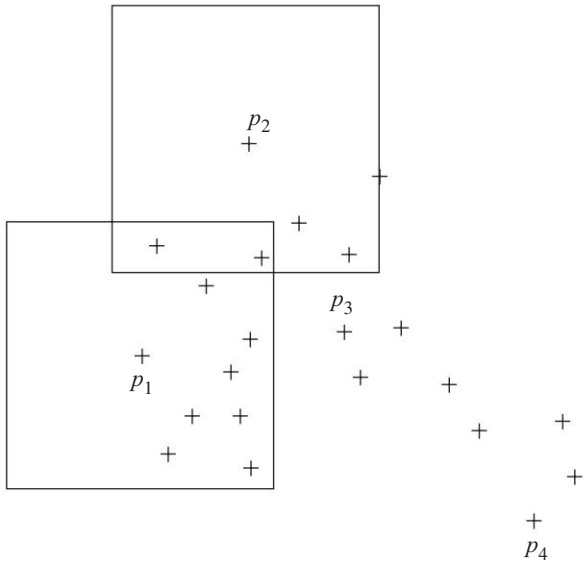
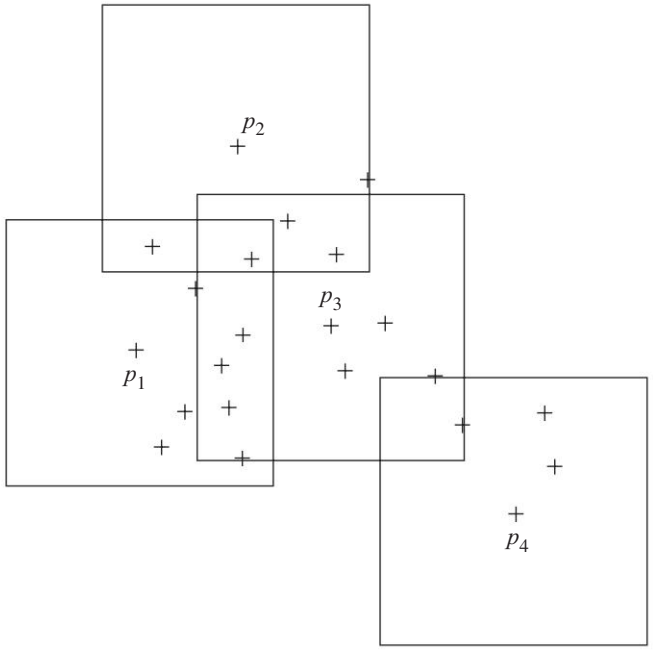


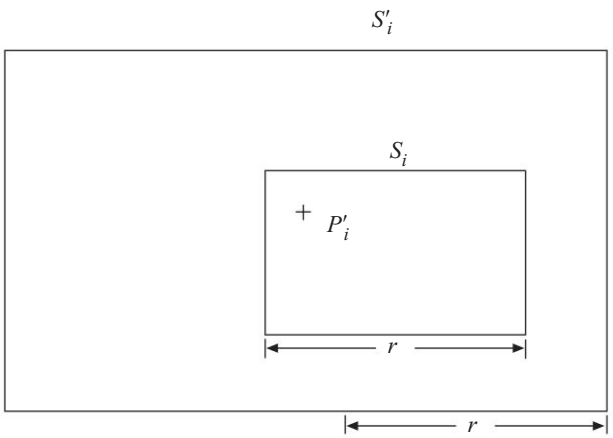
FIGURE 9–28 A feasible solution of the rectilinear 5-center problem.



in the subroutine Test(m, P, r). In case that $S'_1, S'_2, \ldots, S'_m$ are not distinct, $PS = \phi$ in Step 2 of the subroutine Test. In other words, in this case, $S'_1, S'_2, \ldots, S'_m$ always cover all points and form a feasible solution. Next, consider the case that $S'_1, S'_2, \ldots, S'_m$ are distinct. Let the center of S'_i be p'_i for $i = 1, 2, \ldots, m$. That is, any point in S'_i is of rectilinear distance less than or equal to r from p'_i. By the way, we choose S'_i's, p'_j is not covered by $S'_1, S'_2, \ldots, S'_{j-1}$. Thus, $L_\infty(p'_i, p'_j) > r$ for all $i \neq j$. Since the side length of S_i is r for $i = 1, \ldots, m$, the rectilinear distance of any two points in S_i is less than or equal to r. Thus, any S_j at most contains one p'_i, $i = 1, \ldots, m$. On the other hand, since $S_1, S_2, \ldots, S_m$ form a feasible solution, the union of $S_1, S_2, \ldots, S_m$ contains $p'_1, \ldots, p'_m$. Therefore, p'_i's belong to distinct S_j's; that is $p'_i \in S_i$. For $1 \leq i \leq m$, since the size of S_i is r, $L_\infty(p, p'_i) \leq r$ for all $p \in S_i$. Thus, $S_i \subset S'_i$ (Figure 9–29). Therefore, $S'_1, \ldots, S'_m$ contain all the points. We conclude our claimed statement. Because of the above statement, we may also have the following equivalent statement:

If Test(m, P, r) does not return a feasible solution, then r is not feasible.

FIGURE 9–29 The explanation of $S_i \subset S'_i$.



In Algorithm 9–5, Step 1 and Step 2 take $O(n^2 \log n)$ time and Step 3 is executed $O(\log n)$ times. The loop of testing from Step 3 to Step 5 takes $O(mn)$ time. We therefore have an $O(n^2 \log n)$ approximation algorithm with error ratio 2 for the rectilinear m-center problem.

9–7 AN APPROXIMATION ALGORITHM FOR THE MULTIPLE SEQUENCE ALIGNMENT PROBLEM

In Chapter 7, we only studied the alignment between two sequences. It is natural to extend this problem to the multiple sequence alignment problem in which more than two sequences are involved. Consider the following case where three sequences are involved.

S_1 = ATTCGAT
S_2 = TTGAG
S_3 = ATGCT

A very good alignment of these three sequences is now shown below.

S_1 = ATTCGAT
S_2 = –TT–GAG
S_3 = AT– –GCT

It is noted that the alignment between every pair of sequences is quite good.

The multiple alignment problem is similar to the two sequence alignment problem. Instead of defining a fundamental scoring function $\sigma(x, y)$ we must define a scoring function involving more variables. Let us assume that there are, say three sequences. Instead of considering a_i and b_j, we now must consider matching a_i, b_j and c_k. That is, a $\sigma(x, y, z)$ must be defined and we must find $A(i, j, k)$. The problem is: when we determine $A(i, j, k)$, we need to consider the following.

$$A(i-1, j, k),\ A(i, j-1, k),\ A(i, j, k-1),\ A(i-1, j-1, k)$$
$$A(i, j-1, k-1),\ A(i-1, j, k-1),\ A(i-1, j-1, k-1)$$

Let a linear scoring function be defined between two sequences. Given k input sequences, the sum of pair multiple sequence alignment problem is to find an alignment of these k sequences which maximizes the sum of scores of all pairs of sequence alignments among them. If k, the number of input sequences, is a variable, this problem was proved to be NP-hard. Thus, there is not much hope for using a polynomial algorithm to solve this sum of pair multiple sequence alignment problem, and instead approximation algorithms are needed. In this

section, we shall introduce an approximation algorithm, proposed by Gusfield, to solve the sum of pair multiple sequence alignment problem.

Consider the following two sequences:

S_1 = GCCAT

S_2 = GAT.

A possible alignment between these two sequences is

S'_1 = GCCAT

S'_2 = G– –AT.

For this alignment, there are three exact matches and two mismatches. We define $\sigma(x, y) = 0$ if $x = y$ and $\sigma(x, y) = 1$ if $x \neq y$. For an alignment $S'_1 = a'_1, a'_2, \ldots, a'_n$ and $S'_2 = b'_1, b'_2, \ldots, b'_n$, the distance between the two sequences induced by the alignment is defined as

$$\sum_{i=1}^{n} \sigma(a'_i, b'_i).$$

The reader can easily see that this distance function, denoted as $d(S_i, S_j)$, has the following characteristics:

(1) $d(S_i, S_i) = 0$

(2) $d(S_i, S_j) + d(S_i, S_k) \geq d(S_j, S_k)$

The second property is called the triangular inequality.

Another point which we must emphasize is that for a 2-sequence alignment, the pair (–, –) will never occur. But, in a multiple sequence alignment, it is possible to have a "–" matched with a "–". Consider S_1, S_2 and S_3 as follows:

S_1 = ACTC

S_2 = AC

S_3 = ATCG.

Let us first align S_1 and S_2 as follows:

S_1 = ACTC

S_2 = A– –C.

Then suppose we align S_3 with S_1 as follows:

S_1 = ACTC–
S_3 = A–TCG.

The three sequences are finally aligned as below:

S_1 = ACTC–
S_2 = A– –C–
S_3 = A–TCG.

This time, "–" is matched with "–" twice. Note that when S_3 is aligned with S_1, "–" is added to the already aligned S_1.

Given two sequences S_i and S_j, the minimum aligned distance is denoted as $D(S_i, S_j)$. Let us now consider the following four sequences. We first find out the sequence with the shortest distances to all other sequences

S_1 = ATGCTC
S_2 = AGAGC
S_3 = TTCTG
S_4 = ATTGCATGC.

We align the four sequences in pairs.

S_1 = ATGCTC
$D(S_1, S_2) = 3$
S_2 = A–GAGC

S_1 = ATGCTC
$D(S_1, S_3) = 3$
S_3 = TT–CTG

S_1 = AT–GC–T–C
$D(S_1, S_4) = 3$
S_4 = ATTGCATGC

S_2 = AGAGC
$D(S_2, S_3) = 5$

S_3 = TTCTG

S_2 = A– –G–A–GC

$D(S_2, S_4) = 4$

S_4 = ATTGCATGC

S_3 = –TT–C–TG–

$D(S_3, S_4) = 4$

S_4 = ATTGCATGC.

Then we have

$$D(S_1, S_2) + D(S_1, S_3) + D(S_1, S_4) = 3 + 3 + 3 = 9$$
$$D(S_2, S_1) + D(S_2, S_3) + D(S_2, S_4) = 3 + 5 + 4 = 12$$
$$D(S_3, S_1) + D(S_3, S_2) + D(S_3, S_4) = 3 + 5 + 4 = 12$$
$$D(S_4, S_1) + D(S_4, S_2) + D(S_4, S_3) = 3 + 4 + 4 = 11.$$

It can be seen that S_1 has the shortest distances to all other sequences. We may say that S_1 is most similar to others. We shall call this sequence the center of the sequences. Let us now formally define this concept.

Given a set S of k sequences, the *center* of this set of sequences is the sequence that minimizes

$$\sum_{X \in S \setminus \{S_i\}} D(S_i, X).$$

There are $k(k-1)/2$ pairs of sequences. Each pair can be aligned by using the dynamic programming approach. It can be seen that it takes polynomial time to find a center. Our approximation algorithm works as described in Algorithm 9–7.

Algorithm 9–7 □ A 2-approximation algorithm to find an approximate solution for the sum of pair multiple sequence alignment problem

Input: k sequences.
Output: An alignment of the k sequences with performance ratio not larger than 2.

Step 1: Find the center of these k sequences. Without losing generality, we may assume that S_1 is the center.
Step 2: Let $i = 2$.
Step 3: While $i \leq k$
Find an optimal alignment between S_i and S_1.
Add spaces to the already aligned sequences $S_1, S_2, \ldots, S_{i-1}$ if necessary.
$i = i + 1$
End while
Step 4: Output the final alignment.

Let us consider the four sequences discussed by us in the above paragraphs:

S_1 = ATGCTC
S_2 = AGAGC
S_3 = TTCTG
S_4 = ATTGCATGC.

As we showed before, S_1 is the center. We now align S_2 with S_1 as follows:

S_1 = ATGCTC
S_2 = A–GAGC.

Add S_3 by aligning S_3 with S_1.

S_1 = ATGCTC
S_3 = –TTCTG.

Thus, the alignment becomes:

S_1 = ATGCTC
S_2 = A–GAGC
S_3 = –TTCTG.

Add S_4 by aligning S_4 with S_1.

S_1 = AT–GC–T–C
S_4 = ATTGCATGC.

This time, spaces are added to the aligned S_1. Thus, spaces must be added to aligned S_2 and S_3. The final alignment is:

S_1 = AT–GC–T–C
S_2 = A– –GA–G–C
S_3 = –T–TC–T–G
S_4 = ATTGCATGC.

As we can see, this is a typical approximation algorithm as we only align all sequences with respect to S_1. Let $d(S_i, S_j)$ denote the distance between S_i and S_j induced by this approximation algorithm. Let $App = \sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq i}}^{k} d(S_i, S_j)$. Let $d^*(S_i, S_j)$ denote the distance between S_i and S_j induced by an optimal multiple sequence alignment. Let $Opt = \sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq i}}^{k} d^*(S_i, S_j)$. We shall now show that $App \leq 2Opt$.

Before the formal proof, let us note that $d(S_1, S_i) = D(S_1, S_i)$. This can be easily seen by examining the above example.

S_1 = ATGCTC
S_2 = A–GAGC.

Thus, $D(S_1, S_2) = 3$. At the end of the algorithm, S_1 and S_2 are aligned as follows:

S_1 = AT–GC–T–C
S_2 = A– –GA–G–C.

Then $d(S_1, S_2) = 3 = D(S_1, S_2)$. This distance is not changed because $\sigma(-, -) = 0$.

The proof of $App \leq 2Opt$ is as follows:

$$App = \sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq i}}^{k} d(S_i, S_j)$$

$$\leq \sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq i}}^{k} d(S_i, S_1) + d(S_1, S_j) \qquad \text{(triangle inequality)}$$

$$= 2(k-1) \sum_{i=2}^{k} d(S_1, S_i) \qquad (d(S_1, S_i) = d(S_i, S_1)).$$

Since $d(S_1, S_i) = D(S_1, S_i)$ for all i, we have

$$App \leq 2(k-1) \sum_{i=2}^{k} D(S_1, S_i). \tag{9–1}$$

Let us now find $Opt = \sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq i}}^{k} d^*(S_i, S_j)$. First, let us note that $D(S_i, S_j)$ is the distance induced by an optimal 2-sequence alignment. Thus,

$$D(S_i, S_j) \leq d^*(S_i, S_j),$$

and

$$Opt = \sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq i}}^{k} d^*(S_i, S_j)$$

$$\geq \sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq i}}^{k} D(S_i, S_j).$$

But, note that S_1 is the center. Thus,

$$Opt \geq \sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq 1}}^{k} D(S_i, S_j)$$

$$\geq \sum_{i=1}^{k} \sum_{j=2}^{k} D(S_1, S_j)$$

$$= k \sum_{j=2}^{k} D(S_1, S_j). \tag{9–2}$$

Considering Equations (9–1) and (9–2), we have $App \leq 2Opt$.

9–8 A 2-APPROXIMATION ALGORITHM FOR THE SORTING BY TRANSPOSITION PROBLEM

In this section, we shall introduce the sorting by transposition algorithms for comparing two genomes. We may simply view a genome to be a sequence of genes whose order in the sequence is critically important. Thus, in this section, we label each gene by an integer and a genome to be a sequence of integers.

The comparison of two genomes is significant because it provides us some insight as to how far away genetically these species are. If two genomes are similar to each other, they are genetically close; otherwise they are not. The question is how we measure the similarity of two genomes. Essentially, we measure the similarity of two genomes by measuring how easy it is to transform one genome to another by some operations. In this section, we shall introduce one kind of operation, namely transposition.

Since we are transforming a sequence of numbers into another sequence, without losing generality, we may always assume that the target sequence is 1, 2, … , n. A transposition swaps two adjacent substrings of any length without changing the order of two substrings. One example describing such an operation is as follows,

Genome X: 3 1 5 2 4 → Genome Y: 3 2 4 1 5.

The similarity between two sequences will be measured by the minimum number of operations required to transform a sequence into another. Because the target sequence is always 1, 2, … , n, we may view the problem as a sorting problem. But this is not a usual sorting problem which we are familiar with. Our sorting problem is to find the minimum number of a specified operation needed to sort a sequence. Thus, our sorting problem is an optimization problem.

Consider a sequence 1 4 5 3 2. We may sort this sequence by transposition as follows:

1 4 5 3 2
1 3 2 4 5
1 2 3 4 5.

We now start to introduce sorting by transposition. First of all, although we are talking about sorting, our input is different from ordinary inputs for sorting. It must satisfy the following conditions:

(1) The input sequence cannot contain two identical numbers. For instance, it cannot contain two 5's.
(2) No negative number may appear in the input sequence.
(3) If i and j appear in the sequence and $i < k < j$, k must appear in the sequence. That is, we do not allow the case where 5 and 7 appear, but 6 does not appear.

In summary, we may simply define our input to be a permutation of 1, 2, … , n. That is, the input genome is represented by a permutation $\pi = \pi_1\pi_2 \cdots \pi_n$. For reasons which will become clear later, we extend the permutation to include $\pi_0 = 0$ and $\pi_{n+1} = n + 1$. For instance, a typical input permutation is 0 2 4 1 3 5.

For a permutation π, a transposition, denoted by $\rho(i, j, k)$ (defined for all $1 \le i < j \le n + 1$ and all $1 \le k \le n + 1$ such that $k \notin [i, j]$), swaps the substrings π_i, π_{i+1}, … , π_{j-1}, and π_j, π_{j+1}, … , π_{k-1} if $k > j$ or π_k, π_{k+1}, …, π_{i-1} and π_i, π_{i+1}, …, π_{j-1} if $k < i$ in the permutation. For instance, $\rho(2, 4, 6)$ in permutation 0 7 2 3 6 1 5 4 8 swaps substrings (2 3) and (6 1) and results in 0 7 6 1 2 3 5 4 8. Given a permutation π and a transposition ρ, the application of ρ to π is denoted as $\rho \cdot \pi$.

The problem of sorting by transposition is now formally defined as follows: Given two permutations π and σ, the sorting by transposition problem is to find a series of transpositions ρ_1, ρ_2, … , ρ_t such that $\rho_t \ldots \rho_2 \cdot \rho_1 \cdot \pi = \sigma$ and t is minimum. This t is called the transposition distance between π and σ. As indicated before, without losing generality, we shall assume that σ is always an identity permutation in the form of (0, 1, 2, … , n, $n + 1$). In the following, we will introduce the 2-approximation algorithm for sorting by transposition, proposed by Bafna and Pevzner in 1998. Note that the complexity of the problem of sorting by transpositions is unknown.

For all $0 \le i \le n$ in a permutation, there is a *breakpoint* between π_i and π_{i+1} if $\pi_{i+1} \ne \pi_i + 1$. For instance, for a permutation 0 2 3 1 4 5, the permutation with breakpoints added is 0, 2 3, 1, 4 5. A sorted permutation contains no breakpoints. A permutation without breakpoints is called an identity permutation. Consequently, we may say that our job is to sort the input permutation into an identity permutation.

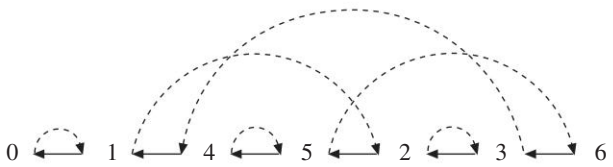
Since the identity permutation is a permutation without breakpoints, sorting a permutation corresponds to decreasing the number of breakpoints. Let $d(\pi)$ denote the minimum number of transpositions needed to transform π to an

identity permutation. For each transposition, at most three breakpoints can be decreased, and therefore, a trivial lower bound of $d(\pi)$ is

(the number of breakpoints in π)/3.

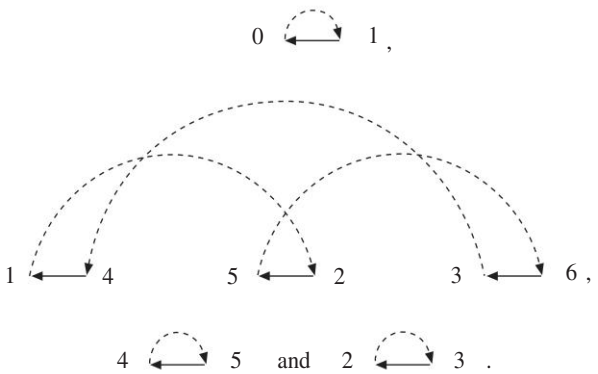
A cycle graph of π, denoted by $G(\pi)$, is the directed edge-colored graph with vertex set $\{0, 1, 2, \ldots, n, n+1\}$ and edge set defined as follows. For all $1 \le i \le n+1$, gray edges are directed from $i-1$ to i and black edges from π_i to π_{i-1}. An example of cycle graphs is shown in Figure 9–30, where dashed and solid arcs represent gray and black edges respectively.

FIGURE 9–30 A cycle graph of a permutation 0 1 4 5 2 3 6.



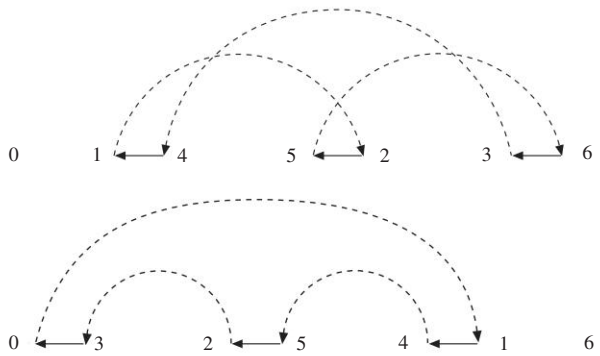
An alternating cycle of a colored graph is a cycle where each pair of adjacent edges is of different colors. For each vertex in $G(\pi)$, every incoming edge is uniquely paired with an outgoing edge of a different color. Hence, the edge set of $G(\pi)$ can be decomposed into alternating cycles. One example for the decomposition of the cycle graph depicted in Figure 9–30 into alternating cycles is shown in Figure 9–31.

FIGURE 9–31 The decomposition of a cycle graph into alternating cycles.



We use k-cycles to refer to an alternating cycle which contains k black edges. We say that a k-cycle is long if $k > 2$, and short otherwise. Two examples about long and short alternating cycles are shown in Figure 9–32. In Figure 9–33, we show the cycle graph of an identity permutation.

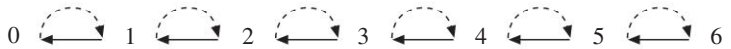
FIGURE 9–32 Long and short alternating cycles.



(a) Long alternating cycles

(b) Short alternating cycles

FIGURE 9–33 The cycle graph of an identity graph.



As suggested in Figure 9–33, in an identity permutation, each vertex x points to another vertex y through a gray edge, and vertex y points backward to x through a black edge. We call this kind of cycle graphs regular. The sorting by transposition problem is to transform a cycle graph which is not regular into a regular cycle graph.

Since we will always be dealing with alternating cycles, we shall simply call an alternating cycle a cycle. There are at most $n + 1$ cycles in $G(\pi)$, and the only

permutation with $n + 1$ cycles is the identity permutation. We denote the number of cycles in $G(\pi)$ as $c(\pi)$ for a permutation π. Consequently, the purpose of sorting π is increasing the number of cycles from $c(\pi)$ to $n + 1$. We also denote the change in the number of cycles due to transposition ρ as $\Delta c(\rho) = c(\rho\pi) - c(\pi)$ for a permutation π.

Let us consider the following permutation:

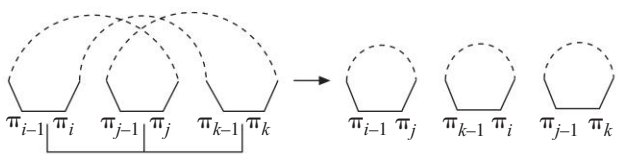
0 3 4 1 2 5.

Suppose we perform a transposition on substrings (3 4) and (1 2), as indicated above, to achieve the following:

0 1 2 3 4 5

The reader can easily see that the above transposition is quite ideal. The original permutation has three breakpoints. After the transposition, there are no breakpoints. This can be explained by using the alternating cycle diagram. Suppose that the transposition is $\rho(i, j, k)$ and suppose its corresponding vertices in $G(\pi)$ involved in the permutation, namely π_{i-1}, π_i, π_{j-1}, π_j, π_{k-1}, π_k are in one cycle, shown in Figure 9–34.

FIGURE 9–34 A special case of transposition with $\Delta c(\rho) = 2$.

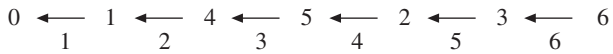


As shown in Figure 9–34, this particular kind of transposition increases the number of cycles by two. Thus, we have a better lower bound $d(\pi) = \dfrac{n+1-c(\pi)}{2}$. Any sorting by transposition algorithm which produces a transposition distance equal to this lower bound $\dfrac{n+1-c(\pi)}{2}$ must be an optimal algorithm. Up to now, there is no such algorithm yet. Of course, there is a possibility that the lower bound is not high enough. In the following, we shall achieve a 2-approximation algorithm.

The case in Figure 9–34 is a very desirable one because it is a transposition which increases the number of cycles by two. Ideally, we hope that there are such kinds of cycles in our permutation at any moment. Unfortunately, this is not true. As a result, we must handle the other cases.

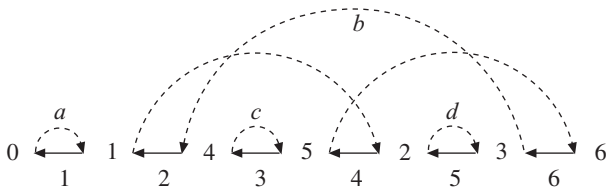
We first assign a number from 1 to $n + 1$ to the black edges of $G(\pi)$, and say that a transposition $\rho\ (i, j, k)$ acts on black edges i, j and k. An example for assigning the number to the black edges is shown in Figure 9–35.

FIGURE 9–35 A permutation with black edges of $G(\pi)$ labeled.



Note that a cycle can be represented by $(i_1, \ldots, i_k)$ according to the visiting black edges from i_1 to i_k, where i_1 is the rightmost black edge in the cycle. For instance, there are four alternating cycles in the $G(\pi)$ in Figure 9–36. The rightmost black edge in cycle b is black edge 6, so cycle b is represented as (6, 2, 4).

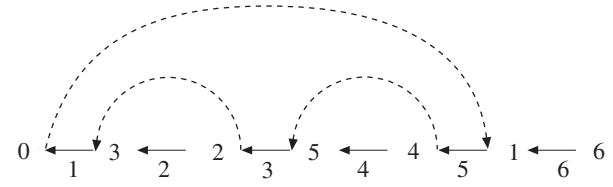
FIGURE 9–36 A permutation with $G(\pi)$ containing four cycles.



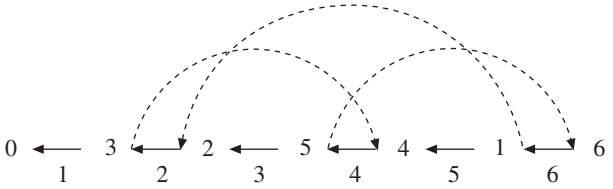
There are two different kinds of cycles, non-oriented and oriented cycles. For all $k > 1$, a cycle $C = (i_1, \ldots, i_k)$ is non-oriented if $i_1, \ldots, i_k$ is a decreasing sequence; C is oriented otherwise. Two examples about non-oriented and oriented cycles are shown in Figure 9–37.

We shall call a transposition ρ as *x-move* if $\Delta c(\rho) = x$. Assume that a cycle $C = (i_1, \ldots, i_k)$ is an oriented cycle. We can prove that there exist an i_t in C, $i_t > i_{t-1}$, $3 \le t \le k$ and a transposition $\rho(i_{t-1}, i_t, i_1)$ such that $\rho\pi$ creates a 1-cycle containing vertices $\pi_{i_{t-1}-1}$ and π_{i_t} and other cycles. Therefore, ρ is a 2-move transposition. In conclusion, there is a 2-move transposition on every oriented cycle. An example is shown in Figure 9–38. In this example, the input permutation

FIGURE 9–37 Oriented and non-oriented cycles.



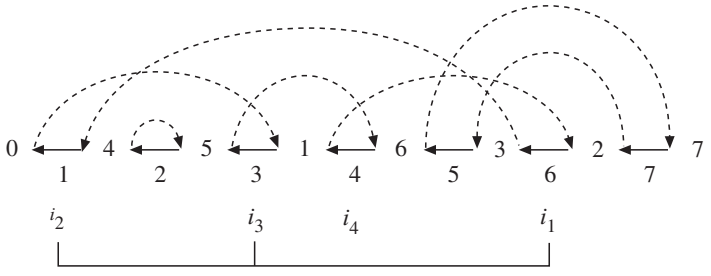
(a) Non-oriented cycle (5, 3, 1)



(b) Oriented cycle (6, 2, 4)

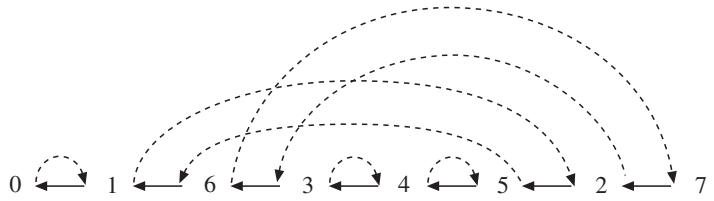
is 0 4 5 1 6 3 2 7. As shown in Figure 9–38(a), there are three cycles. Among them, the cycle (6, 1, 3, 4) is an oriented cycle. As shown in Figure 9–38(a), there is a transposition $\rho\,(i_2, i_3, i_1) = \rho\,(1, 3, 6)$. This transposition corresponds to swapping the substrings (4 5) and (1 6 3). After applying this transposition, the permutation becomes 0 1 6 3 4 5 2 7 and there will be five cycles. As one can see in Figure 9–38(b), the number of cycles is increased by two.

FIGURE 9–38 An oriented cycle allowing a 2-move.



(a) A permutation: 0 4 5 1 6 3 2 7, $t = 3$, an oriented cycle (6, 1, 3, 4) and a transposition $\rho(i_2, i_3, i_1)$

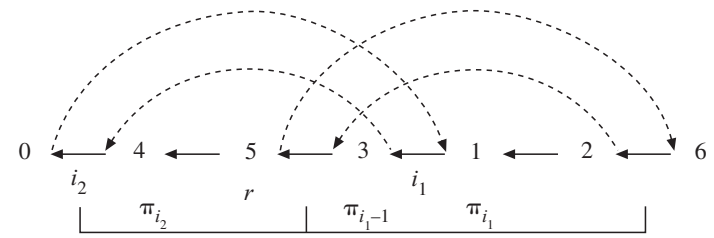
FIGURE 9–38 (*cont'd*)



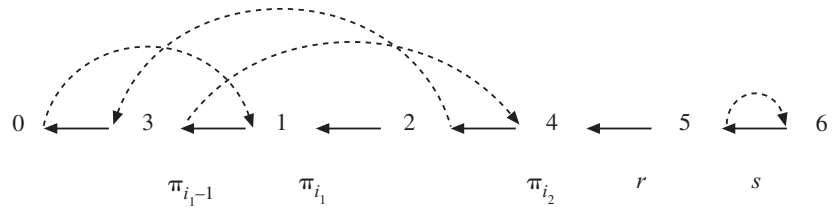
(b) The cycle graph after transposition

Consider Figure 9–39(a). Note that there is no oriented cycle in the graph. It can be easily seen that there must exist more than one non-oriented cycle. Assume that a cycle $C = (i_1, i_2, \ldots, i_k)$ is a non-oriented cycle. Let the position of the maximal element of permutation π in the interval $[i_2, i_1 - 1]$ be r, and the position of $\pi_r + 1$ in π be s. We can show that $s > i_1$ and transposition $\rho(r + 1, s, i_2)$ is a 0-move which transforms a non-oriented cycle C into an oriented cycle C' allowing a 2-move. Therefore, there is a 0-move followed by a 2-move on a non-oriented cycle. An example is shown in Figure 9–39.

FIGURE 9–39 A non-oriented cycle allowing 0, 2-moves.



(a) A permutation: 0 4 5 3 1 2 6, a non-oriented cycle $C = (i_1, i_2)$, $r = 2$, $s = 6$, a transposition $\rho(3, 6, 1)$, $\pi_{i_1} = 1$, $\pi_{i_1-1} = 3$ and $\pi_{i_2} = 4$



(b) The cycle graph after transposition

From the previous discussion, we can see that for an arbitrary permutation π, there exists either a 2-move permutation or a 0-move permutation followed by a 2-move permutation. We thus have obtained an upper bound of the transposition distance. That is $d(\pi) \leq \frac{n+1-c(\pi)}{2/2} = n+1-c(\pi)$ for sorting by transpositions. Thus, we have an algorithm which can produce a transposition distance not higher than $n + 1 - c(\pi)$.

Because of the lower bound, $d(\pi) \geq \frac{n+1-c(\pi)}{2}$, and the upper bound, $d(\pi) \leq n + 1 - c(\pi)$, there is an approximation algorithm for sorting by transpositions with performance ratio 2. The algorithm is listed in Algorithm 9–8.

Algorithm 9–8 □ A 2-approximation algorithm to find an approximation solution for the problem of sorting by transpositions

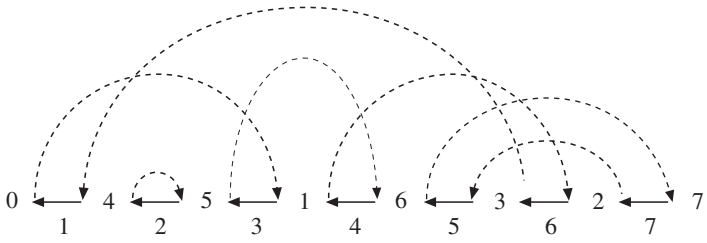
Input: Two permutations π and σ.
Output: The minimum distance between two permutations π and σ.
Step 1: Relabel two permutations for sorting permutation π into the identity permutation.
Step 2: Construct the cycle $G(\pi)$ graph of permutation π. Let the distance $d(\pi) = 0$.
Step 3: While there is an oriented cycle
    Perform a 2-move, $d(\pi) = d(\pi) + 1$
While there is a non-oriented cycle
    Perform a 0-move followed by 2-move, $d(\pi) = d(\pi) + 2$
Step 4: Output the distance $d(\pi)$.

An example for the 2-approximation algorithm is shown in Figure 9–40. The algorithm is as follows:

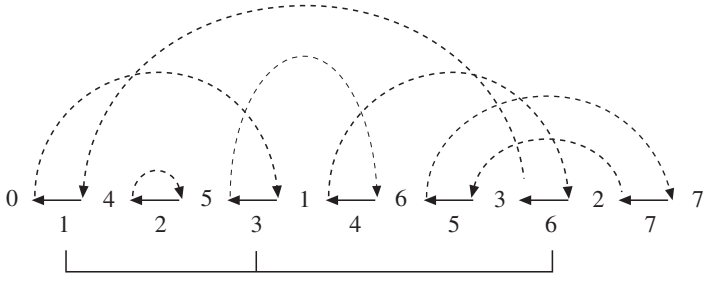
(1) Since there is an oriented cycle (6, 1, 3, 4), we perform a transposition $\rho(6, 3, 1)$ on it. The result is shown in Figure 9–40(c).
(2) As shown in Figure 9–40(b), no oriented cycle exists in the cycle graph, and there are two non-oriented cycles (6, 2) and (7, 3). We perform transposition $\rho(3, 7, 2)$ followed by transposition $\rho(6, 5, 2)$. The result of

transposition $\rho(7, 3, 2)$ is shown in Figure 9–40(d), and the result of applying transposition $\rho(6, 5, 2)$ is shown in Figure 9–40(e).

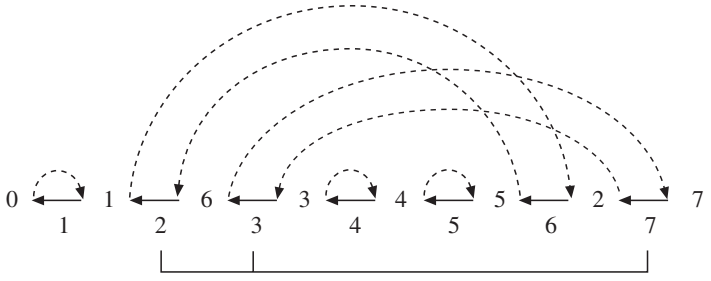
FIGURE 9–40 An example for 2-approximation algorithm.



(a) The cycle graph of a permutation: 4 5 1 6 3 2, an oriented cycle (6, 1, 3, 4), a cycle (2), and a non-oriented cycle (7, 5)

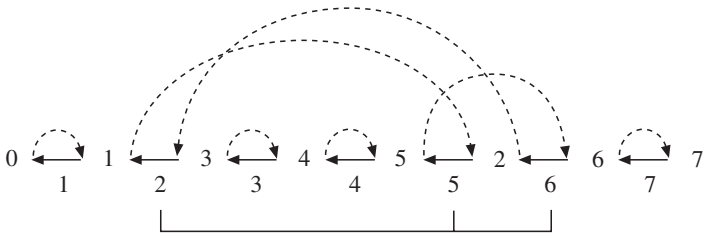


(b) A transposition $\rho(6, 3, 1)$ for the oriented cycle (6, 1, 3, 4)



(c) The cycle graph after a 2-move, two non-oriented cycles (6, 2) and (7, 3), other cycles (1), (4) and (5) and a transposition $\rho(3, 7, 2)$ for the non-oriented cycle

FIGURE 9–40 (*cont'd*)



(d) The cycle graph after (c), an oriented cycle (6, 2, 5), other cycles (1), (3), (4) and (7) and a transposition $\rho(6, 5, 2)$ acting on the oriented cycle

(e) The final cycle graph after three transpositions, a 2-move, a 0-move and a 2-move

The entire process of applying the transpositions is shown as follows:

0 4 5 1 6 3 2 7
0 1 6 3 4 5 2 7
0 1 3 4 5 2 6 7
0 1 2 3 4 5 6 7.

9–9 THE POLYNOMIAL TIME APPROXIMATION SCHEME

For every approximation algorithm, there is error involved. We of course like the error to be as small as possible. That is, we would like to have a family of approximation algorithms such that for every error, there is a corresponding approximation algorithm which achieves this error. Thus, no matter how small the error is specified, we can achieve it by using the corresponding approximation algorithm. Of course, we pay a price for this small error because the time complexity of the algorithm with a smaller error must be higher than that with a larger error. It would be ideal if no matter how small the error is, the time complexity remains polynomial. The above discussion leads to the concept of polynomial time approximation scheme (PTAS).

Let S_{OPT} be the cost of an optimal solution and S_{APX} denote the cost of an approximate solution. The error ratio is now defined as

$$\varepsilon = (S_{OPT} - S_{APX})/S_{OPT}.$$

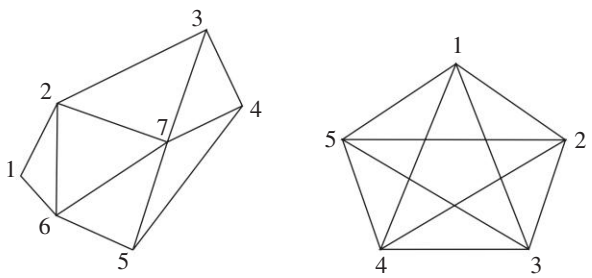
A PTAS for a problem denotes a family of approximation algorithms such that for each prespecified error ratio E, there is an approximation algorithm which maintains polynomial time complexity. For instance, suppose that the time complexity of our algorithm is $O(n/E)$. Then no matter how small E is, we still have a polynomial time approximation algorithm with respect to E, because E is a constant.

The PTAS for the Maximum Independent Set Problem on Planar Graphs

In this section, we shall show that for the maximum independent set problem on planar graphs, there is a PTAS in which the time complexity of each algorithm is $O(8^k kn)$ where $k = \lceil 1/E \rceil - 1$.

We first define planar graphs. *A graph is said to be embedded on a surface S if it can be drawn on S so that its edges intersect only at their end vertices. A graph is a planar graph if it can be embedded on a plane*. Figure 9–41(a) shows a planar graph and the graph in Figure 9–41(b) is not planar.

FIGURE 9–41 Graphs.



(a) A planar graph **(b) A graph which is not planar**

The maximum independent set problem on planar graphs is NP-hard. Thus, approximation algorithms are desirable. Let us first define some terms. A face is a region defined by a planar embedding. The unbounded faces are called exterior faces and all other faces are called interior faces. For instance, for the face in Figure 9–42, face $6 \rightarrow 7 \rightarrow 11 \rightarrow 12 \rightarrow 6$ is an interior face, and face $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 1$ is an exterior face. In a planar graph, we can associate each node with a level. In Figure 9–42, there is only one exterior face, namely $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 1$. Thus, nodes 1, 2, 3, 4 and 5 are of level 1. Subsequently, nodes 6, 7, 11, 12, 8, 9, 10, 13, 14 are of level 2 and nodes 15, 16 and 17 are of level 3. The levels of nodes can be computed in linear time.

A graph is k-outerplanar if it has no nodes with level greater than k. For instance, Figure 9–43 contains a 2-outerplanar graph.

FIGURE 9–42 An embedding of a planar graph.

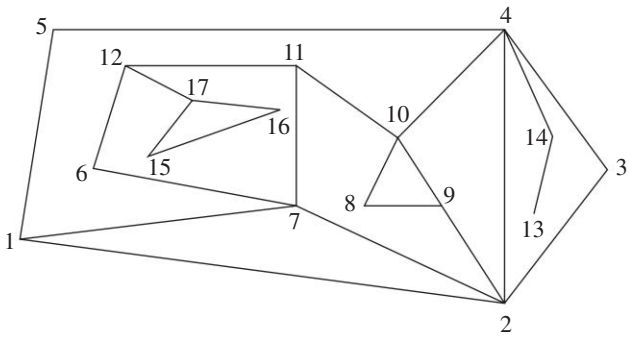
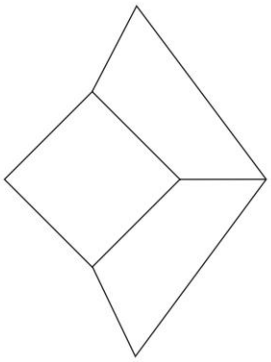


FIGURE 9–43 Example of a 2-outerplanar graph.



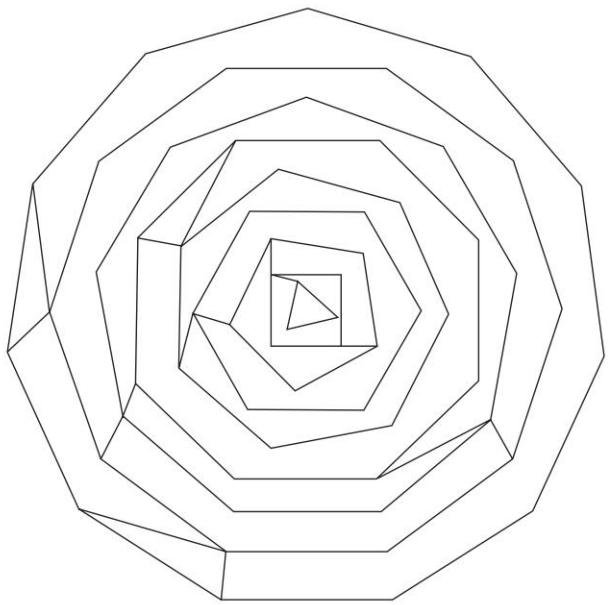
For a k-outerplanar graph, an optimal solution for the maximum independent set problem can be found in $O(8^k n)$ time through the dynamic programming approach where n is the number of vertices. We shall not give details of the dynamic programming approach here.

Given an arbitrary planar graph G, we can decompose it into a set of k-outerplanar graphs. Consider Figure 9–44. Suppose we set k to be 2. Then we group all the nodes in levels 3, 6 and 9 into class 3, nodes in level 1, 4, 7 into class 1, and nodes in level 2, 5 and 8 into class 2, as shown in Table 9–1.

TABLE 9–1 The decomposition of nodes.

Levels	1	4	7	(class 1)
Levels	2	5	8	(class 2)
Levels	3	6	9	(class 3)

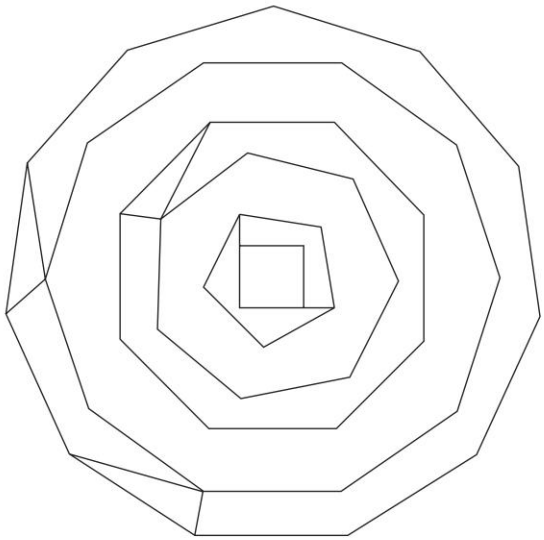
FIGURE 9–44 A planar graph with nine levels.



If we delete all the nodes in class 3, namely nodes in level 3, 6 and 9, we shall have the resulting graph shown in Figure 9–45. It is obvious that all the subgraphs in Figure 9–45 are 2-outerplanar graphs. For each 2-outerplanar graph, we may find its maximum independent set in linear time. Besides, the union of these maximum independent sets is still an independent set (not necessarily maximum) for our original planar graph and thus may serve as an approximate solution for it.

Similarly, we can delete nodes in class 1, namely nodes in levels 1, 4 and 7. The resulting graph again will also consist of a set of 2-outerplanar graphs. Using similar mechanisms, we can obtain an approximate maximum independent set for our original planar graph.

FIGURE 9–45 The graph obtained by removing nodes in levels 3, 6 and 9.



Now, the levels of nodes in class 1 are all congruent to 1 (mod $k + 1$) and the levels of nodes in class 2 are all congruent to 2 (mod $k + 1$). Our approximation algorithm, based upon a certain prespecified k, works as follows:

Algorithm 9–9 □ An approximation algorithm to solve the maximum independent set problem on planar graphs

Step 1. For all $i = 0, 1, \ldots, k$, do

(1.1) Let G_i be the graph obtained by deleting all nodes with levels congruent to i (mod $k + 1$). The remaining subgraphs are all k-outerplanar graphs.

(1.2) For each k-outerplanar graph, find its maximum independent set. Let S_i denote the union of these solutions.

Step 2. Among $S_0, S_1, \ldots, S_k$, choose the S_j with the maximum size and let it be our approximate solution S_{APX}.

The time complexity of our approximation algorithm is obviously $O(8^k kn)$. In the following, we shall show that the k is inversely proportional to the error rate. Thus, the maximum independent set problem on planar graphs has a PTAS.

Note that we have divided all nodes into $(k + 1)$ classes; each class corresponds to a level congruent to $i \pmod{k+1}$ for $i = 0, 1, \ldots, k$. For every independent set S, the average number of nodes in this set for each class is $|S|/(k + 1)$ where $|S|$ is the number of nodes in this independent set. Thus, there is at least one r, such that at most $\frac{1}{k+1}$ of vertices in S_{OPT} are at a level which is congruent to $r \pmod{k+1}$. This means that the solution S_r obtained by deleting the nodes in class r from S_{OPT} will have at least $|S_{OPT}|\left(1-\frac{1}{k+1}\right)=|S_{OPT}|\frac{k}{k+1}$ nodes, because at most $|S_{OPT}|\frac{1}{k+1}$ nodes are deleted. Therefore,

$$|S_r| \geq |S_{OPT}|\frac{k}{k+1}.$$

According to our algorithm,

$$|S_{APX}| \geq |S_r| \geq |S_{OPT}|\frac{k}{k+1}$$

or

$$\varepsilon = \frac{|S_{OPT} - S_{APX}|}{|S_{OPT}|} \leq \frac{1}{k+1}.$$

Thus, if we set $k = \lceil 1/E \rceil - 1$, then the above formula becomes

$$\varepsilon \leq \frac{1}{k+1} = \frac{1}{\lceil 1/E \rceil} \leq E.$$

This shows that for every given error bound E, we have a corresponding k to guarantee that the approximate solution differs from the optimum one within this error ratio. Besides, no matter how small the error, we can always find an algorithm to achieve this error with time complexity $O(8^k kn)$, which is polynomial with respect to n.

The PTAS for the 0/1 Knapsack Problem

The 0/1 knapsack problem is an NP-hard problem and was discussed in Chapter 5. It is indeed surprising that there exists a polynomial time approximation scheme for this problem.

The 0/1 knapsack problem is defined as follows: We are given n items. The ith item has profit p_i and weight w_i. Given an integer M, the 0/1 knapsack problem is to select a subset of these n items such that the sum of profits is maximized under the constraint that the sum of weights does not exceed M. Formally, we want to maximize $\Sigma\delta_i p_i$ where $\delta_i = 1$ or 0 such that $\Sigma\delta_i w_i \leq M$.

Let us consider one example. $M = 92$, $n = 8$ and the profits and weights are shown in Table 9–2, with the items sorted according to the non-increasing order of p_i/w_i.

TABLE 9–2 Profits and weights for eight items.

i	1	2	3	4	5	6	7	8
p_i	90	61	50	33	29	23	15	13
w_i	33	30	25	17	15	12	10	9
p_i/w_i	2.72	2.03	2.0	1.94	1.93	1.91	1.5	1.44

Basically, the approximation algorithm is based upon an error ratio ε. Once this ε is given, we shall, through an elaborate calculation, calculate a threshold, called T. With this threshold T, we divide all the n items into two subsets: *BIG* and *SMALL*. *BIG* contains all items whose profits are larger than T and *SMALL* contains all items smaller than or equal to T.

In our case, T will be found to be 46.8. Thus, $BIG = \{1, 2, 3\}$ and $SMALL = \{4, 5, 6, 7, 8\}$.

After *BIG* and *SMALL* are obtained, we normalize the profits of items in *BIG*. In our case, the normalized profits are

$$p'_1 = 9$$
$$p'_2 = 6$$
and $$p'_3 = 5.$$

We now try to enumerate all possible solutions for this problem instance. There are many such solutions. Let us consider two of them:

Solution 1: We select items 1 and 2. The sum of normalized profits is 15. The corresponding sum of original profits is 90 + 61 = 151. The sum of weights is 63.

Solution 2: We select items 1, 2 and 3. The sum of normalized profits is 20. The corresponding sum of original profits is 90 + 61 + 50 = 201. The sum of weights is 88.

Since the sum of weights is smaller than $M = 92$, we can now add items from *SMALL* to both solutions. This can be done by using the greedy method.

Solution 1: For Solution 1, we can add items 4 and 6. The sum of profits will be 151 + 33 + 23 = 207.

Solution 2: For Solution 2, we cannot add any item from *SMALL*. Thus, the sum of profits is 201.

There are, of course, many such solutions. For each of them, we apply the greedy method and obtain an approximate solution. It can be shown that Solution 1 is the largest. Thus, we output Solution 1 as the approximate solution. It is important to note that an approximation algorithm must be a polynomial algorithm. The greedy part is obviously polynomial. The critical part is the part of algorithm which finds all feasible solutions for the items in *BIG*. One of these feasible solutions is an optimal solution for the 0/1 knapsack problem with the items. If this part of the algorithm is polynomial, does it mean that we used a polynomial algorithm to solve the 0/1 knapsack problem?

We shall show later that for this part, we are actually solving a special version of the 0/1 knapsack problem. We shall show later that in this problem instance, the profits were normalized to such an extent that the sum of the normalized profits will be less than $\lfloor (3/\varepsilon)^2 \rfloor$, where ε is the error ratio. That is, the sum of normalized profits is less than a constant. Once this condition is satisfied, there exists a polynomial time algorithm which produces all possible feasible solutions. Thus, our approximation algorithm is polynomial.

Having given the top level outline of our approximation algorithm, let us now give the details of our algorithm. Again, we shall describe the details by applying the algorithm to our data. Let us set ε to be 0.6.

Step 1: Sort the items according to the non-increasing profit to weight ratio p_i/w_i (Table 9–3).

TABLE 9–3 Sorted items.

i	1	2	3	4	5	6	7	8
p_i	90	61	50	33	29	23	15	13
w_i	33	30	25	17	15	12	10	9
p_i/w_i	2.72	2.03	2.0	1.94	1.93	1.91	1.5	1.44

Step 2: Calculate a number Q as follows:
Find the largest d such that
$W = w_1 + w_2 + \cdots + w_d \leq M$.
If $d = n$ or $W = M$, then

Set $P_{APX} = p_1 + p_2 + \cdots + p_d$ and $INDICES = \{1, 2, \ldots, d\}$ and stop.

In this case, $P_{OPT} = P_{APX}$.

Otherwise, set $Q = p_1 + p_2 + \cdots + p_d + p_{d+1}$.
For our case, $d = 3$ and $Q = 90 + 61 + 50 + 33 = 234$.
What are the characteristics of Q? We now show that

$$Q/2 \leq P_{OPT} \leq Q.$$

Note that $p_1 + p_2 + \cdots + p_d \leq P_{OPT}$.
Since $w_{d+1} \leq M$, therefore p_{d+1} itself is a feasible solution.

$$p_{d+1} \leq P_{OPT}.$$

Therefore, $Q = p_1 + p_2 + \cdots + p_d + p_{d+1} \leq 2P_{OPT}$

or $Q/2 \leq P_{OPT}$.

Since P_{OPT} is a feasible solution and Q is not, we have $P_{OPT} \leq Q$.

Thus, $Q/2 \leq P_{OPT} \leq Q$.

It will be shown later that this is crucial for the error analysis.

Step 3: Calculate a normalizing factor δ as follows:

$$\delta = Q(\varepsilon/3)^2.$$

In our case, $\delta = 234(0.6/3)^2 = 234(0.2)^2 = 9.36$
Then we calculate a parameter g:

$$g = \lfloor Q/\delta \rfloor = \lfloor (3/\varepsilon)^2 \rfloor = \lfloor (3/0.6)^2 \rfloor = 25.$$

Set $T = Q(\varepsilon/3)$.
In our case, $T = 234(0.6/3) = 46.8$.

Step 4:

Step 4.1: Let *SMALL* collect all items whose profits are smaller than or equal to T. Collect all other items into *BIG*.
In our case, $SMALL = \{4, 5, 6, 7, 8\}$
and $BIG = \{1, 2, 3\}$.

Step 4.2: For all items in *BIG*, normalize their profits according to the following formula: $p'_i = \lfloor p_i/\delta \rfloor$.
In our case, $p'_1 = \lfloor 90/9.36 \rfloor = 9$
$p'_2 = \lfloor 61/9.36 \rfloor = 6$
$p'_3 = \lfloor 50/9.36 \rfloor = 5$.

Step 4.3: Initialize an array A with size g. Each entry of array will correspond to a combination of p'_i's. Each entry $A[i]$ consists of three fields I, P, W, representing the index of the combination, the sum of profits and the sum of weights, respectively.

Step 4.4: For each item i in *BIG*, we scan the table and perform the following operations for each entry: For entry $A[j]$, if there is already something in $A[j]$ and adding item i to them will not cause the total weight to exceed the capacity limit M, then we check the weight corresponding to $A[j + p'_i]$, which corresponds to the combination of adding item i to $A[j]$. If there is nothing in $A[j + p'_i]$ or the weight $A[j + p'_i] \cdot W$ is larger than $A[j] \cdot W + w_i$ (which is the weight corresponding to adding item i to $A[j]$), then we update the entry with $A[j] \cdot I \bigcup \{i\}$, and update the corresponding profit and weight.
Our example runs as follows.

When $i = 1$, $p'_i = 9$, see Table 9–4.

TABLE 9–4 The array A for $i = 1$, $p'_i = 9$.

p_i	I	P	W
0		0	0
1			
2			
3			
4			
5			
6			
7			
8			
9	1	90	33
10			
11			
12			
13			
14			
15			
16			
17			
18			
19			
20			
21			
22			
23			
24			
25			

$i = 2$, $p'_2 = 6$, see Table 9–5.

TABLE 9–5 The array A for $i = 2$, $p'_2 = 6$.

p_i	I	P	W
0		0	0
1			
2			
3			
4			
5			
6	2	61	30
7			
8			
9	1	90	33
10			
11			
12			
13			
14			
15	1, 2	151	63
16			
17			
18			
19			
20			
21			
22			
23			
24			
25			

$i = 3$, $p'_3 = 5$, see Table 9–6.

TABLE 9–6 The array A for $i = 3$, $p'_3 = 5$.

p_i	I	P	W
0		0	0
1			
2			
3			
4			
5	3	50	25
6	2	61	30
7			
8			
9	1	90	33
10			
11	2, 3	111	55
12			
13			
14	1, 3	140	58
15	1, 2	151	63
16			
17			
18			
19			
20	1, 2, 3	201	88
21			
22			
23			
24			
25			

Let us examine Table 9–6. Each non-empty entry refers to a feasible solution. For instance, the entry of $A[j] = 5$ corresponds to the selection of only

one item, namely item 3. Its total sum of normalized profits is 5, its total profit is 50 and its total weight is 25. For $j = 15$, we select items 1 and 2. In this case the total normalized profits is 15, its total profit is 151 and its total weight is 63.

Step 5. For each entry of Array A, apply the greedy algorithm to add items in *SMALL* to its corresponding combination.

TABLE 9–7 The final array A.

p_i	I	P	W	FILL	Profit
0		0	0	4, 5, 6, 7, 8	113
1					
2					
3					
4					
5	3	50	25	4, 5, 6, 7, 8	163
6	2	61	30	4, 5, 6, 7	161
7					
8					
9	1	90	33	4, 5, 6, 7	190
10					
11	2, 3	111	55	4, 5	173
12					
13					
14	1, 3	140	58	4, 5	202
15	1, 2	151	63	4, 6	207
16					
17					
18					
19					
20	1, 2, 3	201	88		201
21					
22					
23					
24					
25					

Again, let us examine the meaning of Table 9–7. For $j = 15$, we can add items 4 and 6. This will give us a total profit of 207. For $j = 20$, we select items 1, 2 and 3 from *BIG*. In this case, we cannot add any item from *SMALL* because the weight constraint will exceed. Thus, the total weight is 201.

Step 6: Pick the largest profit obtained in Step 5 to be our approximation solution. In our example, the entry corresponding to $\Sigma p'_i = 15$ is chosen, which generates the largest total profit 207.

The reader may observe that Step 4.4 is an exhaustive scanning step. We have these items with normalized profits 9, 6 and 5. The corresponding weights are 33, 30 and 25, respectively. We start from the item with profit 9. Since its weight is less than M, we may select it alone. The total weight is 33 and the total normalized profit is 9. Then we scan the item with normalized profit 6. We may select this item alone. If we do so, the total normalized profit is 6 and the total weight is 30. If we choose the two items already scanned, the total normalized profit is $9 + 6 = 15$ and total weight is $33 + 30 = 63$. Thus, we have four feasible solutions:

(1) Not selecting any one.
(2) Selecting item 1 alone. We shall have total normalized profit 9 and total weight 33.
(3) Selecting item 2 alone. We shall have total normalized profit 6 and total weight 30.
(4) Selecting both items 1 and 2. The total normalized profit is 15 and total weight is 63.

Intuitively, this step will take exponential time: $2^{|BIG|}$. A critical question arises: Why is this process polynomial?

Next, we shall show that the size of array A is not longer than g.

Let the largest entry of A have normalized profits $p'_{i_1} + p'_{i_2} + \cdots + p'_{i_j}$. Since this corresponds to a feasible solution, we have $p_{i_1} + p_{i_2} + \cdots + p_{i_j} \le P_{OPT} \le Q$, where $p'_{i_j} = \lfloor p_{i_j}/\delta \rfloor$.

Consider $p'_{i_1} + p'_{i_2} + \cdots + p'_{i_j}$.

$$
\begin{aligned}
& p'_{i_1} + p'_{i_2} + \cdots + p'_{i_j} \\
&= \lfloor p_{i_1}/\delta \rfloor + \lfloor p_{i_2}/\delta \rfloor + \cdots + \lfloor p_{i_j}/\delta \rfloor \\
&\le \lfloor (p_{i_1} + p_{i_2} + \cdots + p_{i_j})/\delta \rfloor \\
&\le \lfloor Q/\delta \rfloor \\
&= g.
\end{aligned}
$$

This is why it is sufficient to have an array with size g. Note that g is a constant, which is independent of n. The scanning takes at most ng times and this is why Step 4 is polynomial.

The time complexity of this approximation algorithm follows:

Step 1: $O(n \log n)$

Step 2: $O(n)$

Step 4.1 to 4.2: $O(n)$

Step 4.3: $O(g)$

Step 4.4: $O(ng)$

Step 5: $O(ng)$

Step 6: $O(n)$.

The total time complexity of the approximation algorithm is

$$\begin{aligned} &O(n \log n) + O(n) + O(n) + O(g) + O(ng) + O(ng) + O(n) \\ &= O(n \log n) + O(ng) \\ &= O(n \log n) + O(n(3/\varepsilon)^2). \end{aligned}$$

Thus, our algorithm is a polynomial time approximation scheme. Its time complexity is a function of ε, the error ratio. The smaller we set the error ratio, the larger the time complexity becomes. Yet, it is always polynomial.

We will now perform an error analysis of our approximation algorithm. First, we need some claims:

Claim 1: $p'_i \geq 3/\varepsilon$, $\forall i \in BIG$.
We know $p'_i = \lfloor p_i/\delta \rfloor = \lfloor p_i/Q(\varepsilon/3)^2 \rfloor$,
but $\forall i \in BIG$, $p_i > Q(\varepsilon/3)$,
therefore, $p'_i \geq 3/\varepsilon$.

Claim 2: $p'_i\delta \leq p_i \leq p'_i\delta(1 + \varepsilon/3)$, $\forall i \in BIG$.
Note that $p'_i = \lfloor p_i/\delta \rfloor$.
Therefore, $p'_i \leq p_i/\delta$.
We have $p'_i\delta \leq p_i$.

Besides, we also have $p_i/\delta \leq p'_i + 1 = p'_i(1 + 1/p'_i) \leq p'_i(1 + \varepsilon/3)$, since $p'_i \geq 3/\varepsilon$, by Claim 1.

Claim 3: Let P_{OPT} be the optimum profit of a 0/1 knapsack problem, and P_{Greedy} be the profit obtained by applying the greedy method to the same problem. Then

$$P_{OPT} - P_{Greedy} \leq \max\{p_i\}.$$

Proof: Let b be the first item not chosen by the greedy algorithm.
By the choice of the greedy algorithm, $P_{Greedy} + P_b \geq P_{OPT}$
Thus, $P_{OPT} \leq P_{Greedy} + P_b \leq P_{Greedy} + \max\{p_i\}$,
So $P_{OPT} - P_{Greedy} \leq \max\{p_i\}$.

Now we discuss the error analysis of our approximation algorithm.

Let us first note that P_{OPT} may be expressed as follows:

$$P_{OPT} = p_{i_1} + \cdots + p_{i_k} + \alpha,$$

where $p_{i_1}, p_{i_2}, \ldots, p_{i_k}$ are items in *BIG* and α is the sum of profits corresponding to items in *SMALL*.

Let $c_{i_1}, \ldots, c_{i_k}$ be the associated weights of $p_{i_1}, \ldots, p_{i_k}$. Consider $p'_{i_1} + \cdots + p'_{i_k}$. Since $p'_{i_1} + \cdots + p'_{i_k}$ is not unique, there may be another set of items, namely items $j_1, j_2, \ldots, j_h$, such that $p'_{i_1} + \cdots + p'_{i_k} = p'_{j_1} + \cdots + p'_{j_h}$ and $c_{j_1} + \cdots + c_{j_h} \leq c_{i_1} + \cdots + c_{i_k}$. In other words, our approximation algorithm will choose items $j_1, j_2, \ldots, j_h$, instead of $i_1, i_2, \ldots, i_k$.

The sum $P_H = p_{j_1} + \cdots + p_{j_h} + \beta$ must have been considered by the algorithm during the execution of Step 6.

Obviously, by the choice of P_{APX},

$$P_{APX} \geq P_H.$$

Therefore,

$$P_{OPT} - P_{APX} \leq P_{OPT} - P_H.$$

By Claim 2, we have

$$p'_i\delta \leq p_i \leq p'_i\delta(1 + \varepsilon/3).$$

Substituting this into the formulas $P_{OPT} = p_{i_1} + \cdots + p_{i_k} + \alpha$ and $P_H = p_{j_1} + \cdots\ p_{j_h} + \beta$, we have

$$(p'_{i_1} + \cdots + p'_{i_k})\delta + \alpha \leq P_{OPT} \leq (p'_{i_1} + \cdots + p'_{i_k})\delta(1 + \varepsilon/3) + \alpha$$

and $(p'_{j_1} + \cdots + p'_{j_h})\delta + \beta \leq P_H \leq (p'_{j_1} + \cdots + p'_{j_h})\delta(1 + \varepsilon/3) + \beta.$

Since $p'_{i_1} + \cdots + p'_{i_k} = p'_{j_1} + \cdots + p'_{j_h}$, we have

$$P_{OPT} \le (p'_{i_1} + \cdots + p'_{i_k})\delta(1 + \varepsilon/3) + \alpha$$

and $$P_H \ge (p'_{i_1} + \cdots + p'_{i_k})\delta + \beta.$$

Consequently, we have

$$\begin{aligned}(P_{OPT} - P_H)/P_{OPT} &\le ((p'_{i_1} + \cdots + p'_{i_k})\delta(\varepsilon/3) + \alpha - \beta)/P_{OPT} \\ &\le \varepsilon/3 + (\alpha - \beta)/P_{OPT}. \end{aligned} \quad \textbf{(9–3)}$$

We have to prove further that

$$|\alpha - \beta| \le Q(\varepsilon/3).$$

This can be reasoned as follows:

α is the sum of profits with respect to the 0/1 knapsack problem defined on items of *SMALL* with capacity $M - (c_{i_1} + \cdots + c_{i_k})$.

β is the sum of profits obtained by the greedy method with respect to the 0/1 knapsack problem defined on items of *SMALL* with capacity $M - (c_{j_1} + \cdots + c_{j_h})$. Let us define β' to be the sum of profits obtained by the greedy method with respect to the 0/1 knapsack problem defined on items of *SMALL* with capacity $M - (c_{i_1} + \cdots + c_{i_k})$.

Case 1: $\alpha \ge \beta$

Since we have assumed $c_{i_1} + \cdots + c_{i_k} \ge c_{j_1} + \cdots + c_{j_h}$,

$$M - (c_{i_1} + \cdots + c_{i_k}) \le M - (c_{j_1} + \cdots + c_{j_h}).$$

Moreover, β and β' are obtained by the greedy algorithm applied on *SMALL*, which is sorted by non-increasing order. Hence, $M - (c_{i_1} + \cdots + c_{i_k}) \le M - (c_{j_1} + \cdots + c_{j_h})$ implies $\beta' \le \beta$

Therefore,

$$\alpha - \beta \le \alpha - \beta'.$$

By Claim 3, we also have

$$\alpha - \beta' \le \max\{p_i \mid i \in SMALL\}.$$

Together with the property that every item in *SMALL* has profit less than or equal to $Q(\varepsilon/3)$, we obtain

$$\alpha - \beta \leq Q(\varepsilon/3).$$

Case 2: $\alpha \leq \beta$
Since

$$(p'_{j_1} + \cdots + p'_{j_h})\delta + \beta \leq P_H \leq P_{OPT} \leq (p'_{i_1} + \cdots + p'_{i_k})\delta(1 + \varepsilon/3) + \alpha.$$

Therefore, $\beta - \alpha \leq (p'_{i_1} + \cdots + p'_{i_k})\delta(\varepsilon/3)$
$\leq (p_{i_1} + \cdots + p_{i_k})(\varepsilon/3)$
$\leq P_{OPT} \cdot \varepsilon/3$
$\leq Q(\varepsilon/3).$

Considering Case 1 and Case 2, we have

$$|\alpha - \beta| \leq Q(\varepsilon/3). \qquad \textbf{(9–4)}$$

Substituting (9–4) into (9–3), we have

$$(P_{OPT} - P_H) \leq \varepsilon/3 + (\varepsilon/3)Q/P_{OPT}.$$

By the choice of Q,

$$P_{OPT} \geq Q/2.$$

Together with the inequality that

$$P_{APX} \geq P_H,$$

we have

$$(P_{OPT} - P_{APX})/P_{OPT} \leq \varepsilon/3 + 2\varepsilon/3 = \varepsilon.$$

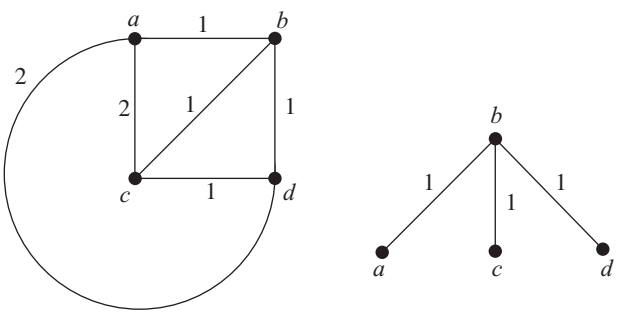
The above equation shows that this approximation algorithm is a polynomial time approximation scheme.

9–10 A 2-APPROXIMATION ALGORITHM FOR THE MINIMUM ROUTING COST SPANNING TREE PROBLEM

The minimum routing cost spanning tree problem (MRCT for short) is similar to the minimum spanning tree problem. In the minimum spanning tree problem, the total cost of all edges on the spanning tree is to be minimized. In the minimum routing cost spanning tree problem, we are interested in the routing cost. For any two nodes u and v on a tree, there is a path between them. The total cost of all edges on this path is called the routing cost of this pair of nodes. In our case, we further stipulate that our given graph is a complete graph and all the edge costs satisfy the triangular inequality. Our minimum routing cost spanning tree problem is thus defined as follows: We are given a complete graph G with edge costs satisfying the triangular inequality, and the minimum routing cost spanning tree problem is to find a spanning tree of G whose total sum of all pairs of routing costs between nodes is minimized.

Consider Figure 9–46. A minimum routing cost spanning tree of the complete graph in Figure 9–46(a) is shown in Figure 9–46(b).

FIGURE 9–46 A minimum routing cost spanning tree of a complete graph.



Let $RC(u, v)$ denote the routing cost between u and v on the tree. The total sum of all pairs of routing costs of this tree is

$$\begin{aligned} &2(RC(a, b) + RC(a, c) + RC(a, d) + RC(b, c) + RC(b, d) + RC(c, d)) \\ &= 2(1 + 2 + 2 + 1 + 1 + 2) \\ &= 18, \end{aligned}$$

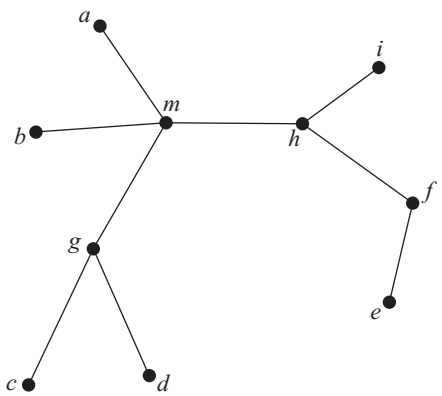
which is minimum among all possible spanning trees.

The minimum routing cost spanning tree problem is NP-hard. In this section, we shall give a 2-approximation algorithm for this problem. In the next section, we shall further introduce a PTAS for this problem.

Before introducing the algorithm, let us point out that we count every pair of nodes twice. That is, the path from u to v and the path from v to u will both be counted. We do this to simplify our discussion.

Our 2-approximation algorithm is based on a term called centroid. A centroid of a tree is a node whose deletion will result in subgraphs such that each subgraph contains no more than $n/2$ nodes, where n is the total number of nodes of the tree. Consider Figure 9–47. Node m is a centroid.

FIGURE 9–47 The centroid of a tree.

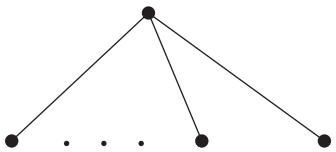


Let us consider any node v in a T rooted by a centroid m of T. By definition, every subtree of m contains no more than $n/2$ nodes. Consider any node v in T. The subtree containing v must contain no more than $n/2$ nodes. This means that there will be at least $n/2$ paths between some node u and v which will pass through m. For instance, consider node e of the tree in Figure 9–47. Only the paths between e and f, h and i will not pass through centroid m. The paths of all other nodes to e will pass through m. Therefore, in the routing cost of this tree, the length of the path from any node u to m will be counted at least $2(n/2) = n$ times. Using $RC(u, v)$ as the routing cost of the path between nodes u and v, the cost of the tree $C(T)$ is

$$C(T) \geq n \sum_{u} RC(u, m). \qquad \textbf{(9–5)}$$

Suppose that the tree T is a minimum routing cost spanning tree and m is a centroid of T. Then the above equation is still valid. Now, we can use m to obtain an approximation algorithm. Let us define a 1-star as a tree with only one internal node and all other nodes as leaf nodes, shown in Figure 9–48.

FIGURE 9–48 A 1-star.



Our approximation algorithm is to connect all nodes to m to form a 1-star S, which is also a spanning tree. Let $w(v, m)$ denote the weight of the edge between v and m in the original graph G. The total sum of the routing costs of this star, denoted as $C(S)$, is as follows:

$$C(S) = (2n - 3)\sum_{v} w(v, m). \tag{9–6}$$

But $w(v, m)$ is less than $RC(v, m)$ of T because of the triangular inequality. Thus, we have

$$C(S) = 2n\sum_{v} RC(v, m) \tag{9–7}$$

This means that

$$C(S) \leq 2C(T). \tag{9–8}$$

Thus, the 1-star S constructed is a 2-approximation solution for the minimum routing cost spanning tree problem. Note that we start from a minimum routing cost spanning tree. But we do not have any minimum routing cost spanning tree. If we do, we would not need any approximation algorithm for the problem.

How can we find the centroid m of that tree without the tree? Note that given a complete graph G, there are only n 1-stars. Thus, we can conduct an exhaustive search to construct all possible n 1-stars and the lowest cost one is selected as our approximate solution. This 1-star must satisfy our requirement. The time complexity of this approximation algorithm is $O(n^2)$.

Perhaps the reader should note that we have just proved the existence of a 1-star satisfying our requirement. The exhaustive search may well find a 1-star whose routing cost is smaller than that of the one we discussed above.

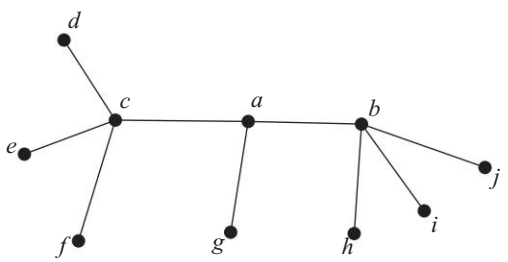
9–11 A PTAS FOR THE MINIMUM ROUTING COST SPANNING TREE PROBLEM

In the above section, we introduced a 2-approximation algorithm for the minimum routing cost spanning tree problem. Our approximation algorithm is based on the idea that for every graph G, there is a 1-star whose routing cost is at most two times the routing cost of the minimum routing cost spanning tree of this graph. Since there are only n 1-stars, we can find an approximate solution in polynomial number of steps.

In this section, we shall introduce a PTAS for the minimum routing cost spanning tree problem. Essentially, we construct k-stars and the larger k is, the smaller the error.

A k-star is a tree with exactly k internal nodes. Figure 9–49 shows a 3-star.

FIGURE 9–49 A 3-star.



In the rest of this section, we shall denote $mcks(G, k)$ as the routing cost of a minimum routing cost k-star of a graph G. Let $mrcst(G)$ denote the routing cost of a minimum routing cost spanning tree of G. Then we will prove the following:

$$mcks(G, k) \leq \left(\frac{k+3}{k+1}\right) mrcst(G). \qquad \textbf{(9–9)}$$

When $k = 1$, we are constructing a 1-star and the above equation becomes:

$$mcks(G, 1) \leq 2mrcst(G) \qquad \textbf{(9–10)}$$

which was proved in the above section.

Equation (9–9) shows that the error ratio of using a k-star to approximate an optimal solution is

$$E = \frac{2}{k+1}. \tag{9–11}$$

For $k = 2$, the error rate is 0.66 and for $k = 5$, the error ratio is reduced to 0.33. Given an error bound, we can select a k by the following equation:

$$k = \left\lceil \frac{2}{E} - 1 \right\rceil. \tag{9–12}$$

Thus, a larger k means a smaller error ratio. For a given prespecified error ratio bound, we can pick a corresponding k large enough to ensure that the error induced by this k-star does not exceed the prespecified error bound. It can be shown that the time complexity required to find a minimum routing cost k-star of a given complete graph is $O(n^{2k})$. For each k, no matter how large it is, the time complexity of our approximation algorithm is still polynomial. That is, we have a PTAS for the minimum routing cost spanning tree problem.

To find a minimum routing cost k-star, we need a concept, called δ-separator where $0 < \delta \leq 1/2$. Given a graph G, a δ-separator of G is a minimum subgraph of G whose deletion will result in subgraphs, each of which contains no more than δn nodes. For $\delta = 1/2$, the δ-separator contains only one point, namely the centroid, which we introduced in the last section. It can be shown that there is a relationship between δ and k as follows:

$$\delta = \frac{2}{k+3}, \tag{9–13}$$

or, conversely,

$$k = \frac{2}{\delta} - 3. \tag{9–14}$$

Substituting (9–14) into (9–9), we obtain

$$mcks(G, k) \leq \left(\frac{1}{1-\delta} \right) mrcst(G). \tag{9–15}$$

Essentially, our reasoning proceeds as follows: Once we have an error rate bound E, we select a k through Equation (9–12) and then determine a δ through Equation (9–13). Through the δ-separator, we can determine a k-star satisfying our error bound requirement. Let us assume that we specify E to be 0.4. Then, using Equation (9–12), we select $k = (2/0.4) - 1 = 4$ and find $\delta = 2/(4 + 3) = 2/7 = 0.28$.

In the following, we shall first discuss the case where $k = 3$ to illustrate the basic concept of the PTAS. In such a case, through Equation (9–13), we have $\delta = 2/(3 + 3) = 1/3$.

Let us assume that we have already found a minimum routing cost spanning tree T. Without losing generality, we may assume that T is rooted at its centroid m. There are at most two subtrees containing more than $n/3$ nodes. Let nodes a and b be the lowest nodes with at least $n/3$ nodes. We ignore the special cases where $m = a$ or $m = b$. These two special cases can be handled similarly and yield the same result. Let P denote the path in T from node a to node b. This path must contain m because none of the subtrees of m contains more than $n/2$ nodes. According to our definition, P is a (1/3)-separator. In the following, we shall show how we can construct a minimum routing cost 3-star based upon P. The routing cost of the 3-star will be no more than

$$\frac{k+3}{k+1} = \frac{3+3}{3+1} = \frac{3}{2}$$

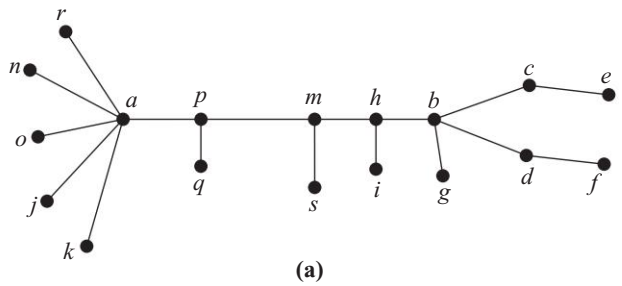
of the routing cost of T.

We first partition all the nodes into V_a, V_b, V_m, V_{am} and V_{bm}. Sets V_a, V_b and V_m consist of nodes whose lowest ancestors on P are a, b and m respectively. Set $V_{am}(V_{bm})$ consists of nodes whose lowest ancestors on P are between a and m (between b and m).

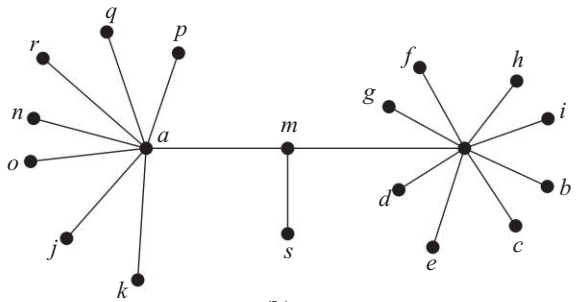
We then replace P by a path with edges (a, m) and (b, m). For each node v in V_a, V_b and V_m, connect v to a, b and m respectively. For all of the nodes in $V_{am}(V_{bm})$, either connect all of them to $a(b)$ or connect all of them to m. Thus, there are four 3-stars and we shall prove that one of them is desirable. A typical case is now shown in Figure 9–50. In Figure 9–50, we show all the four 3-stars.

We now try to find a formula for the routing cost of the minimum routing cost spanning tree. For each node v in this tree, there is a path from v to a node in P. Let the path length from v to that point in P be denoted as $dt(v, P)$. In the total routing cost, this path length must be counted at least $2n/3$ times because P is a (1/3)-separator. For each edge of P, since there are at least $n/3$ nodes on either side of it, the edge is counted at least $(n/3)(2n/3)$ times in the routing cost. Let $w(P)$ denote the total path length of P. Then we have

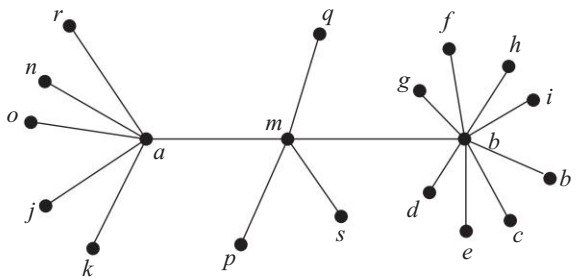
FIGURE 9–50 A tree and its four 3-stars.



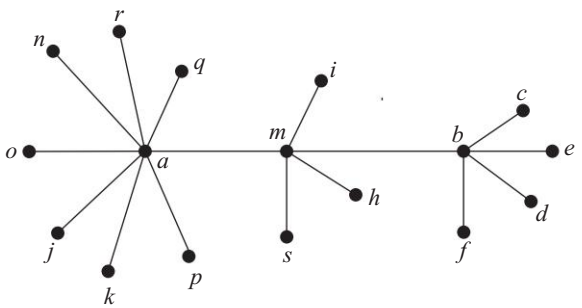
(a)



(b)

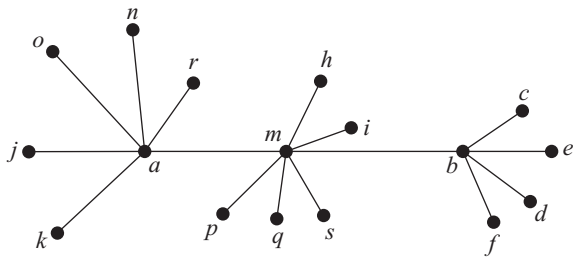


(c)



(d)

FIGURE 9–50 (*cont'd*)



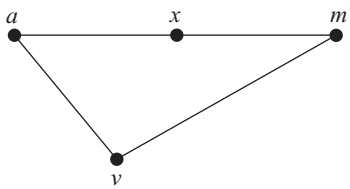
(e)

$$mrcst(G) \geq (2n/3) \sum_{v \notin P} dt(v, P) + (2/9)n^2 w(P). \quad \textbf{(9–16)}$$

Now, let us analyze the routing cost of the 3-star we constructed.

(1) For each edge (v, a), (v, b) and (v, m) in the final routing cost, it will be counted $(n - 1)$ times. For each node v in $V_a \cup V_b \cup V_m$, the only thing we can say is that the weight of the edge is not more than $dt(v, P)$.
(2) For each node in $V_{am} \bigcup V_{bm}$, we can say more. Note that there are four possible 3-stars. For all nodes in V_{am}, they are either all connected to node a or to node m. We do not know which case is better. Consider Figure 9–51. Assume that v was originally connected to a node x between a and m. Then, we have the following two inequalities:

FIGURE 9–51 The connection of a node v to either a or m.



$$w(v, a) \leq w(v, x) + w(a, x). \quad \textbf{(9–17)}$$

$$w(v, m) \leq w(v, x) + w(x, m). \quad \textbf{(9–18)}$$

Because $w(v, x) \leq dt(v, P)$ and $w(a, x) + w(x, m) \leq dt(a, m)$, by summing (9–17) and (9–18), we have

$$w(v, a) + w(v, m) \le 2dt(v, P) + dt(a, m)$$

$$\text{or} \quad \frac{w(v, a) + w(v, m)}{2} \le dt(v, P) + \frac{dt(a, m)}{2}. \tag{9–19}$$

Furthermore, we have the following:

$$\sum_v w(v, a) + \sum_v w(v, m) = \sum_v (w(v, a) + w(v, m)). \tag{9–20}$$

Thus, either $\sum_v w(v, a)$ or $\sum_v w(v, m)$ is smaller than $\sum_v \frac{w(v, a) + w(v, m)}{2}$. There are at most $n/6$ nodes in V_{am} or V_{bm}. Let v be a node in $V_{am} \cup V_{bm}$. The above reasoning leads us to the conclusion that in one of the four 3-stars, the sum of weights of edges from all nodes in $V_{am} \bigcup V_{bm}$ is not more than the following:

$$\frac{n}{6} \sum_v \left(dt(v, P) + \frac{1}{2}(dt(a, m) + dt(b, m)) \right).$$

(3) For each edge (a, m) or (b, m), it is counted no more than $(n/2)(n/2) = (n^2/4)$ times in the final routing cost.

In summary, in one of the four 3-stars, its routing cost, denoted as RC(3-star, G) satisfies the following equation:

$$RC(\text{3-star}, G)$$

$$\le (n-1) \sum_v dt(v, P) + \left(\frac{n}{12}\right)(dt(a, m) + dt(b, m)) + \left(\frac{1}{4}\right) n^2 w(P)$$

$$\le n \sum_v dt(v, P) + \left(\frac{1}{3}\right) n^2 w(P).$$

Using (9–16), we have:

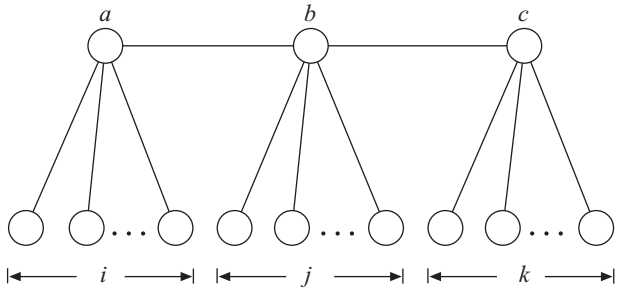
$$RC(\text{3-star}, G) \le \frac{3}{2}\, mrcst(G).$$

Thus, we have proved that for error ratio = 1/2, there exists a 3-star whose routing cost is not more than 3/2 of the routing cost of a minimum routing cost spanning tree.

We have proved the existence of a 3-star as mentioned above. The problem is how to find it. This will be explained next.

Let us imagine that among all the n nodes, we pick three nodes, denoted as a, b and c. These three nodes will be the only internal nodes of our 3-star. Imagine that we are also given a 3-tuple (i, j, k) where i, j and k are all positive integers and $i + j + k = n - 3$. This means that we are going to connect i nodes to a, j nodes to b and k nodes to c, shown in Figure 9–52.

FIGURE 9–52 A 3-star with $(i + j + k)$ leaf nodes.



The question is: Which i nodes should be connected to a, which j nodes should be connected to b and which k nodes should be connected to c? To solve this problem, we must solve a bipartite graph matching problem. In a bipartite graph, there are two sets of nodes, denoted as X and Y. In our case, our original input graph is $G(V, E)$. For the bipartite graph, $X = V - \{a, b, c\}$ and Y contains i copies of node a, j copies of node b, and k copies of node c. The weight of the edge between a node in X and a node in Y can be found in the original input distance matrix. We solve the minimum perfect matching problem on this bipartite graph. If a node v in X is matched with a node u in Y, v will be connected to u in the 3-star. This 3-star is the best 3-star once nodes a, b and c and integers i, j and k have been determined. The total routing cost of this 3-star can be found rather easily. The edge between a leaf node and an internal node is counted exactly $(n - 1)$ times. The edge between a and b is counted $i(j + k)$ times and the edge between b and c is counted $k(i + j)$ times. Each edge between a leaf node and an internal node is counted $(n - 1)$ times.

Our algorithm for finding a 3-star whose total routing cost is not more than 3/2 of a minimum cost routing tree is now given as follows:

Algorithm 9–10 □ An algorithm which produces a 3-star whose total routing cost is not more than 3/2 of that of a minimal routing cost tree

Input: A complete graph $G(V, E)$ with weights on all edges and all the weights satisfying the triangular inequality.

Output: A 3-star whose total routing cost is not more than 3/2 of that of a minimal routing cost tree of G.

Let $RC = \infty$.

For all (a, b, c) where a, b and c are selected from V, do

For all (i, j, k) where $i + j + k = n - 3$ and i, j and k are all positive integers, do

Let $X = V - \{a, b, c\}$ and Y contain i copies of a, j copies of b and k copies of c.

Perform a perfect minimum bipartite matching between X and Y.

If a node v is connected to a, b or c in the matching, connect this node v to a, b or c respectively. This creates a 3-star.

Compute the total routing cost Z of this 3-star.

If Z is less than RC, let $RC = Z$.

Let the present 3-star be denoted as the best 3-star.

The time complexity of the above algorithm is analyzed as follows: There are $O(n^3)$ possible ways to select a, b and c. There are $O(n^2)$ possible ways to select i, j and k. The perfect minimum bipartite matching problem can be solved in $O(n^3)$ time. Therefore, the time complexity needed to find an appropriate 3-star is $O(n^8)$.

A desirable k-star can be found in a similar way. We shall not get into the details of this. In general, it takes $O(n^{2k+2})$ time to find a k-star whose total routing cost is not more than $(k + 3)/(k + 1)$ of that of the minimum routing cost tree. Note that $k = \left\lceil \frac{2}{E} - 1 \right\rceil$. Thus, no matter how small E is, we always have a polynomial approximation algorithm which produces a k-star with error ratio within this error bound. In summary, there is a PTAS for the minimum routing cost tree problem.

9–12 NPO-COMPLETENESS

In Chapter 8, we discussed the concept of NP-completeness. If a decision problem is NP-complete, it is highly unlikely that it can be solved in polynomial time.

In this section, we shall discuss NPO-completeness. We shall show that if an optimization problem is NPO-complete, then it is highly unlikely that there exists a polynomial approximation algorithm which produces an approximate solution of this problem with a constant error ratio. In other words, the class of NPO-complete problems denote a class of optimization problems which is unlikely to have good approximation algorithms.

First, we define the concept of NPO problems. Remember that the set NP consists of decision problems. *The set NPO (non-deterministic polynomial optimization) consists of optimization problems*. Since we are now dealing with optimization problems, we are interested in feasible solutions, which are optimal. If an optimization problem is in NPO, then a feasible solution of this problem can be found by two stages, namely guessing and checking. Assume that this problem does have a feasible solution. Then the guessing stage always correctly locates it. We of course also assume that the checking takes polynomial number of steps. It is important to note that the guessing stage only produces a feasible solution, which is not necessarily optimal.

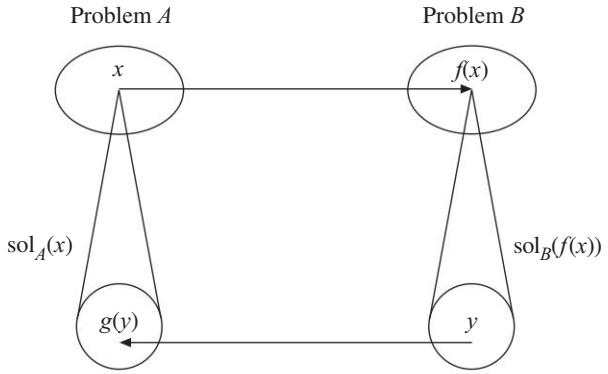
For example, we can see that the traveling salesperson problem is an NPO problem because it is an optimization problem and we can always guess a tour and output the length of this tour. Of course, there is no guarantee that this guessed output is an optimal one.

Now, we shall define the concept of NPO-completeness. First, we shall define a reduction, called strict reduction, which is illustrated in Figure 9–53 and defined as follows:

Given NPO problems A and B, (f, g) is a strict reduction from A to B if

(1) *For every instance x in A, $f(x)$ is an instance in B.*
(2) *For every feasible solution y to $f(x)$ in B, $g(y)$ is a feasible solution in A.*
(3) *The absolute error of $g(y)$ to the optimal of x is less than or equal to the absolute error of y to the optimal of $f(x)$ in B. That is* $|g(y) - OPT_A(x)| \leq |y - OPT_B(f(x))|$.

FIGURE 9–53 The concept of strict reduction.



Having defined NPO problem and strict reduction, we can now define NPO-complete problems. *An NPO problem is NPO-complete if all NPO problems strictly reduce to it.*

It can be seen that, if problem A strictly reduces to problem B, and B has an approximation algorithm whose error relative to the optimal is smaller than ε, then we can use it to construct an approximation algorithm of A whose error is guaranteed to be smaller than ε. Therefore, if an optimization problem A strictly reduces to an optimization problem B, then the fact that problem B has an approximation algorithm with a constant performance ratio will imply that problem A also has an approximation algorithm with a constant performance ratio. Hence, if an NPO-complete problem has any constant-ratio approximation algorithm, then all NPO problems have constant-ratio approximation algorithms. Thus, the hardness of problem A is established.

In Chapter 3, we showed the first formal NP-complete problem: the satisfiability problem. In this section, we shall introduce the weighted satisfiability problem and we shall also describe why the weighted satisfiability problem is NPO-complete.

The minimum (maximum) weighted satisfiability problem is defined as follows: We are given a Boolean formula BF, where each variable x_i is associated with a positive weight $w(x_i)$. Our objective is to find a truth assignment to the variables which satisfies BF and minimizes (maximizes)

$$\sum_{x_i \text{ is true}} w(x_i).$$

We shall omit the formal proof of the NPO-completeness of the weighted satisfiability problem. Just as we did in Chapter 8, we shall give many examples to show how the optimization problems can be strictly reduced to the weighted satisfiability problem.

For example, consider the problem of finding the maximum between two numbers a_1 and a_2. In the strict reduction for this case, $f(x)$ is as follows: For a_1 (and a_2), we associate $x_1(x_2)$ with $a_1(a_2)$ and $f(x)$ maps a_1 (and a_2) into a problem instance of the maximum weighted satisfiability problem as follows:

BF: $(x_1 \vee x_2) \wedge (-x_1 \vee -x_2)$.

Let $w(x_i) = a_i$, for $i = 1, 2$. It can be easily seen that for each truth assignment satisfying BF, exactly one variable will be true. If $a_1(a_2)$ is maximum, then the truth assignment will assign $x_1(x_2)$ to be true.

The function $g(x)$ is defined as

$g(x_1, -x_2) = a_1$
$g(-x_1, x_2) = a_2$.

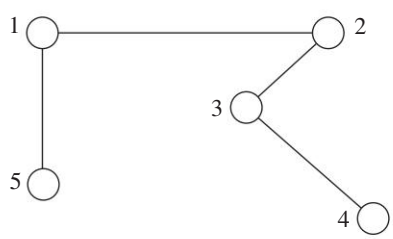
One can see that $f(x)$ and $g(x)$ constitute a strict reduction between the maximization problem and the maximum weighted satisfiability problem.

Example: The Maximum Independent Set Problem

The maximum independent set problem is defined as follows: Given a graph $G = (V, E)$, find $V' \subseteq$ V such that V' is independent and $|V'|$ is maximized. A vertex set S is said to be "independent" if for all $u, v \in S$, the edge $(u, v) \notin E$.

Consider the graph in Figure 9–54.

FIGURE 9–54 An example for maximum independent set problem.



In the strict reduction for this case, $f(x)$ is as follows:

1. For each vertex i, who has neighbors $j_1, j_2, \ldots, j_k$, f maps it to a clause $x_i \rightarrow (-x_{j_1} \,\&\, -x_{j_2} \,\&\, \cdots \,\&\, -x_{j_k})$.
2. $w(x_i) = 1$ for each i.

The instance in Figure 9–54 is transformed to

$$\begin{aligned}\phi = &(x_1 \rightarrow (-x_5 \,\&\, -x_2)) \\ &\&\, (x_2 \rightarrow (-x_1 \,\&\, -x_3)) \\ &\&\, (x_3 \rightarrow (-x_2 \,\&\, -x_4)) \\ &\&\, (x_4 \rightarrow (-x_3)) \\ &\&\, (x_5 \rightarrow (-x_1)).\end{aligned}$$

The function g is defined as: $g(X) = \{i \mid x_i \in \mathrm{X} \text{ and } x_i \text{ is true}\}$.

It can be seen that for each truth assignment X satisfying ϕ, $g(X)$ is an independent set. The strict reduction between the maximum independent set and the maximum weighted satisfiability problem is thus established.

Examples of Proving NPO-Completeness

From the above examples, the readers ought to be convinced that we can always successfully transform any NPO problem to the weighted satisfiability problem (WSAT). This is essentially what we mean by "WSAT is NPO-complete", i.e., all NPO problems can be strictly reduced to it.

Now we shall prove that some problems are NPO-complete. We would like to remind the reader that when we want to prove that a problem A is NPO-complete, we usually do this in two steps:

(1) We first prove that A is an NPO problem.
(2) We then prove that some NPO-complete problem strictly reduces to A.

We note that up to now, we only have an NPO-complete problem: the weighted satisfiability problem. To produce more NPO-complete problems, we must start from it. That is, we should try to prove that the weighted satisfiability problem strictly reduces to the problem which we are interested in.

Example: Zero-One Integer Programming

A zero-one integer programming problem is defined as follows: Given an $m * n$ integer matrix A, integer m-vector b, positive integer n-vector c, we are going to find a zero-one n-vector x that satisfies $Ax \geq b$ and minimizes cx.

Certainly, it is easy to see that this problem is an NPO problem. For any nondeterministic guessed solution X, it can be tested in polynomial time whether $Ax \geq b$, and the measure cx can also be computed in polynomial time.

Before proving the NPO-completeness of the zero-one integer programming problem, we shall first introduce the maximum, or minimum, weighted 3-satisfiability problem (W3SAT). The maximum (minimum) weighted 3-satisfiability problem is defined as follows: Given a Boolean formula ϕ consisting of a conjunction of clauses where each clause contains exactly three literals and a positive integer weight $w(x_i)$, $i = 1, 2, 3$, associated with x_i, find an assignment $\tau(x_i)$ satisfying ϕ, if it exists, which maximizes (minimizes)

$$\sum_{\tau(x_i)\ is\ true} w(x_i).$$

Let $w(x_1) = 3$, $w(x_2) = 5$ and $w(x_3) = 2$. Consider the formula

$$(-x_1 \vee -x_2 \vee -x_3)$$
$$\& (-x_1 \vee -x_2 \vee -x_3)$$
$$\& (-x_1 \vee -x_2 \vee x_3)$$
$$\& (x_1 \vee x_2 \vee x_3).$$

There are four assignments satisfying the above formula:

$(x_1, -x_2, x_3)$ with weight 5
$(x_1, -x_2, -x_3)$ with weight 3
$(-x_1, x_2, x_3)$ with weight 7
and $(-x_1, x_2, -x_3)$ with weight 5.

The assignment $(-x_1, x_2, x_3)((x_1, -x_2, -x_3))$ is the solution for the maximum (minimum) W3SAT problem. The maximum and minimum W3SAT problems are both NPO-complete. To prove that zero-one integer programming problem is NPO-complete, we shall show that the minimum W3SAT can strictly reduce to it.

The function f transforming minimum W3SAT to the zero-one integer programming problem is as follows:

(1) Each variable x in minimum W3SAT corresponds to a variable x in the zero-one integer programming problem.
(2) Variable value 1 represents "TRUE" and value 0 represents "FALSE".
(3) Each clause in W3SAT is transformed to an inequality. The "or" operator is transformed to be "+", and $-x$ is transformed to be $1 - x$. Since the whole clause must be true, we have it to be "≥1".
(4) Let $w(x_i)$, $i = 1, 2, 3$ denote the weight of variable x_i. Then we minimize the following function:

$$\sum_{i=1}^{3} w(x_i)x_i.$$

The minimum W3SAT problem $(x_1 \vee x_2 \vee x_3)\ \&\ (-x_1 \vee -x_2 \vee x_3)$ with $w(x_1) = 3$, $w(x_2) = 5$, $w(x_3) = 5$ is transformed to the following zero-one integer programming problem:

$$\text{Minimize} \quad 3x_1 + 5x_2 + 5x_3, \text{ where } x_i = 0, 1,$$
$$\text{subject to} \quad x_1 + x_2 + x_3 \geq 1,\ 1 - x_1 + 1 - x_2 + x_3 \geq 1.$$

The truth assignment $(-x_1, -x_2, x_3)$ which is a solution of the minimun W3SAT problem corresponds to the vector (0, 0, 1) which is a solution for the zero-one integer programming problem. Thus, we have shown that the weighted 3-satisfiability problem strictly reduces to the zero-one integer programming problem. Hence, the zero-one integer programming problem is NPO-complete.

Example: The Longest and Shortest Hamiltonian Cycle Problems on Graphs

Given a graph, a Hamiltonian cycle is a cycle which traverses all vertices of the graph exactly once. The longest (shortest) Hamiltonian cycle problem on graphs is defined as follows: Given a graph, find a longest (shortest) Hamiltonian cycle of the graph. Both problems have been found to be NP-hard. The shortest Hamiltonian cycle problem is also called the traveling salesperson problem. In the following, we shall prove the NPO-completeness of both problems. It is rather obvious that both problems are NPO problems. The NPO-completeness can be established by showing that the maximum, or minimum, W3SAT problem strictly reduces to it.

First, given a W3SAT problem instance ϕ we shall show the following:

1. There is a function f which maps ϕ to a graph $f(\phi)$ such that the W3SAT is satisfiable if and only if $f(\phi)$ has a Hamiltonian cycle.
2. There is a function g which maps a Hamiltonian cycle C in graph $f(\phi)$ to a satisfying truth assignment $g(C)$ for ϕ.
3. The cost of C is the same as the cost of $g(C)$.

The last property ensures that the absolute error of $g(C)$ to the optimal solution of ϕ is equal to the absolute error of C to the optimal solution of $f(\phi)$. That is, $|g(C) - OPT(\phi)| = |C - OPT(f(\phi))|$. Thus, (f, g) is a strict reduction.

We now illustrate the transformation of a W3SAT problem instance to a graph.

(1) For each clause C_i, there will be a C_i-component, shown in Figure 9–55. The edges e_1, e_2, e_3 correspond to the three literals in clause C_i. For example, if C_i contains $-x_1$, x_2 and $-x_3$, then e_1, e_2 and e_3 correspond to $-x_1$, x_2 and $-x_3$ respectively. Note that a C_i-component has a property that, for any Hamiltonian path from $c_{i,1}$ to $c_{i,4}$, at least one of the edges e_1, e_2, e_3 will not be traversed, shown in Figure 9–56. We shall explain the meaning of an untraversed edge later.

FIGURE 9–55 A C-component corresponding to a 3-clause.

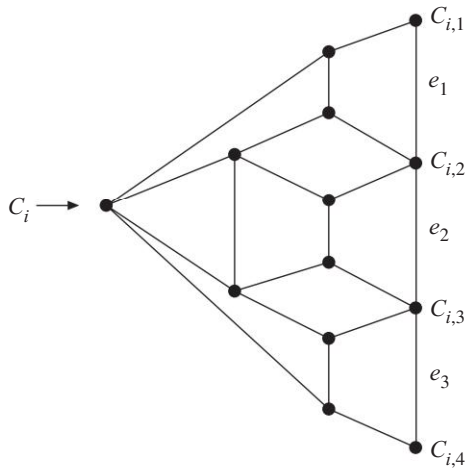
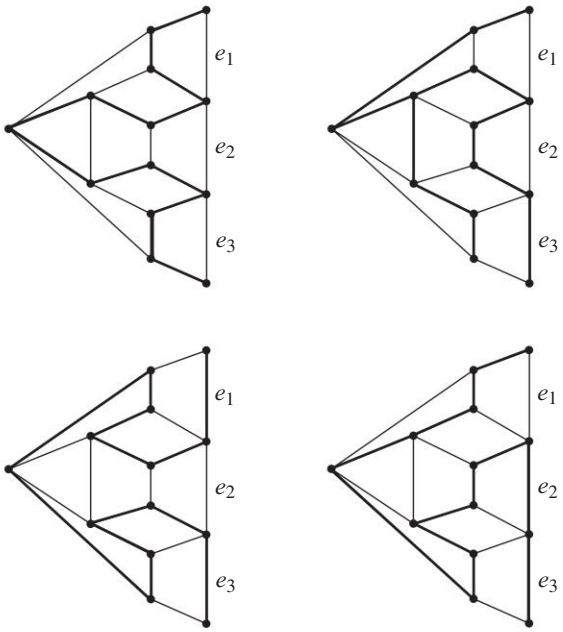
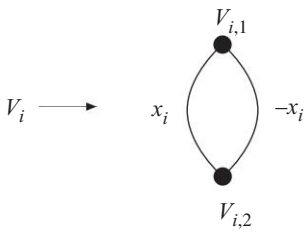


FIGURE 9–56 *C*-component where at least one of the edges e_1, e_2, e_3 will not be traversed.



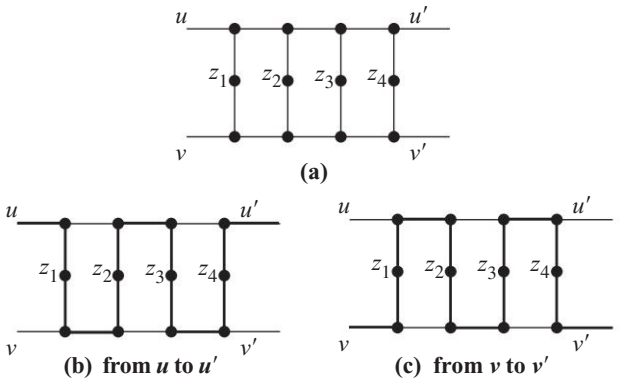
(2) For each variable V_i, we have a V_i-component, shown in Figure 9–57. For a Hamiltonian path, either x_i or $-x_i$ can be traversed.

FIGURE 9–57 The component corresponding to a variable.



(3) A *C*-component will be connected to a *V*-component by a gadget *H*, shown in Figure 9–58. If a graph $G = (V, E)$ contains a gadget $H = (W, F)$ of Figure 9–58(a) in such a way that no vertex in $V - W$ is adjacent to any vertex in $W - \{u, v, u', v'\}$, then it can be seen that any Hamiltonian cycle in G must traverse H in either the one as shown in Figure 9–58(b) or the one in Figure 9–58(c).

FIGURE 9–58 Gadgets.



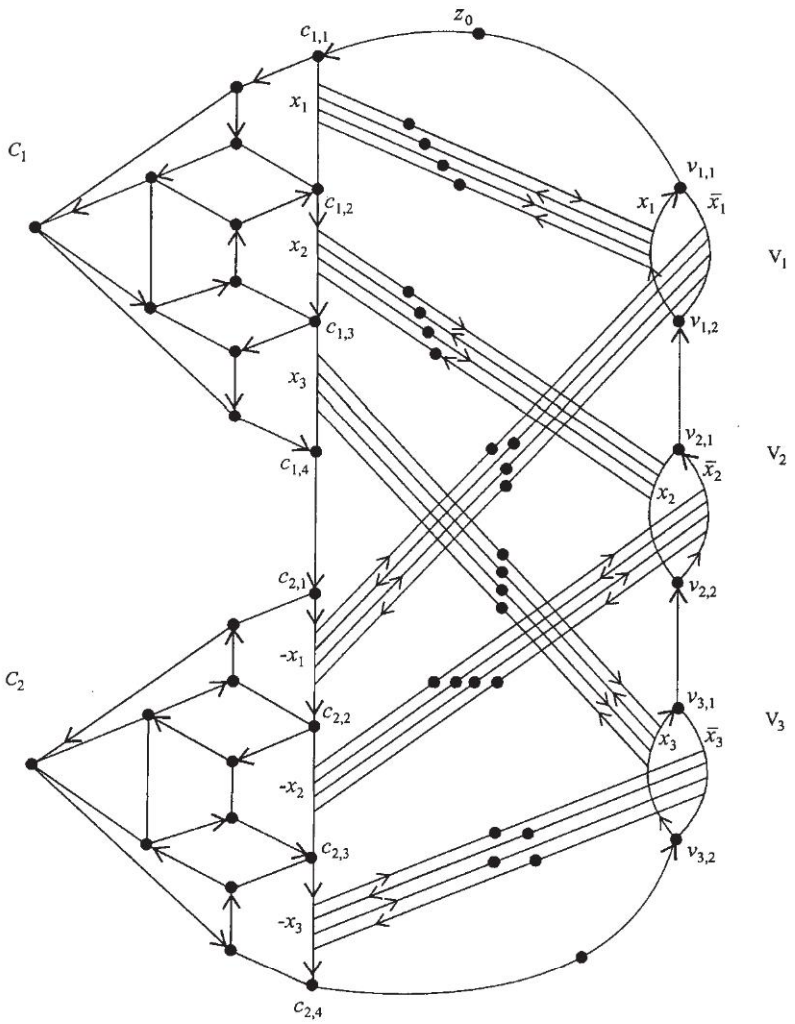
(4) For a given Boolean formula with clauses $C_1, C_2, \ldots, C_m$, the C-components corresponding to these clauses will be connected in series with edges $(c_{i,4}, c_{i+1,1})$ for $i = 1, 2, \ldots, m - 1$.
(5) For a given Boolean formula with variables $x_1, x_2, \ldots, x_n$, their corresponding V_i-components are connected in series with edges $(v_{i,2}, v_{i+1,1})$, $i = 1, 2, \ldots, n - 1$.
(6) There are two special edges, namely $(c_{1,1}, v_{1,1})$ and $(c_{m,4}, v_{n,2})$.
(7) If $x(-x)$ appears in a clause C, then the left (right) edge corresponding to $x(-x)$ in the V-component of variable x will be connected to the edge corresponding to the variable x in the C-component corresponding to clause C.

An example of a Hamiltonian cycle is shown in Figure 9–59. Note that any Hamiltonian cycle must contain the path from $v_{1,1}$ to $c_{1,1}$. From $c_{1,1}$ it has two alternatives, proceeding directly to $c_{1,2}$ or the other way. In our example, it does not traverse the edge corresponding to x_1 in the C_1-component. Throughout the entire cycle, x_1 and x_3 in C_1-component and $-x_2$ in C_2-component are not traversed. This corresponds to the assignment $(x_1, -x_2, x_3)$.

In general, we have the following rule: In a Hamiltonian cycle, if the edge in a C-component corresponding to $x_i(-x_i)$ is not traversed, then assign $x_i(-x_i)$ to be true in the corresponding assignment.

According to the above rule, if the edge corresponding to x_1 in clause C_1 is not traversed in a cycle, we set x_1 to be true in the assignment. The reader may wonder: Will the edge corresponding to $-x_1$ in clause C_2 be traversed? This has to be the case. Otherwise, there will be an inconsistency and we cannot have an

FIGURE 9–59 $(x_1 \vee x_2 \vee x_3)$ and $(-x_1 \vee -x_2 \vee -x_3)$ with the assignment $(-x_1, -x_2, x_3)$.



assignment. That this will be so can be seen by examining the connection of $-x_1$ in the V_1-component. The way of connecting forces the edge corresponding to $-x_1$ to be traversed. It can be proved that for any Hamiltonian cycle in G, if the edge corresponding to $x_i(-x_i)$ in some C_j-component is not traversed, then the edge corresponding to $-x_i(x_i)$ in some C_k-component will be traversed. This means that every Hamiltonian cycle in G corresponds to an assignment for ϕ.

We still have to assign weights to the edges of G. We assign $w(x_i)$ to the left edge of the V_i-component, for $i = 1, 2, \ldots, n$, and 0 to all other edges. It can be proved that the graph G has a Hamiltonian cycle with weight W if and only if the Boolean formula ϕ has a satisfying assignment with weight W.

We have shown that there exists a strict reduction from a maximum, or minimum, W3SAT problem to a longest, or shortest, Hamiltonian cycle problem on graphs. Because of the NPO-completeness of both maximum and minimum W3SAT problems, the NPO-completeness of both longest and shortest Hamiltonian cycle problems on graphs are now established.

9–13 NOTES AND REFERENCES

The NP-hardness of the Euclidean traveling salesperson problem was proved by Papadimitriou (1977). Rosenkrantz, Stearns and Lewis (1977) showed that the Euclidean traveling salesperson problem can be approximated with length less than twice the length of a shortest tour. Christofides (1976) related minimum spanning trees, matchings and the traveling salesperson problem to obtain an approximation algorithm for the Euclidean traveling salesperson problem where the length is within 3/2 of the optimal one. The bottleneck problem discussed in Section 9–3 and Section 9–4 originally appeared in Parker and Rardin (1984) and Hochbaum and Shmoys (1986) respectively. The bin packing approximation algorithm appeared in Johnson (1973). The approximation algorithm for the rectilinear m-center problem was given by Ko, Lee and Chang (1990).

The terminology of NPO-completeness first appeared in Ausiello, Crescenzi and Protasi (1995), but this concept was originally introduced by Orponen and Mannila (1987). In Orponen and Mannila (1987), the zero-one integer programming problem, the traveling salesperson problem and the maximum weighted 3-satisfiability problem were all proved to be NPO-complete. The proof of NPO-completeness of the longest Hamiltonian path problem can be found in Wu, Chao and Lee (1998).

That the traveling salesperson problem can hardly have good approximation algorithms was also proved by Sahni and Gonzalez (1976). They proved that if the traveling salesperson problem has any constant ratio approximations, then NP = P. Note that this is for general cases. There are good approximation algorithms for special cases.

The clever method for finding a PTAS for the 0/1 knapsack problem was discovered by Ibarra and Kim (1975). Baker developed PTAS's for several problems on planar graphs (Baker, 1994). The traveling salesperson problem has PTAS if it is restricted to planar (Grigni, Koutsoupias and Papadimitriou, 1995) or Euclidean (Arora, 1996). The PTAS scheme for the minimum routing cost spanning tree problem was given in Wu,

Lancia, Bafna, Chao, Ravi and Tang (2000). Another interesting PTAS scheme can be found in Wang and Gusfield (1997).

There is another class of problems, denoted as MAX SNP-complete problems, proposed by Papadimitriou and Yannakakis (1991). This class of problems can be expressed in a strict syntactic form (Fagin, 1974), and have constant ratio approximation algorithms for them. It was proved that any problem in this class can have a PTAS if and only if NP = P. Several problems, such as the MAX 3SAT problem, the metric traveling salesperson problem, the maximum leaves spanning tree problem and the unordered labeled tree problems, were shown to be in this class. See Papadimitriou and Yannakakis (1991); Papadimitriou and Yannakakis (1992); Galbiati, Maffioli and Morzenti (1994); and Zhang and Jiang (1994) for more details.

9–14 FURTHER READING MATERIALS

The approximation algorithm about sorting by transpositions can be found in Bafna and Pevzner (1998). For the multiple sequence alignment problem, consult Gusfield (1997). Both Horowitz and Sahni (1978) and Papadimitriou and Steiglitz (1982) have excellent discussions on approximation algorithms. A review of this subject can be found in Garey and Johnson (1976). Many approximation algorithms have been published in recent years. Many of them are related to NP-completeness. We recommend the following papers for further reading: Alon and Azar (1989); Baker and Coffman Jr. (1982); Bruno, Coffman and Sethi (1974); Cornuejols, Fisher and Nemhauser (1977); Friesen and Kuhl (1988); Hall and Hochbaum (1986); Hochbaum and Maass (1987); Hsu (1984); Johnson (1974); Johnson, Demars, Ullman, Garey and Graham (1974); Krarup and Pruzan (1986); Langston (1982); Larson (1984); Mehlhorn (1988); Moran (1981); Murgolo (1987); Nishizeki, Asano and Watanabe (1983); Raghavan (1988); Sahni (1977); Sahni and Gonzalez (1976); Tarhio and Ukkonen (1988); Vaidya (1988) and Wu, Widmayer and Wong (1986).

Latest research results can be found in Agarwal and Procopiuc (2000); Akutsu and Halldorsson (1994); Akutsu and Miyano (1997); Akutsu, Arimura and Shimozono (2000); Aldous (1989); Amir and Farach (1995); Amir and Keselman (1997); Amir and Landau (1991); Arkin, Chiang, Mitchell, Skiena and Yang (1999); Armen and Stein (1994); Armen and Stein (1995); Armen and Stein (1996); Armen and Stein (1998); Arora, Lund, Motwani, Sudan and Szegedy (1998); Arratia, Goldstein and Gordon (1989); Arratia, Martin, Reinert and

Waterman (1996); Arya, Mount, Netanyahu, Silverman and Wu (1998); Avrim, Jiang, Li, Tromp and Yannakakis (1991); Baeza-Yates and Perleberg (1992); Baeza-Yates and Navarro (1999); Bafna and Pevzner (1996); Bafna, Berman and Fujito (1999); Bafna, Lawler and Pevzner (1997); Berman, Hannenhalli and Karpinki (2001); Blum (1994); Blum, Jiang, Li, Tromp and Yannakakis (1994); Bonizzoni, Vedova and Mauri (2001); Breen, Waterman and Zhang (1985); Breslauer, Jiang and Jiang (1997); Bridson and Tang (2001); Cary (2001); Chang and Lampe (1992); Chang and Lawler (1994); Chen and Miranda (2001); Chen (1975); Christos and Drago (1998); Chu and La (2001); Clarkson (1994); Cobbs (1995); Czumaj, Gasieniec, Piotrow and Rytter (1994); Dinitz and Nutov (1999); Drake and Hougardy (2003); Esko (1990); Even, Naor and Zosin (2000); Feige and Krauthgamer (2002); Frieze and Kannan (1991); Galbiati, Maffioli and Morrzenti (1994); Galil and Park (1990); Goemans and Williamson (1995); Gonzalo (2001); Gusfield (1994); Hochbaum and Shmoys (1987); Ivanov (1984); Jain and Vazirani (2001); Jiang and Li (1995); Jiang, Kearney and Li (1998); Jiang, Kearney and Li (2001); Jiang, Wang and Lawler (1996); Jonathan (1989); Jorma and Ukkonen (1988); Kannan and Warnow (1995); Karkkainen, Navarro and Ukkonen (2000); Kececioglu (1991); Kececioglu and Myers (1995a); Kececioglu and Myers (1995b); Kececioglu and Sankoff (1993); Kececioglu and Sankoff (1995); Kleinberg and Tardos (2002); Kolliopoulos and Stein (2002); Kortsarz and Peleg (1995); Krumke, Marathe and Ravi (2001); Landau and Schmidt (1993); Landau and Vishkin (1989); Laquer (1981); Leighton and Rao (1999); Maniezzo (1998); Mauri, Pavesi and Piccolboni (1999); Myers (1994); Myers and Miller (1989); Parida, Floratos and Rigoutsos (1999); Pe'er and Shamir (2000); Pevzner and Waterman (1995); Promel and Steger (2000); Raghavachari and Veerasamy (1999); Slavik (1997); Srinivasan (1999); Srinivasan and Teo (2001); Stewart (1999); Sweedyk (1995); Tarhio and Ukkonen (1986); Tarhio and Ukkonen (1988); Tong and Wong (2002); Trevisan (2001); Turner (1989); Ukkonen (1985a); Ukkonen (1985b); Ukkonen (1990); Ukkonen (1992); Vazirani (2001); Vyugin and V'yugin (2002); Wang and Gusfield (1997); Wang and Jiang (1994); Wang, Jiang and Gusfield (2000); Wang, Jiang and Lawler (1996); Wang, Zhang, Jeong and Shasha (1994); Waterman and Vingron (1994); Wright (1994); Wu and Manber (1992); Wu and Myers (1996) and Zelikovsky (1993).

Exercises

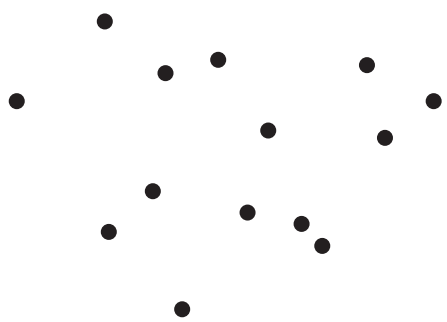
9.1 A simple on-line approximation algorithm for solving the bin packing problem is to put an object into the ith bin as long as it is available and into the $(i + 1)$-th bin if otherwise. This algorithm is called the next-fit (NF) algorithm. Show that the number of bins resulting from this FF algorithm is no more than twice the number of bins needed in an optimal solution.

9.2 Show that if the sequence of items to be considered is $1/2$, $1/2n$, $1/2$, $1/2n$, … , $1/2$ (totally $4n - 1$ items), then the NF algorithm will indeed use nearly twice the number of bins that are really required.

9.3 Show that there does not exist any polynomial time approximation algorithm for the traveling salesperson problem such that the error caused by the approximation algorithm is bounded within $\varepsilon \cdot TSP$ where ε is any constant and TSP denotes an optimal solution.

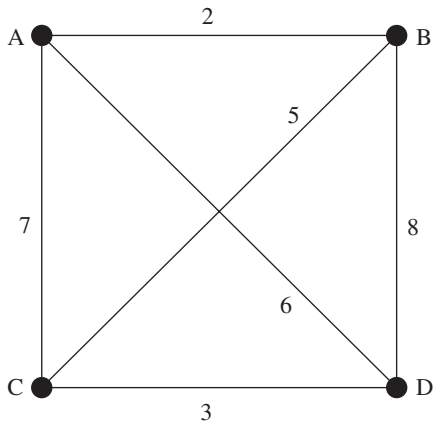
(**Hint:** Show that the Hamiltonian cycle problem can be reduced to this problem.)

9.4 Apply an approximation convex hull algorithm to find an approximate convex hull of the following set of points.

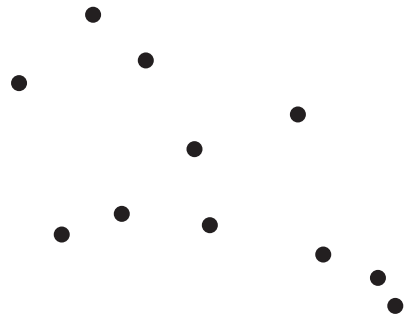


9.5 Let there be a set of points densely distributed on a circle. Apply the approximation Euclidean traveling salesperson algorithm to find an approximate tour for this set of points. Is this result also an optimal one?

9.6 Consider the four points on a square as shown below. Solve the bottleneck traveling salesperson problem approximately by the algorithm introduced in this chapter. Is the result obtained also optimal?



9.7 Use the approximation algorithm for the rectilinear m-center problem to find an approximate solution of the rectilinear 2-center problem for the following set of points.



9.8 Consider the following bottleneck optimization problem. We are given a set of points in the plane and we are asked to find k points such that among these k points, the shortest distance is maximized. This problem is shown to be NP-complete by Wang and Kuo (1988). Try to develop an approximation algorithm to solve this problem.

9.9 Read Section 12.3 of Horowitz and Sahni (1978) about approximation algorithms for scheduling independent tasks. Apply the longest processing time (LPT) rule to the following scheduling problem: There are three processors and seven tasks, where task times are $(t_1, t_2, t_3, t_4, t_5, t_6, t_7) = (14, 12, 11, 9, 8, 7, 1)$.

9.10 Write a program to implement the approximation algorithm for the traveling salesperson problem. Also write a program to implement the branch-and-bound algorithm to find an optimal solution of the traveling salesperson problem. Compare the results. Draw your own conclusion. Is it worthwhile to use such an approximation algorithm?

chapter

10

Amortized Analysis

In this chapter, we are interested in the time complexity analysis of a sequence of operations. Suppose that we consider a sequence of operations $OP_1, OP_2, \ldots, OP_m$ and want to determine the longest possible time that may be consumed by this sequence of operations. One immediate reaction may be to investigate the worst case time complexity of each OP_i. However, it is not always correct to say that the longest possible time of OP_1, $OP_2, \ldots, OP_m$ is the sum of t_i where each t_i is the worst case time spent by OP_i. The reason is rather simple. We cannot expect worst cases to happen so frequently.

To be more specific, let us imagine that in each month we can save money into a bank once and go to a department store to spend money as long as we have money saved in the bank. We can now immediately see that what we do in a certain month depends on what we did in the previous months. If we spend money lavishly in October, we would have to be very thrifty in months before October. Besides, after October, since all the savings are gone, we would have to save again. It will be many months before we can spend money lavishly again.

The above argument illustrates one point. Operations may be interrelated and we cannot assume that they are independent of one another.

The word "amortized" means that we perform many savings now so that we can spend in the future. Why is the word "amortized" related to algorithm analysis? As we shall show later, the word "amortized" implies that in many cases, the actions we perform on our data structure may appear to be wasting our time at this moment. Yet, this action on the data structure may pay back later. That is, after these actions are taken, things may become much easier later. We work hard when we are young; we reap the rewards when we are old.

10–1 AN EXAMPLE OF USING THE POTENTIAL FUNCTION

Potential is a well-known concept for us. In order to utilize the power of water, we must raise the water so that its "potential" is high. Only when the potential is high can we utilize water. The money we save in our bank can also be considered potential. Only after we have saved enough money can we spend.

In this section, we shall show how we can use the concept of potential function to perform some amortized analysis. We consider a sequence of operations $OP_1, OP_2, \ldots, OP_m$ where each OP_i consists of several popping out of elements from a stack and one pushing of an element into this stack. We assume that each pushing and popping takes one time unit. Let t_i denote the time spent by OP_i. Then the total time is

$$T = \sum_{i=1}^{m} t_i$$

and the average time per operation is

$$t_{ave} = \sum_{i=1}^{m} t_i / m$$

and our task is to determine t_{ave}.

The reader can see that t_{ave} is by no means easy to find. Before introducing a method using the potential function concept to find t_{ave}, let us give some examples:

i	1	2	3	4	5	6	7	8
S_1	1 push	1 push	2 pops 1 push	1 push	1 push	1 push	2 pops 1 push	1 pop 1 push
t_i	1	1	3	1	1	1	3	2

$t_{ave} = (1 + 1 + 3 + 1 + 1 + 1 + 3 + 2)/8 = 13/8 = 1.625$

i	1	2	3	4	5	6	7	8
S_2	1 push	1 pop 1 push	1 push	1 push	1 push	1 push	5 pops 1 push	1 push
t_i	1	2	1	1	1	1	6	1

$t_{ave} = (1 + 2 + 1 + 1 + 1 + 1 + 6 + 1)/8 = 14/8 = 1.75$

i	1	2	3	4	5	6	7	8
S_3	1 push	1 push	1 push	1 push	4 pops 1 push	1 push	1 push	1 pop 1 push
t_i	1	1	1	1	5	1	1	2

$t_{ave} = (1 + 1 + 1 + 1 + 5 + 1 + 1 + 2)/8 = 13/8 = 1.625$

The reader must have noted that $t_{ave} \leq 2$ for each case. In fact, t_{ave} is hard to find. But we shall be able to prove that t_{ave} does have an upper bound which is 2. We can do this because after each OP_i, we not only have completed an operation, but have changed the contents of the stack. The number of elements of the stack is either increased or decreased. If it is increased, then the capability that we can execute many pops is also increased. In a certain sense, when we push an element into the stack, we have increased the potential, making it easier to perform many pops. On the other hand, as we pop out an element from the stack, we have somehow decreased the potential.

Let $a_i = t_i + \phi_i - \phi_{i-1}$ where ϕ_i indicates the potential function of the stack after OP_i. Thus $\phi_i - \phi_{i-1}$ is the change of the potential. Then

$$\sum_{i=1}^{m} a_i = \sum_{i=1}^{m} t_i + \sum_{i=1}^{m} (\phi_1 - \phi_{i-1})$$

$$= \sum_{i=1}^{m} t_i + \phi_m - \phi_0 .$$

If $\phi_m - \phi_0 \geq 0$, then $\sum_{i=1}^{m} a_i$ represents an upper bound of $\sum_{i=1}^{m} t_i$. That is,

$$\sum_{i=1}^{m} t_i / m \leq \sum_{i=1}^{m} a_i / m \qquad (1)$$

We now define ϕ_i to be *the number of elements in the stack* immediately after the ith operation. It is easy to see that $\phi_m - \phi_0 \geq 0$ for this case. We assume that before we execute OP_i, there are k elements in the stack and OP_i consists of n pops and 1 push. Thus,

$$\phi_{i-1} = k$$
$$\phi_i = k - n + 1$$

$$
\begin{aligned}
t_i &= n + 1 \\
a_i &= t_i + \phi_i - \phi_{i-1} \\
&= n + 1 + (k - n + 1) - k \\
&= 2.
\end{aligned}
$$

From the above equation, we can easily prove that

$$\sum_{i=1}^{m} a_i / m = 2.$$

Therefore, by (1), $t_{ave} = \sum_{i=1}^{m} t_i / m \le \sum_{i=1}^{m} a_i / m = 2$. That is, $t_{ave} \le 2$.

Actually, we do not need to use the potential function to derive the upper bound 2 for this case. Of course not. For m operations, there are at most m pushes and m pops because the number of pops can never exceed the number of pushes. Thus, there can be at most $2m$ actions and the average time can never exceed 2.

Although, for this simple case, we do not need to use the potential function, we do need to use it in some other cases, as we shall see in the next section.

Finally, we want to remind the reader that $a_i \le 2$ should not be interpreted as $t_i \le 2$. The value of t_i can be very high, much larger than 2. Yet, if t_i is high at present, it will be small in the future because the fact that t_i is high now means that the number of elements in the stack will be tremendously reduced in the future. Or to put it in another way, if it is high, the potential will be tremendously decreased. This is similar to the natural phenomenon. If we use large quantities of the water, the potential of the resources will be decreased; we shall have no water to use in the future.

10–2 AN AMORTIZED ANALYSIS OF SKEW HEAPS

In a skew heap, its basic operation is called "meld". Consider Figure 10–1. To meld the two skew heaps, we first merge the right paths of these two heaps into one and then attach the left parts to nodes in the merged path, shown in Figure 10–2. After this step, we swap the left and right children of every node on the merged path except the lowest.

Figure 10–3 shows the swapping effect. As can be seen, our resulting skew heap tends to have a shortest right path although there is no such guarantee that the right path is the shortest. However, note that it is quite easy to perform the meld operation, and it produces good performance in the amortized sense.

FIGURE 10–1 Two skew heaps.

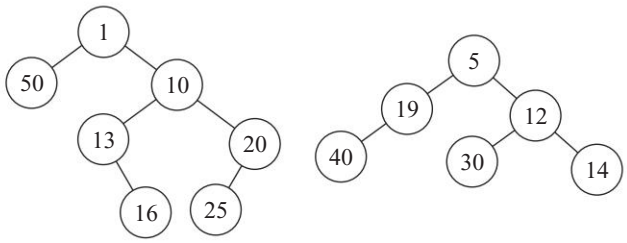


FIGURE 10–2 The melding of the two skew heaps in Figure 10–1.

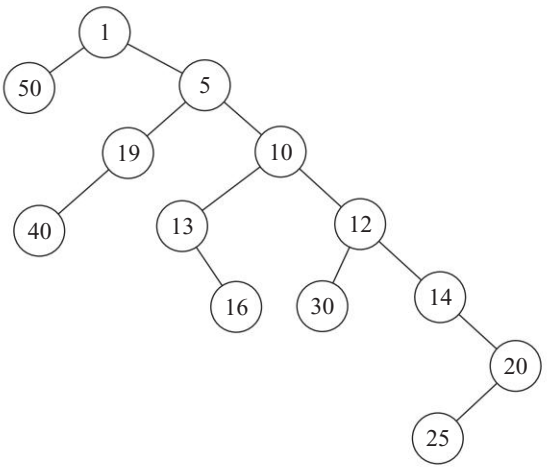
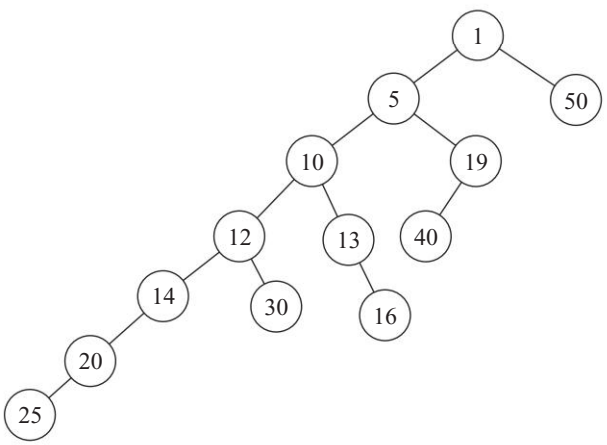


FIGURE 10–3 The swapping of the leftist and rightist paths of the skew heap in Figure 10–2.



A data structure, such as a skew heap, will need many operations. We can actually perform the following operations on a skew heap:

(1) find–min(h): find the minimum of a skew heap h.
(2) insert(x, h): insert x into a skew heap h.
(3) delete–min(h): delete the minimum from a skew heap h.
(4) meld(h_1, h_2): meld two skew heaps h_1 and h_2.

We can implement the first three operations by melding. For instance, to insert x, we consider x as a skew heap with only one element and meld these two skew heaps. Similarly, when we delete the minimum, which always occupies the root of the skew heaps, we effectively break the skew heap into two skew heaps. We then meld these two newly created skew heaps.

Since all operations are based on meld operations, we may consider a sequence of operations which consist of deletion, insertion, and melding as a sequence of meld operations. We therefore only have to analyze the meld operations. Before analyzing, let us note that the meld operation has one interesting property. *If we spend a lot of time for melding, then because of the swapping effect, the right path becomes very short. Thus, the meld operation will not be very time-consuming again; the potential has simply disappeared.*

To perform an amortized analysis, we again let

$$a_i = t_i + \phi_i - \phi_{i-1}$$

where t_i is the time spent for OP_i, ϕ_i and ϕ_{i-1} are the potentials after and before OP_i respectively. Again, we have

$$\sum_{i=1}^{m} a_i = \sum_{i=1}^{m} t_i + \phi_m - \phi_0 .$$

If $\phi_m - \phi_0 \geq 0$, then $\sum_{i=1}^{m} a_i$ serves as an upper bound of the actual time spent by any m operations.

The problem now is to define a good potential function. Let $wt(x)$ of node x denote the number of descendants of node x, including x. Let a heavy node x be a non-root node with $wt(x) > wt(p(x))/2$ where $p(x)$ is the parent node of x. A node x is a light node if it is not a heavy node.

Consider the skew heap in Figure 10–2. The weights of nodes are shown in Table 10–1.

TABLE 10–1 The weights of nodes in Figure 10–2.

Node	Weight	Heavy/Light
1	13	
5	11	Heavy
10	8	Heavy
12	5	Heavy
13	2	Light
14	3	Heavy
16	1	Light
19	2	Light
20	2	Heavy
25	1	Light
30	1	Light
40	1	Light
50	1	Light

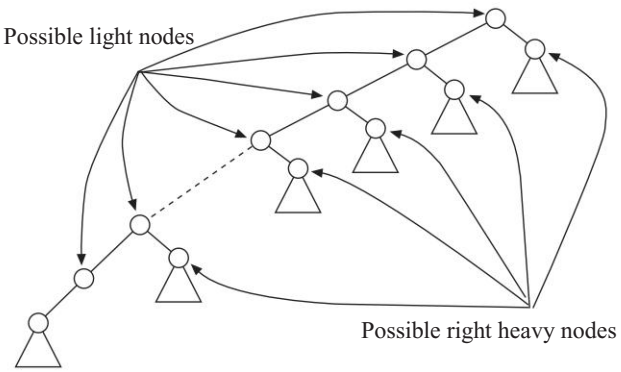
We note that every leaf node must be a light node. Also, if a node is light, then it does not have too many descendants, at least when compared with its brother node. For instance, nodes 13 and 19 in Figure 10–2 are both light nodes. Their brother nodes are heavy. The tree, or the skew heap, is biased to the part with many heavy nodes. Note that in the skew heap in Figure 10–2, all the nodes in the right path are heavy; it is a long path. On the other hand, consider Figure 10–3. The right path of the skew heap in Figure 10–3 has no heavy node; it is a short one.

If a node is the right (left) child of another node, then we call the node a right (left) node. A right (left) heavy node is a right (left) node which is heavy. *We may use the number of right heavy nodes of a skew heap as our potential function*. Thus, for the skew heap in Figure 10–2, the potential is 5 and for the skew heap in Figure 10–3, its potential is 0.

We now investigate a very important question. What are the potentials before the melding and after the melding, or, how much is the difference between the numbers of right heavy nodes before the melding and after the melding?

Let us first note that in order to be a heavy node, it must have more descendants than its brother node. Hence, we can easily see that *among the children of any node, at most one is heavy*. Based on the above statement, we can see in Figure 10–4 that the number of right heavy nodes attached to the left path is related to the number of light nodes in the left path.

FIGURE 10–4 Possible light nodes and possible heavy nodes.



Since only the brother node of a light node can be a heavy node attached to the left path, *the number of right heavy nodes attached to the left path is always less than or equal to the number of light nodes in the left path of a skew heap*. Our problem now is to estimate the number of light nodes in a path.

To estimate the number of light nodes in a path, let us note that *if x is a light node, then the weight of x must be less than or equal to the half weight of p*(x) *where p*(x) *denotes the parent of x*. If y is a descendant of x, and y is a light node, then the weight of y must be less than or equal to one quarter of the weight of $p(x)$. In other words, consider the sequence of light nodes in a path from a node x to y. We can easily see that the weights of these light nodes decrease rapidly and furthermore, *for any path of a skew heap, the number of light nodes in this path must be less than* $\lfloor \log_2 n \rfloor$. This means that the number of right heavy nodes attached to the left path must be less than $\lfloor \log_2 n \rfloor$.

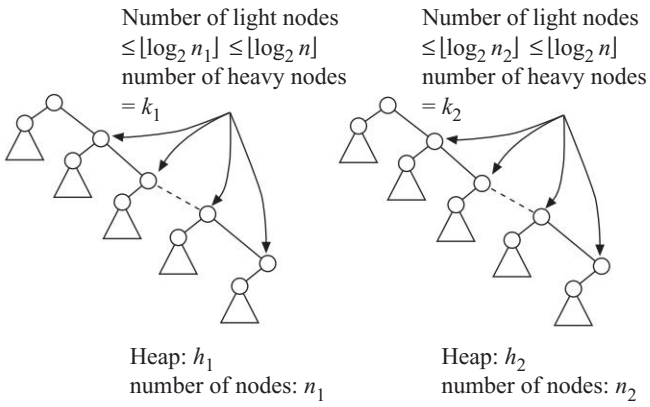
Note that

$$a_i = t_i + \phi_i - \phi_{i-1}$$

where t_i is the time spent by OP_i and ϕ_i denotes the potential after OP_i is executed.

The value of t_i is related to the length of the right path. Consider Figure 10–5. Let $K_1(K_2)$ be the number of heavy nodes of the right path in $h_1(h_2)$. Let $n_1(n_2)$ be the number of nodes in $h_1(h_2)$. Let $L_1(L_2)$ be equal to the number of light nodes plus the number of heavy nodes in the right path of $h_1(h_2)$. And t_i is determined by the number of nodes on the merged path which is equal to $2 + L_1 + L_2$. (The "2" refers to the roots of h_1 and h_2.)

FIGURE 10–5 Two skew heaps.



Thus, $t_i = 2 + L_1 + L_2$

$= 2 +$ number of light nodes of right path of h_1
$+$ number of heavy nodes of right path of h_1
$+$ number of light nodes of right path of h_2
$+$ number of heavy nodes of right path of h_2
$\le 2 + \lfloor \log_2 n_1 \rfloor + K_1 + \lfloor \log_2 n_2 \rfloor + K_2$
$\le 2 + 2\lfloor \log_2 n \rfloor + K_1 + K_2$

where $n = n_1 + n_2$.

We now calculate the value of a_i. *We note that after melding,* $K_1 + K_2$ *right heavy nodes in the right path disappear and fewer than* $\lfloor \log_2 n \rfloor$ *right heavy nodes attached to the left path are created.* Therefore,

$$\phi_i - \phi_{i-1} \le \lfloor \log_2 n \rfloor - K_1 - K_2, \text{ for } i = 1, 2, \ldots, m.$$

Then we have

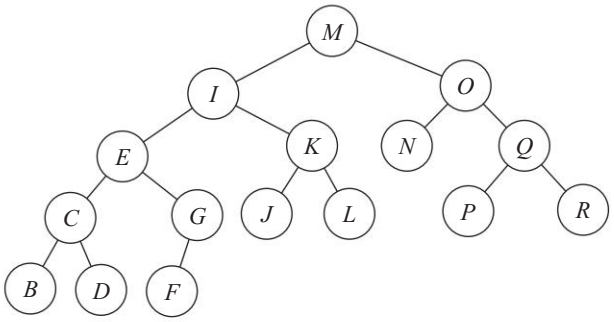
$$\begin{aligned} a_i &= t_i + \phi_i - \phi_{i-1} \\ &\le 2 + 2\lfloor \log_2 n \rfloor + K_1 + K_2 + \lfloor \log_2 n \rfloor - K_1 - K_2 \\ &= 2 + 3\lfloor \log_2 n \rfloor. \end{aligned}$$

Thus $a_i = O(\log_2 n)$ and $\Sigma t_i/m = O(\log_2 n)$.

10–3 AMORTIZED ANALYSIS OF AVL-TREES

In this section, we shall perform an amortized analysis of AVL-trees. *A binary tree is an AVL-tree if the heights of the subtrees of each node differ by at most one*. Figure 10–6 shows an AVL-tree. As we can see, this tree is balanced in the sense that for each node, the heights of its two subtrees differ by at most one.

FIGURE 10–6 An AVL-tree.



For a subtree T with v as its root, the height of T, denoted as H(T), is defined as the length of the longest path from root v to a leaf. Let $L(v)(R(v))$ be the left (right) subtree of the tree with root v. Then for every node v, we define its height balance $hb(v)$ as

$$hb(v) = H(R(v)) - H(L(v)).$$

For an AVL-tree, $hb(v)$ is equal to 0, $+1$ or -1.
For the AVL-tree of Figure 10–6, some of the height balances are as follows:

$$hb(M) = -1 \qquad hb(I) = -1$$
$$hb(E) = 0 \qquad hb(C) = 0$$
$$hb(B) = 0 \qquad hb(O) = +1.$$

Since an AVL-tree is a binary tree, as we add a new item onto the tree, this new data item will become a leaf node. This leaf node will induce a particular path along which all balance factors will have to be changed. Consider Figure 10–7 which shows the AVL-tree in Figure 10–6 with all the height balances. Suppose that we want to add item *A*. The new tree is shown in Figure 10–8. As can be seen, the new tree is not an AVL-tree any more and must be

rebalanced. Comparing Figure 10–8 with Figure 10–7, we can see that only the height balances along the path *M*, *I*, *E*, *C*, *B*, *A* have been changed.

FIGURE 10–7 An AVL-tree with height balance labeled.

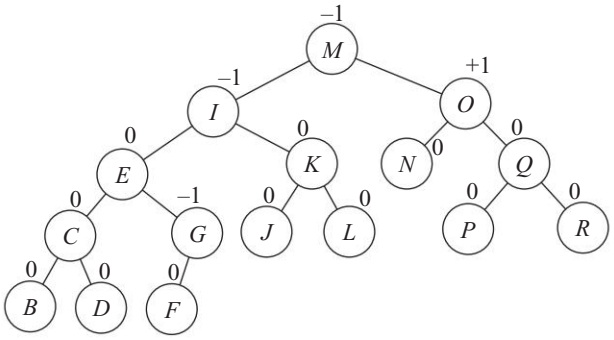
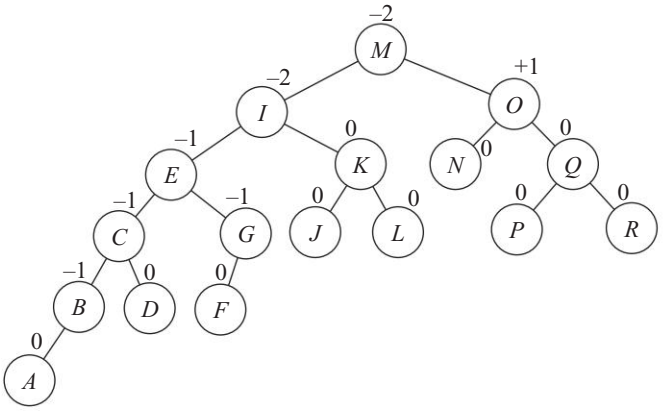


FIGURE 10–8 The new tree with *A* added.



Let V_k be the new leaf node added to an AVL-tree. Let V_0, V_1, … , V_{k-1}, V_k be the newly created path where V_0 is the root of the tree. Let i be the minimum i such that $hb(V_i) = hb(V_{i+1}) = \cdots = hb(V_{k-1}) = 0$ before the insertion. The node V_{i-1} is called the critical node of the path if $i \geq 1$ and V_i, … , V_{k-1} is called the critical path induced by V_k.

Consider Figure 10–7. If the newly added node is attached to *B*, then the critical path is *E*, *C*, *B*. Comparing Figure 10–7 with Figure 10–8, we notice that the balance factors of *E*, *C* and *B* are all changed from 0 to −1. Suppose we add

a new node to, say right of node R. Then the balance factors of Q and R will be changed from 0 to $+1$.

Let T_0 be an empty AVL-tree. We consider a sequence of m insertions into T_0. Let X_1 denote the total number of balance factors changing from 0 to $+1$ or -1 during these m operations. Our problem is to find X_1. The amortized analysis below shows that X_1 is less than or equal to $2.618m$, where m is the number of data elements in the AVL-tree.

Let L_i denote the length of the critical path involved in the ith insertion. Then

$$X_1 = \sum_{i=1}^{m} L_i.$$

In the following, we shall show that when a new item is inserted, there are possibly three cases.

Case 1: There is no need for rebalancing. The tree height is not increased and only an "absorption" is needed to form the new tree.

Case 2: There is need for rebalancing. Either double or single rotations are needed to create a newly balanced tree.

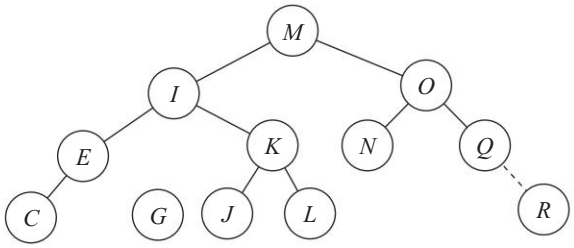
Case 3: There is no need for rebalancing. But the tree height is increased. We only need to record this increase in tree height.

Before illustrating these three cases, let us denote *Val*(T) *as the number of unbalanced nodes in T. That is, Val*(T) *is equal to the number of nodes whose balance factors are not equal to 0.*

Case 1: Consider Figure 10–9. The dotted line indicates that a new data item is inserted. Since the resulting tree is still an AVL-tree, no rebalancing is needed. The old AVL-tree simply absorbs this new node. The height of the tree is not changed either. In this case, the height balance factors of all of the nodes in the critical path are changed from 0 to -1 or $+1$. Furthermore, the balance factor of the critical node will be changed from $+1$ or -1 to 0. Thus,

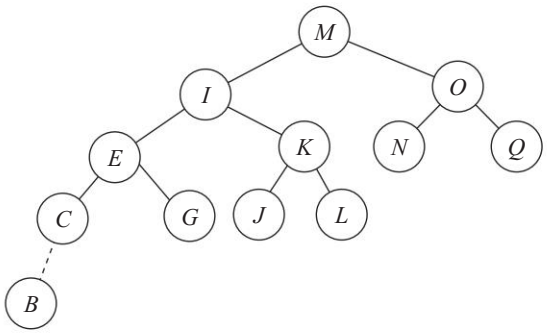
$$Val(T_i) = Val(T_{i-1}) + (L_i - 1).$$

FIGURE 10–9 Case 1 after insertion.



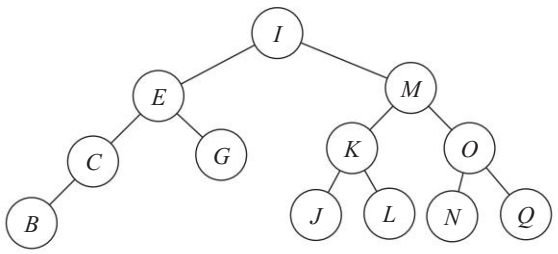
Case 2: Consider Figure 10–10. In this case, the tree needs rebalancing.

FIGURE 10–10 Case 2 after insertion.



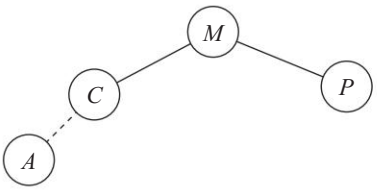
Some kind of double and single rotations are needed. For the case shown in Figure 10–10, the balanced tree is shown in Figure 10–11. It is easy to see that the balance factor of the critical node M is changed from -1 to 0. But we shall have to change the balance factors of all nodes on the critical path from 0 to $+1$ or -1 except the child of the critical node in the critical path which is also a balanced node after rebalancing. Thus, in this case, $Val(T_i) = Val(T_{i-1}) + (L_i - 2)$.

FIGURE 10–11 The tree in Figure 10–10 balanced.



Case 3: Consider Figure 10–12. In this case, we do not need to rebalance the tree. But the height of the tree will be increased.

FIGURE 10–12 Case 3 after insertion.



It is easy to see that

$$Val(T_i) = Val(T_{i-1}) + L_i.$$

Let X_2 denote the number of absorptions needed in Case 1, X_3 the number of single rotations needed in Case 2, X_4 the number of double rotations needed in Case 2 and X_5 the number of height increases in Case 3. Then, using the formula described in the above three cases, we have

$$Val(T_m) = Val(T_0) + \sum_{i=1}^{m} L_i - X_2 - 2(X_3 + X_4).$$

In the above formula, $-X_2$ and $-2(X_3 + X_4)$ correspond to Case 1 and Case 2 respectively. $\sum_{i=1}^{m} L_i$ is due to the L_i term in all the three formulas.

Since $\sum_{i=1}^{m} L_i = X_1$ and $Val(T_0) = 0$, we have

$$\begin{aligned} X_1 &= Val(T_m) + X_2 + 2(X_3 + X_4) \\ &= Val(T_m) + (X_2 + X_3 + X_4 + X_5) + (X_3 + X_4 - X_5) \\ &= Val(T_m) + m + (X_3 + X_4 - X_5) \end{aligned}$$

because $X_2 + X_3 + X_4 + X_5 = m$.

Now, it is obvious that $X_3 + X_4 \le m$ and $X_5 \ge 0$. Therefore,

$$\begin{aligned} X_1 &\le Val(T_m) + m + m \\ &= Val(T_m) + 2m. \end{aligned}$$

It was proved by Knuth in "The Art of Computer Programming Vol. 3" that

$$Val(T_m) \leq 0.618m.$$

Consequently, we have

$$\begin{aligned} X_1 &\leq Val(T_m) + 2m \\ &= 2.618m. \end{aligned}$$

10–4 AMORTIZED ANALYSIS OF SELF-ORGANIZING SEQUENTIAL SEARCH HEURISTICS

Sequential search is the simplest kind of searching. Still, there exist many methods to enhance the performance of the sequential search. One of them is the so called self-organizing method.

Let us imagine that in a dormitory, there are several exceedingly popular students who are constantly receiving phone calls. The operator, if he is smart enough, would put the room numbers of these students at the top of the list. Through this way, phone calls can be quickly answered.

A self-organizing method utilizes this idea. It would move items which are frequently searched towards the top of the list dynamically. That is, whenever a search is completed, the item being searched is moved upwards. There are three famous self-organizing methods:

(1) **Transpose method:** When the item is found, exchange it with the item in front of it. Thus, it is moved one location towards the top of the list.
(2) **Move-to-the-front method:** When the item is found, move this item to the top of the list.
(3) **Count method:** When the item is found, increment its count and move it forward as little as needed to keep the list sorted in decreasing order.

It is customary to call the above methods heuristic methods. Therefore, these three methods are also called transpose heuristics, move-to-the-front heuristics and count heuristics, respectively.

Next, we shall illustrate these heuristics by several examples. We shall always assume that we start with an empty list initially.

(1) Transpose heuristics (see Table 10–2).

TABLE 10–2 Transpose heuristics.

Query	Sequence
B	*B*
D	*D B*
A	*D A B*
D	*D A B*
D	*D A B*
C	*D A C B*
A	*A D C B*

(2) Move-to-the-front heuristics (see Table 10–3).

TABLE 10–3 Move-to-the-front heuristics.

Query	Sequence
B	*B*
D	*D B*
A	*A D B*
D	*D A B*
D	*D A B*
C	*C D A B*
A	*A C D B*

(3) Count heuristics (see Table 10–4).

TABLE 10–4 Count heuristics.

Query	Sequence
B	*B*
D	*B D*
A	*B D A*
D	*D B A*
D	*D B A*
A	*D A B*
C	*D A B C*
A	*D A B C*

We are interested in the amortized analysis of different heuristics. Given a sequence of queries, the cost of this sequence S of queries, under a heuristic R, is defined as the total number of comparisons required for this sequence if the sequential search under this heuristic R is used. We shall denote this cost by $C_R(S)$. Assume that there are m queries. It would be nice if we can find the cost as a function of m. As we did before, we shall instead find an upper bound of this cost. This upper bound is related to the cost if the items are ordered optimally and statically for this sequence. The cost incurred when the items are ordered statically and optimally is now labeled as $C_O(S)$ for this sequence S. For instance, suppose that there are two query sequences. Let the number of B's being queried be higher than that of A's being queried. Then the optimal static ordering for this query sequence is $B\ A$.

In the rest of this section, we shall compare $C_R(S)$ with $C_O(S)$. That we can compare these two costs is based on a very important property, called *pairwise independent property*, possessed by some heuristics. Before introducing this property, let us define the intraword comparisons as comparisons made among unequal items. Intraword comparisons are therefore unsuccessful comparisons. Interword comparisons are comparisons between equal items and are therefore successful comparisons. Now we can introduce the pairwise independent property.

Let us consider the move-to-the-front heuristics. Consider the following query sequence: $C\ A\ C\ B\ C\ A$. The sequence now appears as shown in Table 10–5.

TABLE 10–5 Move-to-the-front heuristics for query sequence: $C\ A\ C\ B\ C\ A$.

Query	Sequence	(A, B) comparison
C	C	
A	$A\ C$	
C	$C\ A$	
B	$B\ C\ A$	√
C	$C\ B\ A$	
A	$A\ C\ B$	√

Next, we focus our attention on the comparison between A and B. The above analysis shows that for this sequence, the total number of comparisons made between A and B is 2.

Suppose that we only consider the subsequences of the sequence consisting of A and B. Then see Table 10–6. Again, the total number of comparisons made between A and B is 2, which is the same as Table 10–5.

TABLE 10–6 Move-to-the-front heuristics for query sequence: *A B A*.

Query	Sequence	(*A*, *B*) comparison
A	*A*	
B	*B A*	√
A	*A B*	√

In other words, for the move-to-the-front heuristics, the total number of comparisons made between any pair of items P and Q depends only on the relative ordering of the P's and Q's in the query sequence and is independent of other items. In other words, for any sequence S and all pairs of P and Q, the number of intraword comparisons of P and Q is exactly the number of intraword comparisons made for the subsequence of S consisting of only P's and Q's. This property is called the pairwise independent property.

For example, let us consider the query sequence *C A C B C A* again. There are three distinct intraword comparisons: (A, B), (A, C) and (B, C). We may consider them separately and then add them up, shown in Table 10–7.

TABLE 10–7 The number of intraword comparisons of move-to-the-front heuristics.

Query	Sequence	*C*	*A*	*C*	*B*	*C*	*A*	
			0		1		1	(A, B)
		0	1	1		0	1	(A, C)
		0		0	1	1		(B, C)
		0	1	1	2	1	2	

The total number of intraword comparisons is equal to $0 + 1 + 1 + 2 + 1 + 2 = 7$. It can be easily checked that this is correct.

Let $C_M(S)$ denote the cost of the move-to-the-front heuristics. Next, we shall show that $C_M(S) \leq 2C_O(S)$. In other words, for any query sequence, we know that for the move-to-the-front heuristics, the cost can never exceed twice that of the optimal static cost for this query sequence.

We first prove that $C_M(S) \leq 2C_O(S)$ for S consisting of two distinct items. Let S consist of a A's and b B's where $a < b$. Thus, an optimal static ordering is *B A*. If move-to-the front heuristics is used, the total number of intraword comparisons is the number of changes from *B A* to *A B* plus the number of changes from *A B* to *B A*. Thus, this number cannot exceed $2a$. Let $Intra_M(S)$ and $Intra_O(S)$ denote the total number of intraword comparisons under the move-to-the-front heuristics and the optimal static ordering respectively. We have

$$Intra_M(S) \le 2Intra_O(S)$$

for any S consisting of two distinct items.

Consider any sequence S consisting of more than two items. Because of the pairwise independent property possessed by the move-to-the-front heuristics, we can easily have

$$Intra_M(S) \le 2Intra_O(S)$$

Let $Intra_M(S)$ and $Intra_O(S)$ denote the numbers of interword comparisons under the move-to-the-front heuristics and under the optimal static ordering respectively. Obviously, we have

$$Inter_M(S) = Inter_O(S)$$

Therefore,

$$Inter_M(S) + Intra_M(S) \le Inter_O(S) + 2Intra_O(S)$$
$$C_M(S) \le 2C_O(S).$$

One may wonder whether coefficient 2 can be further tightened. We shall show below that this cannot be done.

Consider the query sequence $S = (A\ B\ C\ D)^m$. The optimal static ordering for this sequence is $A\ B\ C\ D$. The total number of comparisons, $C_O(S)$, is

$$\underbrace{1 + 2 + 3 + 4 + 1 + 2 + 3 + 4 + \cdots}_{4m} = 10 \cdot \frac{4m}{4} = 10m.$$

Under the move-to-the-front heuristics, the total number of comparisons, $C_M(S)$ can be found as follows:

$$\begin{array}{rllllll}
C_M(S) = 1 & A & A & & & & \\
+2 & B & B & A & & & \\
+3 & C & C & B & A & & \\
+4 & D & D & C & B & A & \\
+4 & A & A & D & C & B & \\
+4 & B & B & A & D & C & \\
\vdots & \vdots & & & \vdots & & \\
\hline
10 + 4(4(m-1)) & & & & & &
\end{array}$$

Therefore, $C_M(S) = 10 + 4(4(m-1)) = 10 + 16m - 16 = 16m - 6$.

If we increase the number of distinct items from 4 to k, we have

$$C_O(S) = \frac{k(k+1)}{2} m = \frac{m(k+1)k}{2}$$

and $$C_M(S) = \frac{k(k+1)}{2} + k(k(m-1)) = \frac{k(k+1)}{2} + k^2(m-1).$$

Thus,

$$\frac{C_M(S)}{C_O(S)} = \frac{\frac{k(k+1)}{2} + k^2(m-1)}{\frac{k(k+1)}{2} \cdot m}$$

$$= \frac{1}{m} + 2\frac{k^2}{k(k+1)} \cdot \frac{m-1}{m} \to 2 \quad \text{as } k \to \infty \text{ and } m \to \infty.$$

This means that the factor 2 cannot be further tightened.

The above discussion shows that the worst case cost of the move-to-the-front heuristics is $2C_o(S)$. The same reasoning can be used to show that $C_c(S) \le 2C_o(S)$ when $C_c(S)$ is the cost of the count heuristics. Unfortunately, the transpose heuristics does not possess the pairwise independent property. Therefore, we cannot have a similar upper bound for the cost of the transpose heuristics. That there is no pairwise independent property can be illustrated by considering the same query sequence, *C A C B C A* again, under the transpose heuristics.

For the transpose heuristics, if we consider pairs of distinct items independently, we shall have the situation depicted in Table 10–8.

TABLE 10–8 The number of intraword comparisons of transpose heuristics.

Query Sequence	*C*	*A*	*C*	*B*	*C*	*A*	
		0		1		1	(A, B)
	0	1	1		0	1	(A, C)
	0		0	1	1		(B, C)
	0	1	1	2	1	2	

The total number of intraword comparisons, with this kind of calculation, will be 1 + 1 + 2 + 1 + 2 = 7. However, this is not correct. The correct intraword comparison is found as follows:

Query Sequence	*C*	*A*	*C*	*B*	*C*	*A*
Data Ordering	C	*AC*	*CA*	*CBA*	*CBA*	*CAB*
Number of Intraword Comparisons	0	1	1	2	0	2

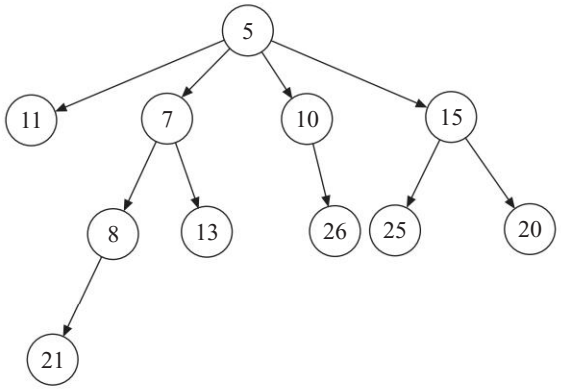
The total number of intraword comparisons is actually $1 + 1 + 2 + 2 = 6$. This shows that pairwise independent property does not hold under the transpose heuristics.

10–5 PAIRING HEAP AND ITS AMORTIZED ANALYSIS

We shall introduce the pairing heap in this section and perform an amortized analysis of this data structure. It will become clear that this pairing heap may degenerate to such an extent that it will be difficult to delete the minimum from it. Still, the amortized analysis will also show that after the pairing deteriorates to such an unpleasant degree, it may soon recover.

Figure 10–13 shows a typical example of a pairing heap. We shall explain why this is called a pairing heap after we introduce the delete minimum operation.

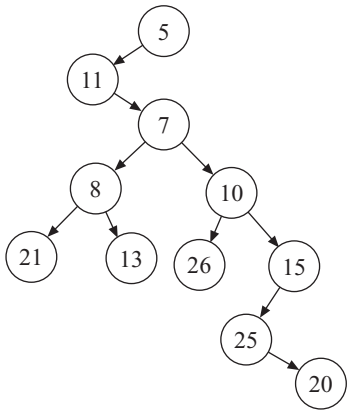
FIGURE 10–13 A pairing heap.



Note that, just as in any heap, the key of the parent of a node is less than the key of the node itself. We, of course, must use a well-structured data structure to implement this heap. For a pairing heap, we use the binary tree technique. The binary tree representation of the heap in Figure 10–13 is now shown in Figure 10–14. As shown in Figure 10–14, each node is connected to its leftmost

descendant and its next right brother, if it has any of them. For instance, in Figure 10–13, node 7 has node 8 as its leftmost descendant and node 10 as its next right brother. Therefore, node 7 is connected to both nodes 8 and 10 in the binary tree representation, shown in Figure 10–14.

FIGURE 10–14 The binary tree representation of the pairing heap shown in Figure 10–13.



There are seven basic operations for a pairing heap. They are

(1) make heap(h): to create a new empty heap named h.
(2) find min(h): to find the minimum of heap h.
(3) insert(x, h): to insert an element x into heap h.
(4) delete min(h): to delete the minimum element from heap h.
(5) meld(h_1, h_2): to create a heap by joining two heaps h_1 and h_2.
(6) decrease (Δ, x, h): to decrease the element x in heap h by the value Δ.
(7) delete(x, h): to delete the element x from heap h.

There is a basic internal operation, called link (h_1, h_2), which links two heaps into a new heap. It will be shown that many operations mentioned above are based on this link operation. Figure 10–15 illustrates a typical link operation.

We shall now illustrate how the seven operations are implemented in pairing heaps.

(1) make heap(h): We just allocate a memory location to the new heap.
(2) find min(h): We return the root of the heap which is trivial.

(3) insert(x, h): This consists of two steps. We first create a new one-node tree and link it with h. Figure 10–16 and Figure 10–17 show how this is done.

FIGURE 10–15 An example of link (h_1, h_2).

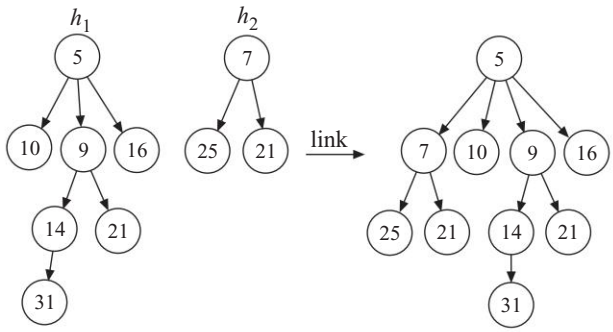


FIGURE 10–16 An example of insertion.

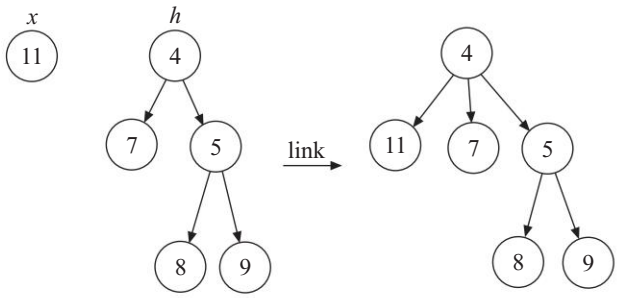
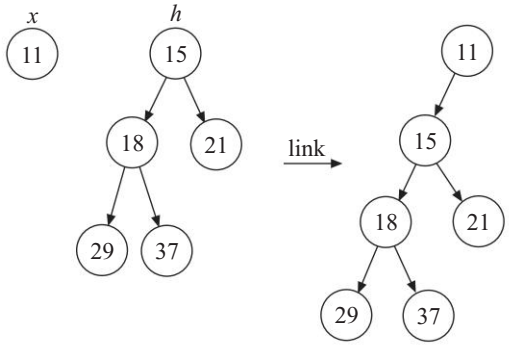


FIGURE 10–17 Another example of insertion.



(4) meld(h_1, h_2): We simply return the heap formed by linking h_1 and h_2.
(5) decrease(Δ, x, h): This operation consists of three steps.
Step 1: Subtract from x.
Step 2: If x is the root, then return.
Step 3: Cut the edge joining x to its parent. This will produce two trees. Link these two resulting trees.

Consider decrease (3, 9, h) where h is shown in Figure 10–18. Figure 10–19 illustrates how this operation is performed.

FIGURE 10–18 A pairing heap to illustrate the decrease operation.

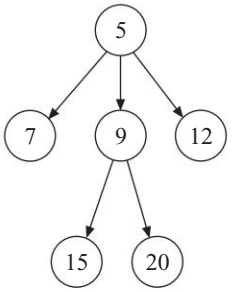
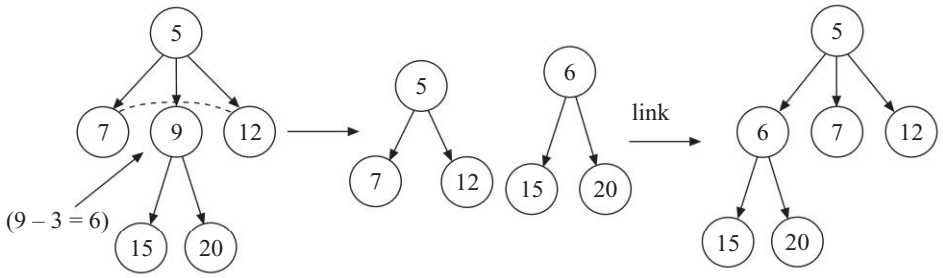


FIGURE 10–19 The decrease (3, 9, h) for the pairing heap in Figure 10–18.

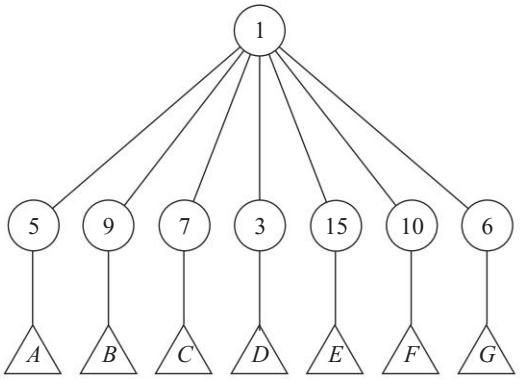


(6) delete(x, h): There are two steps in this operation.
Step 1: If x is the root, then return (delete min(h))
Step 2: Otherwise,
Step 2.1: Cut the edge joining x to its parent.
Step 2.2: Perform a delete min(h) on the tree rooted at x.
Step 2.3: Link the resulting trees.

Note that inside delete(x, h), there is a delete min(h) operation which we have not described yet. We deliberately delay the discussion of this operation because this operation is critically important to the amortized analysis of the pairing heap. In fact, it is this delete min(h) operation which makes pairing heap a beautiful data structure, in the sense of amortized analysis.

Before describing the delete min(h), let us consider the pairing heap in Figure 10–20. Figure 10–21 shows the binary tree representation of this pairing heap. As can be seen in Figure 10–21, this binary tree is highly skewed to the right.

FIGURE 10–20 A pairing heap to illustrate the delete min(h) operation.



After the minimum of the heap, namely the root of the binary tree, is eliminated, we need to reconstruct the heap. This reconstruction step must perform one function: Find the minimum among the second level elements of the heap. Corresponding to the binary tree representation, the second level elements are all in a path, shown in Figure 10–21. If there are many second level elements in the heap, the corresponding path will be relatively long. In such a case, the operation to find the minimum will take a relatively long time, which is undesirable.

We shall demonstrate that there is a high probability that the number of elements in the second level of the heap will be small. In other words, the corresponding binary tree will be quite balanced.

Let us consider the pairing heap in Figure 10–20 again. The delete min(h) operation first cuts off all of the links linking to the root of the heap. This will result in seven heaps, shown in Figure 10–22.

FIGURE 10–21 The binary tree representation of the pairing heap shown in Figure 10–19.

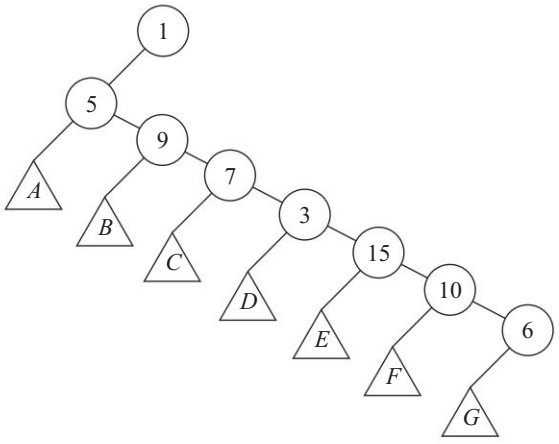
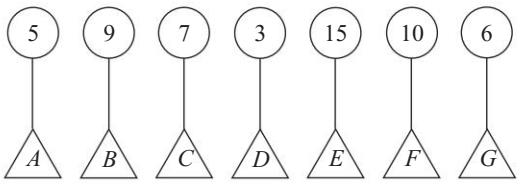
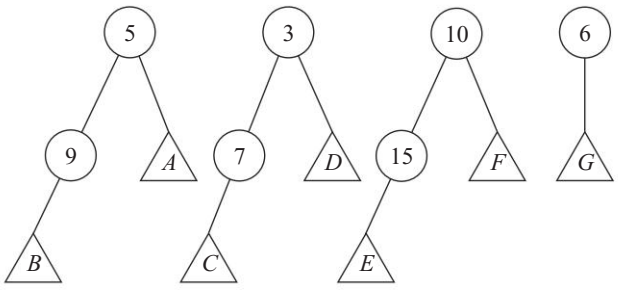


FIGURE 10–22 The first step of the delete min(h) operation.



In the step of the delete min(h) operation, we meld the resulting heaps in pairs, the first and the second, the third and the fourth, etc., shown in Figure 10–23.

FIGURE 10–23 The second step of the delete min(h) operation.



After the pairwise melding operation, we then link the paired heaps one by one to the last heap, starting from the second last heap, then the third last heap and so on, shown in Figure 10–24.

All three steps of delete min(h) can be implemented directly on the binary representation of the heap.

The binary tree representation of the resulting heap in Figure 10–24 is shown in Figure 10–25. It is obvious that this binary tree is much more balanced than that shown in Figure 10–21.

FIGURE 10–24 The third step of the delete min(h) operation.

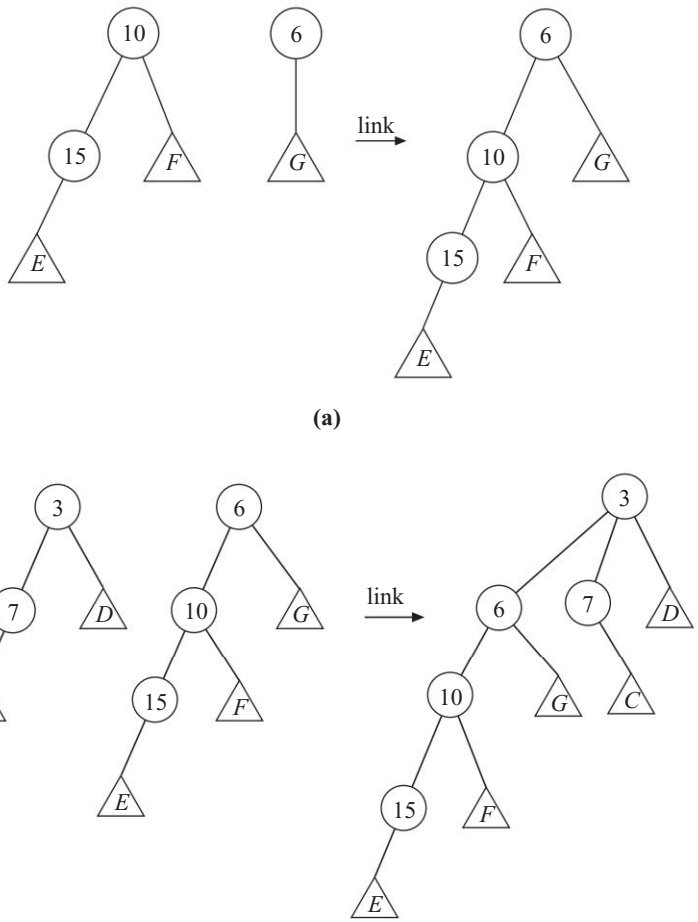
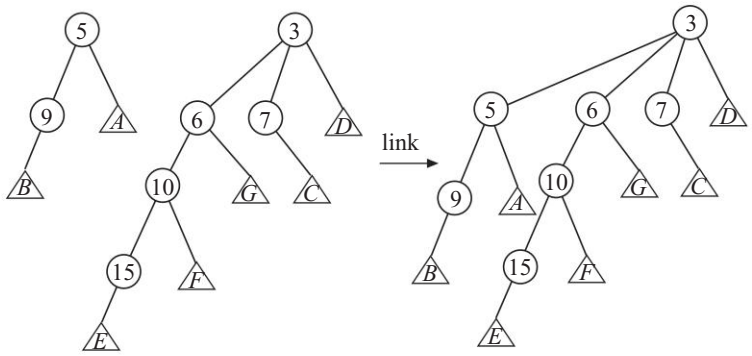
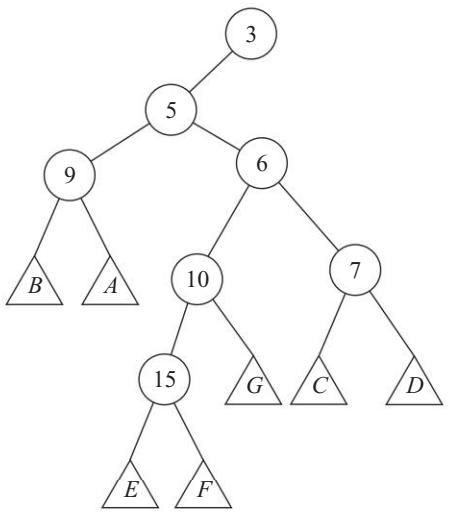


FIGURE 10–24 (*cont'd*)



(c)

FIGURE 10–25 The binary tree representation of the resulting heap shown in Figure 10–24.



Having illustrated the delete min(h) operation, we can now illustrate the delete (x, h) operation. Consider the pairing heap in Figure 10–26. If $x = 6$, then after deleting it, we will have the resulting heap, shown in Figure 10–27.

We shall now give an amortized analysis of the pairing heap operations. As we did before, we shall first define a potential function.

Given a node x of a binary tree, let $s(x)$ denote the number of nodes in its subtree including x. Let the rank of x, denoted as $r(x)$, be defined as $\log(s(x))$. The potential of a tree is the sum of the ranks of all nodes.

FIGURE 10–26 A pairing heap to illustrate the delete(x, h) operation.

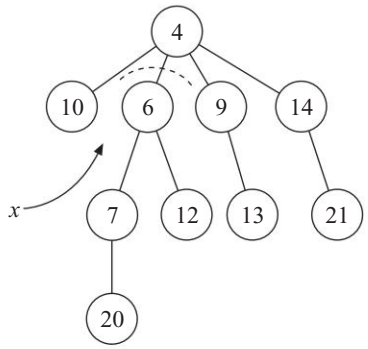
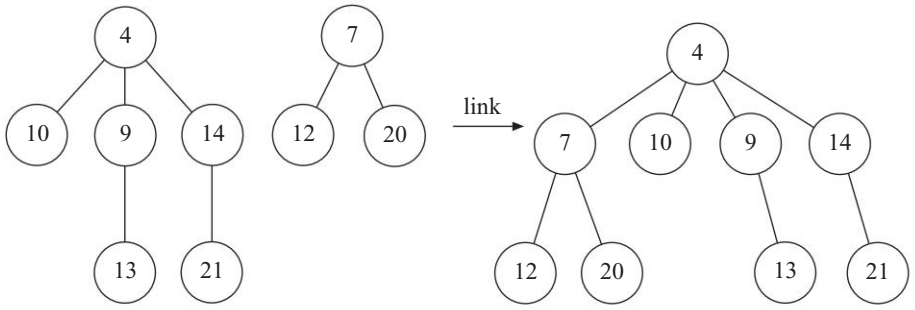


FIGURE 10–27 The result of deleting 6 from the pairing heap in Figure 10–26.



For example, consider Figure 10–28, which shows the two potentials of the two trees.

Of the seven pairing heap operations, make heap and find minimum would not cause any change of potential. As for insert, meld and decrease, it can be easily seen that the change of potential is at most $\log(n + 1)$ and it is caused by the internal link operations. As for delete and delete minimum, the critical operation is delete minimum. Let us now use an example to illustrate how the potential changes.

Consider Figure 10–29. The binary tree of the pairing heap in Figure 10–29(a) is shown in Figure 10–29(b).

Let r_a, r_b, r_c and r_d represent the ranks of A, B, C and D respectively. Let the total numbers of nodes in A, B, C and D be S_a, S_b, S_c and S_d respectively. Then the potential of the binary tree in Figure 10–29(b) is

FIGURE 10–28 Potentials of two binary trees.

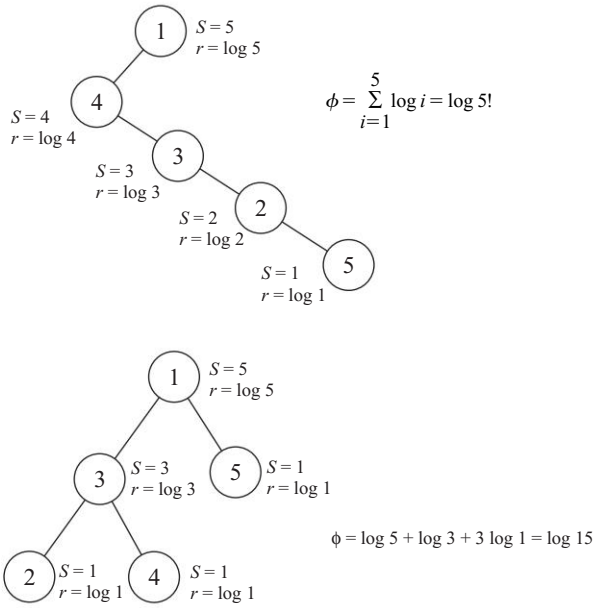
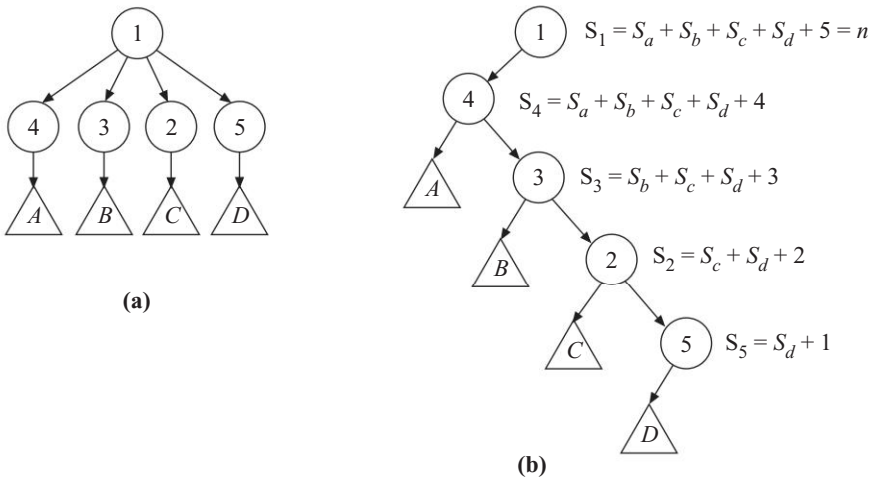


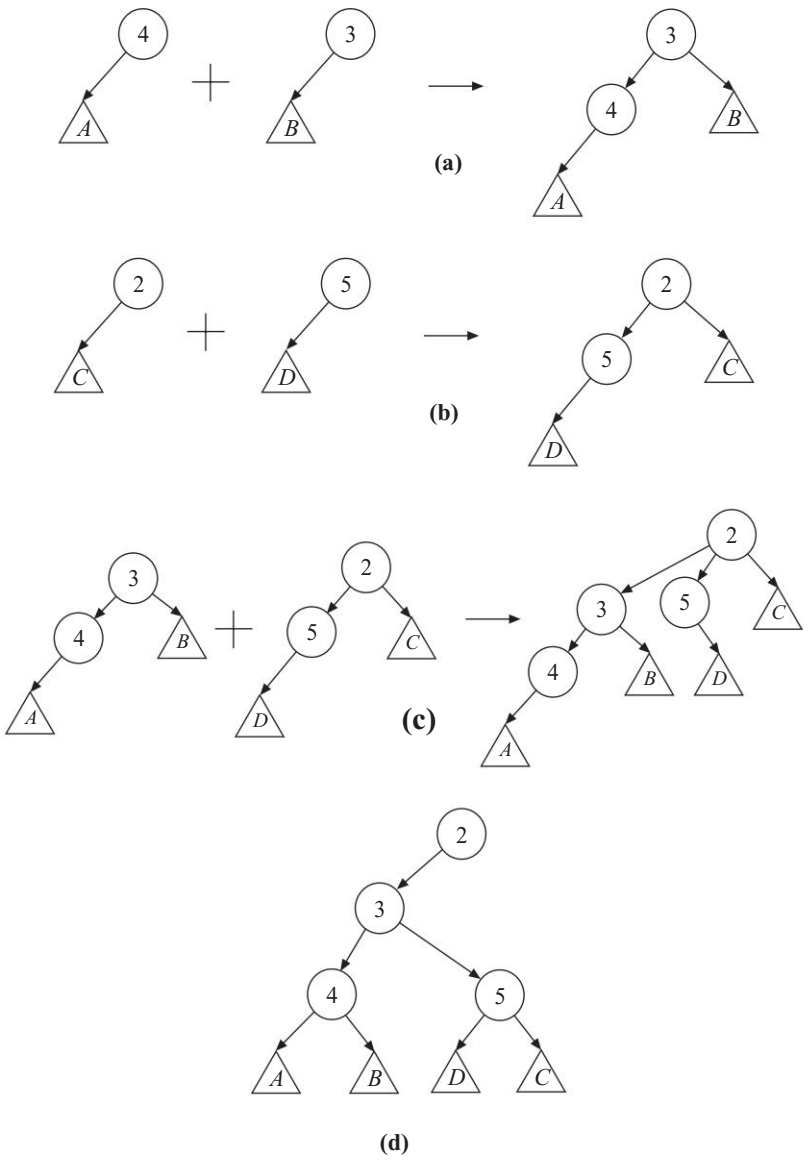
FIGURE 10–29 A heap and its binary tree to illustrate the potential change.



$$\phi = r_a + r_b + r_c + r_d + \log (S_a + S_b + S_c + S_d + 5) + \log (S_a + S_b + S_c + S_d + 4) + \log (S_b + S_c + S_d + 3) + \log (S_c + S_d + 2) + \log (S_d + 1).$$

After the minimum, namely 1, is deleted, there will be a sequence of pairing operations, shown in Figure 10–30(a) to (c). The resulting heap is shown in Figure 10–30 (c), and its binary tree representation is shown in Figure 10–30 (d).

FIGURE 10–30 The sequence of pairing operations after the deleting minimum operation on the heap of Figure 10–29 and its binary tree representation of the resulting heap.



The new potential is changed to

$$\begin{aligned}\phi' &= r_a + r_b + r_c + r_d + \log(S_a + S_b + 1) + \log(S_c + S_d + 1) \\ &\quad + \log(S_a + S_b + S_c + S_d + 3) + \log(n - 1) \\ &= r_a + r_b + r_c + r_d + \log(S_a + S_b + 1) + \log(S_c + S_d + 1) \\ &\quad + \log(S_a + S_b + S_c + S_d + 3) + \log(S_a + S_b + S_c + S_d + 4).\end{aligned}$$

In general, let there be a sequence of operations $OP_1, OP_2, \ldots, OP_q$ and each OP_i may be one of the seven pairing heap operations. Let

$$a_i = t_i + \phi_i - \phi_{i-1}$$

where ϕ_i and ϕ_{i-1} are the potentials after and before OP_i respectively and t_i is the time needed to perform OP_i. Next, we shall concentrate our discussions on the delete minimum operation. We shall derive an amortized upper bound for this operation. It should be obvious that all other operations have the same upper bound.

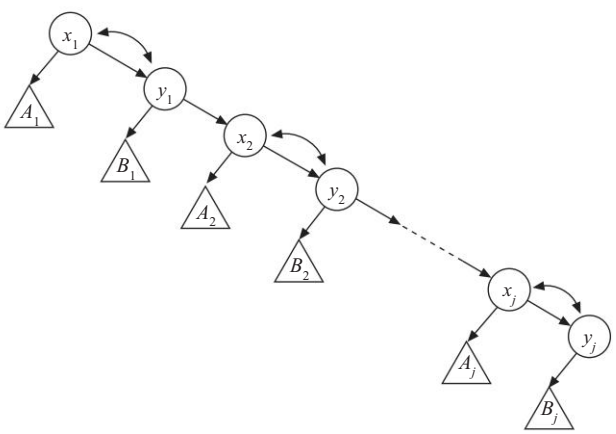
Note that the delete minimum operation consists of the following operations:

(1) Deleting the root.
(2) Pairwise melding.
(3) Melding with the last heap one by one.

For the first operation, it is easy to see that the maximum change of potential is $\Delta\phi_i = -\log n$.

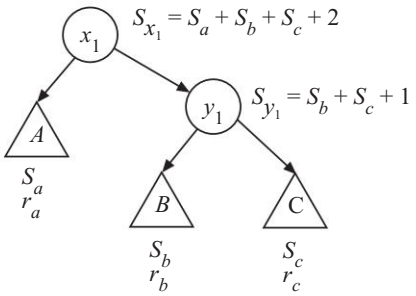
Now, consider the pairwise melding. Originally, the binary tree has a string of nodes, shown in Figure 10–31.

FIGURE 10–31 The pairing operations.



Let us now redraw Figure 10–31, shown in Figure 10–32.

FIGURE 10–32 A redrawing of Figure 10–31.



Let r_a, r_b and r_c denote the ranks of A, B and C respectively. Let S_a, S_b and S_c denote the numbers of nodes in A, B and C respectively. Then the potential before the first melding is

$$\phi_{before} = r_a + r_b + r_c + \log (S_a + S_b + S_c + 2) + \log (S_b + S_c + 1).$$

The first melding operation melds two heaps, shown in Figure 10–33. There are two possibilities depending on the relationships between x_1 and y_1. The binary tree representations of the two resulting heaps are shown in Figure 10–34. For both possible binary tree representations, we must add the remaining part to them, shown in Figure 10–35.

FIGURE 10–33 Two possible heaps after the first melding.

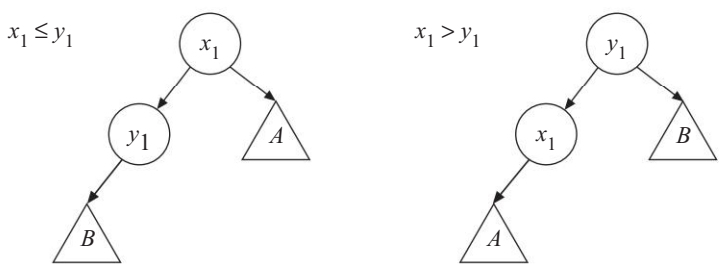


FIGURE 10–34 The binary tree representations of the two heaps in Figure 10–33.

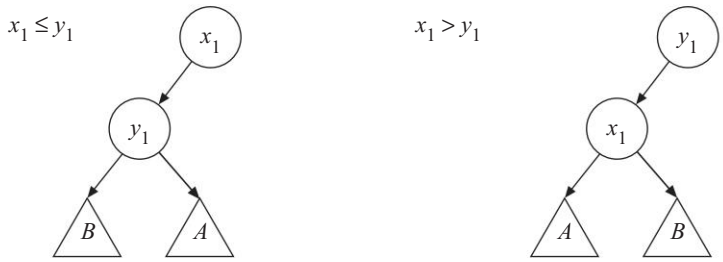
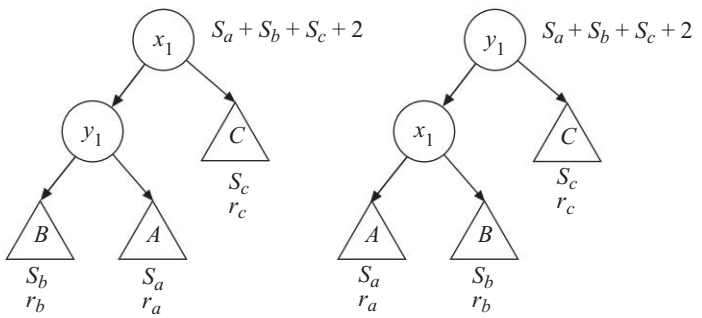


FIGURE 10–35 Resulting binary trees after the first melding.



For both binary trees in Figure 10–35, the potential is

$$\phi_{after} = r_a + r_b + r_c + \log (S_a + S_b + S_c + 2) + \log (S_a + S_b + 1).$$

Therefore, the potential change is

$$\begin{aligned} \Delta\phi_{pairing} &= \phi_{after} - \phi_{before} \\ &= \log (S_a + S_b + 1) - \log (S_b + S_c + 1). \end{aligned}$$

Although this refers to the first melding, it is obvious that $\Delta\phi_{pairing}$ can refer to the general pairing.

We shall show that $\Delta\phi_{pairing} \le 2 \log (S_x) \le 2 \log (S_c) - 2$, where S_x denotes the total number of nodes of the subtree rooted at x before the change. To prove this, let us use the following fact:

If $p, q > 0$, and $p + q \le 1$, then $\log p + \log q \le -2$.

The above property can be easily proved. Now, let

$$p = \frac{S_a + S_b + 1}{S_a + S_b + S_c + 2}$$

and $$q = \frac{S_c}{S_a + S_b + S_c + 2}.$$

Then $$\log\left(\frac{S_a + S_b + 1}{S_a + S_b + S_c + 2}\right) + \log\left(\frac{S_c}{S_a + S_b + S_c + 2}\right) \leq -2,$$

that is $\log (S_a + S_b + 1) - \log (S_c) \leq 2 \log (S_a + S_b + S_c + 2) - 2 \log (S_c) - 2$.

But $\log (S_c) < \log (S_b + S_c + 1)$.

Therefore,

$$\log (S_a + S_b + 1) - \log (S_b + S_c + 1) \leq 2 \log (S_a + S_b + S_c + 2) - 2 \log (S_c) - 2.$$

Thus, we have

$$\begin{aligned}\Delta\phi_{pairing} &\leq 2 \log (S_a + S_b + S_c + 2) - 2 \log (S_c) - 2 \\ &= 2 \log (S_x) - 2 \log (S_c) - 2.\end{aligned}$$

For the last pair, $S_c = 0$. In such a case,

$$\Delta\phi_{pairing} \leq 2 \log (S_x).$$

First of all, let us note that we may safely assume that there are j pairs. If there are $2j + 1$ nodes, the last node will not be melded with others and thus there is no change of potential due to the last node. The total change of the pairwise melding is

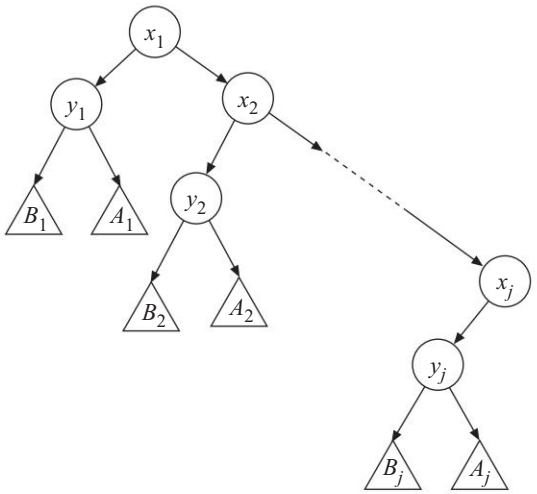
$$\begin{aligned}\Delta\phi_{total\ pairing} &= \sum_{i=1}^{j-1} (\text{the } i\text{th } \Delta\phi_{pairing}) + (\Delta\phi_{pairing} \text{ due to the } j\text{th pair}) \\ &\leq \sum_{i=1}^{j-1} (2 \log (S_{x_i}) - 2 \log (S_{x_{i+1}}) - 2) + 2 \log (S_{x_j}) \\ &= 2(\log (S_{x_1}) - \log (S_{x_2}) + \log (S_{x_2}) - \log (S_{x_3}) + \cdots \\ &\quad - \log (S_{x_j})) + 2 \log (S_{x_j}) - 2(j - 1) \\ &= 2 \log (S_{x_1}) - 2(j - 1) \\ &\leq 2 \log n - 2j + 2.\end{aligned}$$

Thus, for the second step of the delete minimum operation,

$$\Delta\phi_{total\ pairing} \leq 2 \log n - 2j + 2.$$

Finally, we discuss the last step. This step takes place after the above pairing operation. It melds each heap one by one with the last heap. Therefore, we have a binary tree resulting from the pairing operations, illustrated in Figure 10–36. (We assume that $x_i < y_i$ for all i.)

FIGURE 10–36 The binary tree after the pairing operations.



Consider the last pair, shown in Figure 10–37. The potential

$$\begin{aligned}\phi_{before} = {} & r_{A_{j-1}} + r_{B_{j-1}} + r_{A_j} + r_{B_j} + \log (S_{A_{j-1}} + S_{B_{j-1}} + 1) \\ & + \log (S_{A_j} + S_{B_j} + 1) + \log (S_{A_j} + S_{B_j} + 2) \\ & + \log (S_{A_{j-1}} + S_{B_{j-1}} + S_{A_j} + S_{B_j} + 4).\end{aligned}$$

After the melding, depending on whether $x_{j-1} < x_j$ or $x_{j-1} \geq x_j$, the new binary tree either looks like that in Figure 10–38(a) or that in Figure 10–38(b).

Since the new potentials are the same for both binary trees shown in Figure 10–38(a) and Figure 10–38(b), we shall therefore consider only one of them. For the binary tree in Figure 10–38(a), the new potential is

FIGURE 10–37 A pair of subtrees in the third step of the delete minimum operation.

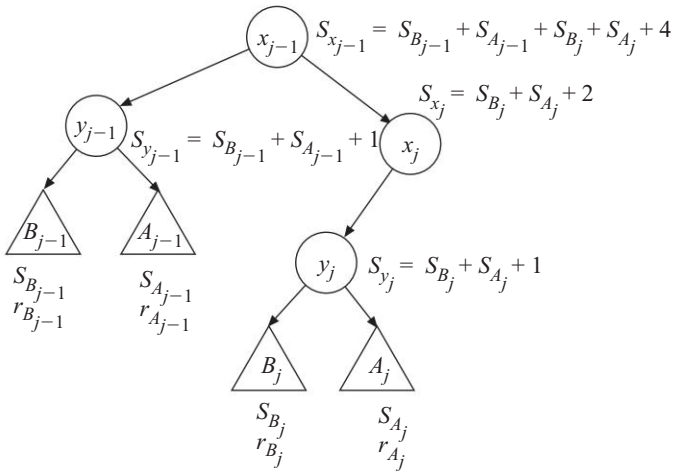
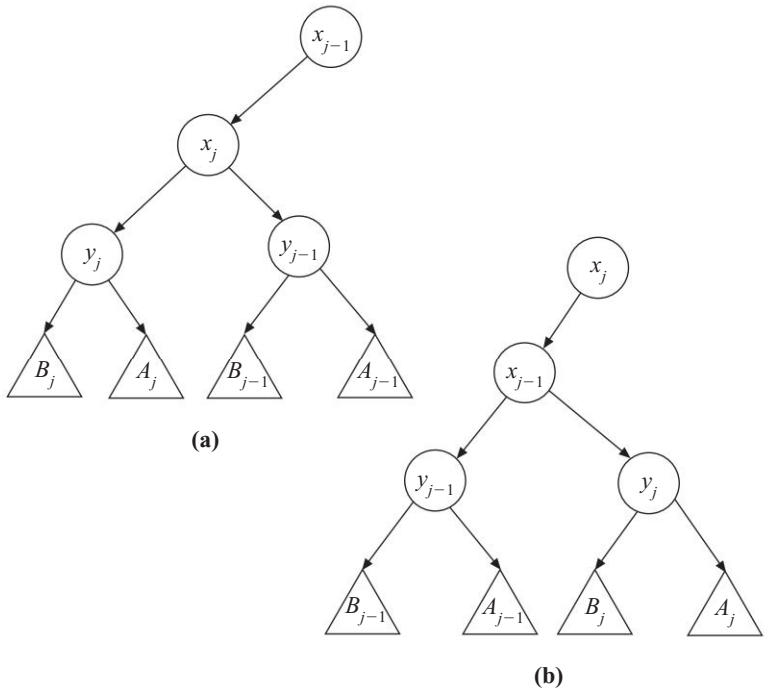


FIGURE 10–38 The result of melding a pair of subtrees in the third step of the delete minimum operation.



$$\begin{aligned}\phi_{after} &= r_{A_{j-1}} + r_{B_{j-1}} + r_{A_j} + r_{B_j} + \log (S_{A_{j-1}} + S_{B_{j-1}} + 1) + \log (S_{A_j} + S_{B_j} + 1) \\ &\quad + \log (S_{A_{j-1}} + S_{B_{j-1}} + S_{A_j} + S_{B_j} + 3) \\ &\quad + \log (S_{A_{j-1}} + S_{B_{j-1}} + S_{A_j} + S_{B_j} + 4) \\ \Delta\phi &= \phi_{after} - \phi_{before} \\ &= \log (S_{A_{j-1}} + S_{B_{j-1}} + S_{A_j} + S_{B_j} + 3) + \log (S_{A_j} + S_{B_j} + 2).\end{aligned}$$

$S_{A_{j-1}} + S_{B_{j-1}} + S_{A_j} + S_{B_j} + 4$ is the total number of nodes of the entire tree and $S_{A_j} + S_{B_j} + 2$ is the total number of the last binary tree. Let the number of the nodes of the tree consisting of x_i, y_i, A_i and B_i be n_i. Then the number of nodes in the subtrees are $n_1, n_2, \ldots, n_j$. The total change of potential is

$$\begin{aligned}\Delta\phi_{\text{third step}} &= \log (n_1 + n_2 + \cdots + n_j - 1) - \log (n_2 + n_3 + \cdots + n_j) \\ &\quad + \log (n_2 + n_3 + \cdots + n_j - 1) - \log (n_3 + n_4 + \cdots + n_j) \\ &\quad + \cdots + \log (n_{j-1} + n_j - 1) - \log (n_j) \\ &< \log (n - 2) - \log (n_j) \\ &< \log (n - 1).\end{aligned}$$

We can now see that for the entire delete minimum operation,

$$\begin{aligned}a_i &= t_i + \phi_i - \phi_{i-1} \\ &\le 2j + 1 - \log n + (2 \log n - 2j + 2) + \log (n - 1) \\ &\le 2 \log n + 3 = O(\log n).\end{aligned}$$

Although $O(\log n)$ is an upper bound for the delete minimum operation, it can be easily seen that this can also serve as the upper bound for all other operations. Consequently, we may conclude that the amortized time for the pairing heap operation is $O(\log n)$.

10-6 AMORTIZED ANALYSIS OF A DISJOINT SET UNION ALGORITHM

In this section, we shall discuss an algorithm which is easy to implement. Yet amortized analysis of it reveals a remarkable, almost-linear running time. The algorithm solves the problem of maintaining a collection of disjoint sets under the operation of union. More precisely, the problem is to carry out three kinds of operations on disjoint sets: makeset, which creates a new set; find, which locates the set containing a given element; and link, which combines two sets into one. As a way of identifying the sets, we shall assume that the algorithm maintains

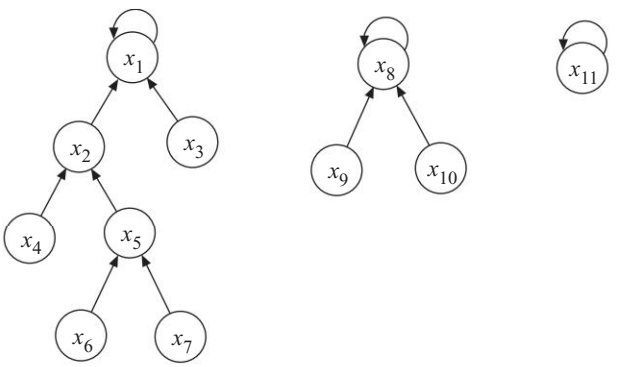
within each set an arbitrary but unique representative called the *canonical* element of the set. We formulate three set operations as follows:

(1) makeset(x): Create a new set containing the single element x, previously non-existent.
(2) find(x): Return the canonical element of the set containing element x.
(3) link(x, y): Form a new set that is the union of the two sets whose canonical elements are x and y. Destroy the two old sets. Select and return a canonical element for the new set. This operation assumes that $x \neq y$.

Note that there is no deletion in these operations.

To solve this problem, we represent each set by a rooted tree. The nodes of the tree are the elements of the set with the canonical element as the root of the tree. Each node x has a pointer pointing to $p(x)$, its parent in the tree; the root points to itself. To carry out makeset(x), we define $p(x)$ to be x. To carry out find(x), we follow parent pointers from x to the root of the tree containing x and return the root. To carry out link(x, y), we define $p(x)$ to be y and return y as the canonical element of the new set. See Figure 10–39. Operation find(x_6) would return x_1 and link(x_1, x_8) would make x_1 point to x_8.

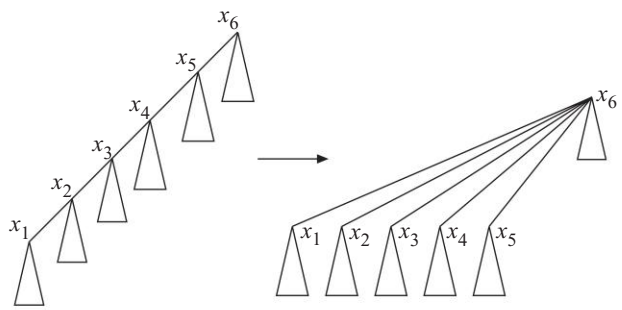
FIGURE 10–39 Representation of sets $\{x_1, x_2, \ldots, x_7\}$, $\{x_8, x_9, x_{10}\}$ and $\{x_{11}\}$.



This naïve algorithm is not very efficient, requiring $O(n)$ time per find operation in the worst case, where n is the total number of elements (makeset operation). By adding two heuristics to the method, we can improve its performance greatly. The first, called path compression, changes the structure of

the tree during a find operation by moving nodes closer to the root: When carrying out find(x), after locating the root r of the tree containing x, we make every node on the path from x to r point directly to r (see Figure 10–40). Path compression increases the time of a single find operation by a constant factor but saves enough time for later find operations.

FIGURE 10–40 Compression of the path $[x_1, x_2, x_3, x_4, x_5, x_6]$.



The second heuristics, called union by rank, keeps the trees shallow. The rank of a node x, denoted as $rank(x)$, is defined as follows:

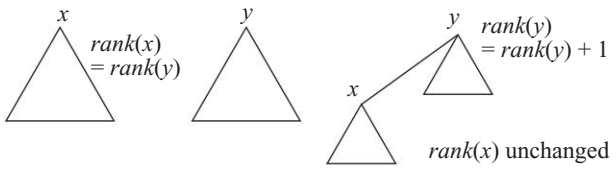
(1) When makeset(x) is performed, $rank(x)$ is assigned to 0.
(2) When the link operation is performed, let x and y be the two roots. There are two cases.
   (a) Case 1: $rank(x) = rank(y)$.
   In this case, we make x point to y, increase $rank(y)$ by one and return y as the canonical element. (See Figure 10–41(a).) The rank of x is not changed.
   (b) Case 2: $rank(x) < rank(y)$.
   In this case, we make x point to y and return y as the canonical element. The ranks of x and y remain the same. (See Figure 10–41(b).)
(3) When the path compression heuristics is performed, no rank is changed.

We shall now show some interesting properties of ranks as defined. These properties will be useful when we analyze the performance of the algorithm.

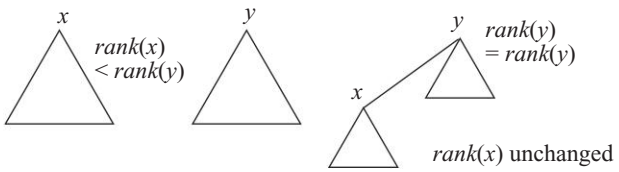
Property 1. *If* $x \neq p(x)$, then $rank(x) < rank(p(x))$.

That this property holds is due to the union by rank heuristics. When two trees are linked, the new root has the highest rank.

FIGURE 10–41 Union by ranks.



(a) Roots of equal ranks



(b) Roots of unequal ranks

Property 2. The value of $rank(x)$ is initially zero and increases as time passes until it is not a root any more; subsequently $rank(x)$ does not change. The value of $rank(p(x))$ is a non-decreasing function of time.

Note that as x is initialized, its rank is zero to start with. Then, as long as x remains as a root, its rank will never decrease. As soon as it has a parent node, its rank will never be changed any more because the path compression heuristics will not offset the rank.

Consider Figure 10–42. Suppose that this is a path which the algorithm traverses during a find operation. Because of Property 1,

$$rank(x_1) < rank(x_2) \ldots < rank(x_r).$$

After the path compression operation, this part of the tree looks like that shown in Figure 10–43. Now, x_1 has a new parent x_r and it is obvious that this new parent x_r has a much higher rank than the previous parent of x_1. Later, x_r may be attached to some other node with an even higher rank. Therefore, it is possible that x_1 may later have another new parent node with an even higher rank. This is why the value of $rank(p(x))$ never decreases.

Property 3. *The number of nodes in a tree with root x is at least* $2^{rank(x)}$.

Property 3 obviously holds for every tree when it is initialized. Now assume that Property 3 is true before two trees are linked. Let the roots of the two trees

FIGURE 10–42 A find path.

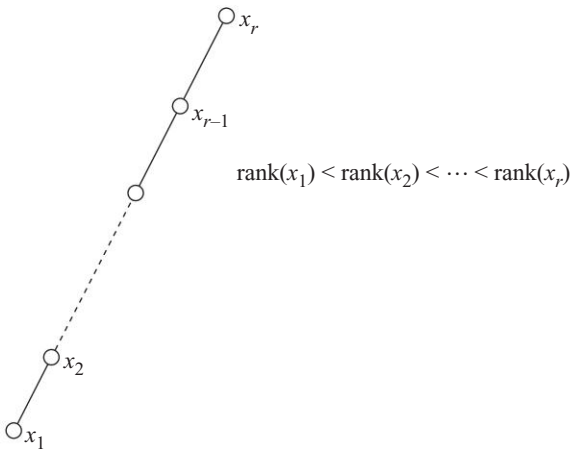
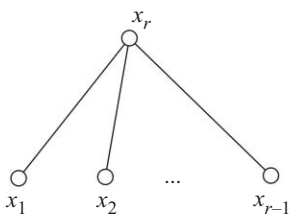


FIGURE 10–43 The find path in Figure 10–42 after the path compression operation.



be x and y. If $rank(x) < rank(y)$, the new tree has y as its root. The rank of y is not changed and the new tree has more nodes than before. Thus, Property 3 holds for this new tree. If $rank(x) = rank(y)$, then the new tree has at least $2^{rank(x)} + 2^{rank(y)} = 2 \times 2^{rank(y)} = 2^{rank(y)+1}$ nodes. But the rank of the new root is $rank(y) + 1$. Therefore, Property 3 again holds for this case.

Property 4. For any $k \geq 0$, *the number of nodes of rank k is at most* $n/2^k$, where n is the number of elements.

We may consider a node with rank k as the root of a tree. By Property 3, this node has at least 2^k nodes. Thus, there can be no more than $n/2^k$ nodes with rank k; otherwise there would be more than n nodes, which is impossible.

Property 5. *The highest rank is* $\log_2 n$.

The highest rank is achieved when there is only one tree. In this case, $2^k \leq n$ as indicated by Property 3. Thus, $k \leq \log_2 n$.

Our goal is to analyze the running time of an intermixed sequence of the three set operations. It is easily seen that both makeset(x) and link(x, y) can be implemented in constant time. Thus, in the following, we shall analyze the time needed for find operations only. We shall use m to denote the number of find operations and n to denote the number of elements. Thus, the number of makeset operations is n, the number of links is at most $n - 1$, and $m \geq n$. The analysis is difficult because the path compressions change the structure of the trees in a complicated way. Yet, by amortized analysis, we can obtain the actual worst case bound, $O(m\alpha(m, n))$, where $\alpha(m, n)$ is a functional inverse of Ackermann's function. For $i, j \geq 1$, Ackermann's function $A(i, j)$ is defined as

$$\begin{aligned} A(1, j) &= 2^j && \text{for } j \geq 1, \\ A(i, 1) &= A(i - 1, 2) && \text{for } i \geq 2, \\ A(i, j) &= A(i - 1, A(i, j - 1)) && \text{for } i, j \geq 2. \end{aligned}$$

And the inverse Ackermann's function $\alpha(m, n)$ for $m, n \geq 1$ is defined as

$$\alpha(m, n) = \min\{i \geq 1 \mid A(i, \lfloor m/n \rfloor) > \log_2 n\}.$$

The most important property of $A(i, j)$ is its explosive growth. Let us examine some of them:

$$\begin{aligned} &A(1, 1) = 2^1 = 2 \\ &A(1, 2) = 2^2 = 4 \\ &A(1, 4) = 2^4 = 16 \\ &A(2, 1) = A(1, 2) = 4 \\ &A(2, 2) = A(1, A(2, 1)) = A(1, 4) = 16 \\ &A(2, 3) = A(1, A(2, 2)) = A(1, 16) = 2^{16} \\ &A(3, 1) = A(2, 2) = 16. \end{aligned}$$

Thus, $\alpha(m, n) \leq 3$ for reasonable values of m/n and for $n < 2^{16} = 65{,}536$. In fact, for all practical purposes, $\alpha(m, n)$ is a constant not larger than four. In the end, we shall derive an upper bound of $O(m\alpha(m, n))$ on the total running time of the m find operations.

Before formally analyzing the algorithm, let us briefly describe the basic ideas of our analysis. Our analysis consists of the following steps:

(1) For each node of the trees, associate it with a level i. This level is related to the rank. We shall show that there are levels $1, 2, \ldots, \alpha(m, n) + 1$. If

a node has a high level, it indicates that this node has a large number of brothers, or this part of the tree is rather flat. For level i, integers are divided into block (i, j)'s. We shall explain this later.

(2) Through some mechanism, we can divide nodes into two kinds: the credit nodes and the debit nodes. For any path, the number of credit nodes is at most a constant. Thus, if a path is long, it is always due to a large number of debit nodes. Conversely, if a path is short, there are mostly credit nodes.

(3) The total time spent by the find operations is the sum of the number of credit nodes traversed by the find operations and the number of debit nodes traversed by the find operations. Since the total number of credit nodes traversed by the find operations is bounded by a constant, it is important to find an upper bound of the number of debit nodes traversed by the find operations.

(4) For a debit node belonging to block (i, j), if it is traversed by the algorithm of the find operation $b_{ij} - 1$ times where b_{ij} will be explained later, the level of this node will be increased to $i + 1$.

(5) Let n_{ij} denote the number of debit nodes in block (i, j). The value of $n_{ij}(b_{ij} - 1)$ is the number of time units spent by traversing the debit nodes by the find operations, which will raise all of the nodes in block (i, j) to level $i + 1$.

(6) Since the highest level is $\alpha(m, n) + 1$, $\sum_{i=1}^{\alpha(m,n)+1} \sum_{j \geq 0} n_{ij}(b_{ij} - 1)$ is the upper bound of the total number of debit nodes traversed by these m find operations.

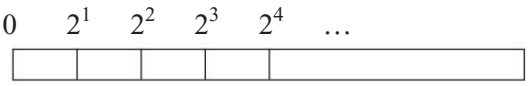
To start with, we should define levels. Let us review Ackermann's function again. Ackermann's function essentially partitions integers into blocks for different levels. Let us rewrite Ackermann's function again:

$$\begin{aligned} A(1, j) &= 2^j && \text{for } j \geq 1, \\ A(i, 1) &= A(i - 1, 2) && \text{for } i \geq 2, \\ A(i, j) &= A(i - 1, A(i, j - 1)) && \text{for } i, j \geq 2. \end{aligned}$$

We shall use the above function to partition integers. That is, for level i, the integers are partitioned into blocks defined by $A(i, j)$. Block $(i, 0)$ contains integers from 0 to $A(i, 1) - 1$ and block (i, j) contains integers from $A(i, j)$ to $A(i, j + 1) - 1$ for $j \geq 1$. For instance, for level 1,

$$A(1, j) = 2^j.$$

Thus, the blocks can be illustrated as follows:



For level 2,

$$A(2, 1) = 2^2$$
$$A(2, 2) = 2^4 = 16$$
$$A(2, 3) = 2^{16} = 65{,}536.$$

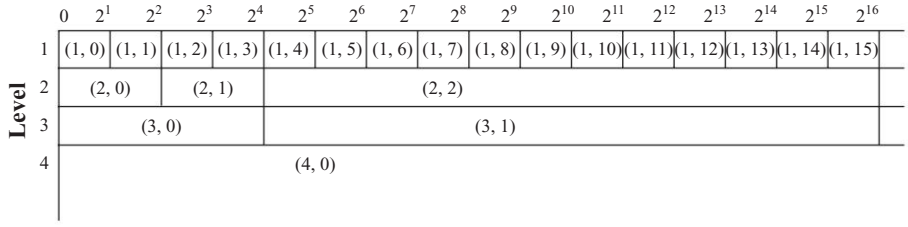
For level 3,

$$\begin{aligned} A(3, 1) &= 2^4 = 16 \\ A(3, 2) &= A(2, A(3, 1)) \\ &= A(2, 16) \\ &= A(1, A(2, 15)) \\ &= 2^{A(2, 15)}. \end{aligned}$$

Since $A(2, 3)$ is already a very large number, $A(2, 15)$ is so large that for practical purposes, $2^{A(2, 15)}$ can be regarded as infinity.

Combining these three levels, we have a diagram, shown in Figure 10–44.

FIGURE 10–44 Different levels corresponding to Ackermann's function.



Let us imagine that we have an integer $2^3 \le i < 2^4$. Then at level 1, it is within block (1, 3), at level 2, it is within block (2, 1) and at level 3, it is within block (3, 0).

As defined before, $p(x)$ denotes the parent of a node x. The level of x is now defined as the minimum level i such that rank(x) and rank($p(x)$) are in a common block of the level i partition. If $x = p(x)$, then the level of x is 0. Let us assume that $rank(x)$ is in block (1, 1) and $rank(p(x))$ is in block (1, 3), then x is in level 3. On the other hand, if $rank(x)$ is in block (1, 3) and $rank(p(x))$ is in block (1, 6), then x is in level 4.

How high can the level be? Certainly, if the level is so high that the first block of that level includes $\log_2 n$, then this level is high enough for one simple reason: $\log_2 n$ is the largest possible rank by Property 5. Let us now examine one example, say $n = 2^{16}$, which is already large enough. In this case, $\log_2 n = 16$. As shown in Figure 10–44, this is included in the first block of level 4. Recalling that

$$\alpha(m, n) = \min\{i \geq 1 \mid A(i, \lfloor m/n \rfloor) > \log_2 n\},$$

we can easily see that $A(\alpha(m, n), m/n) > \log_2 n$. In other words, $\alpha(m, n) + 1$ is the highest level that the nodes can be associated with.

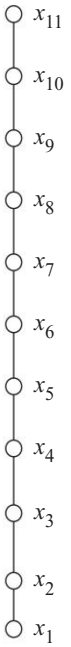
Let us now consider that a find operation which involves a path $x_1, \ldots, x_r$. A typical example is shown in Figure 10–45. In Figure 10–45, each node has a rank associated with it. Using $rank(x_i)$ and $rank(x_{i+1})$, we can determine the level of each x_i.

For each i, $0 \leq i \leq \alpha(m, n) + 1$, we assign the last node of level i on the path to be a credit node and the other nodes as debit nodes. All the credit nodes and debit nodes are also shown in Figure 10–45. According to the definition of credit nodes, *the total number of credit nodes of any find path is bounded by $\alpha(m, n) + 2$. Thus, the total number of credit nodes traversed by the algorithm is bounded by $m(\alpha(m, n) + 2)$.*

Let us now discuss how the level of a node can be very high. By definition, the level of a node x is high if the difference between $rank(x)$ and $rank(p(x))$ is very high. Therefore, we may ask the following question: How can the rank of the parent node of x be so much higher than that of x? There are only two operations: union and compression. When a union operation is performed, the difference between the rank of x and the rank of $p(x)$ can be only one.

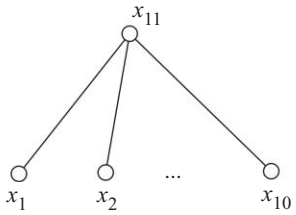
However, when a compression operation is performed, the difference between the rank of a node x and that of its parent node can be very marked. Imagine that we performed a find operation on the path shown in Figure 10–45. After the find operation, path will be compressed, shown in Figure 10–46. Although the rank of x_1 is still 1, suddenly the rank of its parent is 150. Thus, the level of x_1 is now increased from 1 to 4.

FIGURE 10–45 Credit nodes and debit nodes.



Rank	Level	Credit/Debit
150	0	Credit
18	2	Credit
17	1	Credit
13	4	Credit
12	1	Debit
10	1	Debit
7	2	Debit
5	1	Debit
3	3	Credit
2	1	Debit
1	2	Debit

FIGURE 10–46 The path of Figure 10–45 after a find operation.



In general, for a node on a find path, after a find operation, it will have a new parent node and this new parent node must have a much higher rank. We shall show that if x is in level i and is a debit node and $rank(x)$ is in block $(i-1, j')$, then its new parent, actually the root of the tree, will have a rank which is in block $(i-1, j')$ where $j' > j$.

Assume that the level of a node x is i and $rank(x)$ is in block $(i-1, j)$. Let the rank of $p(x)$, the parent of x, be in block $(i-1, j')$. Then, certainly $j' \geq j$ because $rank(p(x)) \geq rank(x)$. Furthermore, $j' \neq j$; otherwise, x will be in level $i-1$. Thus, we conclude that if the level of node x is i and $rank(x)$ is in block $(i-1, j')$, then the rank of $p(x)$ is in block $(i-1, j')$, where $j' > j$.

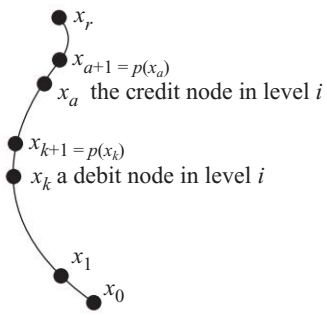
Consider Figure 10–47, where x_{k+1} is the parent of x_k and x_k is a debit node in level i. Since x_k is a debit node, by definition, there must be a credit node, say x_a, between x_k and x_r, the root of the path. Let x_{a+1} be the parent of x_a. Now, let the ranks of x_k, x_{k+1}, x_a, x_{a+1} and x_r be in blocks $(i-1, j_1)$, $(i-1, j_2)$, $(i-1, j_3)$, $(i-1, j_4)$ and $(i-1, j_5)$ respectively, as shown below.

	rank
x_k	$(i-1, j_1)$
x_{k+1}	$(i-1, j_2)$
x_a	$(i-1, j_3)$
x_{a+1}	$(i-1, j_4)$
x_r	$(i-1, j_5)$

Now, since the level of x_k is i, we have $j_2 > j_1$.
Since x_a is between x_{k+1} and x_r, we have $j_3 \geq j_2$.
Since the level of x_a is also i, we have $j_4 > j_3$.
Finally, we have

$j_5 \geq j_4$.

FIGURE 10–47 The ranks of nodes in a path.



Summarizing, we have

$j_1 < j_3 < j_5$.

The above discussion answers a general case. It is possible that $x_a = x_{k+1}$ and $x_r = x_{a+1}$. In this case, there are only three nodes relevant to our discussion, namely, x_k, x_{k+1} and x_r. It is easy to show that for the general and special cases, the following is true: Let x_k be a debit node in level i. Let x_{k+1} be its parent node and x_r be the root node. Let $rank(x_k)$, $rank(x_{k+1})$ and $rank(x_r)$ be in blocks $(i-1, j)$, $(i-1, j')$ and $(i-1, j'')$ respectively. Then $j < j' < j''$, shown in Figure 10–48.

FIGURE 10–48 The ranks of x_k, x_{k+1} and x_r in level $i-1$.

	$rank(x_k)$		$rank(x_{k+1})$		$rank(x_r)$	
	$i-1, j$	…	$i-1, j'$	…	$i-1, j''$	

After a find operation, each node of a path, except the root node, will become a leaf node, because of the path compression heuristics. A leaf node, by definition, is a credit node. Thus, a debit node will become a credit node after a find operation. It will remain a credit node as long as there is no union operations. After a link operation, it may become a debit node again.

For a debit node, each find operation will force it to have a new parent node whose rank will be larger than the rank of its previous parent node. Besides, as we showed above, the rank of the new parent node will be in a higher level $(i-1)$ block. Let b_{ij} denote the number of level $(i-1)$ blocks whose intersection with block (i, j) is non-empty. For example, $b_{22} = 12$ and $b_{30} = 2$. Then, we say that after $b_{ij} - 1$ find operations which traverse x while x is a debit node each time, $rank(x)$ and $rank(p(x))$ will be in different level i blocks. Or, more precisely, the level of x will be increased to $i+1$ after $b_{ij} - 1$ debit node changes. We may also say that after a maximum of $b_{ij} - 1$ debit node changes, the level of x will be increased by one.

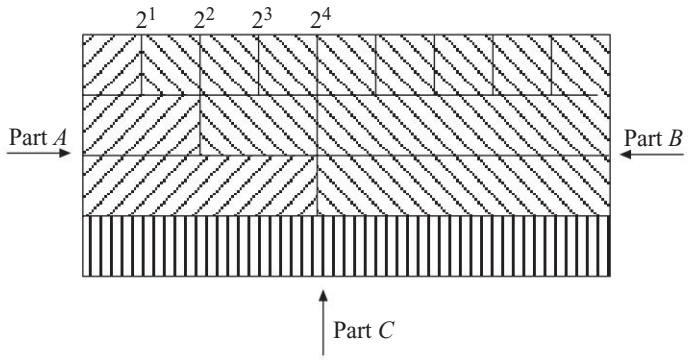
Let n_{ij} denote the total number of nodes whose ranks are in block (i, j). The total number of times that the find operations can traverse debit nodes is at most

$$Q = \sum_{i=1}^{\alpha(m,n)+1} \sum_{j \geq 0} n_{ij}(b_{ij} - 1).$$

In the following paragraphs, we shall show how an upper bound of Q can be derived. The derivation is rather complicated. In fact, if a reader is solely interested in the basic principles of amortized analysis, he may simply ignore this part of discussion.

Let us consider the partition diagram shown in Figure 10–44 again. We may divide this diagram into three parts, shown in Figure 10–49.

FIGURE 10–49 The division of the partition diagram in Figure 10–44.



Thus, we have

$$
\begin{aligned}
Q &= \sum_{i=1}^{\alpha(m,n)+1} \sum_{j\geq 0} n_{ij}(b_{ij}-1) \\
&= \sum_{i=1}^{\alpha(m,n)} n_{i0}(b_{i0}-1) && \text{(Part } A\text{)} \\
&\quad + \sum_{i=1}^{\alpha(m,n)} \sum_{j\geq 1} n_{ij}(b_{ij}-1) && \text{(Part } B\text{)} \\
&\quad + n_{\alpha(m,n)+1,0}(b_{\alpha(m,n)+1,0}-1). && \text{(Part } C\text{)}
\end{aligned}
$$

Let us first compute Part A. By definition,

$$
\begin{aligned}
block(i,0) &= [0, \ldots, A(i,1)-1] \\
&= [0, \ldots, A(i-1,2)-1] \\
&= [0, \ldots, A(i-1,1), \ldots, A(i-1,2)-1].
\end{aligned}
$$

This means that $block(i, 0)$ covers two $(i-1)$-level blocks, namely block $(i-1, 0)$ and block $(i-1, 1)$. Therefore, we have

$$b_{i0} = 2.$$

On the other hand, $n_{i0} \le n$. Thus, for Part A

$$\sum_{i=1}^{\alpha(m,n)} n_{i0}(b_{i0}-1) \le \sum_{i=1}^{\alpha(m,n)} n(2-1) = n\alpha(m, n).$$

Our next job is to compute Part B, which is

$$\sum_{i=1}^{\alpha(m,n)} \sum_{j\ge 1} n_{ij}(b_{ij}-1).$$

We first derive an upper bound for n_{ij}. By Property 4 discussed above,

$$\begin{aligned} n_{ij} &\le \sum_{k=A(i,j)}^{A(i,j+1)-1} n/2^k \\ &\le \sum_{k\ge A(i,j)} n/2^k \\ &\le 2n/2^{A(i,j)} \\ &= n/2^{A(i,j)-1}. \end{aligned} \tag{10–1}$$

For b_{ij}, $1 \le i \le \alpha(m, n)$ and $j \ge 1$, in general, we also have an upper bound.

$$\begin{aligned} block(i, j) &= [A(i, j), \ldots, A(i, j+1) - 1] \\ &= [A(i-1, A(i, j-1)), \ldots, A(i-1, A(i, j)) - 1] \\ &= [A(i-1, A(i, j-1)), A(i-1, A(i, j-1)) + 1), \ldots, \\ &\quad A(i-1, A(i, j)) - 1]. \end{aligned}$$

This means that for $1 \le i \le \alpha(m, n)$ and $j \ge 1$, $block(i, j)$ covers $A(i, j) - A(i, j-1)$ $(i-1)$-level blocks. Thus, in this case,

$$b_{ij} \le A(i, j). \tag{10–2}$$

Substituting (10–1) and (10–2) into Part *B*, we have

$$\begin{aligned}
&\sum_{i=1}^{\alpha(m,n)} \sum_{j\geq 1} n_{ij}(b_{ij}-1) \\
&\leq \sum_{i=1}^{\alpha(m,n)} \sum_{j\geq 1} n_{ij} b_{ij} \\
&\leq \sum_{i=1}^{\alpha(m,n)} \sum_{j\geq 1} (n/2^{A(i,j)-1}) A(i,j) \\
&= n \sum_{i=1}^{\alpha(m,n)} \sum_{j\geq 1} A(i,j)/2^{A(i,j)-1}.
\end{aligned}$$

Let $t = A(i, j)$. We have

$$\begin{aligned}
&\sum_{i=1}^{\alpha(m,n)} \sum_{j\geq 1} n_{ij}(b_{ij}-1) \\
&\leq n \sum_{i=1}^{\alpha(m,n)} \sum_{\substack{t=A(i,j) \\ j\geq 1}} t/2^{t-1} \\
&= n \sum_{i=1}^{\alpha(m,n)} ((A(i,1)/2^{A(i,1)-1}) + (A(i,2)/2^{A(i,2)-1}) + \cdots) \\
&\leq n \sum_{i=1}^{\alpha(m,n)} ((A(i,1)/2^{A(i,1)-1}) + ((A(i,1)+1)/2^{A(i,1)}) \\
&\quad + ((A(i,1)+2)/2^{A(i,1)+1}) + \cdots).
\end{aligned}$$

Let $a = A(i, 1)$. Thus, we have to find the value of the following:

$$S_1 = (a)/2^{a-1} + (a+1)/2^a + (a+2)/2^{a+1} + \cdots \quad \textbf{(10–3)}$$

$$2S_1 = a/2^{a-2} + (a+1)/2^{a-1} + (a+2)/2^a + \cdots \quad \textbf{(10–4)}$$

Taking (10–4) − (10–3), we have

$$\begin{aligned}
S_1 &= a/2^{a-2} + (1/2^{a-1} + 1/2^a + 1/2^{a+1} + \cdots) \\
&= a/2^{a-2} + 1/2^{a-2} = (a+1)/2^{a-2}.
\end{aligned}$$

Thus,

$$
\begin{aligned}
& n \sum_{i=1}^{\alpha(m,n)} ((A(i,1)/2^{A(i,1)-1}) + ((A(i,1)+1)/2^{A(i,1)}) \\
& \quad + ((A(i,1)+2)/2^{A(i,1)+1}) + \cdots) \\
& \quad = n \sum_{i=1}^{\alpha(m,n)} (A(i,1)+1)/2^{A(i,1)-2} \\
& \quad \le n((A(1,1)+1)/2^{A(1,1)-2} + (A(2,1)+1)/2^{A(2,1)-2} \\
& \qquad + (A(3,1)+1)/2^{A(3,1)-2} + \cdots) \\
& \quad \le n((2+1)/2^{2-2} + (3+1)/2^{3-2} + (4+1)/2^{4-2} + \cdots) \\
& \quad = n \sum_{t \ge 2} (t+1)/2^{t-2}.
\end{aligned}
$$

It can easily be proved that

$$
\begin{aligned}
\sum_{t \ge 2} (t+1)/2^{t-2} &= 3/2^{-1} + (1/2^0 + 1/2^1 + 1/2^2 + \cdots) \\
&= 6 + 2 \\
&= 8.
\end{aligned}
$$

Thus, for Part *B*, we have

$$\sum_{i=1}^{\alpha(m,n)} \sum_{j \ge 1} n_{ij}(b_{ij} - 1) \le 8n.$$

Finally, for Part *C*,

$$n_{\alpha(m,n)+1,0}(b_{\alpha(m,n)+1,0} - 1) \le n_{\alpha(m,n)+1,0} b_{\alpha(m,n)+1,0}. \quad \textbf{(10–5)}$$

$$n_{\alpha(m,n)+1,0} \le n. \quad \textbf{(10–6)}$$

$$
\begin{aligned}
& block(\alpha(m,n)+1, 0) \\
& \quad = [0, \ldots, A(\alpha(m,n), \lfloor m/n \rfloor) - 1] \\
& \quad = [0, \ldots, A(\alpha(m,n), 1), \ldots, A(\alpha(m,n), 2), \ldots, A(\alpha(m,n), \lfloor m/n \rfloor) - 1].
\end{aligned}
$$

This means that $block(\alpha(m,n)+1, 0)$ covers $\lfloor m/n \rfloor (i-1)$-level blocks. Thus,

$$b_{\alpha(m,n)+1,0} = \lfloor m/n \rfloor. \quad \textbf{(10–7)}$$

We have, for Part *C*, substituting (10–6) and (10–7) into (10–5),

$$n_{\alpha(m,n)+1,0}(b_{\alpha(m,n)+1,0} - 1) \le n\lfloor m/n \rfloor \le m.$$

In summary, for Q,

$$\begin{aligned} Q &\le n\alpha(m, n) + 8n + m \\ &= (\alpha(m, n) + 8)n + m. \end{aligned}$$

Note that the total time spent by the m find operations is equal to the sum of the number of credit nodes traversed by the find operations and the number of debit nodes traversed by the find operations. There are only two kinds of nodes: credit nodes and debit nodes. The number of credit nodes traversed is bounded by

$$(\alpha(m, n) + 2)m$$

and the number of debit nodes traversed is bounded by

$$(\alpha(m, n) + 8)n + m.$$

Thus, we conclude the average time spent by a find operations is

$$O(\alpha(m, n)).$$

Since $\alpha(m, n)$ is almost a constant, our amortized analysis indicates that the average time spent by a find operation is a constant.

10–7 AMORTIZED ANALYSIS OF SOME DISK SCHEDULING ALGORITHMS

The disk scheduling problem is an interesting and practically important problem in computer science. Consider a single disk. Data are stored in various cylinders. At any time, there are a set of requests to retrieve data. This set of requests is called a waiting queue and these requests are called waiting requests. The disk scheduling problem is to select one of the requests to be served.

For example, assume that there is a sequence of requests retrieving data located on cylinders 16, 2, 14, 5, 21 respectively and the disk head is initially at cylinder 0. Also assume that the time requested for moving from cylinder i to cylinder j is $|i - j|$. Let us now show how different disk scheduling algorithms produce different results.

First consider the first-come-first-serve algorithm (FCFS for short). The disk head will first come to cylinder 16, then to 2, then to 14 and so on. The total time used to serve these requests is $|0 - 16| + |16 - 2| + |2 - 14| + |14 - 5| + |5 - 21| = 16 + 14 + 12 + 9 + 16 = 67$.

Imagine that we use another algorithm, called the shortest-seek-time-first (SSTF for short) algorithm. In this algorithm, the disk head is always moved to the nearest cylinder. Thus, it first moves to cylinder 2, then to 5, then to 14 and so on. Thus, the total time servicing this sequence of requests is $|0 - 2| + |2 - 5| + |5 - 14| + |14 - 16| + |16 - 21| = 2 + 3 + 9 + 2 + 5 = 21$, which is substantially smaller than 48.

In this section, we assume that as the requests are served, new requests keep coming in. We shall assume that the waiting requests contain m requests. Any request outside of these m requests will be ignored. In other words, the maximum number of requests considered is m. On the other hand, we assume that there are at least W requests, where $W \geq 2$. The total number of cylinders is $Q + 1$. Let t_i denote the time used by the ith servicing. We are essentially interested in calculating $\sum_{i=1}^{m} t_i$. As can be expected, we are going to find an upper bound for $\sum_{i=1}^{m} t_i$. As we did before, we let $a_i(x) = t_i(x) + \phi_i(x) - \phi_{i-1}(x)$, where x denotes a particular algorithm, ϕ_i denotes the potential after the ith servicing of the request and $a_i(x)$ denotes the amortized time of the ith servicing.

We have,

$$\sum_{i=1}^{m} a_i(x) = \sum_{i=1}^{m} (t_i(x) - \phi_i(x) - \phi_{i-1}(x))$$

$$= \sum_{i=1}^{m} t_i(x) + \phi_m(x) - \phi_0(x)$$

$$\sum_{i=1}^{m} t_i(x) = \sum_{i=1}^{m} a_i(x) + \phi_0(x) - \phi_m(x).$$

Our strategy of finding an upper bound of $\sum_{i=1}^{m} t_i(x)$ is to find an upper bound of $a_i(x)$. Let us denote this upper bound by $A(x)$. Then

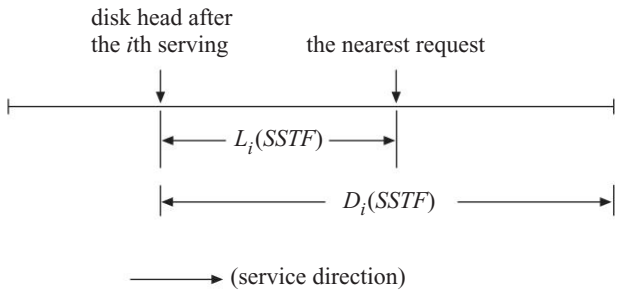
$$\sum_{i=1}^{m} t_i(x) \le mA(x) + \phi_0(x) - \phi_m(x).$$

In the rest of this section, we shall discuss how amortized analysis can be used to analyze the performances of two disk scheduling algorithms: the SSTF algorithm and the SCAN algorithm.

The Analysis of the Shortest-Seek-Time-First (SSTF) Algorithm

In the SSTF algorithm, the nearest request in the waiting queue is serviced. Consider the situation after the ith, $1 \le i \le m$, servicing. Let $N_i(SSTF)$ denote the number of waiting requests located in the present service direction. Let $L_i(SSTF)$ denote the distance between the disk head and the nearest request and $D_i(SSTF)$ be the number of cylinders in the service direction. The definitions of $L_i(SSTF)$ and $D_i(SSTF)$ are shown in Figure 10–50.

FIGURE 10–50 The definitions of $L_i(SSTF)$ and $D_i(SSTF)$.



The potential function for the *SSTF* algorithm can be defined as follows:

$$\phi_i(SSTF) = \begin{cases} L_i(SSTF) & \text{if } N_i(SSTF) = 1 \\ \min\{L_i(SSTF), D_i(SSTF)/2\} & \text{if } N_i(SSTF) > 1. \end{cases} \quad \textbf{(10–8)}$$

From (10–8), it can be easily proved that

$$\phi_i(SSTF) \ge 0. \quad \textbf{(10–9)}$$

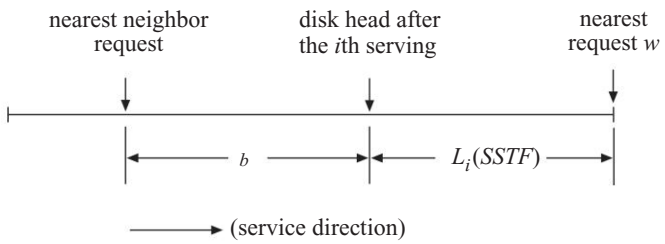
In the following, we shall show that

$$\phi_i(SSTF) \le Q/2. \quad \textbf{(10–10)}$$

We consider two cases:

Case 1. $N_i(SSTF) = 1$.
Since $W \geq 2$, as assumed, there is at least one request in the opposite direction, shown in Figure 10–51. Let the distance between the disk head and the nearest neighbor request in the opposite direction be denoted as b. Then $b + L_i(SSTF) \leq Q$. However, $L_i(SSTF) \leq b$; otherwise, the service direction will be reversed.

FIGURE 10–51 The case of $N_i(SSTF) = 1$.



Therefore,

$$2L_i(SSTF) \leq L_i(SSTF) + b \leq Q.$$

Or,

$$L_i(SSTF) \leq Q/2.$$

This means that

$$\phi_i(SSTF) = L_i(SSTF) \leq Q/2.$$

because of the definitions of $\phi_i(SSTF)$ as depicted in (10–8).

Case 2. $N_i(SSTF) \geq 2$.
Since $D_i(SSTF) \leq Q$, $D_i(SSTF)/2 \leq Q/2$. Therefore, according to (10–8),

$$\begin{aligned}\phi_i(SSTF) &= \min\{L_i(SSTF), D_i(SSTF)/2\} \\ &\leq D_i(SSTF)/2 \\ &\leq Q/2.\end{aligned}$$

Another upper bound of $\phi_i(SSTF)$ can be easily obtained, namely,

$$\phi_i(SSTF) \le L_i(SSTF). \tag{10–11}$$

The above equations will be used to find an upper bound of

$$a_i(SSTF) = t_i(SSTF) + \phi_i(SSTF) - \phi_{i-1}(SSTF).$$

To do this, let us consider the following two cases:

Case 1. $N_{i-1}(SSTF) = 1$
In this case, by (10–8), we have

$$\phi_{i-1}(SSTF) = L_{i-1}(SSTF) = t_i(SSTF). \tag{10–12}$$

Thus,

$$\begin{aligned} a_i(SSTF) &= t_i(SSTF) + \phi_i(SSTF) - \phi_{i-1}(SSTF) \\ &= t_i(SSTF) + \phi_i(SSTF) - L_{i-1}(SSTF) \\ &\le t_i(SSTF) + Q/2 - t_i(SSTF) \qquad \text{(by (10–10) and (10–12))} \\ &= Q/2. \end{aligned}$$

Case 2. $N_{i-1}(SSTF) \ge 2$.
There are again two subcases.

Case 2.1. $N_{i-1}(SSTF) \ge 2$ and $L_{i-1}(SSTF) \le D_{i-1}(SSTF)/2$.
In this case, according to (10–8),

$$\phi_{i-1}(SSTF) = \min\{L_{i-1}(SSTF), D_{i-1}(SSTF)/2\} \le L_{i-1}(SSTF).$$

Thus, we can prove, as we did in Case 1, that

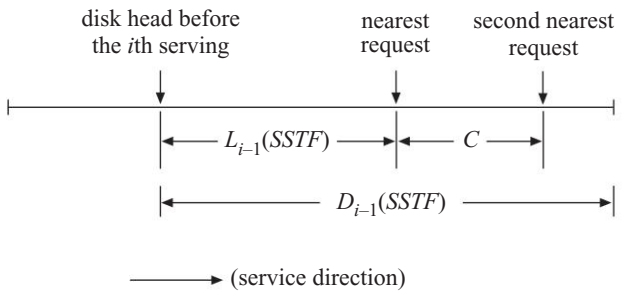
$$a_i(SSTF) \le Q/2.$$

Case 2.2. $N_{i-1}(SSTF) > 1$ and $L_i(SSTF) > D_{i-1}(SSTF)/2$.
In this case, according to (10–9),

$$\phi_{i-1}(SSTF) = D_{i-1}(SSTF)/2.$$

Since $N_{i-1}(SSTF) > 1$, there must be some requests, other than the nearest request, located in the service direction. Let c be the distance between the nearest request and the second nearest request in the service direction. Figure 10–52 describes the situation.

FIGURE 10–52 The situation for $N_{i-1}(SSTF) > 1$ and $L_{i-1}(SSTF) > D_{i-1}(SSTF)/2$.



Since the *SSTF* algorithm is used, $L_i(SSTF) \le c$. Thus, we have

$$L_i(SSTF) \le c \le D_{i-1}(SSTF) - L_{i-1}(SSTF). \quad \textbf{(10–13)}$$

Then

$$\begin{aligned}
a_i(SSTF) &= t_i(SSTF) + \phi_i(SSTF) - \phi_{i-1}(SSTF) \\
&\le t_i(SSTF) + L_i(SSTF) - \phi_{i-1}(SSTF) \quad \text{(by (10–11))} \\
&\le t_i(SSTF) + (D_{i-1}(SSTF) - L_{i-1}(SSTF)) - \phi_{i-1}(SSTF) \quad \text{(by (10–13))} \\
&= L_{i-1}(SSTF) + D_{i-1}(SSTF) - L_{i-1}(SSTF) - \phi_{i-1}(SSTF) \\
&\qquad \text{(because } t_i(SSTF) = L_{i-1}(SSTF)\text{)} \\
&= D_{i-1}(SSTF) - \phi_{i-1}(SSTF) \\
&= D_{i-1}(SSTF)/2 \quad \text{(because } \phi_{i-1}(SSTF) = D_{i-1}(SSTF)/2\text{)} \\
&\le Q/2. \qquad \text{(because } D_{i-1}(SSTF) \le Q\text{)}
\end{aligned}$$

Combining Cases 1 and 2, we have

$$a_i(SSTF) \le Q/2. \quad \textbf{(10–14)}$$

Combining (10–14), (10–9) and (10–10), we can find an upper bound of $\sum_{i=1}^{m} t_i(SSTF)$:

$$\sum_{i=1}^{m} t_i(SSTF) \le \sum_{i=1}^{m} a_i(SSTF) + \phi_0(SSTF) - \phi_m(SSTF).$$

Because the maximum of $\phi_0(SSTF)$ is $Q/2$ and the minimum of $\phi_m(SSTF)$ is 0, we have

$$\begin{aligned}\sum_{i=1}^{m} t_i(SSTF) &\le \sum_{i=1}^{m} a_i(SSTF) + \phi_0(SSTF) - \phi_m(SSTF) \\ &\le mQ/2 + Q/2 - 0 \\ &= (m+1)Q/2. \end{aligned} \quad \textbf{(10–15)}$$

Equation (10–15) indicates that for the SSTF algorithm, the total time consumed by m requests is $(m + 1)Q/2$. Or, equivalently, the average time consumed by a request for this algorithm is roughly $Q/2$.

To show that the upper bound $(m + 1)Q/2$ cannot be tightened, let us consider a sequence of m requests which are located respectively on cylinders as follows:

$$(Q/2, Q, (0, Q/2)^{(m-3)/2}, 0)$$

where x^y means $\overbrace{(x, x, \ldots, x)}^{y}$. Suppose that the number of waiting requests is 2 at any time and $(m - 3)$ can be divided by 2. In this case, the disk head will be oscillating between cylinder $Q/2$ and 0, as follows:

$$((Q/2, 0)^{(m-1)/2}, Q).$$

Thus, the total service time of this request is

$$(Q/2 + Q/2)(m - 1)/2 + Q = (m + 1)Q/2.$$

This means that the upper bound expressed in (10–15) cannot be further tightened.

The Amortized Analysis of SCAN Algorithm

As one can see in the previous discussion, the SSTF algorithm may force the disk head to oscillate. The SCAN algorithm avoids this problem by choosing the nearest request in the present sweep direction. Thus, if the disk head is moving

in one direction, it would continue doing so until there is no request in this direction. Then the disk head reverses the direction.

Consider the status after the ith servicing. We now define two terms, $N_i(SCAN)$ and $D_i(SCAN)$ as follows: If the sweep direction is not changed in the $(i + 1)$-th servicing, $N_i(SCAN)$ and $D_i(SCAN)$ are defined as the number of requests having been serviced and the distance which the disk head has been moved in the current sweep respectively; otherwise, both $N_i(SCAN)$ and $D_i(SCAN)$ are set to zero. $N_0(SCAN)$ and $D_0(SCAN)$ are both zero. The potential function of SCAN can be defined as

$$\phi_i(SCAN) = N_i(SCAN)Q/W - D_i(SCAN). \quad \textbf{(10–16)}$$

For SSTF, we showed that $a_i(SSTF) \le Q/2$. For SCAN, we shall show that $a_i(SCAN) \le Q/W$.

Consider the ith servicing, let $t_i(SCAN)$ denote the time for the ith servicing. There are again two cases to be considered.

Case 1. The sweep direction is not changed for the $(i + 1)$-th servicing. In this case, obviously, we have

$$\begin{aligned}
N_i(SCAN) &= N_{i-1}(SCAN) + 1 \text{ and} \\
D_i(SCAN) &= D_{i-1}(SCAN) + t_i(SCAN) \\
a_i(SCAN) &= t_i(SCAN) + \phi_i(SCAN) - \phi_{i-1}(SCAN) \\
&= t_i(SCAN) + (N_i(SCAN)Q/W - D_i(SCAN)) \\
&\quad - (N_{i-1}(SCAN)Q/W - D_{i-1}(SCAN)) \\
&= t_i(SCAN) + ((N_{i-1}(SCAN) + 1)Q/W - (D_{i-1}(SCAN) \\
&\quad + t_i(SCAN)) - (N_{i-1}(SCAN)Q/W - D_{i-1}(SCAN)) \\
&= Q/W.
\end{aligned}$$

Case 2. The sweep direction is changed from the $(i + 1)$th servicing. In this case, $N_i(SCAN) = D_i(SCAN)$ and $\phi_i(SCAN) = 0$.

Since the disk head was initially located at cylinder 0 and the minimum of the waiting queue is W by assumption, the minimum number of requests serviced in one sweep is W, too. That is, $N_{i-1}(SCAN) > (W - 1)$. Thus,

$$\begin{aligned}
a_i(SCAN) &\le t_i(SCAN) - ((W - 1)Q/W - D_{i-1}(SCAN)) \\
&\le (t_i(SCAN) + D_{i-1}(SCAN) - Q) + Q/W.
\end{aligned}$$

Since the maximum distance which the disk head can move in one sweep is Q,

$$t_i(SCAN) + D_{i-1}(SCAN) \le Q.$$

Therefore,

$$a_i(SCAN) \le Q/W. \tag{10–17}$$

The above discussion shows that

$$a_i(SCAN) \le A(SCAN) = Q/W,$$

where $A(SCAN)$ is an upper bound of $a_i(SCAN)$.

We now have

$$\begin{aligned}\sum_{i=1}^{m} t_i(SCAN) &\le \sum_{i=1}^{m} a_i(SCAN) + \phi_0(SCAN) - \phi_m(SCAN) \\ &\le mQ/W + \phi_0(SCAN) - \phi_m(SCAN).\end{aligned}$$

But $\phi_0(SCAN) = 0$. Thus,

$$\sum_{i=1}^{m} t_i(SCAN) \le mQ/W - \phi_m(SCAN).$$

To estimate $\phi_m(SCAN)$, we again consider two cases:

Case 1. $N_m(SCAN) = D_m(SCAN) = 0.$ (10–18)
In this case, $\phi_m(SCAN) = 0$.

Case 2. One of $N_m(SCAN)$ and $D_m(SCAN)$ is not zero.
In this case, $N_m(SCAN) \ge 1$ and $D_m(SCAN) \le Q$.
Thus,

$$\begin{aligned}\phi_m(SCAN) &= N_m(SCAN)Q/W - D_m(SCAN) \\ &\ge Q/W - Q.\end{aligned} \tag{10–19}$$

Let us combine (10–18) and (10–19) by noting that

$$Q/W - Q \le 0.$$

We conclude that

$$\phi_m(SCAN) \geq Q/W - Q. \tag{10–20}$$

Finally, we have

$$\begin{aligned}\sum_{i=1}^{m} t_i(SCAN) &\leq mQ/W - \phi_m(SCAN) \\ &= mQ/W - (Q/W - Q) \\ &\leq (m-1)Q/W + Q. \end{aligned} \tag{10–21}$$

Formula (10–21) indicates that the total time consumed by those m requests cannot be greater than $(m-1)Q/W + Q$. The average time consumed by one request for the SCAN algorithm is not greater than $((m-1)Q/W + Q)/m$.

As we did before, we can now prove that the upper bound $(m-1)Q/W + Q$ cannot be further tightened. We prove this by considering a sequence of m requests which are respectively located on cylinders

$$((Q^4, 0^4)^{(m-1)/8}, Q)$$

and assume that the number of waiting requests is 4 at any time. Let $W = 4$. Suppose that $(m-1)$ can be divided by 8. Then, this sequence of requests should be also scheduled and processed by the sequence

$$((Q^4, 0^4)^{(m-1)/8}, Q).$$

The total service time of this sequence is

$$((m-1)/8)2Q + Q = ((m-1)/(2W))2Q + Q = (m-1)Q/W + Q.$$

This shows that the upper bound of the total time expressed in (10–21) cannot be tightened.

Now, let us conclude this section by comparing (10–15) and (10–21). From (10–15), we have

$$t_{ave}(SSTF) = \sum_{i=1}^{m} t_i(SSTF)/m \leq (m+1)Q/(2m). \tag{10–22}$$

As $m \to \infty$, we have

$$t_{ave}(SSTF) \le Q/2. \tag{10–23}$$

From (10–21), we can prove that

$$t_{ave}(SCAN) \le Q/W. \tag{10–24}$$

Since $W \ge 2$, we may conclude that

$$t_{ave}(SCAN) \le t_{ave}(SSTF)$$

through amortized analysis of these two algorithms.

10–8 THE EXPERIMENTAL RESULTS

Amortized analysis often involves a lot of mathematics. It is never intuitively clear to many researchers that the result should be so. For instance, it is really not obvious at all to an ordinary person that the average performance of the disjoint set union algorithm is almost a constant. We therefore implemented the disjoint set union algorithm. We used a set of 10,000 elements. Whether the next instruction is a find, or a link, is determined by a random number generator. The program was written in Turbo Pascal and executed on an IBM PC. Table 10–9 summarizes the result.

TABLE 10–9 Experimental results of the amortized analysis of the disjoint set union algorithm.

No. of operations	Total time (msec)	Average time (μsec)
2000	100	50
3000	160	53
4000	210	53
5000	270	54
7000	380	54
9000	490	54
11000	600	55
13000	710	55
14000	760	54
15000	820	55

From the experimental results, we can see that the amortized analysis accurately predicts the behavior of the disjoint set union algorithm. For a sequence of operations, the total time involved increases. However, the average time per operation is a constant which is predicted by the amortized analysis.

10–9 NOTES AND REFERENCES

The term, amortized analysis, first appeared in Tarjan (1985). However, this concept was used by researchers before 1985 implicitly. For example, the analysis of 2–3 trees in Brown and Tarjan (1980) and weak B-trees in Huddleston and Mehlhorn (1982) used this concept, although "amortized" was not used then.

The amortized analysis of skew heaps can be found in Sleator and Tarjan (1986). Mehlhorn and Tsakalidis (1986) gave an amortized analysis of the AVL-tree. The amortized analysis of self-organizing sequential search heuristics appeared in Bentley and McGeoch (1985). For amortized analysis of pairing heaps, see Fredman, Sedgewick, Sleator and Tarjan (1986). The amortized analysis of the disjoint set union algorithm can be found in Tarjan (1983) and Tarjan and Van Leeuwen (1984).

Amortized analysis is often used within an algorithm involving a pertinent data structure and a sequence of operations repeatedly applied on this data structure. For instance, in Fu and Lee (1991), a data structure is needed so that trees can be handled dynamically and efficiently. That is, many tree operations, such as examining an edge, inserting an edge, and so on, will be performed. In such a case, the dynamic tree (Sleator and Tarjan, 1983) can be used and the analysis is an amortized analysis.

Amortized analysis can be applied to analyze some practical strategies for a sequence of operations. Chen, Yang and Lee (1989) applied amortized analysis to analyze some disk scheduling policies.

Very few textbooks discuss the concept of amortized analysis, but Purdom and Brown (1985a) and Tarjan (1983) do contain discussions on it.

10–10 FURTHER READING MATERIALS

For a review and also introductory discussion of the amortized analysis, see Tarjan (1985). The following papers are all recently published ones on amortized analysis and highly recommended for further reading: Bent, Sleator and Tarjan (1985); Eppstein, Galil, Italiano and Spencer (1996); Ferragina (1997); Henzinger (1995); Italiano (1986); Karlin, Manasse, Rudolph and Sleator (1988);

Kingston (1986); Makinen (1987); Sleator and Tarjan (1983); Sleator and Tarjan (1985a); Sleator and Tarjan (1985b); Tarjan and Van Wyk (1988); and Westbrook and Tarjan (1989).

Exercises

10.1 For the stack problem discussed in Section 10–1, prove that the upper bound is 2 without using a potential function. Give an example to show that this upper bound is also tight.

10.2 Imagine that there is a person whose sole income is his monthly salary, which is k units per month. He can, however, spend any amount of money as long as his bank account has enough such money. He puts k units of income into his bank account every month. Can you perform an amortized analysis on his behavior? (Define your own problem. Note that he cannot withdraw a large amount of money all the time.)

10.3 Amortized analysis somehow implies that the concerned data structure has a certain self-organizing mechanism. In other words, when it becomes very bad, there is a chance that it will become good afterwards. In this sense, can hashing be analyzed by using amortized analysis? Do some research on this topic. You may be able to publish some papers, perhaps.

10.4 Select any algorithm introduced in this chapter and implement it. Perform some experiments to see if the amortized analysis makes sense.

10.5 Read Sleator and Tarjan (1983) on the dynamic tree data structure.

chapter

11

Randomized Algorithms

The concept of randomized algorithms is relatively new. In every algorithm introduced so far, each step in the algorithm is deterministic. That is, we never, in the middle of executing an algorithm, make an arbitrary choice. In randomized algorithms, which we shall introduce in this chapter, we do make arbitrary choices. This means that some actions are taken at random.

Since some actions are executed at random, the randomized algorithms introduced later have the following properties:

(1) In the case of optimization problems, a randomized algorithm gives an optimal solution. However, since random actions are taken in the algorithm, the time complexity of a randomized optimization algorithm is randomized. Thus, the average-case time complexity of a randomized optimization algorithm is more important than its worst-case time complexity.
(2) In the case of decision problems, a randomized algorithm does make mistakes from time to time. Yet, it can be said that the probability of producing wrong solutions is exceedingly small; otherwise, this randomized algorithm is not going to be useful.

11-1 A Randomized Algorithm to Solve the Closest Pair Problem

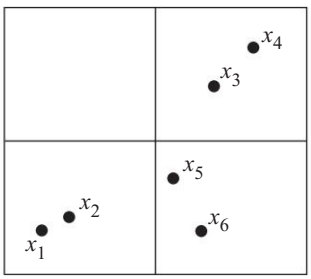
The closest pair problem was introduced in Chapter 4. In Chapter 4, we showed that this problem can be solved by the divide-and-conquer approach in $O(n \log n)$ time. This time complexity is for worst cases. In this section, we shall show that there exists a randomized algorithm and the average-case time complexity of using this algorithm to solve the closest pair problem is $O(n)$.

Let $x_1, x_2, \ldots, x_n$ be n points in the 2-dimensional plane. The closest pair problem is to find the closest pair x_i and x_j for which the distance between x_i and x_j is the smallest, among all possible pairs of points. A straightforward approach to solve this problem is to evaluate all the $n(n-1)/2$ inter-distances and find the minimum among all these distances.

The main idea of the randomized algorithm is based upon the following observation: *If two points x_i and x_j are distant from each other, then their distance will unlikely be the shortest and thus can well be ignored*. With this idea, the randomized algorithm would first partition the points into several clusters in such a way that points within each cluster are close to one another. We then only calculate distances among points within the same cluster.

Let us consider Figure 11–1. As shown in Figure 11–1, there are six points. If we partition these six points into three clusters: $S_1 = \{x_1, x_2\}$, $S_2 = \{x_5, x_6\}$ and $S_3 = \{x_3, x_4\}$, then we shall now only calculate three distances, namely $d(x_1, x_2)$, $d(x_5, x_6)$ and $d(x_3, x_4)$. Afterwards, we find the minimum among these three distances. If we do not partition points into clusters, we must calculate $(6 \cdot 5)/2 = 15$ distances.

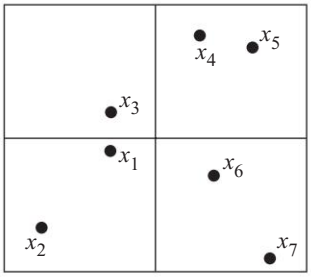
FIGURE 11–1 The partition of points.



Of course, this discussion is quite misleading because there is no guarantee that the strategy would work. In fact, the strategy may be seen as a divide-without-conquer strategy. There is a dividing process, but no merging process. Let us consider Figure 11–2. In Figure 11–2, we can see that the closest pair is $\{x_1, x_3\}$. Yet, we have partitioned these two points into two different clusters.

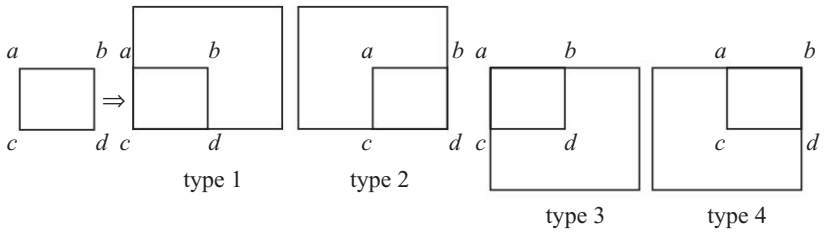
If we partition the entire space into squares with side lengths equal to δ which is not smaller than the shortest distance, then after all the within-cluster

FIGURE 11–2 A case to show the importance of inter-cluster distances.



distances are calculated, we may double the side length of the square and produce larger squares. The shortest distance must be within one of these enlarged squares. Figure 11–3 shows how four enlarged squares correspond to one square. Each enlarged square belongs to a certain type, indicated in Figure 11–3.

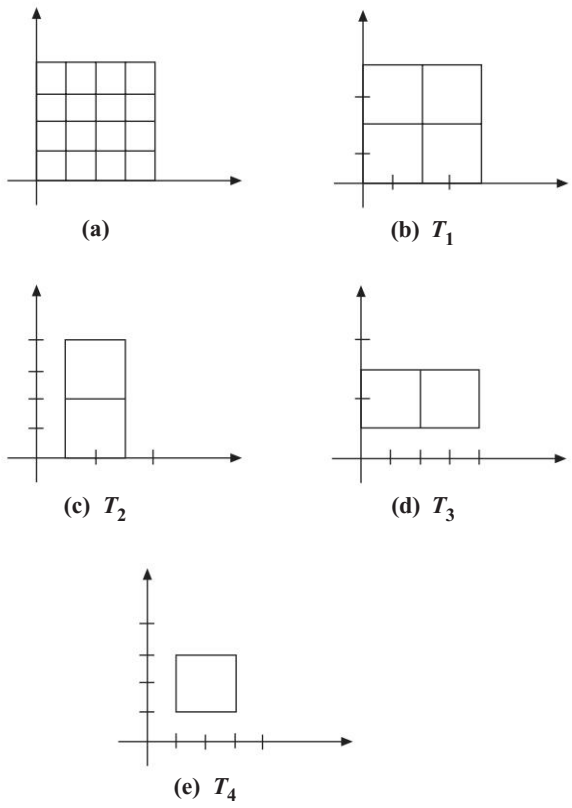
FIGURE 11–3 The production of four enlarged squares.



Let us imagine that the entire space is already divided into squares with width δ. Then the enlargement of these squares will induce four sets of enlarged squares, denoted as T_1, T_2, T_3 and T_4 corresponding to type 1, type 2, type 3 and type 4 squares respectively. A typical case is illustrated in Figure 11–4.

Of course, the critical question is to find the appropriate mesh size δ. If δ is too large, the original square will be very large and a large number of distances will be calculated. In fact, if δ is very large, there is almost no dividing and our problem becomes the original problem. On the other hand, δ cannot be too small because it cannot be smaller than the shortest distance. In the randomized algorithm, we randomly select a subset of points and find the shortest distance among this subset of points. This shortest distance will become our δ.

FIGURE 11–4 Four sets of enlarged squares.



Algorithm 11–1 □ A randomized algorithm for finding a closest pair

Input: A set S consisting of n elements $x_1, x_2, \ldots, x_n$, where $S \subseteq R^2$.

Output: The closest pair in S.

Step 1. Randomly choose a set $S_1 = \{x_{i1}, x_{i2}, \ldots, x_{im}\}$ where $m = n^{\frac{2}{3}}$. Find the closest pair of S_1 and let the distance between this pair of points be denoted as δ.

Step 2. Construct a set of squares T with mesh size δ.

Step 3. Construct four sets of squares T_1, T_2, T_3 and T_4 derived from T by doubling the mesh size to 2δ.

Step 4. For each T_i, find the induced decomposition $S = S_1^{(i)} \cup S_2^{(i)} \cup \cdots \cup S_j^{(i)}$, $1 \le i \le 4$, where $S_j^{(i)}$ is a non-empty intersection of S with a square of T_i.

Step 5. For each x_p, x_q, $\in S_j^{(i)}$, compute $d(x_p, x_q)$. Let x_a and x_b be the pair of points with the shortest distance among these pairs. Return x_a and x_b as the closest pair.

Example 11–1

We are given a set S of 27 points, shown in Figure 11–5. In Step 1, we randomly choose $27^{\frac{2}{3}} = 9$ elements $x_1, x_2, \ldots, x_9$. It can be seen that the closest pair is (x_1, x_2). We use the distance δ between x_1 and x_2 as the mesh size to construct 36 squares as required in Step 2. There will be four sets of squares T_1, T_2, T_3 and T_4:

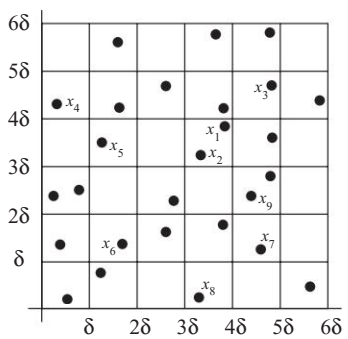
$$T_1 = \{[0 : 2\delta, 0 : 2\delta], [2\delta : 4\delta, 0 : 2\delta], [4\delta : 6\delta, 0 : 2\delta], \ldots, [4\delta : 6\delta, 4\delta : 6\delta]\}.$$

$$T_2 = \{[\delta : 3\delta, 0 : 2\delta], [3\delta : 5\delta, 0 : 2\delta] \ldots, [3\delta : 5\delta, 4\delta : 6\delta]\}.$$

$$T_3 = \{[0 : 2\delta, \delta : 3\delta], [2\delta : 4\delta, \delta : 3\delta], [4\delta : 6\delta, \delta : 3\delta], \ldots, [4\delta : 6\delta, 3\delta : 5\delta]\}.$$

$$T_4 = \{[\delta : 3\delta, \delta : 3\delta], [3\delta : 5\delta, \delta : 3\delta], \ldots, [3\delta : 5\delta, 3\delta : 5\delta]\}.$$

FIGURE 11–5 An example illustrating the randomized closest pair algorithm.



The total number of mutual distance computations is

$$N(T_1): \; C_2^4 + C_2^3 + C_2^2 + C_2^3 + C_2^3 + C_2^3 + C_2^3 + C_2^3 + C_2^3 = 28.$$

$$N(T_2): \; C_2^3 + C_2^3 + C_2^2 + C_2^5 + C_2^3 + C_2^4 = 26.$$

$$N(T_3): \; C_2^4 + C_2^3 + C_2^3 + C_2^3 + C_2^4 + C_2^3 = 24.$$

$$N(T_4): \; C_2^3 + C_2^4 + C_2^3 + C_2^5 = 22.$$

Among $28 + 24 + 22 + 26 = 100$ pairs, the closest one is found to be within $[3\delta : 5\delta, 3\delta : 5\delta]$.

11–2 THE AVERAGE PERFORMANCE OF THE RANDOMIZED CLOSEST PAIR PROBLEM

In the randomized closest pair problem presented in the previous section, the first step is to find the closest pair among $n^{\frac{2}{3}}$ points. This closest pair can be found by recursively applying this randomized algorithm. That is, we randomly choose $(n^{\frac{2}{3}})^{\frac{2}{3}} = n^{\frac{4}{9}}$ points from the original set. To find the closest pair among these $n^{\frac{4}{9}}$ points, we can use the straightforward method which requires $(n^{\frac{4}{9}})^2 = n^{\frac{8}{9}} < n$ distance computations. Still, this does not mean that Step 1 can be done in $O(n)$ time. We shall temporarily suspend the discussion of the time complexity of Step 1 at present and show later that Step 1 will indeed take only $O(n)$ time.

Obviously, Step 2 and Step 3 can all be done in $O(n)$ time. Step 4 can be executed by using the hashing technique. Using the hashing technique, we can easily decide which points are located in a certain square. This means that Step 4 can be computed in $O(n)$ time.

The expected number of distance computations in Step 5 is by no means easy to decide. Actually, we do not have a formula for this expected number of distance computations. Indeed, we shall show that the probability that the expected number of distance computations in Step 5 is $O(n)1-2e^{-cn^{\frac{1}{6}}}$ which rapidly approaches one as n grows. Or, to put it in another way, the probability that the expected number of distance computations is $O(n)$ is very high.

Why can we make the above conclusion? Let us note that in our randomized closest pair algorithm, the mesh size is δ. Let us denote this partition by T and the total number of distance computations for this partition by $N(T)$. We shall show later that there exists a particular partition, called T_0, whose mesh size is δ_0 with the following two properties:

(1) $n \le N(T_0) \le C_0 n$.

(2) The probability that $\delta \le \sqrt{2}\,\delta_0$ is $1-2e^{-cn^{\frac{1}{6}}}$, which is very high.

We shall explain why such a partition exists later. Meanwhile, let us simply assume that this is true and proceed from here.

Imagine that we quadruple the mesh size from δ_0 to $4\delta_0$. This quadrupling will induce 16 sets of squares. Let us denote these sets by T_i, $i = 1, 2, \ldots, 16$. Since the probability that $\delta \le \sqrt{2}\,\delta_0$ is $1 - 2e^{-cn^{\frac{1}{6}}}$, the probability that each square in T falls into at least one square of T_i, $i = 1, 2, \ldots, 16$, is $1 - 2e^{-cn^{\frac{1}{6}}}$. Let the total number of distance computations for partition T_i be $N(T_i)$. Then the probability that

$$N(T) \le \sum_{i=1}^{16} N(T_i)$$

is true is $1 - 2e^{-cn^{\frac{1}{6}}}$.

We now calculate $N(T)$. We start from $N(T_i)$. Each square in T_i is 16 times larger than a square in T_0. Let the square in T_0 with the largest number of elements have k elements. Let S_{ij} denote the set of the 16 squares in T_0 which belong to the jth square of T_i. Let the square in S_{ij} with the largest number of elements have k_{ij} elements. The total number of distance computations in T_0 is greater than $\sum_j k_{ij}(k_{ij} - 1)/2$. That is, $\sum_j k_{ij}(k_{ij} - 1)/2 \le N(T_0) \le C_0 n$. And, the number of distance computations in the jth square of T_i is less than $16k_{ij}(16k_{ij} - 1)/2$. Thus, $N(T_i) \le \sum_j 16k_{ij}(16k_{ij} - 1)/2 \le C_i n$, where C_i is a constant. Therefore, $N(T) \le \sum_{i=1}^{16} N(T_i) = O(n)$ with probability $1 - 2e^{-cn^{\frac{1}{6}}}$. As n is large, $N(T) \le O(n)$ with probability 1, as $e^{-cn^{\frac{1}{6}}}$ diminishes to zero rapidly.

In the following, we shall now explain why we may conclude that there exists a partition which has the above two properties. The reasoning is rather complicated. Before giving the main reasoning, we shall first define some terms.

Let D be a partition of a set of points. That is, $S = S_1 \bigcup S_2 \bigcup \cdots \bigcup S_r$ and $S_i \bigcap S_j = \varnothing$ if $i \ne j$. If $T \subseteq S$ is a choice of m elements, then we call T a success on D if at least two elements of T were chosen from the same part S_i of the partition for some i. If D' is another partition of S, we say that D dominates D' if for every m, the probability of success on D with a choice of m elements is greater than or equal to the success on D' with a choice of m elements.

According to the above definition, it can be easily seen that the following statements are true:

(1) The partition (2, 2, 2) dominates (3, 1, 1, 1). This means that any partition of a set of six elements into three pairs dominates any partition of the same set into a triplet and three singletons. Why does it dominate? The reason is quite easy to understand. For the partition (2, 2, 2), there is a much larger probability that two points will be drawn from the same square than the partition (3, 1, 1, 1) because there is only one square with more than one element.
(2) The partition (3, 3) dominates (4, 1, 1).
(3) The partition (4, 4) dominates (5, 1, 1, 1).
(4) The partition (p, q), $p \geq 5$, $q \geq 5$, dominates $(\ell, 1, 1, \ldots, 1)$ where the number of ones is $p + q - \ell$, $\ell(\ell - 1) \leq p(p - 1) + q(q - 1)\ell \leq (\ell + 1)$.

It can be seen that if partition D dominates partition D', then the number of distance computations in D' is less than or equal to that in D.

Let $N(D)$ be the number of distance computations required in D. According to the preceding statements and discussion, it is obvious that for every partition D of any finite set S of points, there exists another partition D' of the same set such that $\lambda N(D) \leq N(D')$, where D dominates D' and all sets of D' with one exception are singletons where λ is a positive number smaller than or equal to one.

Let us assume that D is a partition of the set S, $|S| = n$ and $n \leq N(D)$. Let D' be a partition such that $S = H_1 \bigcup H_2 \cdots \bigcup H_{k'}$, which is dominated by D, where $|H_1| = b$, $|H_i| = 1$ for $i = 2, 3, \ldots, k'$ and $\lambda n \leq N(D')$. This implies that $\lambda n \leq b(b - 1)/2$. Thus, we have $b \geq \sqrt{2\lambda n}$. Let $c = \sqrt{2\lambda}$, we have $b \geq c\sqrt{n}$.

For each selection of points, the probability that this point is not drawn from H_1 is $1 - b/n \leq 1 - c/\sqrt{n}$. Suppose that $n^{\frac{2}{3}}$ points are randomly selected from S. The probability that they all miss H_1 is smaller than

$$\left(1-\frac{c}{\sqrt{n}}\right)^{n^{\frac{2}{3}}} = \left(\left(1-\frac{c}{\sqrt{n}}\right)^{\sqrt{n}}\right)^{n^{\frac{1}{6}}} = e^{-cn^{\frac{1}{6}}}$$

When we randomly choose $n^{\frac{2}{3}}$ points from S, the probability that at least two of them are drawn from H_1 is greater than $1-2e^{-cn^{\frac{1}{6}}}$. Since D dominates D′, we may conclude that if points are randomly chosen from S, then the probability that

at least two points are drawn from the same set of D is at least $\mu(n) = 1 - 2e^{cn^{\frac{1}{6}}}$ *where c is a constant.*

We still have one problem: Is there a partition T_0 such that $n \le N(T_0) \le C_0 n$ where C_0 is a constant? This partition can be found by using the following algorithm.

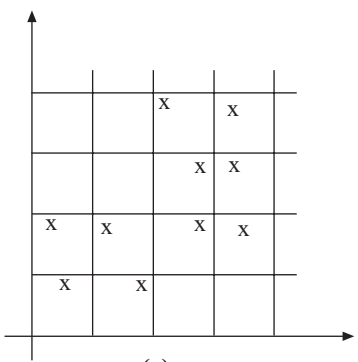
Algorithm 11–2 □ Partition algorithm

Input: A set S of n points.
Output: A partition T_0 where $n \le N(T_0) \le C_0 n$.
Step 1. Find a partition T with a fine enough mesh size such that each square in T contains at most one point of S and no point of S lies on any of the grid lines. Hence, $N(T) = 0$.
Step 2. While $N(T) < n$, double the mesh size of T.

Let us now explain the meaning of Algorithm 11–2. The first step of this algorithm partitions the entire point set into squares such that each square contains at most one point. Of course, in this case, $N(T)$ will be 0 and is not useful for us. We therefore double the size of each square gradually until we hit the first partition such that $N(T_0) \ge n$.

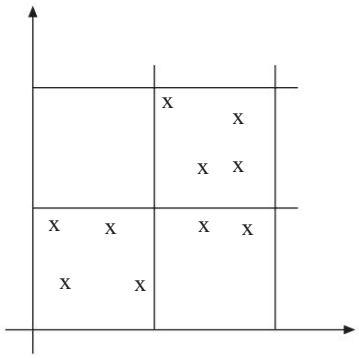
Consider Figure 11–6 where $n = 10$. Figure 11–6(a) shows the initial partition which is the result of Step 1 of Algorithm 11–1. Suppose we double the mesh size. We shall obtain the partitions shown in Figure 11–6(b). Now, the total number of distances computed will be $N(T) = 4 \times 3/2 + 2 \times 1/2 + 4 \times 3/2 = 6 + 1 + 6 = 13 > n = 10$. Thus, this particular partitioning will be a desirable T_0. Note that in this case, $C_0 = 1.3$.

FIGURE 11–6 An example illustrating Algorithm 11–1.



(a)

FIGURE 11–6 (*cont'd*)



(b)

Let δ_0 denote the mesh size of T_0. For any randomly chosen set S_a with $n^{\frac{2}{3}}$ points, if there exist two points of S_a which are within the same square of T_0, then the smallest distance in S_a is less than or equal to $\sqrt{2}\delta_0$ and this inequality holds with probability greater than $\mu(n)$. Therefore, we have proved that there exists a critical T_0 whose size is δ_0 with the following properties:

(1) $n \le N(T_0) \le C_0 n$.

(2) The probability that $\delta \le \sqrt{2}\,\delta_0$ is $1 - 2e^{-cn^{\frac{1}{6}}}$, which is very high, where δ is the smallest distance within a set of $n^{\frac{2}{3}}$ randomly selected points.

We thus explained that in Algorithm 11–1, Step 5 needs $O(n)$ steps. Note that Step 1 is a recursive step. However, since Step 5 is a dominating step, we can conclude that the entire randomized closest pair algorithm is an $O(n)$ algorithm.

11–3 A RANDOMIZED ALGORITHM TO TEST WHETHER A NUMBER IS A PRIME

The prime number problem is to determine whether a given positive number is a prime or not. This is a very difficult problem and no polynomial time algorithm was found to solve this problem until 2004. In this section, we shall introduce a randomized algorithm. This randomized algorithm executes a sequence of m tests. If any one of these tests succeeds, we conclude that this number is a

composite number and we guarantee that this conclusion is absolutely correct. On the other hand, if all of these m tests fail, then we conclude that this number is a prime. However, this time, we are not sure. This conclusion is correct with probability $1 - 2^{-m}$. Thus, if m is sufficiently large, we can draw this conclusion with very high confidence.

Algorithm 11–3 □ A randomized prime number testing algorithm

Input: A positive number N, and a parameter m, where $m \geq \lceil \log_2 \varepsilon \rceil$.

Output: Whether N is a prime number or not, with probability of being correct $1 - \varepsilon = 1 - 2^{-m}$.

Step 1. Randomly choose m numbers $b_1, b_2, \ldots, b_m$, $1 < b_1, b_2, \ldots, b_m < N$.

Step 2. For each b_i, test whether $W(b_i)$ holds where $W(b_i)$ is defined as follows:

(1) $b_i^{N-1} \not\equiv 1 \bmod N$.

or (2) $\exists j$ such that $\dfrac{N-1}{2^j} = k$ is an integer and the greatest common divisor of $b_i^k - 1$ and N is less than N and greater than 1.

If any $W(b_i)$ holds, then return N as a composite number; otherwise, return N as a prime.

Let us illustrate this randomized algorithm by some examples. Consider $N = 12$. Suppose we choose 2, 3 and 7. $2^{12-1} = 2048 \not\equiv 1 \bmod 12$. Therefore, we conclude that 12 is a composite number.

Consider $N = 11$. Suppose that we choose 2, 5 and 7.

$b_1 = 2$: $2^{11-1} = 1024 \equiv 1 \bmod 11$.

Let $j = 1$. $(N - 1)/2^j = (11 - 1)/2^1 = 10/2 = 5$.
This $j = 1$ is the only j where $(N - 1)/2^j$ is an integer.
Yet, the greatest common divisor of $2^5 - 1 = 31$ and 11 is 1.

$W(2)$ does not hold.

$b_2 = 5$: $5^{11-1} = 5^{10} = 9765625 \equiv 1 \bmod 11$.

As discussed above, $j = 1$ is the only j which makes $(N - 1)/2^j = k = 5$, an integer.

$b_2^k = 5^5 = 3125$.

The greatest common divisor of $5^5 - 1 = 3124$ and 11 is 11.
Again, $W(5)$ does not hold.

$b_3 = 7$: $7^{11-1} = 282475249 \equiv 1 \bmod 11$.

Again, let $j = 1$.

$b_3^k = 7^5 = 16807$.

The greatest common divisor of $7^5 - 1 = 16806$ and 11 is 1.
$W(7)$ does not hold.

We may conclude that 11 is a prime number with the probability of correctness being $1 - 2^{-3} = 1 - 1/8 = 7/8$.

If we also choose 3, it can be proved that $W(3)$ also fails. We have now $m = 4$ and the probability of correctness will be increased to $1 - 2^{-4} = 15/16$.

For any N, if m is 10, the probability of correctness will be $1 - 2^{-10}$ which is almost 1.

The correctness of this randomized prime number testing algorithm is based upon the following theorem.

THEOREM 11–1

(1) If $W(b)$ holds for any $1 < b < N$, N is a composite number.

(2) If N is composite, $\dfrac{N-1}{2} \le |\{b \mid 1 \le b < N,\ W(b) \text{ holds}\}|$.

From the above theorem, we know that if N is composite, then at least half of the b_i's ($b_i < N$) have the property that $W(b)$ holds. If it is found that there exists one b_i such that $W(b_i)$ holds, then N must be composite. If none of $W(b_i)$ holds, for m b_i's, then according to Theorem 11–1, the probability that N is composite is $(\frac{1}{2})m$. Therefore, the probability that N is a prime is greater than $1 - 2^{-m}$.

11–4 A RANDOMIZED ALGORITHM FOR PATTERN MATCHING

In this section, we shall introduce a randomized pattern matching algorithm. This algorithm can be used to solve the following problem: Given a pattern string X

of length n and a text string Y of length m, $m \geq n$, find the first occurrence of X as a consecutive substring of Y. Without loss of generality, we assume that both X and Y are binary strings.

For instance, let $X = 01001$ and $Y = 1010100111$. Then we can see that X does occur in Y, as underlined.

Let the patterns X and Y be respectively

$$X = x_1x_2 \ldots x_n, x_i \in \{0, 1\}$$
$$\text{and} \quad Y = y_1y_2 \ldots y_m, y_i \in \{0, 1\}.$$

Let $Y(1) = y_1y_2 \ldots y_n$, $Y(2) = y_2y_3 \ldots y_{n+1}$ and so on. In general, let $Y(i) = y_iy_{i+1} \ldots y_{i+n-1}$. Then, a match occurs if $X = Y(i)$ for some i.

There is another way of checking X and $Y(i)$. Let the binary value of a string s be denoted as $B(s)$. Then

$$B(X) = x_1 2^{n-1} + x_2 2^{n-2} + \cdots + x_n, \text{ and}$$

$B(Y_i) = y_i 2^{n-1} + y_{i+1} 2^{n-2} + \cdots + y_{i+n-1}$, $1 \leq i \leq m - n + 1$. Thus, we may simply check whether $B(X)$ is equal to $B(Y_i)$.

The trouble is as n is large, the computation of $B(X)$ and $B(Y_i)$ may be difficult. Therefore, we shall present a randomized algorithm to solve this problem. Since this is a randomized algorithm, it is possible that we make mistakes.

Our approach is to compute the residue of $B(X)$ and a prime number P. Let the residue of two integers u and v be denoted as $(u)_v$. Obviously, if $(B(X))_p \neq (B(Y_i))_p$, then $X \neq Y_i$, but not vice versa. For example, consider $X = 10110$ and $Y = 10011$.

$$B(X) = 2^4 + 2^2 + 2^1 = 16 + 4 + 2 = 22$$
$$B(Y) = 2^4 + 2^1 + 2^0 = 16 + 2 + 1 = 19.$$

Let $P = 3$. Then $(B(X))_p = (22)_3 = 1$ and $(B(Y))_p = (19)_3 = 1$. Although $(B(X))_p = (B(Y))_p$, we still should not conclude that $X = Y(i)$.

Our randomized algorithm consists of the following ideas:

(1) Select k prime numbers $p_1, p_2, \ldots, p_k$.
(2) If for any j, $(B(X))p_j \neq (B(Y_i))p_j$, conclude that $X \neq Y(i)$.

If $(B(X))p_j = (B(Y))p_j$ for $j = 1, 2, \ldots, k$, then conclude that $X = Y(i)$.

We shall show that if our conclusion is $X \neq Y(i)$, we are absolutely certain. On the other hand, if we conclude that $X = Y(i)$, there is a chance that we are making a mistake.

The advantage of our approach is that $(B(X))_p$ and $(B(Y_i))_p$ can be computed easily. We really don't have to compute $B(X)$ and $B(Y_i)$. Recall that

$$B(X) = x_1 2^{n-1} + x_2 2^{n-2} + \cdots + x_n.$$

It can be easily seen that

$$(B(X_i))_p = ((((x_1 \cdot 2)_p + x_2)_p \cdot 2\ )_p + x_3)_p \cdot 2 + \cdots$$

Similarly,

$$(B(Y(i)))_p = ((((y_i \cdot 2)_p + y_{i+1})_p \ldots 2)_p + y_{i+2})_p \cdot 2 + \cdots$$

Through this kind of mechanism, we never have to worry about whether $B(X)$ or $B(Y(i))$ is a very large number. We now present an example to illustrate this point.

Let $X = 10110$

$x_1 = 1$
$x_2 = 0$
$x_3 = 1$
$x_4 = 1$
$x_5 = 0$

Let p be 3

$$\begin{aligned}
(x_1 \cdot 2)_p &= (1 \cdot 2)_3 = (2)_3 = 2. \\
(2 + x_2)_p &= (2 + 0)_3 = (2)_3 = 2. \\
(2 \cdot 2)_p &= (4)_3 = 1 \\
(1 + x_3)_p &= (1 + 1)_3 = (2)_3 = 2. \\
(2 \cdot 2)_p &= (4)_3 = 1 \\
(1 + x_4)_p &= (1 + 1)_3 = (2)_3 = 2. \\
(2 \cdot 2)_p &= (4)_3 = 1 \\
(1 + x_5)_p &= (1 + 0)_3 = (1)_3 = 1.
\end{aligned}$$

That is $(B(X))_p = (B(X))_3 = 1$.

One can see that through the computation, all numbers involved will be relatively small.

Next, we present the randomized algorithm for pattern matching.

Algorithm 11–4 □ A randomized algorithm for pattern matching

Input: A pattern $X = x_1x_2 \ldots x_n$, a text $Y = y_1y_2 \ldots y_m$ and a parameter k.

Output: (1) No, there is no consecutive substring in Y which matches with X.
(2) Yes, $Y(i) = y_iy_{i+1} \ldots y_{i+n-1}$ matches with X.
If the answer is "No", there is no mistake.
If the answer is "Yes", there is some probability that a mistake is made.

Step 1. Randomly choose k prime numbers $p_1, p_2, \ldots, p_k$ from $\{1, 2, \ldots, nt^2\}$, where $t = m + n - 1$.

Step 2. $i = 1$.

Step 3. $j = 1$.

Step 4. If $B(X)p_j \neq (B(Y_i))p_j$, then go to Step 5.
If $j = k$, return $Y(i)$ as the answer
$j = j + 1$
Go to Step 3.

Step 5. If $i = t$, return "No, there is no consecutive substring in Y which matches with X."
$i = i + 1$.
Go to Step 3.

We now perform a theoretical analysis of this randomized algorithm. Essentially, we shall prove that if k is sufficiently large enough, the probability that we draw a wrong conclusion will be very small.

Perhaps we should ask, when a wrong conclusion is drawn, what is happening? When a wrong conclusion is drawn, we have the following conditions:

(1) $B(X) \neq (B(Y_i)$
(2) $(B(X))p_j = (B(Y_i))p_j$ for $j = 1, 2, \ldots, k$

Because of the above conditions, we may state that

$$B(X) = a_j p_j + c_j. \tag{11–1}$$

and $$B(Y_i) = b_j p_j + c_j. \tag{11–2}$$

Thus, $$B(X) - B(Y_i) = (a_j - b_j)p_j. \tag{11–3}$$

Equation (11–3) indicates that when a wrong conclusion is drawn, then $B(X) - B(Y_i) \neq 0$ and p_j divides $B(X) - B(Y_i)$ for all p_j's.

A critical question that we now ask is: How many prime numbers are there which divide $|B(X) - B(Y(i))|$?

To answer this question, remember that we are given a pattern X with n bits and a text with m bits. Let Q denote the product

$$\prod_{i}^{k=m-n+1} |B(X) - B(Y(i))|$$

where p_j divides $|B(X) - B(Y(i))|$.

Note that p_j also divides Q.

The value of Q is less than $2^{n(m-n+1)}$. With this value, we shall be able to derive the probability that we make a mistake. We need the following statement:

If $u \geq 29$ and $a \geq 2^u$, then a has fewer than $\pi(u)$ different prime number divisors where $\pi(u)$ denotes the number of prime numbers smaller than u.

We shall not go into the details of how the above statement was obtained. To use that statement, let us note that Q is less than $2^{n(m-n+1)} - 2^{nt}$. Therefore, Q has fewer than $\pi(nt)$ different prime number divisors assuming that $nt \geq 29$.

In our algorithm, p_j is a prime number selected from $\{1, 2, \ldots, nt^2\}$. Therefore, the probability that p_j divides Q is less than $\pi(nt)/\pi(nt^2)$. This means that the probability that p_j would give us a wrong answer is less than $\pi(nt)/\pi(nt^2)$. Since we choose k prime numbers, the probability that we shall make a wrong conclusion based upon these numbers is less than $(\pi(nt)/\pi(nt^2))^k$.

Our discussion is summarized as follows:

If k different prime numbers are selected for our randomized pattern matching algorithm, the probability that a mistake is made is less than $\left(\frac{\pi(nt)}{\pi(nt^2)}\right)^k$ provided $nt \geq 29$.

Our next question is: How do we estimate $\pi(x)$? We have the following estimation formula:

For all $u \geq 17$,

$$\frac{u}{\ln u} \leq \pi(u) \leq \pi(u) \leq 1.25506 \frac{u}{\ln u}.$$

We can now see that in general, the probability of drawing a wrong conclusion is quite small even if k is not too large.

Let us assume that $nt \geq 29$. Then

$$\begin{aligned}
\frac{\pi(nt)}{\pi(nt^2)} &\leq 1.25506 \frac{nt}{\ln nt} \cdot \frac{\ln (nt^2)}{nt^2} \\
&= \frac{1.25506}{t} \left(\frac{\ln (nt^2)}{\ln (nt)} \right) \\
&= \frac{1.25506}{t} \left(\frac{\ln (nt) + \ln (t)}{\ln (nt)} \right) \\
&= \frac{1.25506}{t} \left(1 + \frac{\ln (t)}{\ln (nt)} \right).
\end{aligned}$$

For instance, let $n = 10$ and $m = 100$. Then

$t = 91$.

$$\begin{aligned}
\frac{\pi(nt)}{\pi(nt^2)} &\leq \frac{1.25506}{t} \left(1 + \frac{\ln t}{\ln (nt)} \right) \\
&= \frac{1.25506}{91} \left(1 + \frac{\ln (91)}{\ln (910)} \right) \\
&= 0.013792 \cdot \left(1 + \frac{4.5109}{6.8134} \right) \\
&= 0.013792 \cdot (1 + 0.6621) \\
&= 0.013792 \cdot 1.6621 \\
&= 0.0229.
\end{aligned}$$

Suppose that $k = 4$. Then the probability that a wrong conclusion is drawn is

$(0.0229)^4 \approx 2.75 * 10^{-7}$

which is very very small.

11–5 A RANDOMIZED ALGORITHM FOR INTERACTIVE PROOFS

Consider the following problem. It was during the cold war era and a British MI5 (Military Intelligence Unit 5, consult *Spy Catcher*, by P. Wright) agent wanted to talk to a spy whom Her Majesty's Government had planted in the KGB for years.

Let us call the MI5 agent and the spy planted in the KGB B and A respectively. The problem now is: Can B know that A is the real A, not a KGB officer inpersonating A? Well, a trivial method is to ask, say, the maiden name of A's mother. The trouble is: If A answers correctly, some KGB officer may inpersonate A easily the next time. Therefore, B may ask A to do something which is rather difficult, so difficult that ordinary people can hardly do. He may, for example, ask A to determine the satisfiability of a Boolean formula. We assume that A is a brilliant person and knows how to solve this NP-complete problem pretty well. So, every time he receives a Boolean formula from B, he solves the problem. If it is satisfiable, he sends an assignment to B and simply "NO" if it is not satisfiable. B is not that smart. Yet, he knows the definition of satisfiability. He can at least check whether the assignment satisfies the formula or not. If A solves the satisfiability problem correctly each time, B will be happy and convinced that A is the real A.

Still, an interceptor may gradually figure out that B is always sending a Boolean formula to A and if this formula is satisfiable, A will send a satisfying assignment back to B. This interceptor begins to study methods of mechanical theorem proving and sooner or later he can inpersonate A.

We may say that A and B have both revealed too much. It will be nice that B sends some data to A and after A performs some calculation on this set of data, sends some resulting data back to B. B then verifies A's results by performing some computation on it. If he is satisfied, then he can be sure that A is the right person. In this way, an interceptor can still intercept some data. Yet it will be very difficult for him to know what is going on.

In this section, we shall show that B can ask A to solve a quadratic non-residue problem and we will see, with great interest, that the data can be sent back and forth without revealing much information. Of course, this is a randomized algorithm and there is a certain degree of error involved.

Let x and y be two positive integers, $0 < y < x$, such that $gcd(x, y) = 1$. We define $Z_x = \{z \mid 0 < z < x, gcd(z, x) = 1\}$. We say that y is a quadratic residue mod x if $y = z^2$ mod x for some $z \in Z_x$ and y is a quadratic non-residue mod x if otherwise. We further define two sets:

$QR = \{(x, y) \mid y \text{ is a quadratic residue mod } x\}$

$QNR = \{(x, y) \mid y \text{ is a quadratic non-residue mod } x\}$

For example, let $y = 4$ and $x = 13$. We can see that there exists a z, namely 2, such that $z \in Z_x$ and $y = z^2 \bmod x$. (This can be easily verified: $4 = 2^2 \bmod 13$) Thus, 4 is a quadratic residue mod 13. We can also prove that 8 is a quadratic non-residue mod x.

Now, how can B and A communicate in such a way that A solves the quadratic non-residue problem for B without revealing much information?

They may proceed as follows:

(1) Both A and B know the value of x and keep this confidential. B knows the value of y.
(2) Action of B:
  (a) Flip coins to obtain m bits: $b_1, b_2, \ldots, b_m$ where m is the length of the binary representation of x.
  (b) Find $z_1, z_2, \ldots, z_m$, $0 < z_i < x$, such that $gcd(z_i, x) = 1$, for all i.
  (c) Compute $w_1, w_2, \ldots, w_m$ from $b_1, b_2, \ldots, b_m, z_1, z_2, \ldots, z_m$, as follows:

$$w_i = z_i^2 \bmod x \text{ if } b_i = 0.$$
$$w_i = (z_i^2 \cdot y) \bmod x \text{ if } b_i = 1.$$

  (d) Send $w_1, w_2, \ldots, w_m$ to A.
(3) Action of A:
  (a) Receive $w_1, w_2, \ldots, w_m$ from B.
  (b) Set $c_1, c_2, \ldots, c_m$ as follows:

$$c_i = 0 \text{ if } (x, w_i) \in QR.$$
$$c_i = 1 \text{ if } (x, w_i) \in QNR.$$

  (c) Send $c_1, c_2, \ldots, c_m$ to B.
(4) Action of B:
  (a) Receive $c_1, c_2, \ldots, c_m$ from A.
  (b) Return "YES, y is a quadratic non-residue mod x" if $b_i = c_i$ for all i. And return "NO, y is a quadratic residue mod x" if otherwise.

It can be proved that if y is a quadratic non-residue mod x, then B accepts it so with probability $1 - 2^{-|x|}$. If y is a quadratic residue mod x, then B accepts it as a quadratic non-residue with probability $2^{-|x|}$.

B can certainly reduce the error by repeating this process. Note that A must be a smart person in the sense that he can solve the quadratic non-residue problem.

A is thus a "prover". B is only a "verifier".

Let us give some examples.

Suppose that $(x, y) = (13, 8)$, $|x| = 4$.

Action of *B*:

(a) Suppose that $b_1, b_2, b_3, b_4 = 1, 0, 1, 0$.

(b) Suppose that $z_1, z_2, z_3, z_4 = 9, 4, 7, 10$ which are all pairwise prime with respect to $x = 13$.

(c) w_1, w_2, w_3, w_4 are calculated as follows:

$b_1 = 1$, $w_1 = (z_1^2 \cdot y) \bmod x = (9^2 \cdot 8) \bmod 13 = 648 \bmod 13 = 11$

$b_2 = 0$, $w_2 = (z_2^2) \bmod x = 4^2 \bmod 13 = 16 \bmod 13 = 3$

$b_3 = 1$, $w_3 = (z_3^2 \cdot y) \bmod x = (7^2 \cdot 8) \bmod 13 = 392 \bmod 13 = 2$

$b_4 = 0$, $w_4 = (z_4^2) \bmod x = 10^2 \bmod 13 = 100 \bmod 13 = 9$

Thus, $(w_1, w_2, w_3, w_4) = (11, 3, 2, 9)$.

(d) Send $(11, 3, 2, 9)$ to A.

Action of *A*:

(a) Receive $(11, 3, 2, 9)$ from B.

(b) $(13, 11) \in QNR$, $c_1 = 1$.

$(13, 3) \in QR$, $c_2 = 0$.

$(13, 2) \in QNR$, $c_3 = 1$.

$(13, 9) \in QR$, $c_4 = 0$.

(c) Send $(c_1, c_2, c_3, c_4) = (1, 0, 1, 0)$ to B.

Action of *B*:

Since $b_i = c_i$ for all i, B accepts that 8 is a quadratic non-residue of 13, which is true.

Let us consider another example.

$(x, y) = (13, 4)$, $|x| = 4$.

Action of *B*:

(a) Suppose that again, $b_1, b_2, b_3, b_4 = 1, 0, 1, 0$.
(b) Suppose that $z_1, z_2, z_3, z_4 = 9, 4, 7, 10$.
(c) It can be easily proved that $(w_1, w_2, w_3, w_4) = (12, 3, 1, 9)$.
(d) Send $(12, 3, 1, 9)$ to A.

Action of *A*:

(a) Receive $(12, 3, 1, 9)$ from B.
(b) $(13, 12) \in QR$, $c_1 = 0$.
$(13, 3) \in QR$, $c_2 = 0$.
$(13, 1) \in QR$, $c_3 = 0$.
$(13, 9) \in QR$, $c_4 = 0$.
(c) Send $(0, 0, 0, 0)$ to B.

Action of *B*:

Since it is not true that $b_i = c_i$ for all i, B accepts the fact that 4 is a quadratic residue mod 13.

The theoretical basis of this randomized algorithm is number theory and is beyond the scope of this book.

11–6 A RANDOMIZED LINEAR TIME ALGORITHM FOR MINIMUM SPANNING TREES

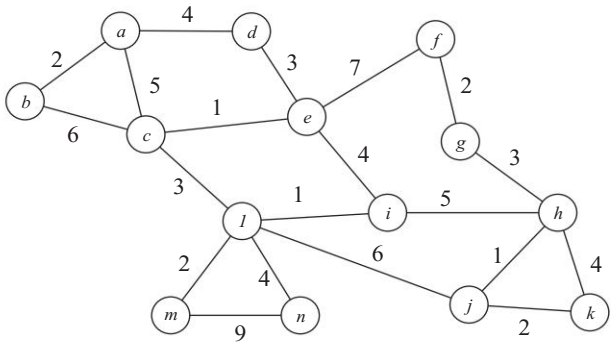
In Chapter 3, we presented two minimum spanning tree algorithms based on the greedy method. One of these tree algorithms is the Kruskal's algorithm whose time complexity is $O(n^2 \log n)$ and the other algorithm is the Prim's algorithm. The time complexity of a sophisticated version of this algorithm is $O(n + m\alpha(m, n))$ where $n(m)$ is the number of nodes (edges) of a graph and $\alpha(m, n)$ is the inverse Ackermann's function. In this section, we shall present a randomized minimum spanning tree algorithm whose expected time complexity is $O(n + m)$.

This algorithm is based upon a so-called Boruvka step, proposed by Boruvka in 1926. The following lemma illustrates the idea behind the Boruvka step:

LEMMA 1: Let V_1 and V_2 be non-empty vertex sets where $V_1 \bigcup V_2 = V$ and $V_1 \bigcap V_2 = \phi$ and let the edge (v, u) be the minimum-weighted edge with one end point in V_1 and another end point in V_2. Then, (v, u) must be contained in the minimum spanning tree in G.

Lemma 1 can be depicted in another form as follows: In a graph G, for any node u, among all edges incident on u, if edge (u, v) has the smallest weight, then (u, v) must be an edge in the minimum spanning tree of G. It is easy to prove Lemma 1. Consider Figure 11–7. For node c, among all edges incident on c, edge (c, e) has the smallest weight. Therefore, edge (c, e) must be included in the minimum spanning tree of G. Similarly, it can be easily proved that edge (f, g) must also be included.

FIGURE 11–7 A graph.



Now, let us select all edges of the graph in Figure 11–7 which must be included in the minimum spanning tree based upon Lemma 1. The resulting connected components are shown in Figure 11–8. In the graph in Figure 11–8, all of the dotted lines are connected components edges.

Let us contract all nodes in each connected component to one node. There are thus five nodes now, shown in Figure 11–9. After we eliminate multiple edges and loops, the result is shown in Figure 11–10.

Since the resulting graph consists of more than one node, we may apply Lemma 1 again. The result is shown in Figure 11–11 and the selected edges are (a, d),(c, l) and (g, h).

After contracting the nodes in each connected components, we have now two nodes, shown in Figure 11–12.

FIGURE 11–8 The selection of edges in the Boruvka step.

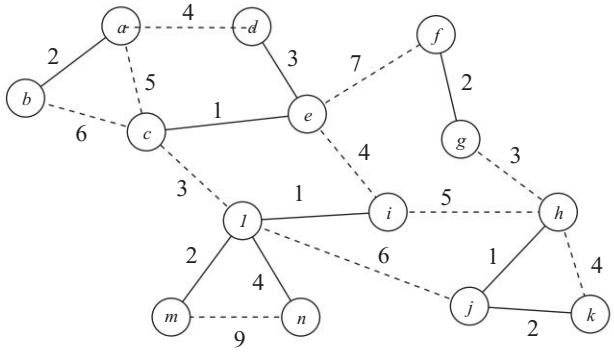


FIGURE 11–9 The construction of nodes.

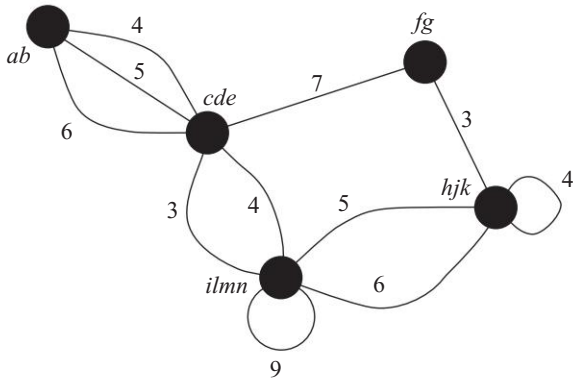


FIGURE 11–10 The result of applying the first Boruvka step.

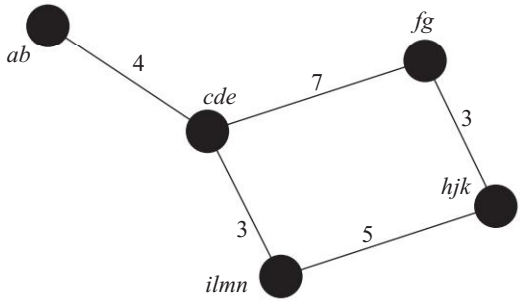


FIGURE 11–11 The second selection of edges.

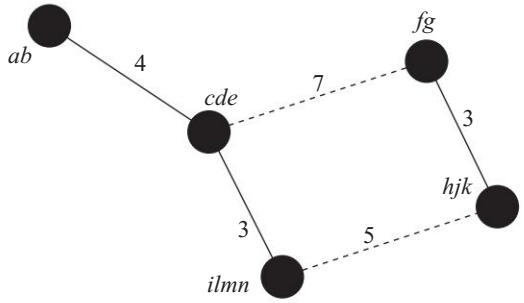
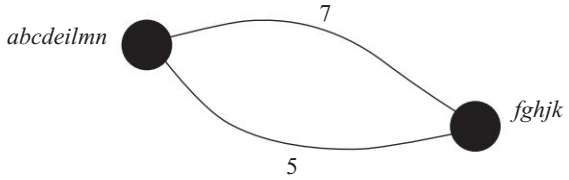
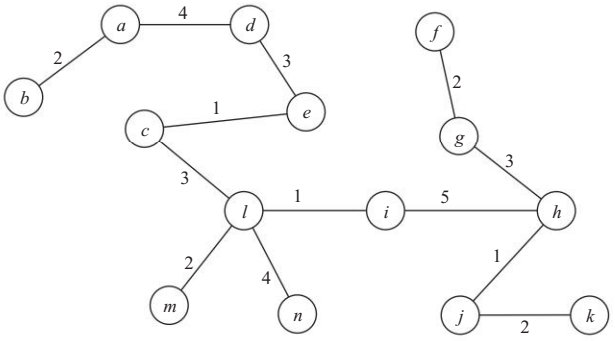


FIGURE 11–12 The second construction of edges.



Again, after we eliminate multiple edges and select edge (i, h), we can now contract all nodes into one node. The process is completed. All the edges selected constitute the minimum spanning tree which is shown in Figure 11–13.

FIGURE 11–13 The minimum spanning tree obtained by Boruvka step.



Boruvka's algorithm for finding the minimum spanning tree applies the Boruvka step recursively until the resulting graph is reduced to a single vertex. Let the input of the Boruvka step be a graph $G(V, E)$ and the output be a graph $G'(V', E')$. The Boruvka step is described next.

The Boruvka Step

1. For each node u, find the edge (u, v) with the smallest weight connected to it. Find all the connected components determined by the marked edges.
2. Contract each connected component determined by the marked edges into one single vertex. Let the resulting graph be $G'(V', E')$. Eliminate multiple edges and loops.

The time complexity for one Boruvka step is $O(n + m)$ where $|V| = n$ and $|E| = m$. Since G is connected, $m > n$. Hence, $O(n + m) = O(m)$. Since each connected component determined by the marked edges contains at least two vertices, after each Boruvka step is executed, the number of remaining edges is smaller than one half of the original one. Hence, the total number of Boruvka steps executed is $O(\log n)$. The total time complexity of the Boruvka algorithm is $O(m \log n)$.

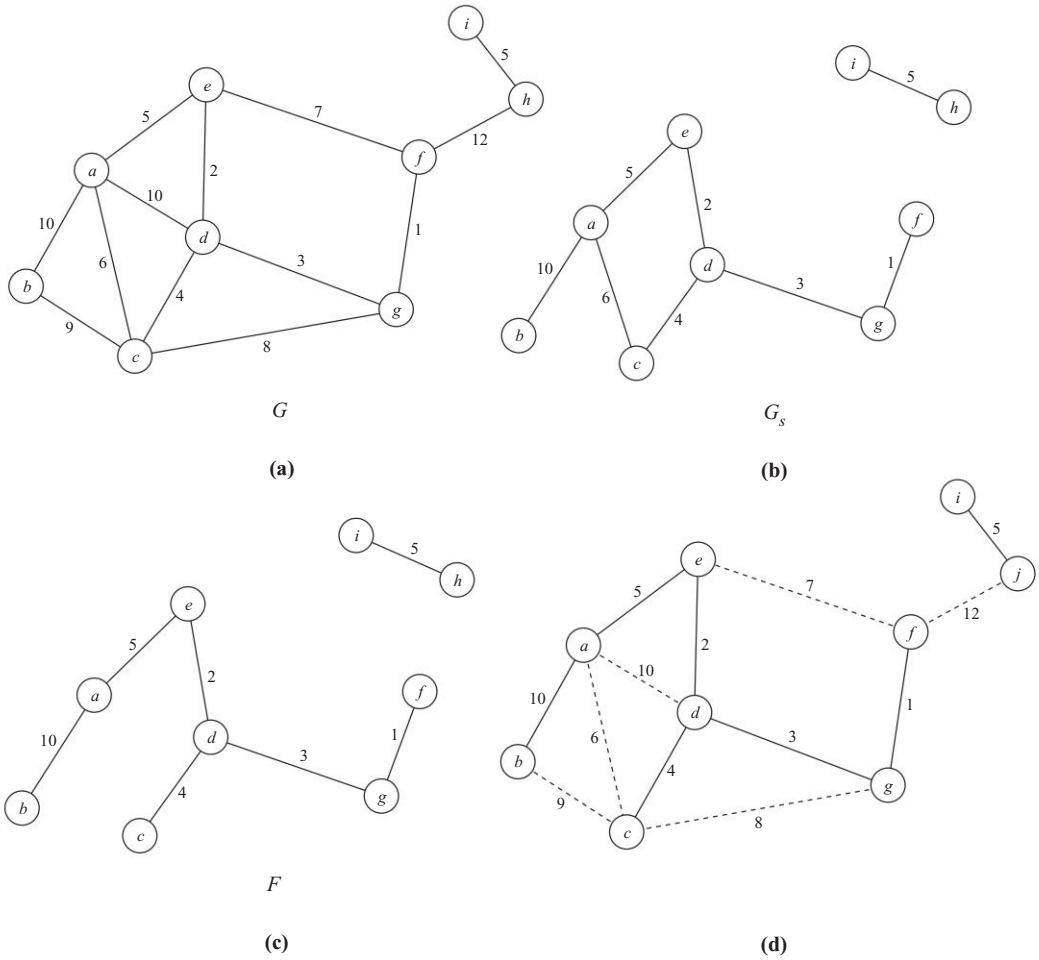
To use the Boruvka step efficiently, we must use a new concept. Consider Figure 11–14. In Figure 11–14(b), graph G_s is a subgraph of graph G in Figure 11–14(a). A minimum spanning forest F of G_s is shown in Figure 11–14(c). In Figure 11–14(d), the minimum spanning forest F is imbedded in the original graph G. All the edges which are not edges of F are now dotted edges. Let us consider edge (e, f). The weight of (e, f) is 7. Yet, there is a path between e and f in the forest F, namely $(e, d) \rightarrow (d, g) \rightarrow (g, f)$. The weight of (e, f) is larger than the maximum weight of edges in this path. According to a lemma which will be presented below, the edge (e, f) cannot be an edge of a minimum spanning tree of G. Before presenting the lemma, let us define a term called F-heavy.

Let $w(x, y)$ denote the weight of edge (x, y) in G. Let G_s denote a subgraph of G. Let F denote a minimum spanning forest of G_s. Let $w_F(x, y)$ denote the maximum weight of an edge in the path connecting x and y in F. If x and y are not connected in F, let $w_F(x, y) = \infty$. We say that edge (x, y) is F-heavy (F-light) with respect to F if $w(x, y) > w_F(x, y)(w(x, y) \leq w_F(x, y))$.

Consider Figure 11–14(d). We can see that edges (e, f), (a, d) and (c, g) are all F-heavy with respect to F. Having defined this new concept, we can have the following lemma which is quite important for us to use the Boruvka step.

LEMMA 2: Let G_s be a subgraph of a graph $G(V, E)$. Let F be a minimum spanning forest of G_s. The F-heavy edges in G with respect to F cannot be minimum spanning tree edges of G.

FIGURE 11–14 *F*-heavy edges.



We shall not prove the above lemma. With this lemma, we know that edges (e, f), (a, d) and (c, g) cannot be minimum spanning tree edges.

We need another lemma for fully utilizing the Boruvka step mechanism. This is lemma 3.

LEMMA 3: Let H be a subgraph obtained from G by including each edge independently with probability p, and let F be the minimum spanning forest of H. The expected number of F-light edges in G is at most n/p where n is the number of vertices of G.

The randomized minimum spanning tree algorithm is described next.

Algorithm 11–5 □ A randomized minimum spanning tree algorithm

Input: A weighted connected graph G.

Output: A minimum spanning tree of G.

Step 1. Apply the Boruvka step three times. Let the resulting graph be $G_1(V_1, E_1)$. If G_1 contains one node, return the set of edges marked in Step 1 and exit.

Step 2. Obtain a subgraph H of G_1 by selecting each edge independently with probability 1/2. Apply the algorithm recursively to H to obtain a minimum spanning forest F of H. Get a graph $G_2(V_2, E_2)$ by deleting all F-heavy edges in G_1 with respect to F.

Step 3. Apply the algorithm recursively to G_2.

Let us analyze the time complexity of Algorithm 11–5 next.

Let $T(|V|, |E|)$ denote the expected running time of the algorithm for graph $G(V, E)$.

Each execution of Step 1 takes $O(|V| + |E|)$ time. After Step 1 is executed, we have $|V_1| \le |V|/8$ and $|E_1| < |E|$. For Step 2, the time needed to compute H is $O(|V_1| + |E_1|) = O(|V| + |E|)$. The time needed for computing F is $T(|V_1|, |E_1|/2) = T(|V|/8, |E|/2)$. The time needed for deleting all F-heavy edges is $O(|V_1| + |E_1|) = O(|V| + |E|)$. Using Lemma 3, we have the expected value for $|E_2|$ is at most $2|V_2| \le |V|/4$. Hence, the expected time needed to execute Step 3 is $T(|V_2|, |E_2|) = T(|V|/8, |V|/4)$. Let $|V| = n$ and $|E| = m$. Then we have the following recurrence relation:

$$T(n, m) \le T(n/8, m/2) + T(n/8, n/4) + c(n + m),$$

for some constant c. It can be proved that

$$T(n, m) \le 2c \cdot (n + m).$$

We encourage the reader to check this solution by substituting (2) into (1). Hence, the expected running time of the algorithm is $O(n + m)$.

11–7 NOTES AND REFERENCES

A survey of randomized algorithms can be found in Maffioli (1986). Gill (1987) and Kurtz (1987) also surveyed randomized algorithms. It is appropriate to point out that the randomized algorithm means different things to different people. It sometimes denotes an algorithm which is good in average case analysis. That is, this algorithm behaves differently for different sets of data. We insist, in this book, that a randomized algorithm is an algorithm which uses flipping of coins in the process. In other words, for the same input data, because of the randomized process, the program may behave very much differently.

That the closest pair problem can be solved by a randomized algorithm was proposed by Rabin (1976). For recent results, see Clarkson (1988). For testing primes by randomized algorithms, see Solovay and Strassen (1977) and Rabin (1980). That the prime number problem is a polynomial problem was proved lately by Agrawal, Kayal and Saxena (2004). The randomized algorithm for pattern matching appeared in Karp and Rabin (1987). The randomized algorithm for interactive proofs can be found in Goldwasser, Micali and Rackoff (1988). Galil, Haber and Yung (1989) suggested a further improvement of their method. The randomized minimum spanning tree algorithm can be found in Karger, Klein and Tarjan (1995). The Boruvka step can be found in Boruvka (1926) and the method to delete F-heavy edges can be found in Dixon, Rauch and Tarjan (1992).

Randomized algorithms are also discussed extensively in Brassard and Bratley (1988).

11–8 FURTHER READING MATERIALS

Randomized algorithms can be classified into two kinds: sequential and parallel. Although this textbook is restricted to sequential algorithms, we are still going to recommend some randomized parallel algorithms.

For randomized sequential algorithms, we recommend Agarwal and Sharir (1996); Anderson and Woll (1997); Chazelle, Edelsbrunner, Guibas, Sharir and Snoeyink (1993); Cheriyan and Harerup (1995); Clarkson (1987); Clarkson (1988); d'Amore and Liberatore (1994); Dyer and Frieze (1989); Goldwasser and Micali (1984); Kannan, Mount and Tayur (1995); Karger and Stein (1996); Karp (1986); Karp, Motwani and Raghavan (1988); Matousek (1991); Matousek (1995); Megiddo and Zemel (1986); Mulmuley, Vazirani and Vazirani (1987); Raghavan and Thompson (1987); Teia (1993); Ting and Yao (1994); Wenger (1997); Wu and Tang (1992); Yao (1991); and Zemel (1987).

For randomized parallel algorithms, we recommend Alon, Babai and Itai (1986); Anderson (1987); Luby (1986) and Spirakis (1988).

A large batch of newly-published papers include Aiello, Rajagopalan and Venkatesan (1998); Albers (2002); Alberts and Henzinger (1995); Arora and Brinkman (2002); Bartal and Grove (2000); Chen and Hwang (2003); Deng and Mahajan (1991); Epstein, Noga, Seiden, Sgall and Woeginger (1999); Froda (2000); Har-Peled (2000); Kleffe and Borodovsky (1992); Klein and Subramanian (1997); Leonardi, Spaccamela, Presciutti and Ros (2001); Meacham (1981) and Sgall (1996).

Exercises

11.1 Write a program to implement the randomized algorithm for solving the closest pair problem. Test your algorithms.

11.2 Use the randomized prime number testing algorithm to determine whether the following numbers are prime or not:

13, 15, 17.

11.3 Use the randomized pattern matching algorithm on the following two strings:

$X = 0101$

$Y = 0010111$

11.4 Use the algorithm introduced in Section 11–5 to determine whether 5 is a quadratic residue of 13 or not. Show an example in which you would draw a wrong conclusion.

11.5 Read Section 8–5 and 8–6 of Brassard and Bratley 1988.

chapter

12

On-Line Algorithms

In all the previous chapters, algorithms were designed under the assumption that the whole picture of the data is available for us before the algorithm is executed. That is, problems are solved together with complete information of the data. However, this is not necessarily true in reality. Consider the disk scheduling problem. The requests of disk servers are totally unknown to the algorithm. They arrive one by one. The paging problem which occurs in operating systems design is also an on-line problem. We simply cannot know which pages will be accessed before executing our programs. If the data arrive on-line, the algorithm still must take action to take care of each datum which has just arrived. Since no complete information is available, the action may seem to be correct at this time, but may turn out to be wrong later. Thus, on-line algorithms are all approximation algorithms in the sense that they can never guarantee to produce optimal solutions. Let us consider the on-line minimum spanning tree problem. In this case, we first must drop the word "minimum" because the spanning tree will not be minimum. Thus, we call the problem an on-line small spanning tree problem. An on-line algorithm to handle the situation may work as follows: Each time as a data item arrives, connect it to its nearest neighbor. Suppose that we have six points as shown in Figure 12–1. Let the data arrive in the order as specified. Thus, one on-line algorithm may produce a spanning tree as shown in Figure 12–2. It is obvious that this tree is not optimal. An optimal spanning tree, constructed with full knowledge of the data, is shown in Figure 12–3.

Since any on-line algorithm must be an approximation algorithm, its performance is naturally measured by comparing its result with that obtained by executing an optimal off-line algorithm. Let $C_{onl}(C_{off})$ denote the cost of executing an on-line (optimal off-line) algorithm on the same data set. If $C_{onl} \leq c \cdot C_{off} + b$ where b is a constant, then we say that the performance ratio of this on-line algorithm is c and this algorithm is c-competitive. If c cannot be smaller, we say this on-line algorithm is optimal. On-line

FIGURE 12–1 A set of data to illustrate an on-line spanning tree algorithm.

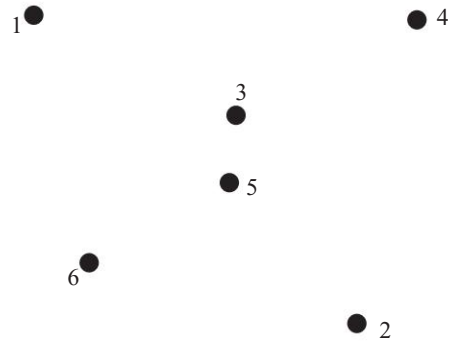


FIGURE 12–2 A spanning tree produced by an on-line small spanning tree algorithm.

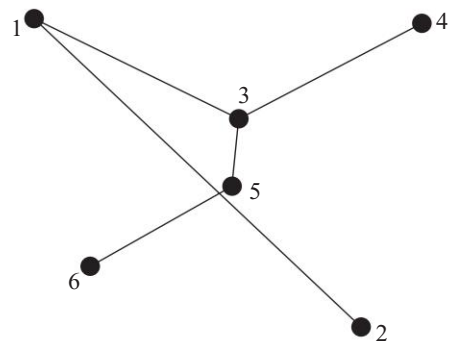
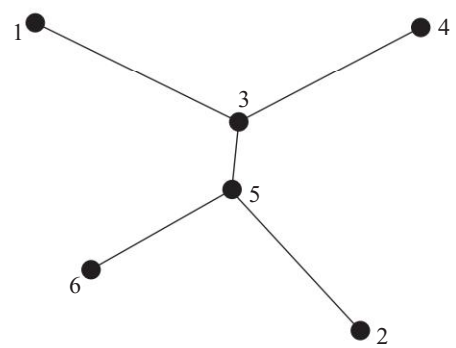


FIGURE 12–3 The minimum spanning tree for the data shown in Figure 12–1.



algorithms are by no means easy to design because the design must be coupled with an analysis. In this chapter, we shall introduce several on-line algorithms and the analyses of them.

12–1 THE ON-LINE EUCLIDEAN SPANNING TREE PROBLEM SOLVED BY THE GREEDY METHOD

In the Euclidean minimum spanning tree problem, we are given a set of points in the plane and our job is to construct a minimum spanning tree out of these points. For the on-line version of this problem, the points are revealed one by one and whenever a point is revealed, some action must be taken to connect this point to the already constructed tree. Besides, this action cannot be reversed. Obviously, trees constructed by an on-line algorithm must be approximate ones. The greedy algorithm for solving this Euclidean spanning tree can be described as follows: Let $v_1, v_2, \ldots, v_{k-1}$ be revealed and T be the spanning tree presently constructed. Add the shortest edge between v_k and $v_1, v_2, \ldots, v_{k-1}$ to T. Consider the set of points shown in Figure 12–4. The spanning tree constructed by the greedy method is shown in Figure 12–5.

FIGURE 12–4 A set of five points.

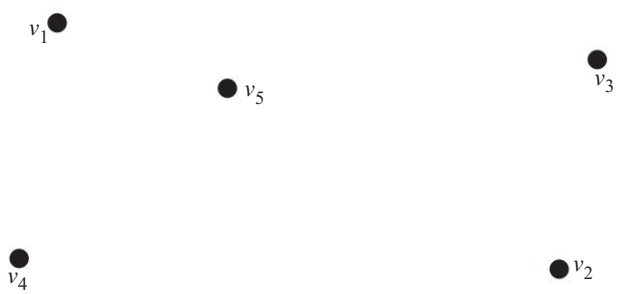
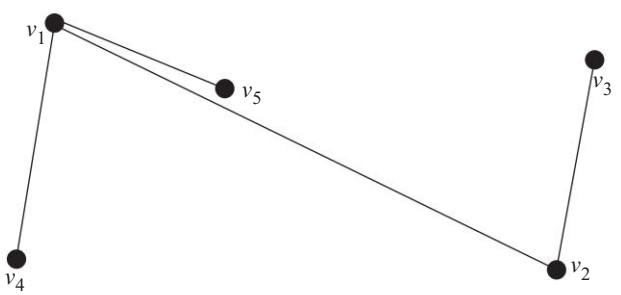


FIGURE 12–5 The tree constructed by the greedy method.



Next, we shall analyze the performance of this on-line algorithm based on the greedy method. Let S denote a set of n points. Let l denote the length of a minimum spanning tree constructed on S. Let T_{onl} denote the Steiner tree constructed by our on-line algorithm.We shall prove that our algorithm is $O(\log n)$ competitive.

Essentially, we shall prove that on T_{onl}, the kth largest edge has length at most $2l/k$, $1 \le k \le n-1$. Another way of saying is that on T_{onl}, there are at most $(k-1)$ edges whose lengths are greater than $2l/k$. Given a sequence of vertices which will produce T_{onl}, let S_k be the set of points whose additions to the tree T_{onl} will cause T_{onl} to have edges with lengths larger than $2l/k$. The original statement now becomes the following: The cardinality of S_k is less than k. To prove this, note that by the definition of our greedy on-line algorithm, the distance between every pair of points in S_k must be larger than $2l/k$. Thus, the length of an optimal traveling salesperson problem tour on S_k must be larger than

$$|S_k| \frac{2l}{k}.$$

Since the length of an optimal traveling salesperson problem tour on a set of points is at most two times the length of a minimum Steiner tree of the same set of points, the length of a minimum Steiner tree on S_k is greater than

$$|S_k| \frac{l}{k}.$$

Since the length of a minimum Steiner tree on S_k is less than that on S, we have

$$|S_k| \frac{l}{k} < l.$$

or equivalently,

$$|S_k| < k.$$

This means that the cardinality of S_k is less than k. That is, we have proved that our original statement: *The length of the kth largest edge of T_{onl} is at most $2l/k$*. The total length of T_{onl} is thus at most

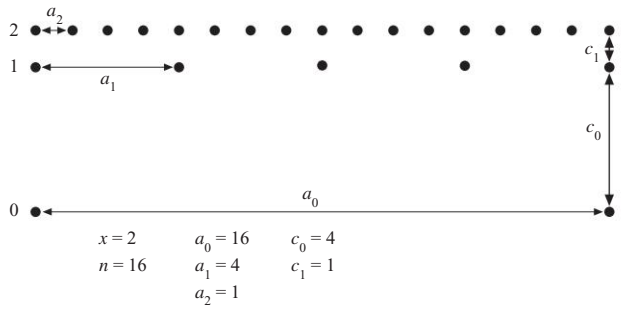
$$\sum_{k=1}^{n-1} \frac{2l}{k} = 2l \sum_{k=1}^{n-1} \frac{1}{k} = l \times (\log n).$$

This indicates that our greedy on-line spanning tree algorithm is $O(\log n)$-competitive. Is this algorithm optimal? Certainly not, because we can prove that a lower bound of the competitive ratio is $\Omega(\log n/\log\log n)$. This is discussed below. We shall find an input σ for the on-line spanning tree problem and prove that there exists no on-line algorithm A such that the ratio between the length of tree constructed by A with σ and the length of minimum spanning tree with σ is $\Omega(\log n/\log\log n)$. Next, we describe the input σ. All the points of input σ lie in a grid. Assume that x is an integer and $x \geq 2$. Let $x^{2x} = n$. Then $x \geq 1/2(\log n/\log\log n)$ and $n \geq 16$. The input consists of $x + 1$ layer of points, where each layer has a set of equally spaced points on a horizontal line of length $n = x^{2x}$. The coordinates of the points in layer i, $0 \leq i \leq x$, are (ja_i, b_i) where $a_i = x^{2x-2i}$, $b_i = 0$ for $i = 0$, $b_i = \sum_{k=1}^{i} a_k$ for $i \neq 0$, and $0 \leq j \leq n/a_i$. Thus, $a_0 = x^{2x}(= n)$ and $a_x = 1$. The vertical distance between layer i and layer $i + 1$ is $c_i = b_{i+1} - b_i = a_{i+1}$ for all i. The number of points in layer i is $\frac{n}{a_i} + 1$. The total number of input points is

$$\begin{aligned}
\sum_{i=0}^{x}\left(\frac{n}{a_i}+1\right) &= \sum_{i=0}^{x}\left(\frac{x^{2x}}{x^{2x-2i}}+1\right)\\
&= \sum_{i=0}^{x}(x^{2i}+1)\\
&= \frac{x^{2x+2}-1}{x^2-1}+x+1\\
&= \frac{nx^2-1}{x^2-1}+x+1\\
&= n+\frac{n-1}{x^2-1}+x+1\\
&= O(n).
\end{aligned}$$

An example input σ for $x = 2$ and $n = 16$ is shown in Figure 12–6.

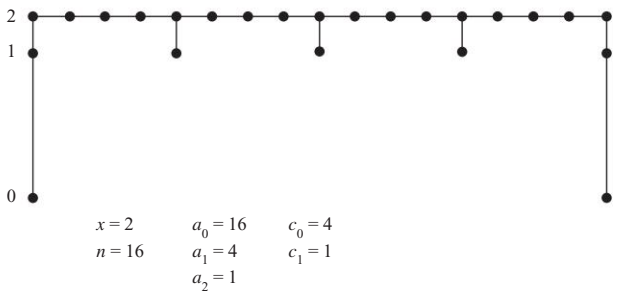
FIGURE 12–6 An example input σ.



A minimum spanning tree of this input can be constructed as follows:

(1) Each point in layer x is linked to its neighbors horizontally.
(2) Each other point is linked to its neighbors vertically. (See Figure 12–7 for a minimum spanning tree of the points depicted in Figure 12–6.)

FIGURE 12–7 The minimum spanning tree of the points in Figure 12–6.



The total length of this tree is

$$n + \sum_{i=0}^{x-1} c_i \left(\frac{n}{a_i} + 1 \right) = n + \sum_{i=0}^{x-1} a_{i+1} \left(\frac{n}{a_i} + 1 \right)$$

$$= n + \sum_{i=0}^{x-1} a_{i+1} + n \sum_{i=0}^{x-1} \frac{a_{i+1}}{a_i}$$

$$= n + \sum_{i=0}^{x-1} \frac{n}{x^{2i+2}} + n \sum_{i=0}^{x-1} \frac{1}{x^2}$$

$$\leq n + n \sum_{i=0}^{x-1} \frac{1}{x^2} + n \sum_{i=0}^{x-1} \frac{1}{x^2}$$

$$= n + 2n \frac{x}{x^2}$$

$$\leq 3n.$$

Suppose the points of σ are given on-line layer by layer, from layer 0 and ending with layer x. Let T_{i-1} denote the graph of the on-line algorithm just before it gets the points in layer i. There are $\left(\frac{n}{a_i} + 1\right) \geq \frac{n}{a_i}$ points arranged in layer i. Note that the distance between two consecutive points in layer i is a_i. For each point, the length of any edge of $T_i - T_{i-1}$ must be at least a_i. That is, for layer i, the total length of the spanning tree increased by the on-line algorithm is at least

$$\frac{n}{a_i} \cdot a_i = n.$$

Thus, the total length of the tree constructed by the on-line algorithm is at least $n \cdot x$.

Since $C_{off} \leq 3n$ and $C_{onl} \geq nx$, we have

$$\frac{C_{onl}}{C_{off}} \geq \frac{1}{3n} \cdot nx = \frac{1}{3} x > \frac{\log n}{6 \log \log n}.$$

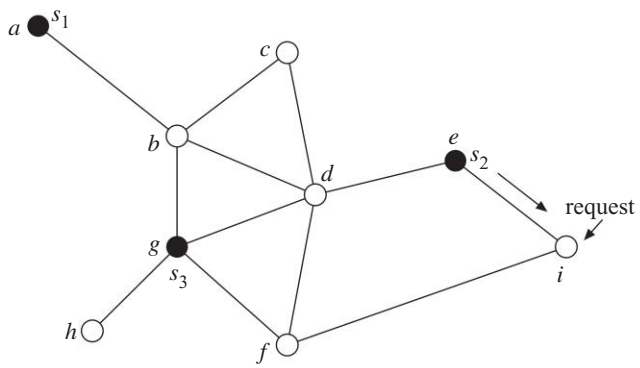
This means that a lower bound of the competitive ratio of this problem is $O(\log n / \log \log n)$. Since our algorithm is $O(\log n)$-competitive, it is not optimal.

12–2 THE ON-LINE *k*-SERVER PROBLEM AND A GREEDY ALGORITHM TO SOLVE THIS PROBLEM DEFINED ON PLANAR TREES

Let us consider the k-server problem. We are given a graph with n vertices and each edge is associated with a positive edge length. Let there be k servers stationed at k vertices where $k < n$. Given a sequence of requests of servers, we must decide how to move servers around to satisfy the requests. The cost of

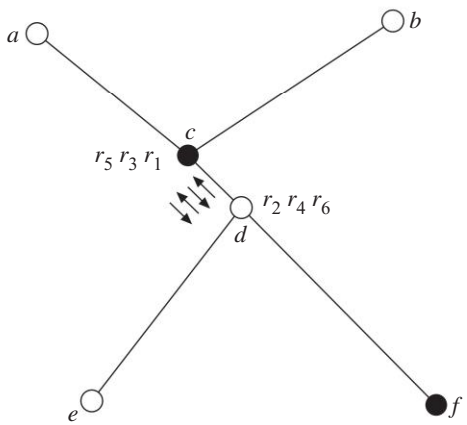
serving a request is the total distances between the servers moved to satisfy the request. Consider Figure 12–8. There are three servers s_1, s_2, and s_3, located at a, e and g, respectively. Suppose that we have a request at vertex i. Then one possible move is to move s_2, presently located at vertex e, to vertex i because e is close to i.

FIGURE 12–8 A k-server problem instance.



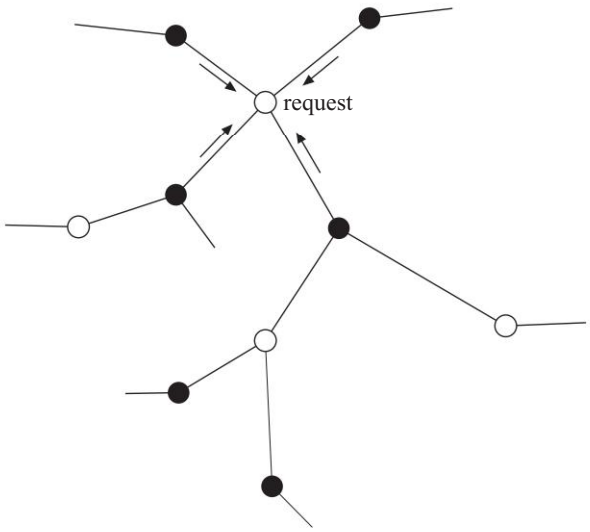
An on-line algorithm to solve this k-server problem based upon the greedy strategy would move the nearest server to the vertex where the request is located. This simple-minded greedy on-line algorithm suffers from one drawback, shown in Figure 12–9.

FIGURE 12–9 A worst case for the greedy on-line k-server algorithm.



Suppose that there is a server located at vertex d and the first request r_1 is located at c. Our greedy algorithm would move the server located at d to vertex c. Then the second request arrives and unfortunately, it is located at d. Thus, we would move the server now located at c to vertex d. As illustrated in Figure 12–9, if the requests are located alternatively at c and d, the server will be moved between c and d all the time. In fact, as shown below, if we move the server at f towards d gradually, this server will finally be located in d at some time and this phenomenon of moving between c and d continuously will not occur. Our modified greedy algorithm to solve the problem moves many so-called active servers towards the vertex where the request is presently located. Let us restrict ourselves to the k-server problem on a planar tree T. Since T is a planar tree, the lengths of edge satisfy the triangular inequality. Let x and y be two points on T. The distance between x and y, denoted as $|x, y|$, is defined as the length of the simple path from x to y through T. Let s_i and d_i denote server i and the location of server i respectively. Let interval $(x, y]$ denote the path from x to y through T, excluding x. We call a server s_i active with respect to a request located at x, if there are no more servers in the interval $(d_i, x]$. The modified greedy on-line k-server algorithm works as follows: When a request is located at x, move all the active servers with respect to x continuously with the same speed towards x until one of them reaches x. If during this moving period, an active server becomes inactive, then it halts. Figure 12–10 illustrates the above modified greedy algorithm, in response to a request.

FIGURE 12–10 The modified greedy on-line k-server algorithm.

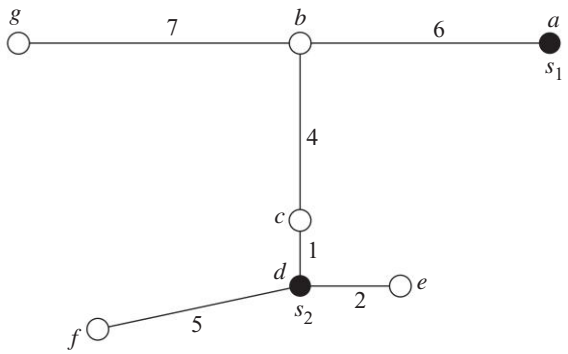


We now present an analysis of this algorithm. We first define a fully informed on-line k-server algorithm. This on-line k-server algorithm is absolutely on-line. That is, after each request, this algorithm will move a server to the vertex where the request is located. However, since the algorithm is fully informed, it has the complete information of the sequence of requests. Therefore, its behavior will be quite different from an ordinary on-line algorithm which is by definition not fully informed. In fact, it is an optimal algorithm because it produces an optimal result.

Consider Figure 12–11. There are two servers, s_1 and s_2, located at a and d respectively. Suppose that the request sequence is as follows:

(1) r_1 at b.
(2) r_2 at e.

FIGURE 12–11 An instance illustrating a fully informed on-line k-server algorithm.



For a greedy on-line algorithm which has no full "information", the moving of servers will be as follows:

(1) Move s_2 from d to b. The cost is 5.
(2) Move s_2 from b to e. The cost is 7.

The total cost is consequently 12.
For the fully informed on-line algorithm, it would do the following:

(1) Move s_1 from a to b. The cost is 6.
(2) Move s_2 from d to e. The cost is 2.

The total cost is only 8. This fully informed on-line algorithm moves s_1 instead of s_2 at the very beginning because it knows that r_2 is located at e.

Our analysis is to compare the cost of the modified greedy on-line k-server algorithm with the cost of the fully informed on-line k-server algorithm. Let us imagine that we are playing a game. Each time a request arrives, our adversary, who is fully informed, makes a move. Then we make a move. It is important to note that our adversary moves only one server while we move all the servers active with respect to the request. Just as we did in the amortized analysis, we define a potential function which maps all the locations of our and adversary k servers to a non-negative real number.

Let $\tilde{\psi}_i$ denote the value of the potential function after the adversary moves in response to the ith request and before our algorithm makes any move for the ith request. Let ψ_i denote the value of the potential function after our algorithm makes the move after the ith request and before the $(i + 1)$th request. Let ψ_0 denote the potential value of the potential function. These terms can be best illustrated in Figure 12–12.

FIGURE 12–12 The value of potential functions with respect to requests

⋮
- ith request:
  adversary moves;
  $\leftarrow \tilde{\psi}_i$

  we move;
  $\leftarrow \psi_i$
- $(i + 1)$th request:

⋮

Let the costs of our algorithm and the adversary algorithm for the ith request be O_i and A_i respectively. Let O and A denote the total costs of our algorithm and the adversary algorithm after all requests are made respectively. Then we shall try to prove the following.

(1) $\tilde{\psi}_i - \psi_{i-1} \le \alpha A_i, \quad 1 \le i \le n$, for some α.

(2) $\tilde{\psi}_i - \psi_i \ge \beta O_i, \quad 1 \le i \le n$, for some β.

We may expand the above equations to:

$$\begin{aligned}
\tilde{\psi}_1 - \psi_0 &\le \alpha A_1 \\
\psi_1 - \tilde{\psi}_1 &\le -\beta O_1 \\
\tilde{\psi}_2 - \psi_1 &\le \alpha A_2 \\
\psi_2 - \tilde{\psi}_2 &\le -\beta O_2 \\
&\vdots \\
\tilde{\psi}_n - \psi_{n-1} &\le \alpha A_n \\
\psi_n - \tilde{\psi}_n &\le -\beta O_n.
\end{aligned}$$

Summing up, we have

$$\begin{aligned}
\psi_n - \psi_0 &\le \alpha(A_1 + A_2 + \cdots + A_n) - \beta(O_1 + O_2 + \cdots + O_n) \\
\beta O &\le \alpha A + \psi_0 - \psi_n.
\end{aligned}$$

Since $\psi_n \ge 0$,

$$\begin{aligned}
\beta O &\le \alpha A + \psi_0 \\
O &\le \frac{\alpha}{\beta} A + \frac{1}{\beta} \psi_0 .
\end{aligned}$$

We now define the potential function. At any time instance, let the k servers of our algorithm be located at $b_1, b_2, \ldots, b_k$, and the k servers of our adversary algorithm be located at $a_1, a_2, \ldots, a_k$. Let us define a bipartite graph with components $v_1, v_2, \ldots, v_k$ and $v'_1, v'_2, \ldots, v'_k$ where $v_i(v'_i)$ represents $b_i(a_i)$ and the weight of edge (v_i, v'_j) is $|b_i, a_j|$. On this bipartite graph, we can perform a minimum weighted matching. Let $M_{\min}$ denote the minimum weighted matching on this bipartite graph. Our potential function is defined as

$$\psi = k|M_{\min}| + \sum_{i<j} |b_i, b_j|$$

Let us consider $\tilde{\psi}_i - \psi_{i-1}$. Since only the adversary makes a move, only one a_i will change. Therefore, in ψ, the only increase possible is related to the change of $|M_{\min}|$, which is equal to A_i. Thus, we have

$$\tilde{\psi}_i - \psi_{i-1} \le kA_i \qquad \textbf{(12–1)}$$

Now, consider $\psi_i - \tilde{\psi}_i$. For the ith request, q, $q \le k$, servers of our algorithm are moved. Let us assume that each of the q servers moves d distance. For $M_{\min}$, at least one matched edge will be reduced to zero because both algorithms will have to satisfy the ith request and the distance between one pair of v_a and v_b will be zero. This means that $M_{\min}$ will increase at most $-d + (q - 1)d = (q - 2)d$.

We now compute the increase of $\sum_{i<j} |b_i, b_j|$. Let d_p be the location of one of the active servers of our algorithm. Let b_r be the location of a server such that d_p is between the ith request and b_r. Let l_p denote the number of b_r's. Since the server located at d_p moves d distance, the sum of distance between d_p and l_p b_r's increase by $(l_p - 1)d$. But for the other $(k - l_p)$ servers of our algorithm, the sum of distances decrease by $(k - l_p)d$. Thus, the total distance among our servers increase at most $\sum_{p=1}^{q} (l_p - 1 + l_p - k)d$ as there are q servers which are moved for this ith request. The change of potential function is therefore at most

$$k(q - 2)d + \sum_{p=1}^{q} (l_p - 1 + l_p - k)d = -qd. \quad \text{(Note that } \sum_{p=1}^{q} l_p = k.\text{)}$$

That is, we have

$$\psi_i - \tilde{\psi}_i \le -qd.$$

Or, equivalently,

$$\tilde{\psi}_i - \psi_i \ge O_i. \tag{12–2}$$

Combining Equations (12–1) and (12–2), we have

$$O \le kA + \psi_0$$

which is desired. Thus, our on-line algorithm for the k-server problem is k-competitive.

We have proved that the on-line k-server algorithm is k-competitive. We naturally will ask: Is our algorithm optimal? That is, can there be any on-line k-server algorithm which has a better performance? For instance, can there be a $k/2$-competitive on-line k-server algorithm? We shall show that this is impossible. In other words, our on-line k-server algorithm is indeed optimal.

We first note that our on-line algorithm is defined on a planar tree where edge lengths satisfy the triangular inequality. Besides, our algorithm moves several, instead of one, servers. For the purpose of proving the optimality of our on-line k-server algorithm, we now define a term called "lazy on-line k-server algorithm". An on-line k-server algorithm is called *lazy* if it moves only one server to handle each request.

We can easily see that we can modify an on-line k-server algorithm which is not lazy to a lazy on-line k-server algorithm without increasing its cost if this algorithm is executed on a graph satisfying the triangular inequality. Let D be an on-line k-server algorithm which is not lazy. Suppose D moves a server s from v to w to satisfy a certain request. But, since D is not lazy, before this step, D moves s from u to v even no request located at v. The cost of moving s from u to w is $d(u, v) + d(v, w)$ which is greater than or equal to $d(u, w)$ because of the triangular inequality. We may therefore eliminate the step of moving s from u to v without increasing the cost.

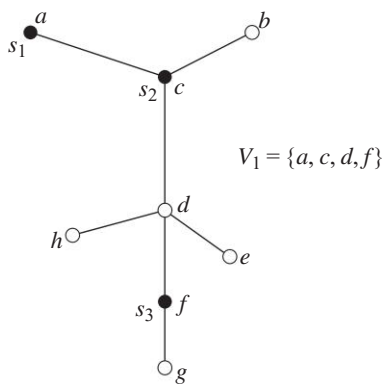
In the following discussion, we shall assume that our on-line k-server algorithm is a lazy one.

Let D be any on-line k-server algorithm. Let $G(V, E)$ be the graph containing k servers located at k distinct vertices of G. Let v_1 be a subset of V which contains k vertices where the k servers are initially located and another arbitrary vertex x of G. Define a request sequence as follows.

(1) $\sigma(1)$ is located at x.
(1) $\sigma(i)$ is the unique vertex in v_1 not covered by D at time i for $i \geq 1$.

Consider Figure 12–13.

FIGURE 12–13 An example for the 3-server problem.



Suppose $k = 3$ and the three servers are located at a, c, and f initially. Let the other vertex chosen be d. Then $v_1 = \{a, c, f, d\}$. Now the first request $\sigma(1)$ must be d. Suppose D vacates c by moving s_2 from c to d. Then $\sigma(2) = c$. If D further moves s_1 from a to c, then $\sigma(3) = a$. With this kind of request sequence, the cost of serving D on $\sigma(1), \sigma(2), \ldots, \sigma(t)$ is

$$C_D(\sigma, t) = \sum_{i=1}^{t} d(\sigma(i+1), \sigma(i)).$$

We now define an on-line k-server algorithm which is more informed than Algorithm D. We call this algorithm Algorithm E. Algorithm E is more informed because it somehow knows $\sigma(1)$. Let v_2 be $\sigma(1)$ and any subset of $v_1 - \sigma(1)$ with $k - 1$ vertices. For instance, for the above case, $v_1 = \{a, c, f, d\}$. Since $\sigma(1) = d$, v_2 may be $\{a, c, d\}$, $\{c, f, d\}$ or $\{a, f, d\}$. With this subset of vertices v_2 in mind, we may define an on-line k-server algorithm $E(v_2)$, which is more informed than D as follows:

> If v_2 contains $\sigma(1)$, do nothing; otherwise, move the server at vertex $\sigma(i-1)$ to $\sigma(i)$ and update v_2 to reflect this change. Initially, the servers occupy the vertices in v_2.

Note that in this case, the initial condition is changed because one server is initially located at where $\sigma(1)$ is. Under this circumstance, $E(v_2)$ does not have to move any server around to meet the first request. This is why we say that $E(v_2)$ is more informed than D. The cost of $E(v_2)$ will never be greater than that of D and can therefore be used as a lower bound of all on-line k-server algorithms. It is easy to see that for all $i > 1$, v_2 always contains $\sigma(i-1)$ when step i begins.

Again, let us illustrate this concept by an example. Consider the case shown in Figure 12–13. Assume that $v_2 = \{a, c, d\}$ and

$$\sigma(1) = d$$
$$\sigma(2) = e$$
$$\sigma(3) = f$$
$$\sigma(4) = a.$$

The three servers s_1, s_2 and s_3 initially occupy a, c and d respectively. Then $E(v_2)$ behaves as follows:

(1) Since v_2 contains $\sigma(1) = d$, do nothing.
(2) Since v_2 does not contain $\sigma(2) = e$, move the server s_3 located at $\sigma(1) = d$ to e. Let $v_2 = \{a, c, e\}$.
(3) Since v_2 does not contain $\sigma(3) = f$, move the server s_3 located at $\sigma(2) = e$ to f. Let $v_2 = \{a, c, f\}$.
(4) Since v_2 contains $\sigma(3) = a$, do nothing.

As v_1 contains $k + 1$ vertices, we can have $\binom{k}{k-1} = k$ distinct v_2's. Thus, there are k distinct $E(v_2)$'s. One may naturally ask: Which $E(v_2)$ performs the best? Instead of finding the cost of the best $E(v_2)$ algorithm, we shall find the expected cost of these $kE(v_2)$ algorithms. The cost of the best $E(v_2)$ algorithm will be smaller than the expected one.

By the definition of $E(v_2)$ algorithms, step 1 does not cost anything. At step $i + 1$, (for $i \geq 1$), each of these algorithms either does nothing with no cost or moves a server from $\sigma(i)$ to $\sigma(i + 1)$ with cost $d(\sigma(i), \sigma(i + 1))$. Among the k algorithms, $\binom{k-1}{k-1} = 1$ of them, the one which contains $\sigma(i)$, but not $\sigma(i + 1)$, incurs this cost, and all of the others incur no cost. So for step $i + 1$, the total cost increased by all of these algorithms is $d(\sigma(i), \sigma(i + 1))$. The total cost of these k algorithms up to and including $\sigma(t)$ is

$$\sum_{i=1}^{t} d(\sigma(i), \sigma(i-1)).$$

Thus, the expected cost of $E(v_2)$ is

$$\begin{aligned} E(v_2) &= \frac{1}{k} \sum_{i=2}^{t} d(\sigma(i), \sigma(i-1)) \\ &= \frac{1}{k} \left(d(\sigma(2), \sigma(1)) + d(\sigma(3), \sigma(2)) + \cdots + d(\sigma(t), \sigma(t-1))\right) \\ &= \frac{1}{k} \left(\sum_{i=1}^{t} d(\sigma(i+1), \sigma(i)) - d(\sigma(t+1), \sigma(t)) \right) \\ &= \frac{1}{k} \left(C_D(\sigma, t) - d(\sigma(t+1), \sigma(t))\right) \end{aligned}$$

because $C_D(\sigma, t) = \sum_{i=1}^{t} d(\sigma(i+1), \sigma(i))$.

The cost of the best $E(v_2)$ algorithm must be lower than this expected one. Let it be $E(v_2')$. The cost of $E(v_2')$ is denoted as $C_{E(v_2')}$. Thus,

$$C_{E(v_2)} \geq C_{E(v_2')}.$$

This means that

$$\frac{1}{k} C_D(\sigma, t) - \frac{1}{k} d(\sigma(t+1), \sigma(t)) \geq C_{E(v_2')}(\sigma, t).$$

That is,

$$C_D(\sigma, t) \geq k \cdot C_{E(v_2')}(\sigma, t).$$

or equivalently,

$$\frac{C_D(\sigma, t)}{C_{E(v_2')}(\sigma, t)} \geq k.$$

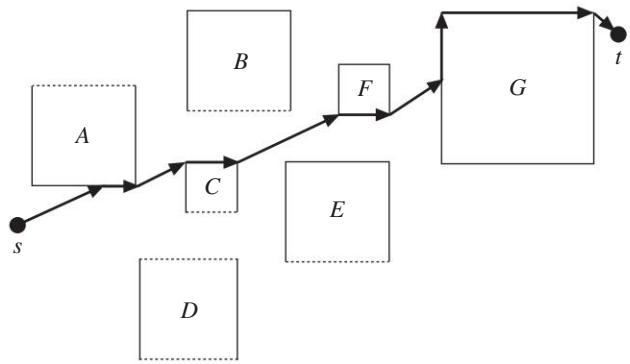
The above equation indicates that for any on-line k-server algorithm which is not informed at all, there exists a more informed on-line k-server algorithm whose cost is k times smaller. This proves that the greedy on-line k-server algorithm defined on planar trees cannot be better.

12–3 AN ON-LINE OBSTACLE TRAVERSAL ALGORITHM BASED ON THE BALANCE STRATEGY

Let us consider the obstacle traversal problem. There is a set of square obstacles. All the sides of these squares are parallel to the axes. The length of each side of the squares is less than or equal to 1. There are a starting point, denoted as s, and a goal point, denoted as t. Our job is to find a shortest path from s to t which avoids the obstacles. Figure 12–14 illustrates a typical example.

The algorithm introduced here for such a problem is an on-line one. That is, we do not have the whole picture of all of these obstacles. Thus, it is impossible to obtain an optimal solution. The searcher starts from s and uses many heuristics to guide him. These heuristics form the on-line algorithm. In the rest of this section, we shall describe these rules.

FIGURE 12–14 An example for the traversal problem.

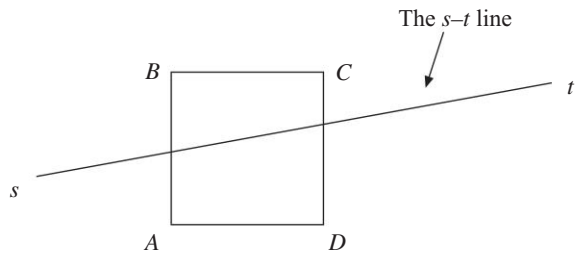


We first note that the line joining s and t is the shortest possible geometric distance between s and t. Thus, this distance, denoted as d, serves as a lower bound of our solution. Later, we shall compare our solutions with d. It will be shown that the distance traveled by our searcher by using our on-line algorithm is no more than $\frac{3}{2}d$ when d is large.

We assume that the line from s to t has an angle $\psi \leq \frac{\pi}{4}$ with the horizontal axis. If otherwise, it will have an angle $\psi' \leq \frac{\pi}{4}$ with the vertical axis and obviously, our algorithm can still be applied with a slight modification.

Let α denote the direction from s to t. Next, we shall present all the rules concerned with different cases. First, let us see how many cases we must deal with. Consider Figure 12–15.

FIGURE 12–15 An obstacle *ABCD*.



There are at least three cases:

Case 1: The searcher is traveling between obstacles.

Case 2: The searcher hits the horizontal side of an obstacle. That is, it hits *AD*.

Case 3: The searcher hits the vertical side of an obstacle. That is, it hits *AB*.

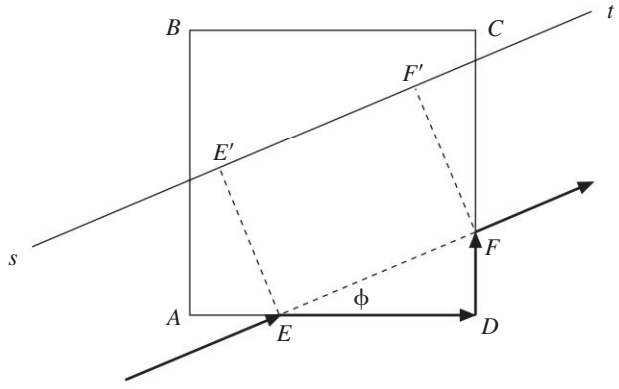
For Case 1, we have Rule 1.

RULE 1: When the searcher is traveling between obstacles, it travels in the direction α. That is, it travels as if there were no obstacles.

Once we have this rule, we may imagine that there are two searchers, denoted as *P* and *Q*. Searcher *P* is our real searcher which will hit obstacles and then try to avoid them by going around them. Searcher *Q* is a fictitious one which travels from *s* to *t* along the straight line joining *s* and *t*. We like to compare the distance traveled by *P* with that by *Q*.

RULE 2: When the searcher hits the horizontal side of a square *AD* at point *E*, it travels from *E* to *D* and goes up to *F* such that *EF* is parallel to the *s*–*t* line as illustrated in Figure 12–16. Afterwards, it resumes the direction α.

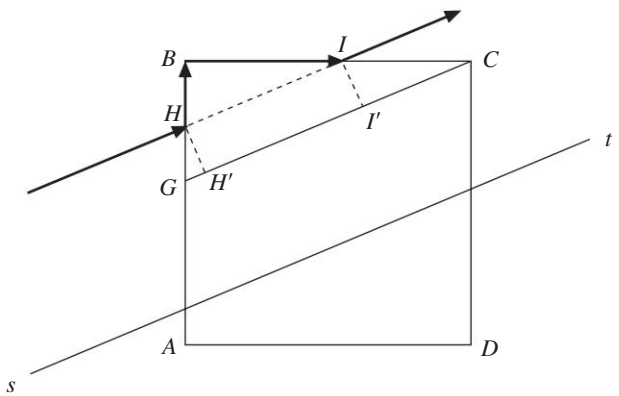
FIGURE 12–16 Hitting side *AD* at *E*.



Our searcher P travels the distance $|ED| + |DF|$. Our fictitious searcher Q, for the same time period, would travel the distance $|EF|$, illustrated in Figure 12–16. $(|ED| + |DF|)/|EF| = \cos\psi + \sin\psi \leq \frac{3}{2}$. Thus, we conclude that if our searcher hits the horizontal side of a square, the distance that it travels is no more than $\frac{3}{2}$ of the distance traveled by the fictitious searcher Q.

If searcher P hits the vertical side, the case becomes more complicated. We first draw a line through C, in the direction α until it hits AB at G, as illustrated in Figure 12–17.

FIGURE 12–17 Hitting side AB at H.



We then have Rule 3.

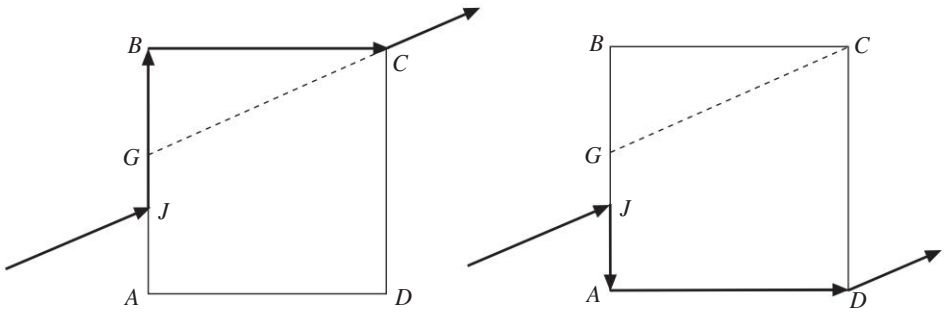
RULE 3: If searcher P hits AB within interval BG at H, it would go up to B, travel right until it hits I such that HI is parallel to the s–t line, illustrated in Figure 12–17. Afterwards, it resumes the direction α.

Again, it is easy to prove that the ratio of the distance traveled by searcher P and the distance traveled by the fictitious searcher Q is no more than $\frac{3}{2}$ for this case.

For the case of searcher P hitting interval AG, more complicated rules exist. In fact, these rules form the core of the balancing strategy.

RULE 4: If searcher P hits AB within interval AG, it either goes up or goes down. If it goes up, it goes to B, turns to the right until it hits corner C as shown in Figure 12–18(a). If it goes down, it goes to A, turns to the right until it hits corner D as shown in Figure 12–18(b). After hitting the corner, it resumes the direction α.

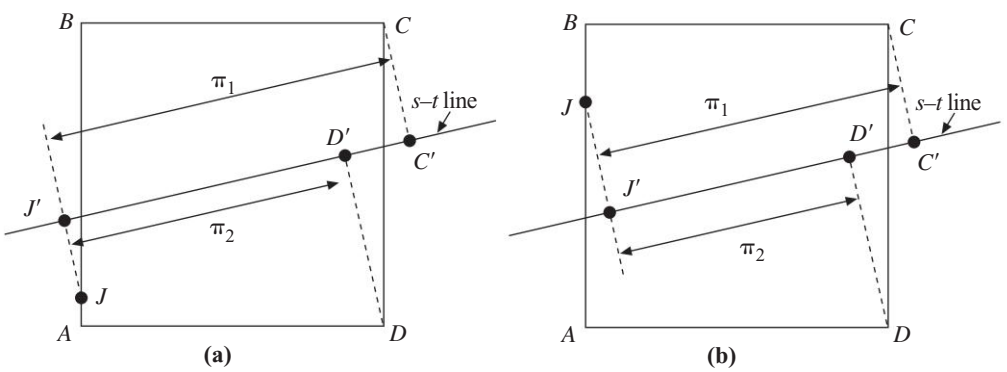
FIGURE 12–18 Two cases for hitting side AB.



Rule 4 only states that the searcher either goes up or goes down if it hits interval AG. Now the question is: How do we decide whether we go up or down?

The distance traveled by P is either $\tau_1 = |JB| + |BC|$ if it goes up or $\tau_2 = |JA| + |AD|$ if it goes down. For this period, the distance traveled by our fictitious searcher Q is denoted by π_1 or π_2 for going up and going down respectively. Distances π_1 and π_2 are obtained by projecting points J, C and D onto the s–t line, illustrated in Figure 12–19. In Figure 12–19(a), J is below the s–t line and in Figure 12–19(b), J is above the s–t line.

FIGURE 12–19 The illustration for π_1 and π_2.



It would be good if both $\frac{\tau_1}{\pi_1}$ and $\frac{\tau_2}{\pi_2}$ are not greater than $\frac{3}{2}$. But it is not the case. Consider the situation in Figure 12–20. Suppose the direction of our traveler is going up. Then clearly, $\frac{\tau_1}{\pi_1}$ is almost 2. Similarly, consider the case in Figure 12–21. If the decision is going down, $\frac{\tau_2}{\pi_2}$ is again almost 2.

FIGURE 12–20 The case for the worst value of τ_1/π_1.

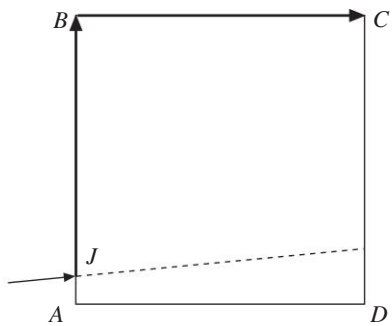
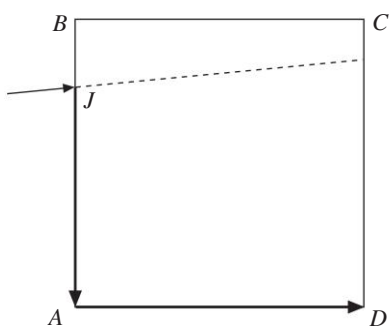


FIGURE 12–21 The case for the worst value of τ_2/π_2.



In 1978, it was proved that at least one of $\frac{\tau_1}{\pi_1}$ and $\frac{\tau_2}{\pi_2}$ is no more than $\frac{3}{2}$. Note that the hitting point J is either below or above the s–t line. It appears that if J is above the s–t line and $\frac{\tau_1}{\pi_1} \leq \frac{3}{2}$, then we should move up and if J is below the s–t

line and $\frac{\tau_2}{\pi_2} \leq \frac{2}{3}$, then we should move down. Still, this may cause trouble because we may be continuously moving up, or down, which is not necessarily healthy as we may be moving farther and farther away from the *s*–*t* line.

Let the side length of a square be k where $k \leq 1$. We partition the interval AG into equal segments of length $k/\sqrt{d}$ where d is the length of the *s*–*t* line as defined before. A segment is labeled up if its lowest point satisfies $\frac{\tau_1}{\pi_1} \leq \frac{3}{2}$ and is labeled down if its lowest point satisfies $\frac{\tau_2}{\pi_2} \leq \frac{3}{2}$. A segment can be labeled as both up and down. We call a segment "pure" if it has only one label and others "mixed".

Let J_i be the lowest point of the ith segment and let J_i' be the intersection of the line from J_i in the direction α with side CD. If the ith segment is pure, we define $\rho_i = |DJ_i'|/|CJ_i'|$ if this segment is labeled only up and $\rho_i = |CJ_i'|/|DJ_i'|$ if this segment is labeled only down. The algorithm maintains a parameter, called *balance* with every segment. All balances are initialized to be 0. As shown later, *balance* is used to control moving up or down.

RULE 5: Assume that searcher P hits the interval at point J which belongs to the ith segment.

Case 1: J is above the *s*–*t* line. Check the label of the ith segment. If it is labeled down, move down. Otherwise, check *balance*. If *balance* $\geq \rho_i k$, move down and subtract $\rho_i k$ from *balance*; otherwise, add k to *balance* and move up.

Case 2: J is below the *s*–*t* line. Check the label of the ith segment. If it is labeled up, then move up. Otherwise, check *balance*. If *balance* $\geq \rho_i k$, move up and subtract $\rho_i k$ from *balance*; otherwise add k to *balance* and move down.

Finally, we have the following rule.

RULE 6: If searcher P hits the same x-coordinate or y-coordinate of the goal, it goes directly to the goal.

We have presented the on-line obstacle traversal algorithm. We now discuss its performance. First of all, let us remind ourselves that if searcher P hits side

AD or the interval BG of side AB, then the ratio of the distance traveled by P and the distance traveled by Q is no more than $\frac{3}{2}$. Thus, we only have to confine ourselves to the case of hitting the interval AG.

By the definition of being mixed, no matter whether searcher P goes up or down, $\frac{\tau_1}{\pi_1}$ or $\frac{\tau_2}{\pi_2}$ is no more than $\frac{3}{2}$. Thus, we shall only consider the case where the segment is pure. Let us consider the ith segment. Assume that it is labeled up and the searcher P is above the s–t line. The case that it is labeled down is similar. Let $a_i(b_i)$ be the sum of sides of the squares for which searcher P hits the ith segment and goes up (down). The total distance traveled by searcher P with respect to the ith segment is $a_i\tau_1 + b_i\tau_2$ while the total distance traveled by searcher Q with respect to the ith segment is $a_i\pi_1 + b_i\pi_2$, where the quantities τ_1, τ_2, π_1 and π_2 are all with respect to the lowest point of the ith segment of a unit square. Then

$$\frac{a_i\tau_1 + b_i\tau_2}{a_i\pi_1 + b_i\pi_2} = \frac{\frac{a_i}{b_i}\tau_1 + \tau_2}{\frac{a_i}{b_i}\pi_1 + \pi_2}.$$

Now, the question is: How large is $\frac{a_i}{b_i}$? Note that both a_i and b_i are related to the balance used in the algorithm. Totally, $\rho_i b_i$ is subtracted from *balance* and a_i is added. Since *balance* never becomes negative, we have $\rho_i b_i \le a_i$. Since the ith segment is labeled up, by definition we have $\frac{\tau_1}{\pi_1} \le \frac{3}{2}$.

We now try to find an upper bound of $\frac{\rho_i\tau_1 + \tau_2}{\rho_i\pi_1 + \pi_2}$.

$$\frac{\rho_i\tau_1 + \tau_2}{\rho_i\pi_1 + \pi_2} = \frac{\frac{|DJ_i'|}{|CJ_i'|}\cdot\tau_1 + \tau_2}{\frac{|DJ_i'|}{|CJ_i'|}\cdot\pi_1 + \pi_2}$$

$$= \frac{|DJ_i'|\tau_1 + |CJ_i'|\tau_2}{|DJ_i'|\pi_1 + |CJ_i'|\pi_2}.$$

Assume that we have a unit square. The numerator

$$
\begin{aligned}
&|DJ_i'|\tau_1 + |CJ_i'|\tau_2 \\
&= (|AJ_i| + \tan\psi)(2 - |AJ_i|) + (1 - |AJ_i| - \tan\psi)(1 + |AJ_i|) \\
&= \frac{1}{\cos\psi}((\sin\psi + \cos\psi) + 2(\cos\psi - \sin\psi)|AJ_i| - 2\cos\psi\,(|AJ_i|)^2).
\end{aligned}
$$

Let $|AJ_i| = y$, $\sin\psi = a$ and $\cos\psi = b$.

$$
\begin{aligned}
&|DJ_i'|\tau_1 + |CJ_i'|\tau_2 \\
&= \frac{1}{b}((a + b) + 2(b - a)y - 2by^2).
\end{aligned}
$$

The denominator

$$
\begin{aligned}
&|DJ_i'|\pi_1 + |CJ_i'|\pi_2 \\
&= (|AJ_i| + \tan\psi)(\sin\psi + \cos\psi - |AJ_i|\sin\psi) \\
&\quad + (1 - |AJ_i| - \tan\psi)(\cos\psi - |AJ_i|\sin\psi) \\
&= \frac{1}{\cos\psi}(\sin^2\psi + \cos^2\psi) \\
&= \frac{1}{\cos\psi} \\
&= \frac{1}{b}.
\end{aligned}
$$

Thus,

$$
\frac{|DJ_i'|\tau_1 + |CJ_i'|\tau_2}{|DJ_i'|\pi_1 + |CJ_i'|\pi_2} = (a + b) + 2(b - a)y - 2by^2.
$$

Let $f(y) = (a + b) + 2(b - a)y - 2by^2$. Then

$$
f'(y) = 2(b - a) - 4by.
$$

When $y = \dfrac{b-a}{2b}$, $f(y)$ is maximized.

$$f\left(\frac{b-a}{2b}\right)=(a+b)+2(b-a)\frac{b-a}{2b}-2b\left(\frac{b-a}{2b}\right)^2$$
$$=(a+b)+\frac{(b-a)^2}{b}-\frac{(b-a)^2}{2b}$$
$$=(a+b)+\frac{(b-a)^2}{2b}$$
$$=\frac{2b^2+b^2+a^2}{2b}.$$

Since $a^2 + b^2 = \sin^2\psi + \cos^2\psi = 1$, we have

$$f\left(\frac{b-a}{b}\right)=\frac{2b^2+1}{2b}.$$

Because $b = \cos\psi$ and $\psi \le \pi/4$, we have

$$f\left(\frac{b-a}{b}\right)=\frac{2b^2+1}{2b}\le\frac{3}{2}.$$

Thus,

$$\frac{\sigma_i\tau_1+\tau_2}{\sigma_i\pi_1+\pi_2}\le\frac{3}{2}.$$

It is easy to see that the above inequality still holds for squares with size k (≤ 1). Then we have

$$\frac{\rho_i\tau_1+\tau_2}{\rho_i\pi_1+\pi_2}\le\frac{3}{2}.$$
$$2\rho_i\tau_1+2\tau_2\le 3\rho_i\pi_1+3\pi_2$$
$$(3\pi_1-2\tau_1)\rho_i\ge 2\tau_2-3\pi_2$$
$$(3\pi_1-2\tau_1)\frac{a_i}{b_i}\ge 2\tau_2-3\pi_2 \quad (\text{by } \rho_i b_i\le a_i)$$

$$\frac{\frac{a_i}{b_i}\tau_1+\tau_2}{\frac{a_i}{b_i}\pi_1+\pi_2}\le\frac{3}{2}. \quad \left(\text{by } \frac{\pi_1}{\tau_1}\le\frac{3}{2}\right).$$

We assume that searcher P hits the lowest point of segment i of an obstacle. If he hits a point within the segment i of an obstacle, there is an $O(1/\sqrt{d})$ error each time this situation occurs. Note that d is the length of the s–t line and the size of each square is limited to some constant $c(< 1)$. Then we at worst hit $O(d)$ squares. Thus, the total error is $O(\sqrt{d})$.

Next, we need to know the distance between searcher P and the s–t line when the algorithm applies Rule 6 to reach t. Assume that searcher P is above the s–t line and hits the ith segment. Clearly, if the ith segment is labeled down, he becomes closer to the s–t line. Consider the case where the ith segment is labeled up and searcher P goes up. By definition, the times of addition to balance the ith segment is at most $\rho_i + 1$. The total vertical distance above the s–t line is $|CJ_i'| \times (\rho_i + 1) \leq 1$, where $|CJ_i'|$ is the vertical distance away off the s–t line when searcher P hits the ith segment. Since there are $\sqrt{d}$ segments in each obstacle, after applying Rule 6, the distance between searcher P and the s–t line is at most $\sqrt{d}$.

Based on the above discussion, we conclude that the ratio between the distance traveled by searcher P and searcher Q is no greater than

$$\frac{3}{2} + O\left(\frac{1}{\sqrt{d}}\right).$$

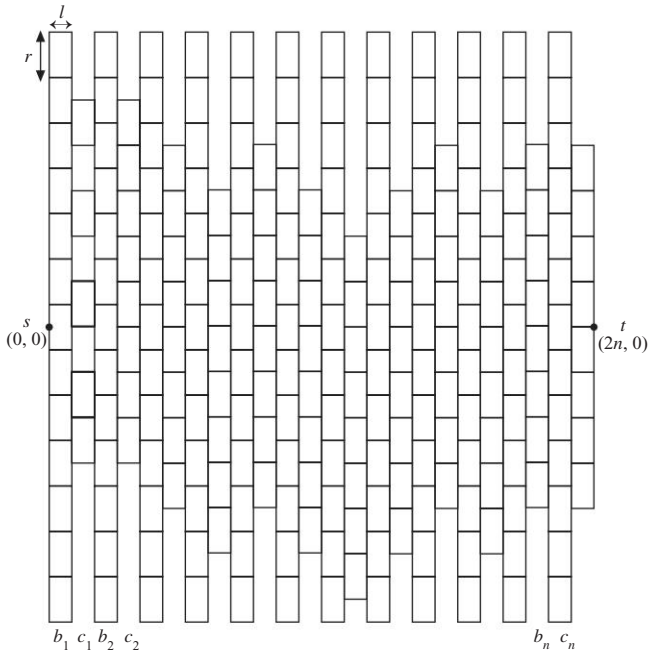
If d is large, the ratio is no greater than $\frac{3}{2}$. Thus, we conclude that the total distance traveled by P is no more than $\frac{3}{2}$ that traveled by the fictitious searcher Q. Thus, our algorithm is (3/2)-competitive.

In the above, we showed that the competitive ratio of our on-line obstacle traversal algorithm is 3/2. Next, we shall show that this is optimal. Assume that the obstacles are rectangles and let r denote the ratio of the length and width of each rectangle. We shall show that it is impossible to have a competitive ratio of any on-line algorithm to solve the obstacle traversal problem which is less than $(r/2 + 1)$. In our case, we assume that all the obstacles are squares and r is equal to 1. Thus, the competitive ratio cannot be less than 3/2 and the on-line algorithm is optimal.

To find a lower bound of the competitive ratios, we must have a special arrangement of the obstacles. Figure 12–22 illustrates this special layout of the

obstacles. Let n be an integer greater than 1. As shown in Figure 12–22, the coordinates of s and t are $(0, 0)$ and $(2n, 0)$, respectively. Note that between obstacles, there is an infinitely small width which separates them and the searcher can squeeze himself to travel between obstacles. Obviously, the searcher goes either horizontally or vertically.

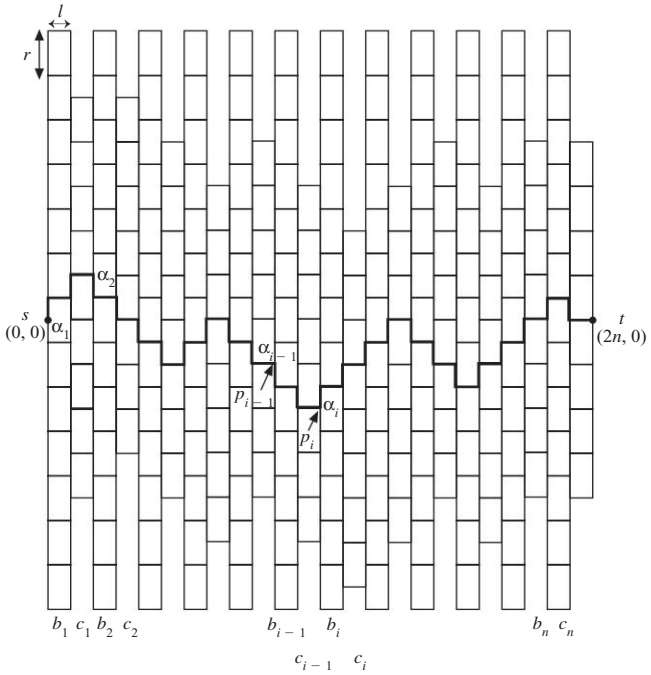
FIGURE 12–22 A special layout.



The obstacles are arranged in columns. There are two kinds of columns, b columns and c columns. Each b column consists of an infinite number of obstacles and there are exactly eight obstacles in each c column. Each b column is next to a c column and each c column is next to a b column. The ith $b(c)$ column is labeled as $b_i(c_i)$.

For $1 \le i \le n$, the locations of obstacles on c_i are arranged according to how the searcher hits b_i. Let the obstacle encountered by searcher P on column b_i be α_i. Let p_i be the point where searcher P reaches obstacle α_i on column b_i. Then the eight obstacles on column c_i are arranged in such a way that α_i is next to their middle. Figure 12–23 shows one possible arrangement of the obstacles where the heavy line indicates the traversal of searcher P.

FIGURE 12–23 The route for the layout in Figure 12–22.



It is clear that any length of the shortest route from p_{i-1} to p_i is at least

$r + 2$.

Thus, the total distance traveled by searcher P is at least

$(r + 2)n$.

For $0 \le m \le \sqrt{n}$, define horizontal line L_m to be

L_m: $y = (8m + 1/2)r$.

Note that L_m does not intersect any obstacle on b_i, and may intersect one obstacle on c_j for any i and j. Since the distance between any two lines L_m and L'_m is at least $8r$, they cannot intersect obstacles which are on the same column c_i, for $1 \le i \le n$. Thus, $L_0, L_1, \ldots, L_{\lfloor\sqrt{n}\rfloor}$ can intersect at most n obstacles on columns $c_1, c_2, \ldots, c_n$. Since there are $\sqrt{n}$ lines and they intersect n obstacles, on average, each line intersects $n/\sqrt{n} = \sqrt{n}$ obstacles. There is at least one

integer $m \leq \sqrt{n}$ such that L_m intersects at most $\sqrt{n}$ obstacles on columns c_1, $c_2, \ldots, c_n$. For any feasible solution, the horizontal distance is at least $2n$. Every time it hits an obstacle, it travels cr for some c because r is the length of each obstacle. Thus, we can construct a feasible route of length at most

$$2n + cr\sqrt{n} \quad \text{for some } c.$$

Finally, we can conclude that the competitive ratio of any on-line algorithm for the obstacle traversal problem is at least

$$\frac{r+n}{2n+nc\sqrt{n}}.$$

If n is large, this ratio becomes at least

$$r/2 + 1.$$

If $r = 1$, this is equal to $\frac{3}{2}$.

12–4 THE ON-LINE BIPARTITE MATCHING PROBLEM SOLVED BY THE COMPENSATION STRATEGY

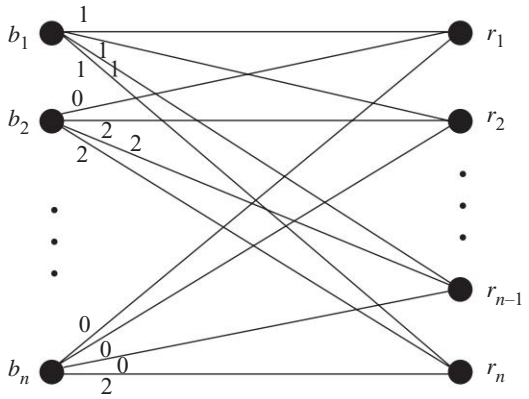
In this section, we consider a bipartite matching problem.We are given a bipartite weighted graph G with vertices bipartition R and B, each of cardinality n. A bipartite matching M is a subset of E where no two edges in the set are incident to one single vertex and each edge in the set is incident to both R and B. The cost of a matching is defined as the total weight of the edges in the matching. The minimum bipartite matching problem is to find a bipartite matching with the minimum cost. In this section, we only consider the minimum bipartite matching problem.

Our on-line version of the bipartite matching problem is described as follows: The vertices in R are all known to us in advance. Then the vertices in B are revealed one by one. After the ith vertex in B arrives, it must be matched with an unmatched vertex in R and this decision cannot be changed later. Our goal is to keep the cost of this on-line matching low. We also put a special constraint on the data: All the weights of edges satisfy the triangle inequality.

Next, we shall first prove a lower bound of the cost resulting from on-line minimum bipartite matching algorithms. We shall prove that the cost of any on-line minimum bipartite matching algorithm cannot be less than $(2n - 1)$ times the cost of an optimal off-line matching.

Let $b_1, \ldots, b_n$ be the n vertices in B. Let r_i, $1 \le i \le n$, denote the matched vertex with b_i when b_i appears. Let the bipartite graph be illustrated by Figure 12–24.

FIGURE 12–24 The bipartite graph for proving a lower bound for on-line minimum bipartite matching algorithms.



As shown in Figure 12–24, this bipartite graph has the following characteristics:

(1) The weight of (b_1, r_i) is 1 for all i.
(2) The weight of (b_i, r_j) is 0 for $i = 2, 3, \ldots, n$ if $j < i$.
(3) The weight of (b_i, r_j) is 2 for $i = 2, 3, \ldots, n$ if $j \ge i$.

For any on-line algorithm with $b_1, b_2, \ldots, b_n$ revealed in order, there exists the total matching cost $1 + 2(n - 1) = 2n - 1$.

An optimal off-line matching for this bipartite graph should be as follows:

r_1 matched with b_2 with cost 0.
r_2 matched with b_3 with cost 0.
$\vdots$
r_n matched with b_1 with cost 1.

Thus, the optimal cost for an off-line algorithm is 1. This means that no on-line bipartite matching algorithm can achieve a competitive ratio less than $2n - 1$.

Let us now illustrate an on-line bipartite matching algorithm. This on-line matching algorithm is based upon an off-line matching. We shall describe our algorithm through an example.

Consider Figure 12–25 where the set of R is shown. Note that none of the nodes are labeled. The weight of the edge between two vertices is their geometric distance.

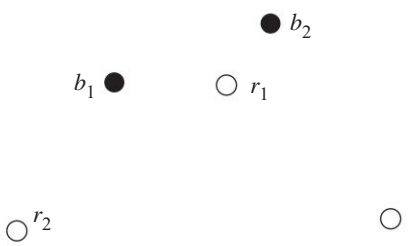
FIGURE 12–25 The set R.

Now, in Figure 12–26, b_1 is revealed. Any optimal off-line matching would find the node in R nearest to b_1. Without losing generality, let this node be r_1. Our on-line matching algorithm matches b_1 with r_1 just as any optimal off-line matching would do.

FIGURE 12–26 The revealing of b_1.

In Figure 12–27, b_2 is revealed. Any optimal off-line matching will now match b_2 with r_1 and match b_1 with a new node in R. Without losing generality, we may denote this node as r_2. The question is, which action will our on-line matching algorithm do?

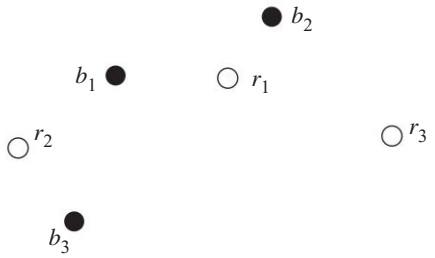
FIGURE 12–27 The revealing of b_2.



Since our on-line matching algorithm has already matched b_1 with r_1, we cannot change this fact. To compensate, since r_2 is the new node matched, we simply match b_2 with r_2.

Finally, b_3 is revealed as shown in Figure 12–28. An optimal off-line matching would match b_1 with r_1, b_2 with r_3, the new node in R added, and b_3 with r_2.

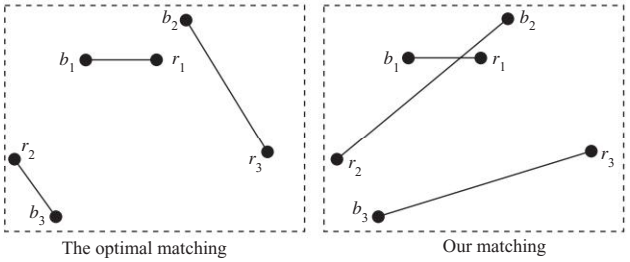
FIGURE 12–28 The revealing of b_3.



Our on-line bipartite matching algorithm notices that r_3 is a newly added point and it matches b_3 with r_3.

In Figure 12–29, we compare the optimum off-line matching with our on-line matching.

FIGURE 12–29 The comparison of on-line and off-line matchings.



We now formally define our on-line algorithm. Let us assume that before b_i is revealed, $b_1, b_2, \ldots, b_{i-1}$ have been already revealed in this order and without losing generality, they are matched with $r_1, r_2, \ldots, r_{i-1}$ respectively by our on-line matching M_{i-1}. When b_i is revealed, consider the matching M_i', which satisfies the following conditions:

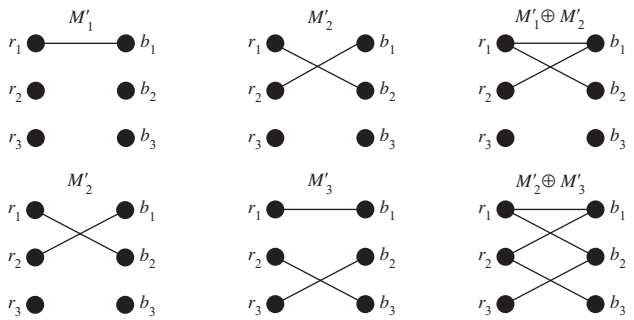
(1) M'_i is an optimal bipartite matching between $\{b_1, b_2, \ldots, b_i\}$ and $r_1, r_2, \ldots, r_i$.
(2) Among all optimal bipartite matchings between $\{b_1, b_2, \ldots, b_i\}$ and $r_1, r_2, \ldots, r_i$, $|M'_i - M'_{i-1}|$ is the smallest.

Let R_i denote the set of elements of R in M'_i. We can prove that R_i only adds one element to R_{i-1}. Without losing generality, we may assume that r_i is added after b_i is revealed. That is, $R_i = \{r_1, r_2, \ldots, r_{i-1}, r_i\}$. Since in M_{i-1}, r_j is matched with b_j for $j = 1, 2, \ldots, i-1$, to compensate, we match r_i with b_i in M_i.

Initially, $M_1 = M'_1$.

We shall now discuss an interesting property of the on-line algorithm. Note that for the on-line algorithm, there is a sequence of matchings $M'_1, M'_2, \ldots, M'_n$. We now consider $M'_i \oplus M'_{i+1}$, in which each edge is in exactly one of M'_i and M'_{i+1}. For the example, the sets are shown in Figure 12–30.

FIGURE 12–30 $M'_i \oplus M'_{i+1}$.



Let E_i and E_j be two sets of edges. An alternating path (cycle) of $E_i \oplus E_j$ is a simple path (cycle) with edges alternating in E_i and E_j. A maximal alternating path (cycle) is an alternating path (cycle) which is not a subpath (subcycle) of any alternating path (cycle). For $M'_1 \oplus M'_2$, there is exactly one maximal alternating path and this path has b_2 as its ending point. This path is

$$r_2 \rightarrow b_1 \rightarrow r_1 \rightarrow b_2.$$

Similarly, in $M'_2 \oplus M'_3$, there is exactly one maximal alternating path which ends in b_3, namely

$$r_3 \rightarrow b_2 \rightarrow r_1 \rightarrow b_1 \rightarrow r_2 \rightarrow b_3.$$

We now try to prove that in $M'_{i-1} \oplus M'_i$ for $i = 2, \ldots, n$, there is exactly one maximal alternating path and this path has b_i as its ending point.

Let H be $M'_{i-1} \oplus M'_i$. We first note the following:

(1) The degree of b_i in H is 1 because b_i is a newly revealed vertex and it must be matched with some vertex in R.
(2) For b_j, $1 \le j \le i-1$, we shall see that the degree of b_j in H is either 0 or 2. This can be seen as follows: If b_j is matched with the same element in R for both M'_{i-1} and M'_i, then the degree of b_j in H is 0; otherwise, the degree of b_j in H is 2 because one edge in M'_{i-1} and one edge in M'_i are incident in b_j.
(3) As for any r_j, the degree of r_j can be 0, 1 or 2. This can be proved because there are only two possible cases:

Case 3.1: r_j is not matched with any vertex in both M'_{i-1} and M'_i. In this case, the degree of r_j in H is 0.

Case 3.2: r_j is matched with some vertex. In this case, it can be shown that the degree of r_j in H is 1 if it is matched with a vertex in either M'_{i-1} or M'_i, but not both, the degree of r_j in H is 0 if it is matched with the same vertex in both M'_{i-1} and M'_i and the degree of r_j is 2 if it is matched with two different vertices in M'_{i-1} and M'_i.

It can be easily seen that in H, each connected component must be a maximal alternating path or cycle. Since the degree of b_i is 1, there must be an alternating path which has b_i as its ending point. Let P denote this path. We shall show that there is no other connected component in H. In other words, P is the only connected component in H.

We first assume that H contains a maximal alternating path L where $L \neq P$. It is clear that b_i is not on L because $L \neq P$, and P is the unique maximal alternating path which has b_i as an endpoint. Since each vertex of $B_i - \{b_i\}$ has degree 0 or 2 and each vertex of R has degree 0 or 1 or 2, the two endpoints of L must be in R and the number of edges of L must be even. In addition, the number of vertices of L which are in R must be larger than that of L which are in $B_i - \{b_i\}$ by 1.Without losing generality, we assume that there are $2m$ edges on L, where $1 \le m \le i-1$, and the sequence of vertices on path L is $r_1, b_1, r_2, b_2, \ldots, r_m, b_m, r_{m+1}$, where r_1 and r_{m+1} are endpoints of L, $r_1, r_2, \ldots$, and r_{m+1}

are in R, and $b_1, \ldots,$ and b_m are in $B_i - \{b_i\}$. Let $A_i(A_{i-1})$ denote the set of edges on L which belong to $M'_i(M'_{i-1})$. Without losing generality, we assume $A_i = \{(b_1, r_1), (b_2, r_2), \ldots, (b_m, r_m)\}$ and $A_{i-1} = \{(b_1, r_2), (b_2, r_3), \ldots, (b_m, r_{m+1})\}$. Note that in H, the vertex of R with degree 1 is matched exactly in either M'_i or M'_{i-1}. Thus, the endpoint r_1 is matched exactly in M'_i and the endpoint r_{m+1} is matched exactly in M'_{i-1}. In other words, $M'_i(M'_{i-1})$ is the set of edges of a matching of $B_i = \{b_1, b_2, \ldots, b_i\}$ ($B_{i-1} = \{b_1, b_2, \ldots, b_{i-1}\}$) with a subset of $R + \{r_{m+1}\}(R - \{r_1\})$.

Since A_i is the set of edges of a matching of $\{b_1, \ldots, b_m\}$ with $\{r_1, \ldots, r_m\}$, $M'_i - A_i$ is the set of edges of a matching of $B_i - \{b_1, \ldots, b_m\}$ with a subset of $R - \{r_1, r_2, \ldots, r_m, r_{m+1}\}$. Since A_{i-1} is the set of edges of a matching of $\{b_1, \ldots, b_m\}$ with $\{r_2, \ldots, r_{m+1}\}$, $(M'_i - A_i) \bigcup A_{i-1}$ is the set of edges of a matching of b_i with a set of $R - \{r_1\}$. Similarly, it can be proved that $(M'_{i-1} - A_{i-1}) \bigcup A_i$ is the set of edges of a matching of B_{i-1} with a subset of $R - \{r_{m+1}\}$. Let $W(S)$ denote the total weight of an edge set S. There are three possible cases for L:

- $W(A_i) > W(A_{i-1})$.
  In this case, there exists a matching $M''_i = (M'_i - A_i) \bigcup A_{i-1}$ such that $W(M''_i) < W(M'_i)$. This contradicts the minimality of M'_i.

- $W(A_i) < W(A_{i-1})$.
  In this case, there exists a matching $M''_{i-1} = (M'_{i-1} - A_{i-1}) \bigcup A_i$ such that $W(M''_{i-1}) < W(M'_{i-1})$. This contradicts the minimality of M'_{i-1}.

- $W(A_i) = W(A_{i-1})$.
  In this case, there exists a matching $M''_i = (M'_i - A_i) \bigcup A_{i-1}$ such that $|M''_i - M'_{i-1}| < |M'_i - M'_{i-1}|$. This contradicts the fact that M'_i was selected to minimize the number of edges in $M'_i - M'_{i-1}$.

Hence, there is no maximal alternating path except P in H.

We conclude that $M'_i \oplus M'_{i-1}$ consists of exactly one maximal alternating path P which has b_i as an endpoint.

Since H consists of exactly one maximal alternating path P and the degree of b_j is 0 or 2, where $1 \le j \le i - 1$, there is a vertex r' of R with degree 1 as the other endpoint of P and each vertex of $R - \{r'\}$ has degree 0 or 2.

Besides, the number of edges of P must be odd. Thus, both the edge linked to the endpoint b_i and the edge linked to the endpoint r' must come from the

same edge set. Since the edge linked to b_i is an edge of M'_i, the edge linked to r' is also an edge of M'_i. Note that in H, the vertex of R with degree 1 is matched exactly in one of M'_i and M'_{i-1}. Thus, r' is matched exactly in M'_i. In H, a vertex of $R - \{r'\}$ with degree 0 is matched with the same vertex of b_i or is matched neither in M'_i nor in M'_{i-1}, and a vertex of $R - \{r'\}$ with degree 2 is matched with different vertices of b_i in M'_i and M'_{i-1}. That is, a vertex of $R - \{r'\}$ is either matched or not matched in both M'_i and M'_{i-1}. So r' is the only vertex of $R_i - R_{i-1}$. We conclude that for $1 < i \leq n$, $|R_i - R_{i-1}| = 1$.

The above discussion is very important because it lays the foundation of the compensation strategy. The on-line algorithm introduced in this section works only if after a new element in B is revealed, compared with the previous optimal matching, the new optimal matching adds only one new element of R.

We have proved the validity of this approach. In the following, we discuss the performance of the on-line algorithm. We shall show that the algorithm is $(2n - 1)$-competitive. That is, let $C(M_n)$ and $C(M'_n)$ denote the costs of matchings M_n and M'_n, respectively. Then

$$C(M_n) \leq (2n - 1)C(M'_n).$$

We prove this by induction. For $i = 1$, it is obvious that $C(M_1) = C(M'_1)$. Therefore,

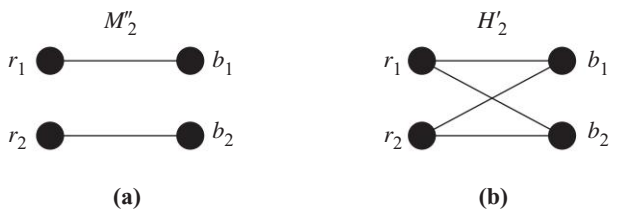
$$C(M_i) \leq (2i - 1)C(M'_i).$$

holds for $i = 1$.

Assume that the above formula holds for $i = 1$ and in M_i, b_i is matched with r_j. Let M''_i be $M'_{i-1} \bigcup (b_i, r_j)$ and H'_i be $M'_i \oplus M''_i$. Since matchings M'_i and M''_i are the matchings of b_i with the subset R_i of R, each vertex in H'_i has degree 0 or 2. Before going on with our proof, let us show some examples.

For the case illustrated in Figure 12–30, $M''_2 = M'_1 \bigcup (b_2, r_1)$ is now shown in Figure 12–31(a) and $H'_2 = M'_2 \oplus M''_2$ is shown in Figure 12–31(b).

FIGURE 12–31 M''_2 and H'_i for the case in Figure 12–30.



Since each vertex in H_i' has degree 0 or 2, there are two possible cases for H.

Case 1: All of the vertices in H_i' have degree 0.
In this case, M_i' and M_i'' are exactly the same. Thus, (b_i, r_j) must be in M_i'. Let $d(b_i, r_j)$ denote the distance between b_i and r_j. Then

$$d(b_i, r_j) \le C(M_i).$$

Since $C(M_{i-1}')$ is non-negative, we have

$$d(b_i, r_j) \le C(M_i') + C(M_{i-1}').$$

Case 2: There is an alternating cycle in H. Again, we can prove that

$$d(b_i, r_j) \le C(M_i') + C(M_{i-1}')$$

in this case. The proof is omitted here.

Since $C(M_i) - C(M_{i-1}) = d(b_i, r_j)$, we have

$$\begin{aligned}
&C(M_i) - C(M_{i-1}) = d(b_i, r_j) \le C(M_{i-1}') + C(M_i').\\
&C(M_i) \le C(M_{i-1}) + C(M_{i-1}') + C(M_i')\\
&\qquad \le (2(i-1)-1)C(M_{i-1}') + C(M_{i-1}') + C(M_i')\\
&\qquad\quad \text{(by inductive hypothesis)}\\
&\qquad \le (2(i-1)-1)C(M_i') + 2C(M_i') \quad (\text{by } C(M_{i-1}') \le C(M_i'))\\
&\qquad \le (2i-1)C(M_i').
\end{aligned}$$

We proved that our algorithm is $(2n - 1)$-competitive. The question is: Is our algorithm optimal? At the beginning of this section, we established the fact that no on-line minimum bipartite matching algorithm can achieve a competitive ratio less than $2n - 1$. Thus, this on-line algorithm based on the compensation strategy is optimal.

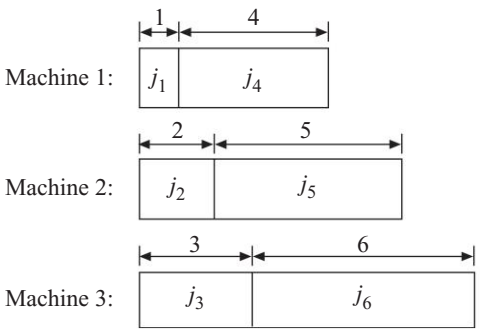
12–5 THE ON-LINE *m*-MACHINE PROBLEM SOLVED BY THE MODERATION STRATEGY

In this section, we shall describe an algorithm based on the moderation strategy for the on-line m-machine scheduling problem. The on-line m-machine scheduling problem is defined as follows: We are given m identical machines and

jobs are arriving one by one. The execution time for the ith job is known when the ith job arrives. As soon as a job arrives, it must be assigned immediately to one of the m machines. The goal is to schedule the jobs non-preemptively on the m machines so as to minimize the makespan, the completion time of the last job.

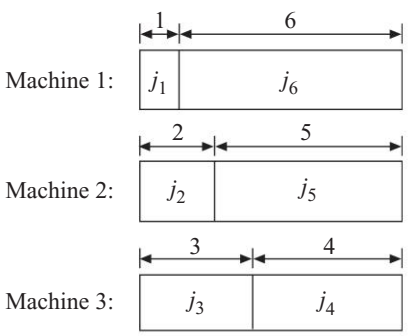
A straightforward strategy is the greedy method, which can be illustrated by the following example. Consider the case where there are six jobs, denoted as j_1, $j_2, \ldots, j_6$ and their execution times are $1, 2, \ldots, 6$ respectively. The greedy method assigns the arriving job to the machine with the least total processing time. The final assignment is shown in Figure 12–32.

FIGURE 12–32 An example for the m-machine scheduling problem.



As shown in Figure 12–32, the longest completion time, as often denoted as the makespan, is 9 for this case. This makespan can be shortened if we do not adopt the greedy algorithm. The scheduling shown in Figure 12–33 produces a makespan which is 7.

FIGURE 12–33 A better scheduling for the example in Figure 12–32.



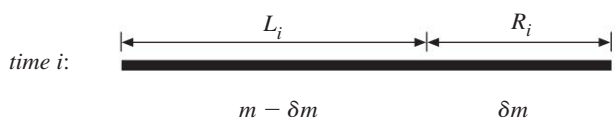
The algorithm based upon the greedy method is called the List algorithm. It is shown that the List algorithm is $\left(2-\frac{1}{m}\right)$-competitive. In the following, we shall describe an algorithm which uses the the moderation strategy. The algorithm is $\left(2-\frac{1}{70}\right)$-competitive for $m \geq 70$. Thus, this algorithm has better performance than the List algorithm when $m \geq 70$.

Basically, the moderation strategy does not assign an arriving job to the machine with the least processing time. Instead, it tries to assign the incoming job to a machine whose processing time is sort of in the middle, neither too short, nor too long. This is why this strategy is called the moderation strategy.

Let us now define some terms. First, let a_i denote the execution time of the ith arriving job, where $1 \leq i \leq n$. Assume $m \geq 70$. Let $\varepsilon = \frac{1}{70}$. Let $\delta \in [0.445 - 1/(2m), 0.445 + 1/(2m)]$ and δm be integral. For instance, let $m = 80$. Then we may let $\delta = \frac{36}{80}$ which satisfies the definition of the range of δ, and $\delta m = 36$ is an integer.

The height of a machine is defined to be the sum of the lengths of the jobs already assigned to it. At any time, the algorithm will maintain a sequence of machines sorted into non-decreasing order by their current heights. At time i, that is, when i jobs have been scheduled, we divide the sequence into two subsequences: L_i and R_i. R_i is the subsequence of the first δm machines on the list, and L_i is the subsequence of the last $m - \delta m$ machines, shown in Figure 12–34.

FIGURE 12–34 L_i and R_i.



Let the sequences of heights of R_i and L_i be denoted as Rh_i and Lh_i, respectively. Let A_i and M_i denote the average and minimum heights over all m machines, respectively, at time i. Let $A(P)$ and $M(P)$ denote the average and minimum, respectively, of the heights in P, where P is a sequence of heights.

The on-line algorithm for the on-line m-machine problem based on the moderation strategy is described in the following:

When job $i + 1$ arrives, place job $i + 1$ on the first machine in L_i, if

$$M(Lh_i) + a_{i+1} \leq (2 - \varepsilon)A(Rh_i);$$

otherwise, place job $i + 1$ on the first machine on the list R_i, the one with least height overall. If necessary, permute the list of machines so that height remains non-decreasing.

Next, we shall prove that the above on-line algorithm is $(2 - \varepsilon)$-competitive. The proof is rather complicated and we cannot present the entire proof. Only the major parts of the proof will be presented.

Let ON_t and OPT_t denote the makespans of the on-line and optimal algorithms at time t, respectively. Assume that the algorithm is not $(2 - \varepsilon)$-competitive.Thus, there exists a t such that

$$ON_t \leq (2 - \varepsilon)OPT_t \quad \text{and} \quad ON_{t+1} > (2 - \varepsilon)OPT_{t+1}.$$

Let us consider $M(Lh_t)$ and $A(Rh_t)$. Assume that $M(Lh_t) + a_{t+1} \leq (2 - \varepsilon)A(Rh_t)$. Then the on-line algorithm places a_{t+1} on the shortest machine of L_i. Thus,

$$\begin{aligned} ON_{t+1} &= M(Lh_t) + a_{t+1} \\ &\leq (2 - \varepsilon)A(Rh_t) \\ &\leq (2 - \varepsilon)A_t \\ &\leq (2 - \varepsilon)A_{t+1}. \end{aligned} \tag{12–3}$$

Since OPT_i is an optimal makespan for jobs $a_1, a_2, \ldots, a_i$, we have $m \times OPT_i \geq a_1 + a_2 + \cdots + a_i = m \times A_i$. Hence,

$$OPT_i \geq A_i. \tag{12–4}$$

Substituting (12–4) into (12–3), we obtain

$$ON_{t+1} \leq (2 - \varepsilon)OPT_{t+1}. \tag{12–5}$$

This contradicts with the assumptions. Thus, $M(Lh_t) + a_{t+1} < (2 - \varepsilon)A(Rh_t)$ and the on-line algorithm places a_{t+1} on the machine with least height overall. Hence,

$$ON_{t+1} = M_t + a_{t+1}. \tag{12–6}$$

We now prove that $M_t > (1 - \varepsilon)A_t$. Suppose $M_t \le (1 - \varepsilon)A_t$. Then

$$\begin{aligned} ON_{t+1} &= M_t + a_{t+1} \quad \text{(by Equation (12–6))} \\ &\le (1 - \varepsilon)A_t + a_{t+1} \\ &\le (1 - \varepsilon)A_{t+1} + a_{t+1}. \end{aligned}$$

Because $A_{t+1} \le OPT_{t+1}$ and $a_{t+1} \le OPT_{t+1}$

$$ON_{t+1} \le (2 - \varepsilon)OPT_{t+1}.$$

This is impossible as we have assumed that $ON_{t+1} > (2 - \varepsilon)OPT_{t+1}$. Therefore, we conclude that

$$M_t > (1 - \varepsilon)A_t. \tag{12–7}$$

Inequality (12–7) indicates that the height of the shortest machine at time t has a height longer than $(1 - \varepsilon)A_t$. In fact, we can prove an even stronger statement: Every machine at time t contains a job of length at least $\frac{1}{2(1-\varepsilon)} A_t \ge \frac{1}{2(1-\varepsilon)} M_t$. We shall present the proof of this statement later. Meanwhile, let us assume that this is true. Then we shall quickly establish the fact that the algorithm is $(2 - \varepsilon)$-competitive.

There are two possible cases for a_{t+1}.

$a_{t+1} \le (1 - \varepsilon)M_t$.
Then

$$\begin{aligned} ON_{t+1} &= M_t + a_{t+1} \quad \text{(by Equation (12–6))} \\ &\le (2 - \varepsilon)M_t \\ &\le (2 - \varepsilon)M_{t+1} \\ &\le (2 - \varepsilon)OPT_{t+1}. \end{aligned}$$

This contradicts $ON_{t+1} > (2 - \varepsilon)OPT_{t+1}$.

$a_{t+1} > (1 - \varepsilon)M_t$.
Then

$$a_{t+1} > (1 - \varepsilon)M_t$$
$$\geq \frac{1}{2(1-\varepsilon)} M_t. \quad (\text{by } (1 - \varepsilon) \geq \frac{1}{2(1-\varepsilon)})$$

Since every machine at time t contains a job of length at least $\frac{1}{2(1-\varepsilon)} M_t$ and $a_{t+1} \geq \frac{1}{2(1-\varepsilon)} M_t$, there are at least $m + 1$ jobs of length at least $\frac{1}{2(1-\varepsilon)} M_t$, two of which must be on the same machine. So

$$OPT_{t+1} \geq \max\left\{a_{t+1}, \frac{1}{2(1-\varepsilon)} M_t\right\}.$$

Thus,

$$ON_{t+1} = M_t + a_{t+1} \quad (\text{by Equation (12–6)})$$
$$\leq (1 - \varepsilon)OPT_{t+1} + OPT_{t+1}$$
$$(\text{by } a_{t+1} \leq OPT_t + 1 \text{ and } \frac{1}{2(1-\varepsilon)} M_t \leq OPT_{t+1})$$
$$= (2 - \varepsilon)OPT_{t+1}.$$

This also contradicts $ON_{t+1} > (2 - \varepsilon)OPT_{t+1}$.

In either case, we have the contradiction. Therefore, we conclude that the on-line algorithm is $(2 - \varepsilon)$-competitive.

We now prove the strong statement which claims that at time t, every machine contains a job of length at least $\frac{1}{2(1-\varepsilon)} A_t$. To prove this statement, note that we have proved that $M_t > (1 - \varepsilon)A_t$. Because of this, there must exist time r such that $r < t$ and the machines in R_r have heights at most $(1 - \varepsilon)A_t$ at time r.

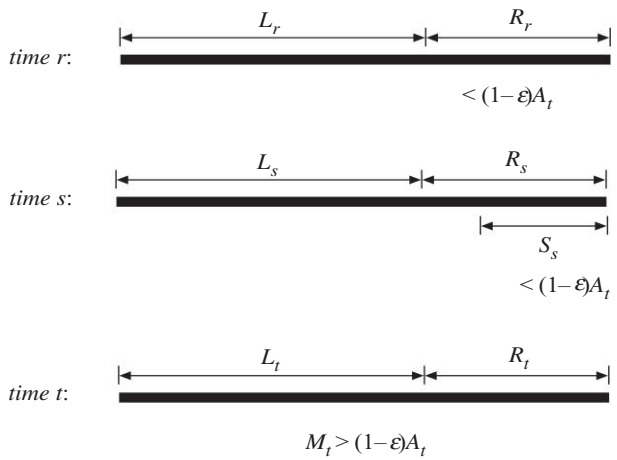
Let $\tau \in [0.14 - 1/(2\delta m), 0.14 + 1/(2\delta m)]$ and m be integral. Let S_i and Sh_i denote the sequence of m machines of smallest height at time i and the sequence of their heights, respectively. Let s denote the time that only the machines in S_s have height at most $(1 - \varepsilon)A_t$. It can be easily seen that $r < s < t$.

Let us summarize the three critical times as follows:

(1) At time r, the machines in R_r have heights at most $(1 - \varepsilon)A_t$.
(2) At time s, the machines in S_s have heights at most $(1 - \varepsilon)A_t$.
(3) At time t, $M_t > (1 - \varepsilon)A_t$.

We illustrate these times in Figure 12–35.

FIGURE 12–35 The illustration for times r, s and t.



We further note that $S_s \in R_r$. The following claims can be made:

Claim 1. At time t, every machine in S_s contains a job of size at least $\frac{1}{2(1-\varepsilon)}A_t$.

Claim 2. At time t, every machine in R_r, other than those in S_s, contains a job of size at least $\frac{1}{2(1-\varepsilon)}A_t$.

Claim 3. At time t, every machine in L_r contains a job of at least $\frac{1}{2(1-\varepsilon)}A_t$.

It is obvious that once we have proved the above three claims, we have proved that at time t, every machine contains a job with length at least $\frac{1}{2(1-\varepsilon)}A_t$. In the following, we shall prove Claim 1 because the other two can be proved similarly.

Proof of Claim 1. Since every machine in S_s has height at most $(1-\varepsilon)A_t$ at time s, and $M_t > (1-\varepsilon)A_t$ (inequality (12–7)), every machine in S_s must receive at least one more job during the period from time s to time t. Let p be a machine in S_s. Let job j_{i+1} be the first job machine p receives after time s, where $s < i \le t-1$. According to our algorithm, this job is always either put on the first machine in L_{i+1} or the shortest machine overall. Since $p \in S_s$, it cannot belong to L_{i+1}. Therefore, p must be the shortest machine overall at time $i+1$. Thus, we have

$$M(Lh_i) + a_{i+1} > (2-\varepsilon)A(Rh_i)$$

so that

$$a_{i+1} > (2-\varepsilon)A(Rh_i) - M(Lh_i). \tag{12–8}$$

We also have

$$\begin{aligned} A_t &\ge A_i \\ &= (1-\delta)A(Lh_i) + \delta A(Rh_i) \\ &\ge (1-\delta)M(Lh_i) + \delta A(Rh_i). \end{aligned}$$

This implies that

$$M(Lh_i) \le \frac{A_t - \delta A(Rh_i)}{1-\delta}. \tag{12–9}$$

Besides,

$$\begin{aligned} A(Rh_i) &\ge A(Rh_s) \\ &= (1-\tau)(\text{the average of heights in } R_s - S_s) + \tau A(Sh_s) \\ &> (1-\tau)[(1-\varepsilon)A_t] + \tau A(Sh_s). \qquad \text{(12–10)} \\ &\quad \text{(because the machine in } R_s - S_s \text{ has height at least } (1-\varepsilon)A_t) \end{aligned}$$

Then we obtain

$$
\begin{aligned}
a_{i+1} &> (2-\varepsilon)A(Rh_i) - M(Lh_i) \quad \text{(by inequality (12–8))} \\
&> (2-\varepsilon)A(Rh_i) - \left[\frac{A_t - \delta A(Rh_i)}{1-\delta}\right] \quad \text{(by inequality (12–9))} \\
&= \left(2-\varepsilon+\frac{\delta}{1-\delta}\right)A(Rh_i) - \frac{\delta}{1-\delta}A_t \\
&= \left(2-\varepsilon+\frac{\delta}{1-\delta}\right)[(1-\tau)[(1-\varepsilon)A_t] + \tau A(Sh_s)] - \frac{\delta}{1-\delta}A_t \\
&\quad \text{(by inequality (12–10))} \\
&= \left\{\left[2-\varepsilon+\frac{\delta}{1-\delta}\right](1-\tau)(1-\varepsilon) - \frac{\delta}{1-\delta}\right\}A_t \\
&\quad + \left\{\left[2-\varepsilon+\frac{\delta}{1-\delta}\right]\tau\right\}A(Sh_s) \\
&> \left\{\left[2-\varepsilon+\frac{\delta}{1-\delta}\right](1-\tau)(1-\varepsilon) - \frac{\delta}{1-\delta}\right\}A_t.
\end{aligned}
$$

Since $\dfrac{1}{2(1-\varepsilon)} \approx 0.51$ and

$$
\begin{aligned}
&\left(2-\varepsilon+\frac{\delta}{1-\delta}\right)(1-\tau)(1-\varepsilon) - \frac{\delta}{1-\delta} \\
&\quad \approx \left(2-\frac{1}{70}+\frac{0.455}{1-0.455}\right)(1-0.14)\left(1-\frac{1}{70}\right) - \frac{0.455}{1-0.455} \\
&\quad = 0.56,
\end{aligned}
$$

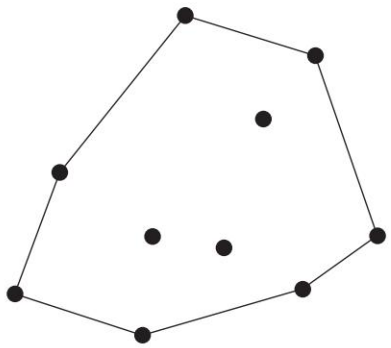
we have for $s < i \le t-1$,

$$
a_{i+1} > \frac{1}{2(1-\varepsilon)}A_t.
$$

12–6 ON-LINE ALGORITHMS FOR THREE COMPUTATIONAL GEOMETRY PROBLEMS BASED ON THE ELIMINATION STRATEGY

In this section, we shall introduce on-line algorithms for three computational geometry problems. They are the convex hull problem, the farthest pair problem and the 1-center problem. The farthest pair problem is as follows: Find a pair of points for a set S of points in the plane such that the distance between them is the largest one. All our on-line algorithms will give approximate solutions. We shall show that all these approximate solutions are close to precise ones. For these three problems, we shall use the parallel lines containment approach. Consider Figure 12–36, which contains a convex hull.

FIGURE 12–36 A convex hull.



Now, imagine that a new point is added. If this point is inside the convex hull, we do not need to take any action. This means that we may only remember the boundary of this convex hull. We of course must change the boundary of this convex hull if a newly arrived point is outside the boundary.

We use a set of parallel lines to form the boundary of our input points. For instance, in Figure 12–37, there are four pairs of parallel lines, namely (l_1, l'_1), (l_2, l'_2), (l_3, l'_3) and (l_4, l'_4), forming a boundary of input points.

Whenever a point arrives, we check whether this point is outside the boundary. If it is, then an action is taken; otherwise, let l_i and l'_i be a pair of parallel lines which do not enclose p and without losing generality, let l_i be closer to p than l'_i. Then we move l_i, without changing its slope, to touch the point p. Consider Figure 12–38. Since p is not enclosed by l_3 and l'_3, we move l_3 to touch p.

FIGURE 12–37 A boundary formed by four pairs of parallel lines.

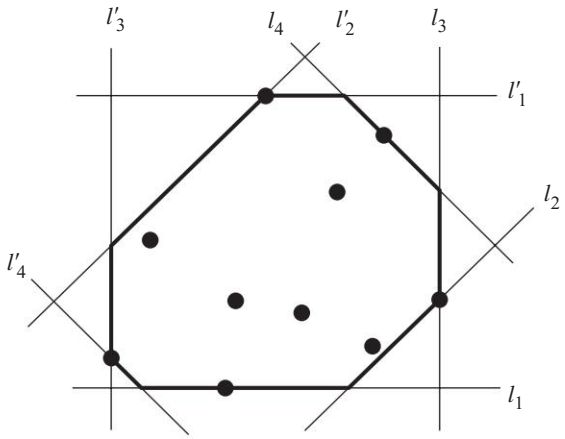
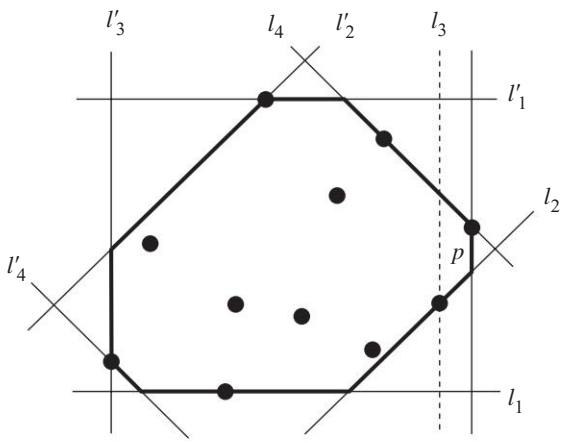


FIGURE 12–38 Line movements after receiving a new input point.



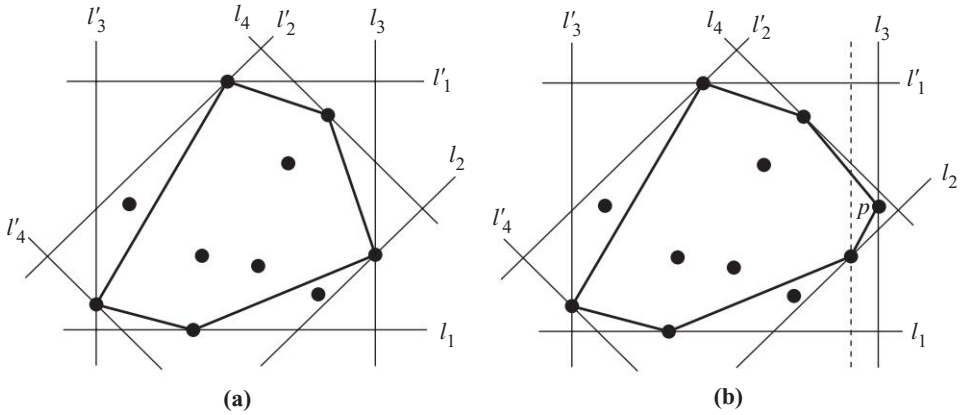
In general, let there be m pairs of parallel lines forming the boundary. The slopes of these m pairs of parallel lines are 0, $\tan(\pi/m)$, $\tan(2\pi/m)$, … , $\tan((m-1)\pi/m)$, respectively. These pairs of parallel lines must satisfy the following conditions:

Rule 1: Every input point is inside of every pair of parallel lines.

Rule 2: Each pair of parallel lines must be as close as possible.

An on-line algorithm for constructing an approximate convex hull is now presented. Since each line is associated with at least one input point, we can connect these points in the clockwise fashion to form a convex hull which serves as our approximate convex hull. Consider Figure 12–39. The original approximate convex hull is shown in Figure 12–39(a). After a new point p is added, the new approximate convex hull is shown in Figure 12–39(b). Note that a point may fall outside the convex polygon. This is why our algorithm is an approximation algorithm.

FIGURE 12–39 The approximate convex hull.



An on-line approximation convex hull algorithm, Algorithm A, is presented below.

Algorithm 12–1 □ Algorithm A for computing on-line convex hulls

Input: A sequence of points $p_1, p_2, \ldots$, and the number m of pairs of parallel lines being used.

Output: A sequence of approximate convex hulls $a_1, a_2, \ldots$, where a_i is an on-line approximate convex hull covering the points $p_1, p_2, \ldots, p_i$.

Initialization: Construct m pairs of parallel lines with slopes 0, $\tan(\pi/m)$, $\tan(2\pi/m)$, $\ldots$, $\tan((m-1)\pi/m)$, respectively and locate all lines so that they all intersect at the first input point p_1. Set $i = 1$, i.e. the current input point is p_1.

Step 1: For each of the m pairs of parallel lines, if point p_i is between them, nothing is changed; otherwise, move the line nearest to point p_i, without changing its slope, to touch p_i and associate p_i with this line.

Step 2: Construct an approximate convex hull by connecting the $2m$ points with respect to each line in the clockwise fashion and denote this approximate convex hull as a_i.

Step 3: If no other point will be input, then stop; otherwise, set $i = i + 1$ and receive the next input point as p_i. Go to Step 1.

The time complexity of Algorithm A is obviously $O(mn)$ where m is the number of pairs of parallel lines being used and n is the number of input points. Since m is fixed, the algorithm is $O(n)$. In the following, we will discuss the error rate of our approximate convex hulls. There are three polygons which are of interest to our algorithm:

(1) Polygon E: the convex enclosing polygon formed by the $2m$ lines. All input points are inside this polygon.
(2) Polygon C: the convex hull of the input points.
(3) Polygon A: the approximate convex hull produced by Algorithm A.

Let $L(P)$ denote the total side length of polygon P. Then it is obvious that

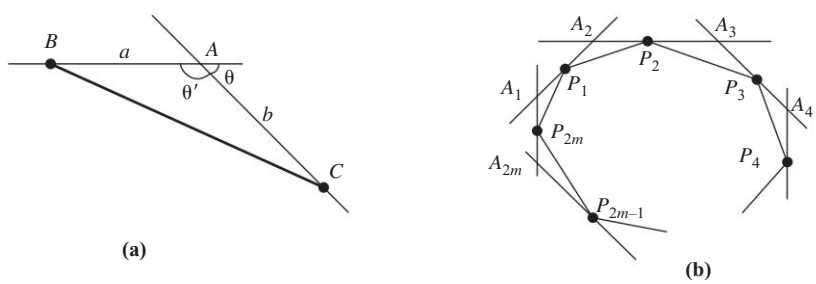
$$L(E) \geq L(C) \geq L(A).$$

The error rate $Err(A)$ for our on-line approximate convex hull A is

$$Err(A) = \frac{L(C) - L(A)}{L(C)}.$$

Consider Figure 12–40(a).

FIGURE 12–40 Estimation of the $Err(A)$.



We have the following:

$$\left(\frac{\overline{AB}+\overline{AC}}{\overline{BC}}\right)^2 = \frac{(a+b)^2}{a^2+b^2-2ab\cdot\cos\theta'}$$

$$= \frac{a^2+b^2+2ab}{a^2+b^2+2ab\cdot\cos\theta}$$

$$= \frac{(a^2+b^2)/2ab+1}{(a^2+b^2)/2ab+\cos\theta}$$

$$\leq \frac{1+1}{1+\cos\theta}$$

$$= \sec^2\frac{\theta}{2}.$$

Hence,

$$\overline{AB}+\overline{AC} \leq \sec\frac{\theta}{2}\cdot\overline{BC}.$$

Consider Figure 12–40(b).

$$L(E) = \overline{P_{2m}A_1}+\overline{A_1P_1}+\overline{P_1A_2}+\cdots+\overline{A_{2m}P_{2m}}$$

$$\leq \sec\frac{\pi}{2m}\cdot(\overline{P_{2m}P_1}+\overline{P_1P_2}+\overline{P_2P_3}+\cdots+\overline{P_{2m-1}P_{2m}})$$

$$= \sec\frac{\pi}{2m}\cdot L(A).$$

We have

$$Err(A) = \frac{L(C)-L(A)}{L(C)} \leq \frac{L(E)-L(A)}{L(A)} \leq \sec\frac{\pi}{2m} - 1.$$

The above equation indicates that the more lines used, the lower the error rate, as can be expected. Table 12–1 shows how $Err(A)$ decreases as m increases.

TABLE 12–1 The upper bound of the error rate.

m	$\sec \frac{\pi}{2m}$	Upper bound of $Err(A)$
2	1.4142	0.4142
3	1.1547	0.1547
4	1.0824	0.0824
5	1.0515	0.0515
10	1.0125	0.0125
20	1.0031	0.0031

The on-line approximation farthest pair algorithm is almost the same as the on-line approximation convex hull algorithm presented in the previous section. Among all pairs of parallel lines, we always find the pair such that the distance between them is the longest one. The two points associated with this pair of parallel lines are used as the on-line approximate farthest pair output by this algorithm.

Thus, our on-line algorithm for the farthest pair problem is almost exactly the same as Algorithm A, except in Step 2. We use now the following:

Step 2: Find the pair of parallel lines whose distance is longest. Set the two points associated with the above pair of parallel lines as the on-line approximate farthest pair A_i. Go to Step 3.

Consider Figure 12–41. Let p_1 and p_2 be the precise farthest pair. Let d_i be the distance between the ith pair of parallel lines. Let $d_{\max}$ be the longest d_i for all i. Consider Figure 12–42. Let N_i be the direction perpendicular to the ith pair of lines. Let θ_i be the sharp angle between the farthest pair $\overline{P_1P_2}$ and N_i. Notice that $0 \le \theta_i \le \frac{\pi}{2}$. Let o_1 and o_2 be the two points associated with the pair of parallel lines whose distance is $d_{\max}$.

FIGURE 12–41 An example of the precise and approximate farthest pairs.

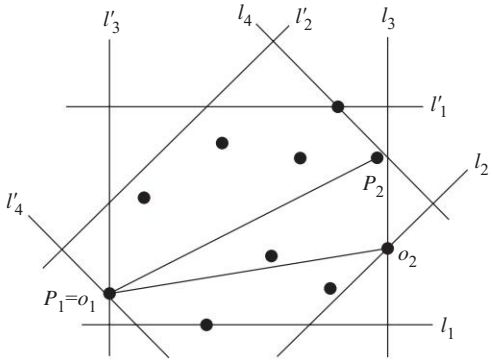
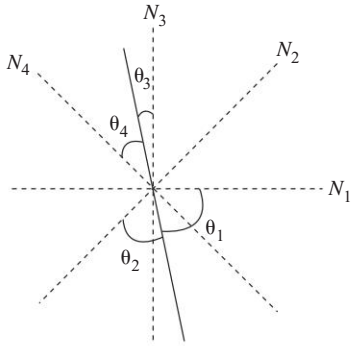


FIGURE 12–42 An example of the angle between the approximate farthest pair and N_i.



We have the following:

$$\begin{aligned}
\overline{P_1P_2} &\le d_i \cdot \sec\theta_i, \ \forall i \in \{1, \ldots, m\} \\
&\le \min_i \{d_i \sec\theta_i\} \\
&\le \min_i \{d_{\max} \cdot \sec\theta_i\} \\
&= d_{\max} \min_i \{\sec\theta_i\} \\
&= d_{\max} \cdot \sec(\min_i \{\theta_i\}) \\
&= d_{\max} \cdot \sec\frac{\pi}{2m} \\
&= \overline{o_1o_2} \cdot \sec\frac{\pi}{2m}.
\end{aligned}$$

Hence, we have the following:

$$\text{Error rate} = \frac{\overline{p_1p_2} - \overline{o_1o_2}}{\overline{p_1p_2}}$$

$$\leq \frac{\overline{p_1p_2} - \overline{o_1o_2}}{\overline{o_1o_2}}$$

$$\leq \sec\frac{\pi}{2m} - 1.$$

The upper bound of the error rate decreases as m increases and is listed in Table 12–1. The 1-center problem, introduced in Chapter 6, is defined as follows: Given a set of n points in the Euclidean space, find a circle to cover all these n points such that the diameter length of this circle is minimized. In Algorithm A, there are always m pairs of parallel lines with particularly fixed slopes. The m pairs of parallel lines form a convex enclosing polygon E. This convex enclosing polygon E has the following properties:

Property 1: E covers all the input points.
Property 2: The number of vertices of E is at most $2m$.
Property 3: The slopes of edges in E are one-to-one equal to those of a regular polygon with $2m$ sides.
Property 4: Every edge of E contains at least one point.

We can use the above properties of E to construct an on-line approximate smallest circle for input points. Our on-line algorithm for the 1-center problem is also almost exactly the same as Algorithm A, except Step 2. We use now the following:

Step 2: Find the vertices of the convex polygon E, i.e. the points which are intersections of two consecutive lines. Use any off-line 1-center algorithm to solve the 1-center problem for the point set of at most $2m$ vertices of E. Let this circle be the on-line approximate circle A_i.

The time complexity of finding vertices of E takes $O(m)$ time (Property 2). The time complexity of finding a circle covering $2m$ points is again a constant because only a constant number of points is involved.

By definition, the circle found by our algorithm covers the convex enclosing polygon E and, therefore by Property 1, it also covers all input points. The

remaining problem is how good the on-line approximate smallest circle is. We follow the approach called competitive analysis. Let the diameter length of a circle C be denoted as $d(C)$. Given a sequence of points, let C_{opt} and C_{onl} denote the smallest circle and the on-line approximate circle covering all points of this sequence, respectively. An on-line algorithm for the 1-center problem is c_k-competitive if for all sequences of points in the plane, $d(C_{onl}) \le c_k \cdot d(C_{opt})$. Later we shall show that c_k is $\sec \frac{\pi}{2m}$.

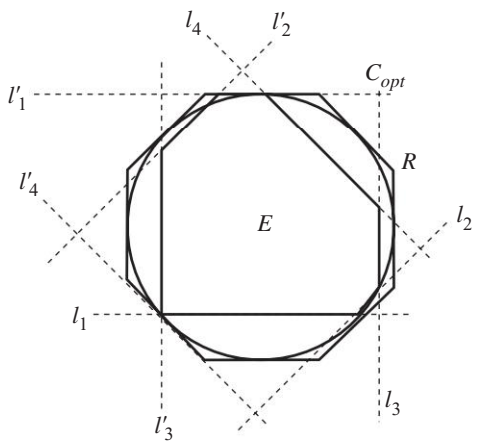
For C_{opt}, we can construct a regular polygon R with $2m$ sides and the same slopes as the $2m$ pairs of parallel lines contain all of the points. Let E denote the polygon formed by those $2m$ lines. As shown in Figure 12–43, $E \subset R$. Let C_R denote the smallest circle covering R. Since $E \subset R$, the smallest circle covering E, which is C_{onl}, must be smaller than or equal to C_R. Since C_{opt} is the optimal circle covering all points, we have $d(C_{opt}) \le d(C_{onl}) \le d(C_R)$. Consider Figure 12–43. $\overline{OB}$ is the radius of C_{opt}. $\overline{OA}$ is the radius of C_R. We have

$$d(C_{opt}) \cdot \sec \frac{\pi}{2m} = d(C_R).$$

Hence,

$$\frac{d(C_{onl})}{d(C_{opt})} \le \frac{d(C_R)}{d(C_{opt})} = \frac{\overline{OA}}{\overline{OB}} = \sec \frac{\pi}{2m}.$$

FIGURE 12–43 The relation between E and R.



That is to say, the on-line algorithm presented in this section is sec($\pi/(2m)$)-competitive.

12–7 AN ON-LINE SPANNING TREE ALGORITHM BASED ON THE RANDOMIZATION STRATEGY

The performance of an on-line algorithm with low space and low time complexity in each decision step is of fundamental importance for on-line problems in general, because in real applications (e.g. real time systems), a decision must be made as soon as possible when data arrive. Therefore, randomization at the decision step becomes a design strategy for on-line algorithms to meet the requirements of low computing complexity and space. In this section, we shall introduce a simple randomized algorithm for the on-line spanning tree problem, defined in Section 12–1.

Our randomized on-line algorithm, Algorithm $R(m)$, for computing a Euclidean tree is presented next.

Algorithm 12–2 □ Algorithm $R(m)$ for computing on-line Euclidean spanning trees

Input: $n(\geq 3)$ points $v_1, \ldots, v_n$ on Euclidean space and a positive integer $m(\leq n-1)$

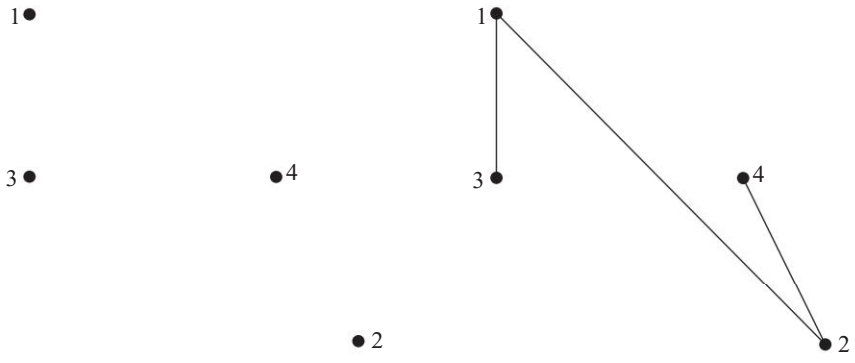
Output: A Euclidean tree T

```
Begin
    T = φ;
    input(m);
    input(v_1);
    input(v_2);
    Add the edge between v_1 and v_2 to T;
    For k = 3 to n do
        input(v_k);
        If k ≤ m + 1 then
        Add the minimum edge between v_k and v_1, ... , v_{k-1} to T;
        else
        Randomly choose m edges from the k − 1 edges between v_k
        and v_1, ... , v_{k-1}, and add the minimum edge of the m edges
        to T;
    Endfor
    output(T);
End
```

Algorithm $R(m)$ accepts the points one by one according to their order in the input sequence. When a point arrives, the algorithm makes a decision to construct a spanning tree which spans the points given so far, and all decisions are not changed after the decisions are made. Suppose that the currently examined point is the ith input point (i.e. v_i). If $2 \le i \le m + 1$, Algorithm $R(m)$ adds the minimum edge between the currently examined point and the previously examined points (i.e. $v_1, \ldots, v_{i-1}$) to the previous tree to form a new spanning tree; otherwise, it adds the minimum edge of m edges, which are randomly chosen from the edges between the currently examined point and the previously examined points, to the previous tree to form a new spanning tree. In each examining step, it takes $O(m)$ time. It is not hard to see that Algorithm $R(n - 1)$ is the deterministic greedy algorithm introduced before.

Suppose we are given four points in the plane, shown in Figure 12–44(a), and the input sequence of the points is 1, 2, 3, 4, we will construct a spanning tree, shown in Figure 12–44(b), by Algorithm $R(2)$. Table 12–2 shows the actions during the execution of Algorithm $R(2)$ with the input sequence 1, 2, 3, 4, where $e(a, b)$ denotes the edge between point a and point b, and $|e(a, b)|$ denotes the Euclidean distance of $e(a, b)$.

FIGURE 12–44 An example for algorithm $R(2)$.



(a) Input points **(b) Input sequence 1, 2, 3, 4**

TABLE 12–2 The actions during the execution of algorithm $R(2)$ with the input sequence 1, 2, 3, 4.

Current input point	Action
1	No action.
2	Add $e(1, 2)$ to T.
3	Add $e(3, 1)$ to T. ($\|e(3, 1)\| = \min\{\|e(3, 1)\|, \|e(3, 2)\|\}$)
4	Choose $e(4, 1)$ and $e(4, 2)$, and add $e(4, 2)$ to T. ($\|e(4, 2)\| = \min\{\|e(4, 1)\|, \|e(4, 2)\|\}$)

A randomized on-line Algorithm A is called c-competitive if there is a constant b such that

$$E(C_A(\sigma)) \le c \cdot C_{opt}(\sigma) + b$$

where $E(C_A(\sigma))$ denotes the expected cost incurred by Algorithm A with input σ.

Next, we shall show that if m is a fixed constant, the competitive ratio of Algorithm $R(m)$ for the on-line spanning tree problem is $\Theta(n)$.

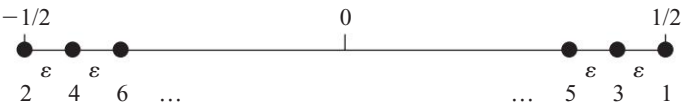
Assume that we are given a set σ of n points $1, 2, \ldots, n$, and the input sequence is $1, 2, \ldots, n$. Let $D(a, b)$ denote the distance between point a and point b and $N(a, k)$ point a's kth nearest point among the points given so far. We denote the expected length of the tree produced by Algorithm $R(m)$ with input σ by $E(L_{R(m)}(\sigma))$. Suppose that Algorithm $R(m)$ reads point i now. If $2 \le i \le m + 1$, the algorithm links point i and its nearest point among the $i - 1$ points given before. However, if $m + 2 \le i \le n$, the algorithm links point i and its jth nearest point among the $i - 1$ points given before with probability $\binom{i-1-j}{m-1} \Big/ \binom{i-1}{m}$, where $1 \le j \le i - m$. Then we have the following formula for $E(L_{R(m)}(\sigma))$:

$$E(L_{R(m)}(\sigma)) = \sum_{i=2}^{m+1} D(i, N(i, 1)) + \sum_{i=m+2}^{n} \sum_{j=1}^{i-m} \binom{i-1-j}{m-1} \Big/ \binom{i-1}{m} \cdot D(i, N(i, j)). \quad \textbf{(12–11)}$$

Suppose that the n points are all on a line and point i is at

$$\frac{(-1)^{i+1}}{2} + (-1)^i(\lceil i/2 - 1 \rceil)\varepsilon, \text{ where } \varepsilon > 0 \text{ (see Figure 12–45).}$$

FIGURE 12–45 The n points on a line.



If the value of ε is significantly small, we have

$$D(i, N(i, j)) = \begin{cases} 0 & \text{if } 1 \le j \le i - \lceil i/2 \rceil - 1 \\ 1 & \text{if } i \le \lceil i/2 \rceil \le j - i - 1 \end{cases}.$$

Let σ^* be the above input points. If the input sequence is $1, 2, \ldots, n$, by Equation (12–11), we get

$$\begin{aligned}
&E(L_{R(m)}(\sigma^*)) \\
&= 1 + \sum_{i=m+2}^{n} \sum_{j=1-\lfloor i/2 \rfloor}^{i-m} \binom{i-1-j}{m-1} \Big/ \binom{i-1}{m} \\
&= 1 + \sum_{i=m+2}^{n} 1 \Big/ \binom{i-1}{m} \sum_{k=0}^{\lfloor i/2 \rfloor - m} \binom{m-1+k}{k} \\
&= 1 + \sum_{i=m+2}^{n} \binom{\lfloor i/2 \rfloor}{m} \Big/ \binom{i-1}{m} \quad \left(\text{by } \sum_{k=0}^{n} \binom{m+k}{k} = \binom{m+n+1}{n}\right) \\
&\ge 1 + \sum_{i=2m+1}^{n} \binom{(i-1)/2}{m} \Big/ \binom{i-1}{m} \\
&= 1 + \frac{1}{2^m} \sum_{i=2m+1}^{n} \frac{\prod_{k=\lceil m/2 \rceil + 1}^{m} (i - 2k + 1)}{\prod_{k=1}^{\lfloor m/2 \rfloor} (i - 2k)} \\
&= 1 + \frac{1}{2^m} \sum_{i=2m+1}^{n} \prod_{k=1}^{\lfloor m/2 \rfloor} \left(1 - \frac{2\lceil m/2 \rceil - 1}{i - 2k}\right).
\end{aligned}$$

Since m is a constant, we have

$$\sum_{i=2m+1}^{n} \prod_{k=1}^{\lfloor m/2 \rfloor} \left(1 - \frac{2\lceil m/2 \rceil - 1}{i - 2k}\right)$$

$$> \sum_{i=2m+1}^{n} 1 - \sum_{k=1}^{\lfloor m/2 \rfloor} \frac{2\lceil m/2 \rceil - 1}{i - 2k}$$

$$= (n - 2m) - (2\lceil m/2 \rceil - 1)\left[\frac{1}{2m-1} + \frac{1}{2m-3} + \cdots + \frac{1}{2m+1-2\lceil m/2 \rceil}\right.$$

$$+ \frac{1}{2m} + \frac{1}{2m-2} + \cdots + \frac{1}{2m+2-2\lceil m/2 \rceil}$$

$$+ \frac{1}{2m-1} + \frac{1}{2m-1} + \cdots + \frac{1}{2m+3-2\lceil m/2 \rceil} + \cdots$$

$$\left. + \frac{1}{n-2} + \frac{1}{n-4} + \cdots + \frac{1}{n-2\lceil m/2 \rceil}\right]$$

$$= (n - 2m) - (2\lceil m/2 \rceil - 1)\left[\sum_{k=1}^{\lceil m/2 \rceil} (H_{n-2k} - H_{2m-2k})\right] \quad (\text{where } H_n = \sum_{i=1}^{n} \frac{1}{i})$$

$$= n - 2m - o(n).$$

Thus,

$$E(L_{R(m)}(\sigma^*))$$

$$\geq 1 + \frac{1}{2^m} \sum_{i=2m+1}^{n} \prod_{k=1}^{\lfloor m/2 \rfloor} \left(1 - \frac{2\lceil m/2 \rceil - 1}{i - 2k}\right)$$

$$= 1 + \frac{n - 2m - o(n)}{2^m}$$

Let $L_{span}(\sigma^*)$ be the length of the minimum spanning tree with input σ^*. It is easy to see that $L_{span}(\sigma^*) = 1$. Consequently, we obtain

$$\frac{E(L_{R(m)}(\sigma^*))}{L_{span}(\sigma^*)} \geq 1 + \frac{n - 2m - o(n)}{2^m}.$$

This means that the competitive ratios of Algorithm $R(m)$ for the on-line spanning tree problem is $\Omega(n)$, where n is the number of points and m is a fixed constant.

Let D be the diameter of n points. The lengths of minimum Steiner tree and minimum spanning trees are equal to or larger than D. Since the length of each edge is less than or equal to D, the length of the tree produced by any on-line algorithm is less than or equal to $(n - 1) \cdot D$. Therefore, for the on-line spanning tree problem, there is no on-line algorithm whose competitive ratio is larger than $n - 1$, where n is the number of points.

Based on those results, we can conclude that the competitive ratio of Algorithm $R(m)$ for the on-line spanning tree problem is $\Theta(n)$, where n is the number of points and m is a fixed constant.

12–8 NOTES AND REFERENCES

The on-line Steiner tree problem, which is similar to the on-line spanning tree problem, is to find a Steiner tree on-line, where the points are revealed one by one. In Imase and Waxman (1991) and Alon and Azar (1993), the competitive ratio of the greedy algorithm in Section 12–1 was shown to be $\log_2 n$ for these two problems. In addition, Alon and Azar (1993) showed that the lower bounds of competitive ratio of any on-line algorithm for the on-line spanning tree and Steiner tree problems are $\Omega(\log n/\log \log n)$.

A server problem is symmetric if the distance from vertex i to vertex j is the same as the distance from j to i, and is asymmetric if otherwise. An on-line algorithm for solving the k-server problem operate under the additional constraint that it must decide which server to move to satisfy a given request without knowing what the future requests will be.

The cache problem, the paging problem, the two disk heads problem, linear search, etc., can be abstracted as the server problem after choosing the distances between vertices and the number of servers.

Manasse, McGeoch and Sleator (1990) introduced the server problem in 1990. They proved that k is a lower bound of competitive ratio for the k-server problem. The optimal on-line algorithm for the k-server problem on planar trees was proposed in Chrobak and Larmore (1991).

The obstacle traversal problem was first introduced by Papadimitriou and Yannakakis (1991b). They showed a lower bound of $\Omega(\sqrt{d})$ for any deterministic algorithm on the competitive ratio when all obstacles are with sides parallel to the axes, where d is the distance between s and t. They also proposed an asymptotic optimal 3/2-competitive algorithm for which all obstacles are squares with the same size, and with sides parallel to the axes.

The inequality $\frac{\tau_1}{\pi_1} < \frac{2}{3}$ or $\frac{\tau_2}{\pi_2} < \frac{2}{3}$, used in Section 12–3, was proved in Fejes (1978).

The bipartite matching problem was introduced in Kalyanasundaram and Pruhs (1993). The algorithm and lower bound for this problem, presented in Section 12–4, are also from Kalyanasundaram and Pruhs (1993).

In 1966, Graham proposed a simple greedy algorithm, called List, for the m-machine scheduling problem. Graham showed that the List algorithm is $\left(2 - \frac{1}{m}\right)$-competitive. The $\left(2 - \frac{1}{70}\right)$-competitive algorithm presented in Section 12–5 was proposed by Bartal, Fiat, Karloff and Vahra (1995) for the m-machine scheduling problem, where $m \geq 70$.

The on-line algorithms for the geometry problems in Section 12–6 can be found in Chao (1992). The randomized algorithm in Section 12–7, for the on-line spanning tree problem is discussed in Tsai and Tang (1993).

12–9 FURTHER READING MATERIALS

Manasse, McGeoch and Sleator (1990) showed that the optimal competitive ratio of a deterministic on-line algorithm for the symmetric 2-server problem is 2, and conjectured that k-server problems have no deterministic on-line c-competitive algorithms for $c < k$, for any metric space with at least $k + 1$ vertices. Koutsoupias and Papadimitriou (1995) proved that the work function algorithm for the k-server problem has a competitive ratio at most $2k - 1$. The conjecture has not been proved or disproved up today.

Chan and Lam (1993) proposed an asymptotic optimal $(1 + r/2)$-competitive algorithm for the obstacle traversal problem, where each obstacles has aspect ratio (the length ratio between the long side and the short side) bounded by some constant r.

Khuller, Mitchell and Vazirani (1994) proposed on-line algorithms for weighted matching and stable marriages. The bipartite matching problem for unweighted graphs was discussed in Karp, Vazirani and Vazirani (1990) and Kao and Tate (1991).

Karger, Phillips and Torng (1994) presented an algorithm to solve the m-machine scheduling problem, which has a competitive ratio at most 1.945 for all m and outperforms the List algorithm for $m \geq 6$.

Many interesting on-line problems were also exploited, such as metrical task systems: Borodin, Linial and Saks (1992); on-line bin packing problems: Csirik (1989); Lee and Lee (1985); Vliet (1992); on-line graph coloring problems: Vishwanathan (1992); on-line financial problems: El-Yaniv (1998), etc.

The following lists interesting and new results: Adamy and Erlebach (2003); Albers (2002); Albers and Koga (1998); Albers and Leonardi (1999); Alon, Awerbuch and Azar (2003); Aspnes, Azar, Fiat, Plotkin and Waarts (1997); Awerbuch and Peleg (1995); Awerbuch and Singh (1997); Awerbuch, Azar and Meyerson (2003); Azar, Blum and Mansour (2003); Azar, Kalyanasundaram, Plotkin, Pruhs and Waarts (1997); Azar, Naor and Rom (1995); Bachrach and El-Yaniv (1997); Bareli, Berman, Fiat and Yan (1994); Bartal and Grove (2000); Berman, Charikar and Karpinski (2000); Bern, Greene, Raghunathan and Sudan (1990); Blum, Sandholm and Zinkevich (2002); Caragiannis, Kaklamanis and Papaioannou (2003); Chan (1998); Chandra and Vishwanathan (1995); Chazelle and Matousek (1996); Chekuri, Khanna and Zhu (2001); Conn and Vonholdt (1965); Coppersmith, Doyle, Raghavan and Snir (1993); Crammer and Singer (2002); El-Yaniv (1998); Epstein and Ganot (2003); Even and Shiloach (1981); Faigle, Kern and Nawijn (1999); Feldmann, Kao, Sgall and Teng (1993); Galil and Seiferas (1978); Garay, Gopal, Kutten, Mansour and Yung (1997); Goldman, Parwatikar and Suri (2000); Gupta, Konjevod and Varsamopoulos (2002); Haas and Hellerstein (1999); Halldorsson (1997); Halperin, Sharir and Goldberg (2002); Irani, Shukla and Gupta (2003); Janssen, Krizanc, Narayanan and Shende (2000); Jayram, Kimbrel, Krauthgamer, Schieber and Sviridenko (2001); Kalyanasundaram and Pruhs (1993); Keogh, Chu, Hart and Pazzani (2001); Khuller, Mitchell and Vazirani (1994); Klarlund (1999); Kolman and Scheideler (2001); Koo, Lam, Ngan, Sadakane and To (2003); Kossmann, Ramsak and Rost (2002); Lee (2003a); Lee (2003b); Lueker (1998); Manacher (1975); Mandic and Cichocki (2003); Mansour and Schieber (1992); Megow and Schulz (2003); Oza and Russell (2001); Pandurangan and Upfal (2001); Peserico (2003); Pittel and Weishaar (1997); Ramesh (1995); Seiden (1999); Seiden (2002); Sgall (1996); Tamassia (1996); Tsai, Lin and Hsu (2002); Tsai, Tang and Chen (1994); Tsai, Tang and Chen (1996); Ye and Zhang (2003); and Young (2000).

12.1 Consider the bipartite matching problem introduced in Section 12–4. When a vertex b_i of B arrives, we let b_i match the nearest unmatched vertex of set R. Is such an algorithm $(2n - 1)$-competitive? Prove your answer.

12.2 Given a bipartite weighted graph with vertices bipartition R and B, each of cardinality n, the maximum bipartite matching problem is to find a bipartite matching with the maximum cost. Assume that all the weights of edges satisfy the triangle inequality. If the vertices in R are all known to us in advance and the vertices in B are revealed one by one, what is the competitive ratio of a greedy algorithm which always matches an arriving vertex with the farthest unmatched vertex of R?

12.3 Prove that the on-line k-server algorithm introduced in Section 12–2 is also k-competitive for a line.

12.4 Is the obstacle traversal algorithm introduced in Section 12–3 3/2-competitive for square obstacles with arbitrary directions? Prove your answer.

Bibliography

Abel, S. (1990): A Divide and Conquer Approach to Least-Squares Estimation, *IEEE Transactions on Aerospace and Electronic Systems*, Vol. 26, pp. 423–427.

Adamy, U. and Erlebach, T. (2003): Online Coloring of Intervals with Bandwidth, *LNCS 2909*, pp. 1–12.

Agarwal, P. K. and Procopiuc. C. M. (2000): Approximation Algorithms for Projective Clustering, *Journal of Algorithms*, Vol. 34, pp. 128–147.

Agarwal, P. K. and Sharir, M. (1996): Efficient Randomized Algorithms for Some Geometric Optimization Problems, *Discrete Computational Geometry*, Vol. 16, No. 4, pp. 317–337.

Agrawal, M., Kayal, N. and Saxena, N. (2004): PRIMES is in P, *Annals of Mathematics* (forthcoming).

Aho, A. V., Ganapathi, M. and Tjang, S. (1989): Code Generation Using Tree Matching and Dynamic Programming, *ACM Transactions on Programming Languages and Systems*, Vol. 11, pp. 491–516.

Aho, A. V., Hopcroft, J. E. and Ullman, J. D. (1974): *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass.

Ahuja, R. K. (1988): Minimum Cost-Reliability Ratio Path Problem, *Computer and Operations Research*, Vol. 15, No. 1.

Aiello, M., Rajagopalan, S. R. and Venkatesan, R. (1998): Design of Practical and Provably Good Random Number Generators, *Journal of Algorithms*, pp. 358–389.

Akiyoshi, S. and Takeaki, U. (1997): A Linear Time Algorithm for Finding a *k*-Tree Core, *Journal of Algorithms*, Vol. 23, pp. 281–290.

Akutsu, T. (1996): Protein Structure Alignment Using Dynamic Programming and Iterative Improvement, *IEICE Transactions on Information and Systems*, Vol. E78-D, No. 12, pp. 1629–1636.

Akutsu, T., Arimura, H. and Shimozono, S. (2000): On Approximation Algorithms for Local Multiple Alignment, *Proceedings of the Fourth Annual International Conference on Computational Molecular Biology*, ACM Press, Tokyo, pp. 1–7.

Akutsu, T. and Halldorsson, M. M. (1994): On the Approximation of Largest Common Subtrees and Largest Common Point Sets, *Lecture Notes in Computer Science*, pp. 405–413.

Akutsu, T. and Miyano, S. (1997): On the Approximation of Protein Threading, *RECOMB*, pp. 3–8.

Akutsu, T., Miyano, S. and Kuhara, S. (2003): A Simple Greedy Algorithm for Finding Functional Relations: Efficient Implementation and Average Case Analysis, *Theoretical Computer Science*, Vol. 292, pp. 481–495.

Albers, S. (2002): On Randomized Online Scheduling, *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, ACM Press, Montreal, Canada, pp. 134–143.

Albers, S. and Koga, H. (1998): New On-Line Algorithms for the Page Replication Problem, *Journal of Algorithms*, Vol. 27, pp. 75–96.

Albers, S. and Leonardi, S. (1999): On-Line Algorithms, *ACM Computing Surveys (CSUR)*, Vol. 31, No. 3, Article No. 4, September.

Alberts, D. and Henzinger, M. R. (1995): Average Case Analysis of Dynamic Graph Algorithms, *Symposium in Discrete Algorithms*, pp. 312–321.

Aldous, D. (1989): *Probability Approximations via the Poisson Clumping Heuristic*, Springer-Verlag, Berlin.

Aleksandrov, L. and Djidjev, H. (1996): Linear Algorithms for Partitioning Embedded Graphs of Bounded Genus, *SIAM Journal on Discrete Mathematics*, Vol. 9, No. 1, pp. 129–150.

Alon, N., Awerbuch, B. and Azar, Y. (2003): Session 2B: The Online Set Cover Problem, *Proceedings of the 35th ACM Symposium on Theory of Computing*, ACM Press, San Diego, California, pp. 100–105.

Alon, N. and Azar, Y. (1989): Finding an Approximate Maximum, *SIAM Journal on Computing*, Vol. 18, No. 2, pp. 258–267.

Alon, N. and Azar, Y. (1993): On-Line Steiner Trees in the Euclidean Plane, *Discrete Computational Geometry*, Vol. 10, No. 2, pp. 113–121.

Alon, N., Babai, L. and Itai, A. (1986): A Fast and Simple Randomized Parallel Algorithm for the Maximal Independent Set Problem, *Journal of Algorithms*, Vol. 7, No. 4, pp. 567–583.

Alpert, C. J. and Kahng, A. B. (1995): Multi-Way Partitioning via Geometric Embeddings; Orderings; and Dynamic Programming, *IEEE Transactions on CAD*, Vol. 14, pp. 1342–1358.

Amini, A. A., Weymouth, T. E. and Jain, R. C. (1990): Using Dynamic Programming for Solving Variational Problems in Vision, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 12, No. 9, pp. 855–867.

Amir, A. and Farach, M. (1995): Efficient 2-Dimensional Approximate Matching of Half-Rectangular Figures, *Information and Computation*, Vol. 118, pp. 1–11.

Amir, A. and Keselman, D. (1997): Maximum Agreement Subtree in a Set of Evolutionary Trees: Metrics and Efficient Algorithms, *SIAM Journal on Computing*, Vol. 26, pp. 1656–1669.

Amir, A. and Landau, G. (1991): Fast Parallel and Serial Multidimensional Approximate Array Matching, *Theoretical Computer Science*, Vol. 81, pp. 97–115.

Anderson, R. (1987): A Parallel Algorithm for the Maximal Path Problem, *Combinatorica*, Vol. 7, No. 4, pp. 315–326.

Anderson, R. J. and Woll, H. (1997): Algorithms for the Certified Write-All Problem, *SIAM Journal on Computing*, Vol. 26, No. 5, pp. 1277–1283.

Ando, K., Fujishige, S. and Naitoh, T. (1995): A Greedy Algorithm for Minimizing a Separable Convex Function over a Finite Jump System, *Journal of the Operations Research Society of Japan*, Vol. 38, pp. 362–375.

Arkin, E. M., Chiang, Y. J., Mitchell, J. S. B., Skiena, S. S. and Yang, T. C. (1999): On the Maximum Scatter Traveling Salesperson Problem, *SIAM Journal on Computing*, Vol. 29, pp. 515–544.

Armen, C. and Stein, C. (1994): A 2 3/4-Approximation Algorithm for the Shortest Superstring Problem, *Technical Report* (PCS-TR94-214), Computer Science Department, Dartmouth College, Hanover, New Hampshire.

Armen, C. and Stein, C. (1995): Improved Length Bounds for the Shortest Superstring Problem (Shortest Common Superstring: 2 3/4-Approximation), *Lecture Notes in Computer Science*, Vol. 955, pp. 494–505.

Armen, C. and Stein, C. (1996): A 2 2/3 Approximation Algorithm for the Shortest Superstring Problem, *Lecture Notes in Computer Science*, Vol. 1075, pp. 87–101.

Armen, C. and Stein, C. (1998): 2 2/3 Superstring Approximation Algorithm, *Discrete Applied Mathematics*, Vol. 88, No. 1–3, pp. 29–57.

Arora, S. (1996): Polynomial Approximation Schemes for Euclidean TSP and Other Geometric Problems, *Foundations of Computer Science*, pp. 2–13.

Arora, S. and Brinkman, B. (2002): A Randomized Online Algorithm for Bandwidth Utilization, *Symposium on Discrete Algorithms*, pp. 535–539.

Arora, S., Lund, C., Motwani, R., Sudan, M. and Szegedy, M. (1998): Proof Verification and the Hardness of Approximation Problems (Prove NP-Complete Problem), *Journal of the ACM*, Vol. 45, No. 3, pp. 501–555.

Arratia, R., Goldstein, L. and Gordon, L. (1989): Two Moments Suffice for Poisson Approximation: The Chen-Stein Method, *The Annals of Probability*, Vol. 17, pp. 9–25.

Arratia, R., Martin, D., Reinert, G. and Waterman, M. S. (1996): Poisson Process Approximation for Sequence Repeats and Sequencing by Hybridization, *Journal of Computational Biology*, Vol. 3, pp. 425–464.

Arya, S., Mount, D. M., Netanyahu, N. S., Silverman, R. and Wu, A. Y. (1998): An Optimal Algorithm for Approximate Nearest Neighbor Searching in Fixed Dimensions, *Journal of the ACM*, Vol. 45, No. 6, pp. 891–923.

Ashenhurst, R. L. (1987): *ACM Turing Award Lectures: The First Twenty Years: 1966–1985*, ACM Press, Baltimore, Maryland.

Aspnes, J., Azar, Y., Fiat, A., Plotkin, S. and Waarts, O. (1997): On-Line Routing of Virtual Circuits with Applications to Load Balancing and Machine Scheduling, *Journal of the ACM*, Vol. 44, No. 3, pp. 486–504.

Atallah, M. J. and Hambrusch, S. E. (1986): An Assignment Algorithm with Applications to Integrated Circuit Layout, *Discrete Applied Mathematics*, Vol. 13, No. 1, pp. 9–22.

Auletta, V., Parente, D. and Persiano, G. (1996): Dynamic and Static Algorithms for Optimal Placement of Resources in Trees, *Theoretical Computer Science*, Vol. 165, No. 2, pp. 441–461.

Ausiello, G., Crescenzi, P. and Protasi, M. (1995): Approximate Solution of NP Optimization Problems, *Theoretical Computer Science*, Vol. 150, pp. 1–55.

Avis, D., Bose, P., Shermer, T. C., Snoeyink, J., Toussaint, G. and Zhu, B. (1996): On the Sectional Area of Convex Polytopes, *Proceedings of the Twelfth Annual Symposium on Computational Geometry*, ACM Press, Philadelphia, Pennsylvania, pp. 411–412.

Avrim, B., Jiang, T., Li, M., Tromp, J. and Yannakakis, M. (1991): Linear Approximation of Shortest Superstrings, *Proceedings of the 23rd ACM Symposium on Theory of Computation*, ACM Press, New Orleans, Louisiana, pp. 328–336.

Awerbuch, B., Azar, Y. and Meyerson, A. (2003): Reducing Truth-Telling Online Mechanisms to Online Optimization, *Proceedings of the 35th ACM Symposium on Theory of Computing*, ACM Press, San Diego, California, pp. 503–510.

Awerbuch, B. and Peleg, D. (1995): On-Line Tracking of Mobile Users, *Journal of the ACM*, Vol. 42, No. 5, pp. 1021–1058.

Awerbuch, B. and Singh, T. (1997): On-Line Algorithms for Selective Multicast and Maximal Dense Trees, *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, ACM Press, El Paso, Texas, pp. 354–362.

Azar, Y., Blum, A. and Mansour, Y. (2003): Combining Online Algorithms for Rejection and Acceptance, *Proceedings of the 15th Annual ACM Symposium on Parallel Algorithms and Architectures*, ACM Press, San Diego, California, pp. 159–163.

Azar, Y., Kalyanasundaram, B., Plotkin, S., Pruhs, K. and Waarts, O. (1997): On-Line Load Balancing of Temporary Tasks, *Journal of Algorithms*, Vol. 22, pp. 93–110.

Azar, Y., Naor, J. and Rom, R. (1995): The Competitiveness of On-Line Assignments, *Journal of Algorithms*, Vol. 18, pp. 221–237.

Bachrach, R. and El-Yaniv, R. (1997): Online List Accessing Algorithms and Their Applications: Recent Empirical Evidence, *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, SIAM, New Orleans, Louisiana, pp. 53–62.

Baeza-Yates, R. A. and Navarro, G. (1999): Faster Approximate String Matching, *Algorithmica*, Vol. 23, No. 2, pp. 127–158.

Baeza-Yates, R. A. and Perleberg, C. H. (1992): Fast and Practical Approximate String Matching, *Lecture Notes in Computer Science*, Vol. 644, pp. 185–192.

Bafna, V., Berman, P. and Fujito, T. (1999): A 2-Approximation Algorithm for the Undirected Feedback Vertex Set Problem, *SIAM Journal on Discrete Mathematics*, Vol. 12, No. 3, pp. 289–297.

Bafna, V., Lawler, E. L. and Pevzner, P. A. (1997): Approximation Algorithms for Multiple Sequence Alignment, *Theoretical Computer Science*, Vol. 182, pp. 233–244.

Bafna, V. and Pevzner, P. (1996): Genome Rearrangements and Sorting by Reversals (Approximation Algorithm), *SIAM Journal on Computing*, Vol. 25, No. 2, pp. 272–289.

Bafna, V. and Pevzner, P. A. (1998): Sorting by Transpositions, *SIAM Journal on Discrete Mathematics*, Vol. 11, No. 2, pp. 224–240.

Bagchi, A. and Mahanti, A. (1983): Search Algorithms under Different Kinds of Heuristics—A Comparative Study, *Journal of the ACM*, Vol. 30, No. 1, pp. 1–21.

Baker, B. S. (1994): Approximation Algorithms for NP-Complete Problems on Planar Graphs, *Journal of the ACM*, Vol. 41, No. 1.

Baker, B. S. and Coffman, E. G. Jr. (1982): A Two-Dimensional Bin-Packing Model of Preemptive FIFO Storage Allocation, *Journal of Algorithms*, Vol. 3, pp. 303–316.

Baker, B. S. and Giancarlo, R. (2002): Sparse Dynamic Programming for Longest Common Subsequence from Fragments, *Journal of Algorithm*, Vol. 42, pp. 231–254.

Balas, F. and Yu, C. S. (1986): Finding a Maximum Clique in an Arbitrary Graph, *SIAM Journal on Computing*, Vol. 15, No. 4, pp. 1054–1068.

Bandelloni, M., Tucci, M. and Rinaldi, R. (1994): Optimal Resource Leveling Using Non-Serial Dynamic Programming, *European Journal of Operational Research*, Vol. 78, pp. 162–177.

Barbu, V. (1991): The Dynamic Programming Equation for the Time Optimal Control Problem in Infinite Dimensions, *SIAM Journal on Control and Optimization*, Vol. 29, pp. 445–456.

Bareli, E., Berman, P., Fiat, A. and Yan, P. (1994): Online Navigation in a Room, *Journal of Algorithms*, Vol. 17, pp. 319–341.

Bartal, Y., Fiat, A., Karloff, H. and Vohra, R. (1995): New Algorithms for an Ancient Scheduling Problem, *Journal of Computer and System Sciences*, Vol. 51, No. 3, pp. 359–366.

Bartal, Y. and Grove, E. (2000): The Harmonic *k*-Server Algorithm is Competitive, *Journal of the ACM*, Vol. 47, No. 1, pp. 1–15.

Basse, S. and Van Gelder, A. (2000): *Computer Algorithms: Introduction to Design and Analysis*, Addison-Wesley, Reading, Mass.

Bein, W. W. and Brucker, P. (1986): Greedy Concepts for Network Flow Problems, *Discrete Applied Mathematics*, Vol. 15, No. 2, pp. 135–144.

Bein, W. W., Brucker, P. and Tamir, A. (1985): Minimum Cost Flow Algorithm for Series-Parallel Networks, *Discrete Applied Mathematics*, Vol. 10, No. 3, pp. 117–124.

Bekesi, J., Galambos, G., Pferschy, U. and Woeginger, G. (1997): Greedy Algorithms for On-Line Data Compression, *Journal of Algorithms*, Vol. 25, pp. 274–289.

Bellman, R. and Dreyfus, S. E. (1962): *Applied Dynamic Programming*, Princeton University Press, Princeton, New Jersey.

Ben-Asher, Y., Farchi, E. and Newman, I. (1999): Optimal Search in Trees, *SIAM Journal on Mathematics*, Vol. 28, No. 6, pp. 2090–2102.

Bent, S. W., Sleator, D. D. and Tarjan, R. E. (1985): Biased Search Trees, *SIAM Journal on Computing*, Vol. 14, No. 3, pp. 545–568.

Bentley, J. L. (1980): Multidimensional Divide-and-Conquer, *Communications of the ACM*, Vol. 23, No. 4, pp. 214–229.

Bentley, J. L., Faust, G. M. and Preparata, F. P. (1982): Approximation Algorithms for Convex Hulls, *Communications of the ACM*, Vol. 25, pp. 64–68.

Bentley, J. L. and McGeoch, C. C. (1985): Amortized Analysis of Self-Organizing Sequential Search Heuristics, *Communications of the ACM*, Vol. 28, No. 4, pp. 404–411.

Bentley, J. L. and Shamos, M. I. (1978): Divide-and-Conquer for Linear Expected Time, *Information Processing Letters*, Vol. 7, No. 2, pp. 87–91.

Berger, B. and Leighton, T. (1998): Protein Folding in the Hydrophobic-Hydrophilic (HP) Model is NP-Complete (Prove NP-Complete Problem), *Journal of Computational Biology*, Vol. 5, No. 1, pp. 27–40.

Berger, R. (1966): The Undecidability of the Domino Problem, *Memoirs of the American Mathematical Society*, No. 66.

Berman, P., Charikar, M. and Karpinski, M. (2000): On-Line Load Balancing for Related Machines, *Journal of Algorithms*, Vol. 35, pp. 108–121.

Berman, P., Hannenhalli, S. and Karpinki, M. (2001): 1.375 – Approximation Algorithm for Sorting by Reversals, *Technical Report DIMACS*, TR2001-41.

Berman, P., Karpinski, M., Larmore, L. L., Plandowski, W. and Rytter, W. (2002): On the Complexity of Pattern Matching for Highly Compressed Two-Dimensional Texts, *Journal of Computer System Sciences*, Vol. 65, No. 2, pp. 332–350.

Bern, M., Greene, D., Raghunathan, A. and Sudan, M. (1990): Online Algorithms for Locating Checkpoints, *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, ACM Press, Baltimore, Maryland, pp. 359–368.

Bhagavathi, D., Grosch, C. E. and Olariu, S. (1994): A Greedy Hypercube-Labeling Algorithm, *The Computer Journal*, Vol. 37, pp. 124–128.

Bhattacharya, B. K., Jadhav, S., Mukhopadhyay, A. and Robert, J. M. (1994): Optimal Algorithms for Some Intersection Radius Problems, *Computing*, Vol. 52, No. 3, pp. 269–279.

Blankenagel, G. and Gueting, R. H. (1990): Internal and External Algorithms for the Points-in-Regions Problem, *Algorithmica*, Vol. 5, No. 2, pp. 251–276.

Blazewicz, J. and Kasprzak, M. (2003): Complexity of DNA Sequencing by Hybridization, *Theoretical Computer Science*, Vol. 290, No. 3, pp. 1459–1473.

Blot, J., Fernandez de la Vega, W., Paschos, V. T. and Saad, R. (1995): Average Case Analysis of Greedy Algorithms for Optimization Problems on Set Systems, *Theoretical Computer Science*, Vol. 147, No. 1–2, pp. 267–298.

Blum, A. (1994): New Approximation Algorithms for Graph Coloring, *Journal of the ACM*, Vol. 41, No. 3.

Blum, A., Jiang, T., Li, M., Tromp, J. and Yannakakis, M. (1994): Linear Approximation of Shortest Superstrings, *Journal of the ACM*, Vol. 41, pp. 630–647.

Blum, A., Sandholm, T. and Zinkevich, M. (2002): Online Algorithms for Market Clearing, *Proceedings of the 30th Annual ACM-SIAM Symposium on Discrete Algorithms*, SIAM, San Francisco, California, pp. 971–980.

Blum, M., Floyd, R. W., Pratt, V. R., Rivest, R. L. and Tarjan, R. E. (1972): Time Bounds for Selection, *Journal of Computer and System Sciences*, Vol. 7, No. 4, pp. 448–461.

Bodlaender, H. L. (1988): The Complexity of Finding Uniform Emulations on Fixed Graphs, *Information Processing Letters*, Vol. 29, No. 3, pp. 137–141.

Bodlaender, H. L. (1993): Complexity of Path-Forming Games, *Theoretical Computer Science*, Vol. 110, No. 1, pp. 215–245.

Bodlaender, H. L., Downey, R. G., Fellows, M. R. and Wareham, H. T. (1995): The Parameterized Complexity of Sequence Alignment and Consensus, *Theoretical Computer Science*, Vol. 147, pp. 31–54.

Bodlaender, H. L., Fellows, M. R. and Hallet, M. T. (1994): Beyond NP-Completeness for Problems of Bounded Width: Hardness for the W Hierarchy (Extended Abstract), *Proceedings of the 26th Annual ACM Symposium on the Theory of Computing*, ACM Press, New York, pp. 449–458.

Boffey, T. B. and Green, J. R. (1983): Design of Electricity Supply Networks, *Discrete Applied Mathematics*, Vol. 5, pp. 25–38.

Boldi, P. and Vigna, S. (1999): Complexity of Deciding Sense of Direction, *SIAM Journal on Computing*, Vol. 29, No. 3, pp. 779–789.

Bonizzoni, P. and Vedova, G. D. (2001): The Complexity of Multiple Sequence Alignment with SP-Score that is a Metric, *Theoretical Computer Science*, Vol. 259, pp. 63–79.

Bonizzoni, P., Vedova, G. D. and Mauri, G. (2001): Experimenting an Approximation Algorithm for the LCS, *Discrete Applied Mathematics*, Vol. 110, No. 1, pp. 13–24.

Boppana, R. B., Hastad, J. and Zachos, S. (1987): Does Co-NO Have Short Interactive Proofs? *Information Processing Letters*, Vol. 25, No. 2, pp. 127–132.

Boreale, M. and Trevisan, L. (2000): A Complexity Analysis of Bisimilarity for Value-Passing Processes, *Theoretical Computer Science*, Vol. 238, No. 1, pp. 313–345.

Borodin, A. and El-Yaniv, R. (1998): *Online Computation and Competitive Analysis*, Cambridge University Press, Cambridge, England.

Borodin, A., Linial, N. and Saks, M. (1992): An Optimal On-line Algorithm for Metrical Task System, *Journal of the ACM*, Vol. 39, No. 4, pp. 745–763.

Boros, E., Crama, Y., Hammer, P. L. and Saks, M. (1994): A Complexity Index for Satisfiability Problems, *SIAM Journal on Computing*, Vol. 23, No. 1, pp. 45–49.

Boruvka, O. (1926): O Jistem Problemu Minimalmim. *Praca Moravske Prirodovedecke Spoleconosti*, Vol. 3, pp. 37–58.

Bossi, A., Cocco, N. and Colussi, L. (1983): A Divide-and-Conquer Approach to General Context-Free Parsing, *Information Processing Letters*, Vol. 16, No. 4, pp. 203–208.

Brassard, G. and Bratley, P. (1988): *Algorithmics: Theory and Practice*, Prentice-Hall, Englewood Cliffs, New Jersey.

Breen, S., Waterman, M. S. and Zhang, N. (1985): Renewal Theory for Several Patterns, *Journal of Applied Probability*, Vol. 22, pp. 228–234.

Breslauer, D., Jiang, T. and Jiang, Z. J. (1997): Rotations of Periodic Strings and Short Superstrings (2.596 – Approximation), *Algorithms*, Vol. 24, No. 2, pp. 340–353.

Bridson, R. and Tang, W. P. (2001): Multiresolution Approximate Inverse Preconditioners, *SIAM Journal on Scientific Computing*, Vol. 23, pp. 463–479.

Brigham, E. O. (1974): *The Fast Fourier Transform*, Prentice-Hall, Englewood Cliffs, New Jersey.

Brown, C. A. and Purdom, P. W. Jr. (1981): An Average Time Analysis of Backtracking, *SIAM Journal on Computing*, Vol. 10, pp. 583–593.

Brown, K. Q. (1979): Voronoi Diagrams from Convex Hulls, *Information Processing Letters*, Vol. 9, pp. 223–228.

Brown, M. L. and Whitney, D. E. (1994): Stochastic Dynamic Programming Applied to Planning of Robot Grinding Tasks, *IEEE Transactions on Robotics and Automation*, pp. 594–604.

Brown, M. R. and Tarjan, R. E. (1980): Design and Analysis of a Data Structure for Representing Sorted Lists, *SIAM Journal on Computing*, Vol. 9, No. 3, pp. 594–614.

Bruno, J., Coffman, E. G. Jr. and Sethi, R. (1974): Scheduling Independent Tasks to Reduce Mean Finishing Time, *Communications of the ACM*, Vol. 17, No. 7, pp. 382–387.

Bryant, D. (1998): The Complexity of the Breakpoint Median Problem, *Technical Report CRM-2579*, pp. 1–12.

Caballero-Gil, P. (2000): New Upper Bounds on the Linear Complexity, *Computers and Mathematics with Applications*, Vol. 39, No. 3, pp. 31–38.

Cai, J. Y. and Meyer, G. E. (1987): Graph Minimal Uncolorability Is DP-Complete, *SIAM Journal on Computing*, Vol. 16, No. 2, pp. 259–277.

Caprara, A. (1997a): Sorting by Reversals Is Difficult, *Proceedings of the First Annual International Conference on Computational Molecular Biology*, ACM Press, Santa Fe, New Mexico, pp. 75–83.

Caprara, A. (1997b): Sorting Permutations by Reversals and Eulerian Cycle Decompositions, *SIAM Journal on Discrete Mathematics*, pp. 1–23.

Caprara, A. (1999): Formulations and Hardness of Multiple Sorting by Reversals, *Proceedings of the 3rd Annual International Conference on Computational Molecular Biology*, ACM Press, Lyon, France, pp. 84–93.

Caragiannis, I., Kaklamanis, C. and Papaioannou, E. (2003): Simple On-Line Algorithms for Call Control in Cellular Networks, *Lecture Notes in Computer Science*, Vol. 2909, pp. 67–80.

Cary, M. (2001): Toward Optimal ε-Approximate Nearest Neighbor Algorithms, *Journal of Algorithms*, Vol. 41, pp. 417–428.

Chan, K. F. and Lam, T. W. (1993): An On-Line Algorithm for Navigating in Unknown Environment, *International Journal of Computational Geometry and Applications*, Vol. 3, No. 3, pp. 227–244.

Chan, T. (1998): Deterministic Algorithms for 2-D Convex Programming and 3-D Online Linear Programming, *Journal of Algorithms*, Vol. 27, pp. 147–166.

Chandra, B. and Vishwanathan, S. (1995): Constructing Reliable Communication Networks of Small Weight On-Line, *Journal of Algorithms*, Vol. 18, pp. 159–175.

Chang, C. L. and Lee, R. C. T. (1973): *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York.

Chang, K. C. and Du, H. C. (1988): Layer Assignment Problem for Three-Layer Routing, *IEEE Transactions on Computers*, Vol. 37, pp. 625–632.

Chang, W. I. and Lampe, J. (1992): Theoretical and Empirical Comparisons of Approximate String Matching Algorithms, *Lecture Notes in Computer Science*, Vol. 644, pp. 172–181.

Chang, W. I. and Lawler, E. L. (1994): Sublinear Approximate String Matching and Biological Applications, *Algorithmica*, Vol. 12, No. 4–5, pp. 327–344.

Chao, H. S. (1992): *On-line algorithms for three computational geometry problems*. Unpublished Ph.D. thesis, National Tsing Hua University, Hsinchu, Taiwan.

Chao, M. T. (1985): *Probabilistic analysis and performance measurement of algorithms for the satisfiability problem*. Unpublished Ph.D. thesis, Case Western Reserve University, Cleveland, Ohio.

Chao, M. T. and Franco, J. (1986): Probabilistic Analysis of Two Heuristics for the 3-Satisfiability Problem, *SIAM Journal on Computing*, Vol. 15, No. 4, pp. 1106–1118.

Charalambous, C. (1997): Partially Observable Nonlinear Risk-Sensitive Control Problems: Dynamic Programming and Verification Theorems, *IEEE Transactions on Automatic Control*, Vol. 42, pp. 1130–1138.

Chazelle, B., Drysdale, R. L. and Lee, D. T. (1986): Computing the Largest Empty Rectangle, *SIAM Journal on Computing*, Vol. 15, No. 1, pp. 300–315.

Chazelle, B., Edelsbrunner, H., Guibas, L., Sharir, M. and Snoeyink, J. (1993): Computing a Face in an Arrangement of Line Segments and Related Problems, *SIAM Journal on Computing*, Vol. 22, No. 6, pp. 1286–1302.

Chazelle, B. and Matousek, J. (1996): On Linear-Time Deterministic Algorithms for Optimization Problems in Fixed Dimension, *Journal of Algorithm*, Vol. 21, pp. 579–597.

Chekuri, C., Khanna, S. and Zhu, A. (2001): Algorithms for Minimizing Weighted Flow Time, *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing*, ACM Press, Hersonissos, Greece, pp. 84–93.

Chen, G. H., Chern, M. S. and Jang, J. H. (1990): Pipeline Architectures for Dynamic Programming Algorithms, *Parallel Computing*, Vol. 13, pp. 111–117.

Chen, G. H., Kuo, M. T. and Sheu, J. P. (1988): An Optimal Time Algorithm for Finding a Maximum Weighted Independent Set in a Tree, *BIT*, Vol. 28, pp. 353–356.

Chen, J. and Miranda, A. (2001): A Polynomial Time Approximation Scheme for General Multiprocessor Job Scheduling, *SIAM Journal on Computing*, Vol. 31, pp. 1–17.

Chen, L. H. Y. (1975): Poisson Approximation for Dependent Trials, *The Annals of Probability*, Vol. 3, pp. 534–545.

Chen, T. S., Yang, W. P. and Lee, R. C. T. (1989): Amortized Analysis of Disk Scheduling Algorithms, *Technical Report*, Department of Information Engineering, National Chiao-Tung University, Taiwan.

Chen, W. M. and Hwang, H. K. (2003): Analysis in Distribution of Two Randomized Algorithms for Finding the Maximum in a Broadcast Communication Model, *Journal of Algorithms*, Vol. 46, pp. 140–177.

Cheriyan, J. and Harerup, T. (1995): Randomized Maximum-Flow Algorithm, *SIAM Journal on Computing*, Vol. 24, No. 2, pp. 203–226.

Chin, W. and Ntafos, S. (1988): Optimum Watchman Routes, *Information Processing Letters*, Vol. 28, No. 1, pp. 39–44.

Chou, H. C. and Chung C. P. (1994): Optimal Multiprocessor Task Scheduling Using Dominance and Equivalence Relations, *Computer & Operations Research*, Vol. 21, No. 4, pp. 463–475.

Choukhmane, E. and Franco, J. (1986): An Approximation Algorithm for the Maximum Independent Set Problem in Cubic Planar Graphs, *Networks*, Vol. 16, No. 4, pp. 349–356.

Christofides, N. (1976): Worst-Case Analysis of a New Heuristic for the Traveling Salesman Problem, *Management Sciences Research Report*, No. 388.

Christos, L. and Drago, K. (1998): Quasi-Greedy Triangulations Approximating the Minimum Weight Triangulation, *Journal of Algorithms*, Vol. 27, No. 2, pp. 303–338.

Chrobak, M. and Larmore, L. L. (1991): An Optimal On-Line Algorithm for *k*-Servers on Trees, *SIAM Journal of Computing*, Vol. 20, No. 1, pp. 144–148.

Chu, C. and La, R. (2001): Variable-Sized Bin Packing: Tight Absolute Worst-Case Performance Ratios for Four Approximation Algorithms, *SIAM Journal on Computing*, Vol. 30, pp. 2069–2083.

Chung, M. J. and Krishnamoorthy, M. S. (1988): Algorithms of Placing Recovery Points, *Information Processing Letters*, Vol. 28, No. 4, pp. 177–181.

Chung, M. J., Makedon, F., Sudborough, I. H. and Turner, J. (1985): Polynomial Time Algorithms for the Min Cut Problem on Degree Restricted Trees, *SIAM Journal on Computing*, Vol. 14, No. 1, pp. 158–177.

Cidon, I., Kutten, S., Mansour, Y. and Peleg, D. (1995): Greedy Packet Scheduling, *SIAM Journal on Computing*, Vol. 24, pp. 148–157.

Clarkson, K. L. (1987): New Applications of Random Sampling to Computational Geometry, *Discrete and Computational Geometry*, Vol. 2, pp. 195–222.

Clarkson, K. L. (1988): A Randomized Algorithm for Closest-Point Queries, *SIAM Journal on Computing*, Vol. 17, No. 4, pp. 830–847.

Clarkson, K. L. (1994): An Algorithm for Approximate Closest-Point Queries, *Proceedings of the 10th Annual Symposium on Computational Geometry*, ACM Press, Stony Brook, New York, pp. 160–164.

Cobbs, A. (1995): Fast Approximate Matching Using Suffix Trees, *Lecture Notes in Computer Science*, Vol. 937, pp. 41–54.

Coffman, E. G., Langston, J. and Langston, M. A. (1984): A Performance Guarantee for the Greedy Set-Partitioning Algorithm, *Acta Informatica*, Vol. 21, pp. 409–415.

Coffman, E. G. and Lueker, G. S. (1991): *Probabilistic Analysis of Packaging & Partitioning Algorithms*, John Wiley & Sons, New York.

Colbourn, C. J., Kocay, W. L. and Stinson, D. R. (1986): Some NP-Complete Problems for Hypergraph Degree Sequences, *Discrete Applied Mathematics*, Vol. 14, No. 3, pp. 239–254.

Cole, R. (1994): Tight Bounds on the Complexity of the Boyer-Moore String Matching Algorithm, *SIAM Journal on Computing*, Vol. 23, No. 5, pp. 1075–1091.

Cole, R., Farach-Colton, M., Hariharan, R., Przytycka, T. and Thorup, M. (2000): An $O(n \log n)$ Algorithm for the Maximum Agreement Subtree Problem for Binary Trees, *SIAM Journal on Applied Mathematics*, Vol. 30, No. 5, pp. 1385–1404.

Cole, R. and Hariharan, R. (1997): Tighter Upper Bounds on the Exact Complexity of String Matching, *SIAM Journal on Computing*, Vol. 26, No. 3, pp. 803–856.

Cole, R., Hariharan, R., Paterson, M. and Zwick, U. (1995): Tighter Lower Bounds on the Exact Complexity of String Matching, *SIAM Journal on Computing*, Vol. 24, No. 1, pp. 30–45.

Coleman, T. F., Edenbrandt, A. and Gilbert, J. R. (1986): Predicting Fill for Sparse Orthogonal Factorization, *Journal of the ACM*, Vol. 33, No. 3, pp. 517–532.

Conn, R. and Vonholdt, R. (1965): An Online Display for the Study of Approximating Functions, *Journal of the ACM*, Vol. 12, No. 3, pp. 326–349.

Cook, S. A. (1971): The Complexity of Theorem Proving Procedures, *Proceedings of the 3rd ACM Symposium on Theory of Computing*, ACM Press, Shaker Heights, Ohio, pp. 151–158.

Cooley, J. W. and Tukey, J. W. (1965): An Algorithm for the Machine Calculation of Complex Fourier Series, *Mathematics of Computation*, Vol. 19, pp. 297–301.

Coppersmith, D., Doyle, P., Raghavan, P. and Snir, M. (1993): Random Walks on Weighted Graphs and Applications to On-line Algorithms, *Journal of the ACM*, Vol. 40, No. 3, pp. 421–453.

Cormen, T. H. (1999): Determining an Out-of-Core FFT Decomposition Strategy for Parallel Disks by Dynamic Programming, *Algorithms for Parallel Processing*, Vol. 105, pp. 307–320.

Cormen, T. H. (2001): *Introduction to Algorithms*, McGraw-Hill, New York.

Cormen, T. H., Leiserson, C. E. and Rivest, R. L. (1990): *Introduction to Algorithms*, McGraw-Hill, New York.

Cornuejols, C., Fisher, M. L. and Nemhauser, G. L. (1977): Location of Bank Accounts to Optimize Float: An Analytic Study of Exact and Approximate Algorithms, *Management Science*, Vol. 23, No. 8, pp. 789–810.

Cowureur, C. and Bresler, Y. (2000): On the Optimality of the Backward Greedy Algorithm for the Subset Selection Problem, *SIAM Journal on Matrix Analysis and Applications*, Vol. 21, pp. 797–808.

Crammer, K. and Singer, Y. (2002): Text Categorization: A New Family of Online Algorithms for Category Ranking, *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, ACM Press, Tampere, Finland, pp. 151–158.

Crescenzi, P., Goldman, D., Papadimitriou, C., Piccolboni, A. and Yannakakis, M. (1998): On the Complexity of Protein Folding, *Journal of Computational Biology*, Vol. 5, No. 3, pp. 423–465.

Csirik, J. (1989): An On-Line Algorithm For Variable-Sized Bin Packing, *Acta Informatica*, Vol. 26, pp. 697–709.

Csur, M. and Kao, M. Y. (2001): Provably Fast and Accurate Recovery of Evolutionary Trees through Harmonic Greedy Triplets, *SIAM Journal on Computing*, Vol. 31, pp. 306–322.

Culberson, J. C. and Rudnicki, P. (1989): A Fast Algorithm for Constructing Trees from Distance Matrices, *Information Processing Letters*, Vol. 30, No. 4, pp. 215–220.

Cunningham, W. H. (1985): Optimal Attack and Reinforcement of a Network, *Journal of the ACM*, Vol. 32, No. 3, pp. 549–561.

Czumaj, A., Gasieniec, L., Piotrow, M. and Rytter, W. (1994): Parallel and Sequential Approximation of Shortest Superstrings, *Lecture Notes in Computer Science*, Vol. 824, pp. 95–106.

d'Amore, F. and Liberatore, V. (1994): List Update Problem and the Retrieval of Sets, *Theoretical Computer Science*, Vol. 130, No. 1, pp. 101–123.

Darve, E. (2000): The Fast Multipole Method I: Error Analysis and Asymptotic Complexity, *SIAM Journal on Numerical Analysis*, Vol. 38, No. 1, pp. 98–128.

Day, W. H. (1987): Computational Complexity of Inferring Phylogenies from Dissimilarity Matrices, *Bulletin of Mathematical Biology*, Vol. 49, No. 4, pp. 461–467.

Decatur, S. E., Goldreich, O. and Ron, D. (1999): Computational Sample Complexity, *SIAM Journal on Computing*, Vol. 29, No. 3.

Dechter, R. and Pearl, J. (1985): Generalized Best-First Search Strategies and the Optimality of A*, *Journal of the ACM*, Vol. 32, No. 3, pp. 505–536.

Delcoigne, A. and Hansen, P. (1975): Sequence Comparison by Dynamic Programming, *Biometrika*, Vol. 62, pp. 661–664.

Denardo, E. V. (1982): *Dynamic Programming: Model and Applications*, Prentice-Hall, Englewood Cliffs, New Jersey.

Deng, X. and Mahajan, S. (1991): Infinite Games: Randomization Computability and Applications to Online Problems, *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, ACM Press, New Orleans, Louisiana, pp. 289–298.

Derfel, G. and Vogl, F. (2000): Divide-and-Conquer Recurrences: Classification of Asymptotics, *Aequationes Mathematicae*, Vol. 60, pp. 243–257.

Devroye, L. (2002): Limit Laws for Sums of Functions of Subtrees of Random Binary Search Trees, *SIAM Journal on Applied Mathematics*, Vol. 32, No. 1, pp. 152–171.

Devroye, L. and Robson, J. M. (1995): On the Generation of Random Binary Search Trees, *SIAM Journal on Applied Mathematics*, Vol. 24, No. 6, pp. 1141–1156.

Dietterich, T. G. (2000): The Divide-and-Conquer Manifesto, *Lecture Notes in Artificial Intelligence*, Vol. 1968, pp. 13–26.

Dijkstra, E. W. (1959): A Note on Two Problems in Connexion with Graphs, *Numerische Mathematik*, Vol. 1, pp. 269–271.

Dinitz, Y. and Nutov, Z. (1999): A 3-Approximation Algorithm for Finding Optimum 4, 5-Vertex-Connected Spanning Subgraphs, *Journal of Algorithms*, Vol. 32, pp. 31–40.

Dixon, B., Rauch, M. and Tarjan, R. E. (1992): Verification and Sensitivity Analysis of Minimum Spanning Trees in Linear Time, *SIAM Journal on Computing*, Vol. 21, pp. 1184–1192.

Dobkin, D. P. and Lipton, R. J. (1979): On the Complexity of Computations Under Varying Sets of Primitives, *Journal of Computer and System Sciences*, Vol. 18, pp. 86–91.

Dolan, A. and Aldus, J. (1993): *Networks and Algorithms: An Introductory Approach*, John Wiley & Sons, New York.

Dolev, D., Lynch, N. A., Pinter, S. S., Stark, E. W. and William, E. W. (1986): Reaching Approximate Agreement in the Presence of Faults, *Journal of the ACM*, Vol. 33, No. 3, pp. 499–516.

Drake, D. E. and Hougardy, S. (2003): A Simple Approximation Algorithm for the Weighted Matching Problem, *Information Processing Letters*, Vol. 85, pp. 211–213.

Dreyfus, S. E. and Law, A. M. (1977): *The Art and Theory of Dynamic Programming*, Academic Press, London.

Du, D. Z. and Book, R. V. (1989): On Inefficient Special Cases of NP-Complete Problems, *Theoretical Computer Science*, Vol. 63, No. 3, pp. 239–252.

Du, J. and Leung, J. Y. T. (1988): Scheduling Tree-Structured Tasks with Restricted Execution Times, *Information Processing Letters*, Vol. 28, No. 4, pp. 183–188.

Dwyer, R. A. (1987): A Faster Divide-and-Conquer Algorithm for Constructing Delaunay Triangulations, *Algorithmics*, Vol. 2, pp. 137–151.

Dyer, M. E. (1984): Linear Time Algorithm for Two- and Three-Variable Linear Programs, *SIAM Journal on Computing*, Vol. 13, No. 1, pp. 31–45.

Dyer, M. E. and Frieze, A. M. (1989): Randomized Algorithm for Fixed-Dimensional Linear Programming, *Mathematical Programming*, Vol. 44, No. 2, pp. 203–212.

Edelsbrunner, H. (1987): *Algorithms in Combinatorial Geometry*, Springer-Verlag, Berlin.

Edelsbrunner, H. and Guibas, L. J. (1989): Topologically Sweeping an Arrangement, *Journal of Computer and System Sciences*, Vol. 38, No. 1, pp. 165–194.

Edelsbrunner, H., Maurer, H. A., Preparata, F. P., Rosenberg, A. L., Welzl, E. and Wood, D. (1982): Stabbing Line Segments, *BIT*, Vol. 22, pp. 274–281.

Edwards, C. S. and Elphick, C. H. (1983): Lower Bounds for the Clique and the Chromatic Numbers of a Graph, *Discrete Applied Mathematics*, Vol. 5, No. 1, pp. 51–64.

Eiter, T. and Veith, H. (2002): On the Complexity of Data Disjunctions, *Theoretical Computer Science*, Vol. 288, No. 1, pp. 101–128.

Ekroot, L. and Dolinar, S. (1996): A* Decoding of Block Codes, *IEEE Transactions on Communications*, pp. 1052–1056.

ElGindy, H., Everett, H. and Toussaint, G. (1993): Slicing an Ear Using Prune-and-Search, *Pattern Recognition Letters*, Vol. 14, pp. 719–722.

El-Yaniv, R. (1998): Competitive Solutions for Online Financial Problems, *ACM Computing Surveys*, Vol. 30, No. 1, pp. 28–69.

El-Zahar, M. H. and Rival, I. (1985): Greedy Linear Extensions to Minimize Jumps, *Discrete Applied Mathematics*, Vol. 11, No. 2, pp. 143–156.

Eppstein, D., Galil, Z., Giancarlo, R. and Italiano, G. F. (1992a): Sparse Dynamic Programming I: Linear Cost Functions, *Journal of the ACM*, Vol. 39, No. 3, pp. 519–545.

Eppstein, D., Galil, Z., Giancarlo, R. and Italiano, G. F. (1992b): Sparse Dynamic Programming II: Convex and Concave Cost Functions, *Journal of the ACM*, Vol. 39, pp. 516–567.

Eppstein, D., Galil, Z., Italiano, G. F. and Spencer, T. H. (1996): Separator Based Sparsication I. Planarity Testing and Minimum Spanning Trees, *Journal of Computer and System Sciences*, Vol. 52, No. 1, pp. 3–27.

Epstein, L. and Ganot, A. (2003): Optimal On-line Algorithms to Minimize Makespan on Two Machines with Resource Augmentation, *Lecture Notes in Computer Science*, Vol. 2909, pp. 109–122.

Epstein, L., Noga, J., Seiden, S., Sgall, J. and Woeginger, G. (1999): Randomized Online Scheduling on Two Uniform Machines, *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, SIAM, Baltimore, Maryland, pp. 317–326.

Erdmann, M. (1993): Randomization for Robot Tasks: Using Dynamic Programming in the Space of Knowledge States, *Algorithmica*, Vol. 10, pp. 248–291.

Erlebach, T. and Jansen, K. (1999): Efficient Implementation of an Optimal Greedy Algorithm for Wavelength Assignment in Directed Tree Networks, *ACM Journal of Experimental Algorithmics*, Vol. 4.

Esko, U. (1990): A Linear Time Algorithm for Finding Approximate Shortest Common Superstrings, *Algorithmica*, Vol. 5, pp. 313–323.

Evans, J. R. and Minieka, E. (1992): *Optimization Algorithms for Networks and Graphs*, 2nd ed., Marcel Dekker, New York.

Even, G., Naor, J., Rao, S. and Schieber, B. (2000): Divide-and-Conquer Approximation Algorithms via Spreading Metrics, *Journal of the ACM*, Vol. 47, No. 4, pp. 585–616.

Even, G., Naor, J. and Zosin, L. (2000): An 8-Approximation Algorithm for the Subset Feedback Vertex Set Problem, *SIAM Journal on Computing*, Vol. 30, pp. 1231–1252.

Even, S. (1987): *Graph Algorithms*, Computer Science Press, Rockville, Maryland.

Even, S., Itai, A. and Shamir, A. (1976): On the Complexity of Timetable and Multicommodity Problems, *SIAM Journal on Computing*, Vol. 5, pp. 691–703.

Even, S. and Shiloach, Y. (1981): An On-Line Edge-Deletion Problem, *Journal of the ACM*, Vol. 28, No. 1, pp. 1–4.

Fagin, R. (1974): Generalized First-Order Spectra and Polynomial-Time Recognizable Sets, *SIAM-AMS Proceedings*, Vol. 7, pp. 43–73.

Faigle, U. (1985): On Ordered Languages and the Optimization of Linear Functions by Greedy Algorithms, *Journal of the ACM*, Vol. 32, No. 4, pp. 861–870.

Faigle, U., Kern, W. and Nawijn, W. (1999): A Greedy On-Line Algorithm for the *k*-Track Assignment Problem, *Journal of Algorithms*, Vol. 31, pp. 196–210.

Farach, M. and Thorup, M. (1997): Sparse Dynamic Programming for Evolutionary Tree Comparison, *SIAM Journal on Computing*, Vol. 26, pp. 210–230.

Farber, M. and Keil, J. M. (1985): Domination in Permutation Graphs, *Journal of Algorithms*, Vol. 6, pp. 309–321.

Feder, T. and Motwani, R. (2002): Worst-Case Time Bounds for Coloring and Satisfiability Problems, *Journal of Algorithms*, Vol. 45, pp. 192–201.

Feige, U. and Krauthgamer, R. (2002): A Polylogarithmic Approximation of the Minimum Bisection, *SIAM Journal on Computing*, Vol. 31, No. 4, pp. 1090–1118.

Fejes, G. (1978): Evading Convex Discs, *Studia Science Mathematics Hungar*, Vol. 13, pp. 453–461.

Feldmann, A., Kao, M., Sgall, J. and Teng, S. H. (1993): Optimal Online Scheduling of Parallel Jobs with Dependencies, *Proceedings of the 25th*

Annual ACM Symposium on Theory of Computing, ACM Press, San Diego, California, pp. 642–651.

Fellows, M. R. and Langston, M. A. (1988): Processor Utilization in a Linearly Connected Parallel Processing System, *IEEE Transactions on Computers*, Vol. 37, pp. 594–603.

Fernandez-Beca, D. and Williams, M. A. (1991): On Matroids and Hierarchical Graphs, *Information Processing Letters*, Vol. 38, No. 3, pp. 117–121.

Ferragina, P. (1997): Dynamic Text Indexing under String Updates, *Journal of Algorithms*, Vol. 22, No. 2, pp. 296–238.

Ferreira, C. E., de Souza, C. C. and Wakabayashi, Y. (2002): Rearrangement of DNA Fragments: A Branch-and-Cut Algorithm (Prove NP-Complete Problem), *Discrete Applied Mathematics*, Vol. 116, pp. 161–177.

Fiat, A. and Woeginger, G. J. (1998): Online Algorithms: The State of the Art, *Lecture Notes in Computer Science*, Vol. 1442.

Fischel-Ghodsian, F., Mathiowitz, G. and Smith, T. F. (1990): Alignment of Protein Sequence Using Secondary Structure: A Modified Dynamic Programming Method, *Protein Engineering*, Vol. 3, No. 7, pp. 577–581.

Flajolet, P. and Prodinger, H. (1986): Register Allocation for Unary-Binary Trees, *SIAM Journal on Computing*, Vol. 15, No. 3, pp. 629–640.

Floyd, R. W. (1962): Algorithm 97: Shortest Path, *Communications of the ACM*, Vol. 5, No. 6, p. 345.

Floyd, R. W. and Rivest, R. L. (1975): Algorithm 489 (SELECT), *Communications of the ACM*, Vol. 18, No. 3, p. 173.

Fonlupt, J. and Nachef, A. (1993): Dynamic Programming and the Graphical Traveling Salesman Problem, *Journal of the Association for Computing Machinery*, Vol. 40, No. 5, pp. 1165–1187.

Foulds, L. R. and Graham, R. L. (1982): The Steiner Problem in Phylogeny is NP-Complete, *Advances Application Mathematics*, Vol. 3, pp. 43–49.

Franco, J. (1984): Probabilistic Analysis of the Pure Literal Heuristic for the Satisfiability Problem, *Annals of Operations Research*, Vol. 1, pp. 273–289.

Frederickson, G. N. (1984): Recursively Rotated Orders and Implicit Data Structures: A Lower Bound, *Theoretical Computer Science*, Vol. 29, pp. 75–85.

Fredman, M. L. (1981): A Lower Bound on the Complexity of Orthogonal Range Queries, *Journal of the ACM*, Vol. 28, No. 4, pp. 696–705.

Fredman, M. L., Sedgewick, R., Sleator, D. D. and Tarjan, R. E. (1986): The Pairing Heap: A New Form of Self-Adjusting Heap, *Algorithmica*, Vol. 1, No. 1, pp. 111–129.

Friesen, D. K. and Kuhl, F. S. (1988): Analysis of a Hybrid Algorithm for Packing Unequal Bins, *SIAM Journal on Computing*, Vol. 17, No. 1, pp. 23–40.

Friesen, D. K. and Langston, M. A. (1986): Variable Sized Bin Packing, *SIAM Journal on Computing*, Vol. 15, No. 1, pp. 222–230.

Frieze, A. M. and Kannan, R. (1991): A Random Polynomial-Time Algorithm for Approximating the Volume of Convex Bodies, *Journal of the ACM*, Vol. 38, No. 1.

Frieze, A. M., McDiarmid, C. and Reed, B. (1990): Greedy Matching on the Line, *SIAM Journal on Computing*, Vol. 19, No. 4, pp. 666–672.

Froda, S. (2000): On Assessing the Performance of Randomized Algorithms, *Journal of Algorithms*, Vol. 31, pp. 344–362.

Fu, H. C. (2001): Divide-and-Conquer Learning and Modular Perceptron Networks, *IEEE Transactions on Neural Networks*, Vol. 12, No. 2, pp. 250–263.

Fu, J. J. and Lee, R. C. T. (1991): Minimum Spanning Trees of Moving Points in the Plane, *IEEE Transactions on Computers*, Vol. 40, No. 1, pp. 113–118.

Galbiati, G., Maffioli, F. and Morrzenti, A. (1994): A Short Note on the Approximability of the Maximum Leaves Spanning Tree Problem, *Information Processing Letters*, Vol. 52, pp. 45–49.

Galil, Z. and Giancarlo, R. (1989): Speeding up Dynamic Programming with Applications to Molecular Biology, *Theoretical Computer Science*, Vol. 64, pp. 107–118.

Galil, Z., Haber, S. and Yung, M. (1989): Minimum-Knowledge Interactive Proofs for Decision Problems, *SIAM Journal on Computing*, Vol. 18, No. 4, pp. 711–739.

Galil, Z., Hoffman, C. H., Luks, E. M., Schnorr, C. P. and Weber, A. (1987): An $O(n^3 \log n)$ Deterministic and an $O(n^3)$ Las Vagas Isomorphism Test for Trivalent Graphs, *Journal of the ACM*, Vol. 34, No. 3, pp. 513–531.

Galil, Z. and Park, K. (1990): An Improved Algorithm for Approximate String Matching, *SIAM Journal on Computing*, Vol. 19, pp. 989–999.

Galil, Z. and Seiferas, J. (1978): A Linear-Time On-Line Recognition Algorithm for "Palstar", *Journal of the ACM*, Vol. 25, No. 1, pp. 102–111.

Galil, Z. and Park, K. (1992): Dynamic Programming with Convexity Concavity and Sparsity, *Theoretical Computer Science*, Vol. 49–76.

Gallant, J., Marier, D. and Storer, J. A. (1980): On Finding Minimal Length Superstrings (Prove NP-Hard Problem), *Journal of Computer and System Sciences*, Vol. 20, pp. 50–58.

Garay, J., Gopal, I., Kutten, S., Mansour, Y. and Yung, M. (1997): Efficient On-Line Call Control Algorithms, *Journal of Algorithms*, Vol. 23, pp. 180–194.

Garey, M. R. and Johnson, D. S. (1976): Approximation algorithms for combinatorial problem: An annotated bibliography. In J. F. Traub (Ed.), *Algorithms and Complexity: New Directions and Recent Results*, Academic Press, New York, pp. 41–52.

Garey, M. R. and Johnson, D. S. (1979): *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, San Francisco, California.

Geiger, D., Gupta, A., Costa, L. A. and Vlontzos, J. (1995): Dynamic Programming for Detecting Tracking and Matching Deformable Contours, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 17, pp. 294–302.

Gelfand, M. S. and Roytberg, M. A. (1993): Prediction of the Exon-Intron Structure by a Dynamic Programming Approach, *BioSystems*, Vol. 30, pp. 173–182.

Gentleman, W. M. and Sande, G. (1966): Fast Fourier Transforms for Fun and Prot, *Proceedings of AFIPS Fall Joint Computer Conference*, Vol. 29, pp. 563–578.

Giancarlo, R. and Grossi, R. (1997): Multi-Dimensional Pattern Matching with Dimensional Wildcards: Data Structures and Optimal On-Line Search Algorithms, *Journal of Algorithms*, Vol. 24, pp. 223–265.

Gilbert, E. N. and Moore, E. F. (1959): Variable Length Encodings, *Bell System Technical Journal*, Vol. 38, No. 4, pp. 933–968.

Gilbert, J. R., Hutchinson, J. P. and Tarjan, R. E. (1984): A Separator Theorem for Graphs of Bounded Genus, *Journal of Algorithms*, Vol. 5, pp. 391–407.

Gill, I. (1987): Computational Complexity of Probabilistic Turing Machine, *SIAM Journal on Computing*, Vol. 16, No. 5, pp. 852–853.

Godbole, S. S. (1973): On Efficient Computation of Matrix Chain Products, *IEEE Transactions on Computers*, Vol. 22, No. 9, pp. 864–866.

Goemans, M. X. and Williamson, D. P. (1995): Improved Approximation Algorithms for Maximum Cut and Satisfiability Problems Using Semidefinite Programming, *Journal of the ACM*, Vol. 42, No. 6, pp. 1115–1145.

Goldberg, L. A., Goldberg, P. W. and Paterson, M. (2001): The Complexity of Gene Placement (Prove NP-Complete Problem), *Journal of Algorithms*, Vol. 41, pp. 225–243.

Goldman, S., Parwatikar, J. and Suri, S. (2000): Online Scheduling with Hard Deadlines, *Journal of Algorithms*, Vol. 34, pp. 370–389.

Goldstein, L. and Waterman, M. S. (1987): Mapping DNA by Stochastic Relaxation (Prove NP-Complete Problem of Double Digest Problem), *Advances in Applied Mathematics*, Vol. 8, pp. 194–207.

Goldwasser, S. and Micali, S. (1984): Probabilistic Encryption, *Journal of Computer and System Sciences*, Vol. 28, No. 2, pp. 270–298.

Goldwasser, S., Micali, S. and Rackoff, C. (1988): The Knowledge Complexity of Interactive Proof Systems, *SIAM Journal on Computing*, Vol. 18, No. 1, pp. 186–208.

Golumbic, M. C. (1980): *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York.

Gonnet, G. H. (1983): *Handbook of Algorithms and Data Structures*, Addison-Wesley, Reading, Mass.

Gonzalez, T. F. and Lee, S. L. (1987): A 1.6 Approximation Algorithm for Routing Multiterminal Nets, *SIAM Journal on Computing*, Vol. 16, No. 4, pp. 669–704.

Gonzalo, N. (2001): A Guide Tour to Approximate String Matching, *ACM*, Vol. 33, pp. 31–88.

Goodman, S. and Hedetniemi, S. (1980): *Introduction to the Design and Analysis of Algorithms*, McGraw-Hill, New York.

Gorodkin, J., Lyngso, R. B. and Stormo, G. D. (2001): A Mini-Greedy Algorithm for Faster Structural RNA Stem-Loop Search, *Genome Informatics*, Vol. 12, pp. 184–193.

Gotieb, L. (1981): Optimal Multi-Way Search Trees, *SIAM Journal on Computing*, Vol. 10, No. 3, pp. 422–433.

Gotieb, L. and Wood, D. (1981): The Construction of Optimal Multiway Search Trees and the Monotonicity Principle, *International Journal of Computer Mathematics*, Vol. 9, pp. 17–24.

Gould, R. (1988): *Graph Theory*, Benjamin Cummings, Redwood City, California.

Graham, R. L. (1972): An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set, *Information Processing Letters*, Vol. 1, pp. 132–133.

Grandjean, E. (1988): A Natural NP-Complete Problem with a Nontrivial Lower Bound, *SIAM Journal on Computing*, Vol. 17, No. 4, pp. 786–809.

Greene, D. H. and Knuth, D. E. (1981): *Mathematics for the Analysis of Algorithms*, Birkhauser, Boston, Mass.

Grigni, M., Koutsoupias, E. and Papadimitriou, C. (1995): Approximation Scheme for Planar Graph TSP, *Foundations of Computer Science*, pp. 640–645.

Grove, E. (1995): Online Bin Packing with Lookahead, *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, SIAM, San Francisco, California, pp. 430–436.

Gudmundsson, J., Levcopoulos, C. and Narasimhan, G. (2002): Fast Greedy Algorithms for Constructing Sparse Geometric Spanners, *SIAM Journal on Computing*, Vol. 31, pp. 1479–1500.

Gueting, R. H. (1984): Optimal Divide-and-Conquer to Compute Measure and Contour for a Set of Iso-Rectangles, *Acta Informatica*, Vol. 21, pp. 271–291.

Gueting, R. H. and Schilling, W. (1987): Practical Divide-and-Conquer Algorithm for the Rectangle Intersection Problem, *Information Sciences*, Vol. 42, No. 2, pp. 95–112.

Gupta, S., Konjevod, G. and Varsamopoulos, G. (2002): A Theoretical Study of Optimization Techniques Used in Registration Area Based Location Management: Models and Online Algorithms, *Proceedings of the 6th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, ACM Press, Atlanta, Georgia, pp. 72–79.

Gusfield, D. (1994): Faster Implementation of a Shortest Superstring Approximation, *Information Processing Letters*, Vol. 51, pp. 271–274.

Gusfield, D. (1997): *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge University Press, Cambridge, England.

Guting, R. H. (1984): Optimal Divide-and-Conquer to Compute Measure and Contour for a Set of Iso-Rectangles, *Acta Informatica*, Vol. 21, pp. 271–291.

Haas, P. and Hellerstein, J. (1999): Ripple Joins for Online Aggregation, ACM SIGMOD Record, *Proceedings of the 1999 ACM-SIGMOD International Conference on Management of Data*, Vol. 28.

Hall, N. G. and Hochbaum, D. S. (1986): A Fast Approximation Algorithm for the Multicovering Problem, *Discrete Applied Mathematics*, Vol. 15, No. 1, pp. 35–40.

Halldorsson, M. (1997): Parallel and On-Line Graph Coloring, *Journal of Algorithms*, Vol. 23, pp. 265–280.

Halperin, D., Sharir, M. and Goldberg, K. (2002): The 2-Center Problem with Obstacles, *Journal of Algorithms*, Vol. 42, pp. 109–134.

Han, Y. S., Hartmann, C. R. P. and Chen, C. C. (1993): Efficient Priority: First Search Maximum-Likelihood Soft-Decision Decoding of Linear Block Codes, *IEEE Transactions on Information Theory*, pp. 1514–1523.

Han, Y. S., Hartmann C. R. P. and. Mehrotra, K. G. (1998): Decoding Linear Block Codes Using a Priority-First Search: Performance Analysis and Suboptimal Version, *IEEE Transactions on Information Theory*, pp. 1233–1246.

Hanson, F. B. (1991): Computational Dynamic Programming on a Vector Multiprocessor, *IEEE Transactions on Automatic Control*, Vol. 36, pp. 507–511.

Hariri, A. M. A. and Potts, C. N. (1983): An Algorithm for Single Machine Sequencing with Release Dates to Minimize Total Weighted Completion Time, *Discrete Applied Mathematics*, Vol. 5, No. 1, pp. 99–109.

Har-Peled, S. (2000): Constructing Planar Cuttings in Theory and Practice, *SIAM Journal on Computing*, Vol. 29, No. 6, pp. 2016–2039.

Hasegawa, M. and Horai, S. (1991): Time of the Deepest Root for Polymorphism in Human Mitochondrial DNA, *Journal of Molecular Evolution*, Vol. 32, pp. 37–42.

Hasham, A. and Sack, J. R. (1987): Bounds for Min-Max Heaps, *BIT*, Vol. 27, pp. 315–323.

Hashimoto, R. F. and Barrera J. (2003): A Greedy Algorithm for Decomposing Convex Structuring Elements, *Journal of Mathematical Imaging and Vision*, Vol. 18, No. 3, pp. 269–286.

Haussmann, U. G. and Suo, W. (1995): Singular Optimal Stochastic Controls. II. Dynamic Programming, *SIAM Journal on Control and Optimization*, Vol. 33, pp. 937–959.

Hayward, R. B. (1987): A Lower Bound for the Optimal Crossing-Free Hamiltonian Cycle Problem, *Discrete and Computational Geometry*, Vol. 2, pp. 327–343.

Hein, J. (1989): An Optimal Algorithm to Reconstruct Trees from Additive Distance Data, *Bulletin of Mathematical Biology*, Vol. 51, pp. 597–603.

Held, M. and Karp, R. M. (1962): A Dynamic Programming Approach to Sequencing Problems, *SIAM Journal on Applied Mathematics*, Vol. 10, pp. 196–210.

Hell, P., Shamir, R. and Sharan, R. (2001): A Fully Dynamic Algorithm for Recognizing and Representing Proper Interval Graphs, *SIAM Journal on Computing*, Vol. 31, pp. 289–305.

Henzinger, M. R. (1995): Fully Dynamic Biconnectivity in Graphs, *Algorithmica*, Vol. 13, No. 6, pp. 503–538.

Hirosawa, M., Hoshida, M., Ishikawa, M. and Toya, T. (1993): MASCOT: Multiple Alignment System for Protein Sequence Based on Three-Way Dynamic Programming, *Computer Applications in the Biosciences*, Vol. 9, pp. 161–167.

Hirschberg, D. S. (1975): A Linear Space Algorithm for Computing Maximal Common Subsequences, *Communications of the ACM*, Vol. 18, No. 6, pp. 341–343.

Hirschberg, D. S. and Larmore, L. L. (1987): The Least Weight Subsequence Problem, *SIAM Journal on Computing*, Vol. 16, No. 4, pp. 628–638.

Hoang, T. M. and Thierauf, T. (2003): The Complexity of the Characteristic and the Minimal Polynomial, *Theoretical Computer Science*, Vol. 1–3, pp. 205–222.

Hoare, C. A. R. (1961): Partition (Algorithm 63), Quicksort (Algorithm 64) and Find (Algorithm 65), *Communications of the ACM*, Vol. 4, No. 7, pp. 321–322.

Hoare, C. A. R. (1962): Quicksort, *Computer Journal*, Vol. 5, No. 1, pp. 10–15.

Hochbaum, D. S. (1997): *Approximation Algorithms for NP-Hard Problems*, PWS Publisher, Boston.

Hochbaum, D. S. and Maass, W. (1987): Fast Approximation Algorithms for a Nonconvex Covering Problem, *Journal of Algorithms*, Vol. 8, No. 3, pp. 305–323.

Hochbaum, D. S. and Shmoys, D. B. (1986): A United Approach to Approximation Algorithms for Bottleneck Problems, *Journal of the ACM*, Vol. 33, No. 3, pp. 533–550.

Hochbaum, D. S. and Shmoys, D. B. (1987): Using Dual Approximation Algorithms for Scheduling Problems: Theoretical and Practical Results, *Journal of the ACM*, Vol. 34, No. 1, pp. 144–162.

Hoffman, A. J. (1988): On Greedy Algorithms for Series Parallel Graphs, *Mathematical Programming*, Vol. 40, No. 2, pp. 197–204.

Hofri, M. (1987): *Probabilistic Analysis of Algorithms*, Springer-Verlag, New York.

Holmes, I. and Durbin, R. (1998): Dynamic Programming Alignment Accuracy, *Journal of Computational Biology*, Vol. 5, pp. 493–504.

Holyer, I. (1981): The NP-Completeness of Some Edge-Partition Problems, *SIAM Journal on Computing*, Vol. 10, pp. 713–717.

Homer, S. (1986): On Simple and Creative Sets in NP, *Theoretical Computer Science*, Vol. 47, No. 2, pp. 169–180.

Homer, S. and Long, T. J. (1987): Honest Polynomial Degrees and P =? NP, *Theoretical Computer Science*, Vol. 51, No. 3, pp. 265–280.

Horowitz, E. and Sahni, S. (1974): Computing Partitions with Applications to the Knapsack Problem, *Journal of the ACM*, Vol. 21, No. 2, pp. 277–292.

Horowitz, E. and Sahni, S. (1976a): Exact and Approximate Algorithms for Scheduling Nonidentical Processors, *Journal of the ACM*, Vol. 23, No. 2, pp. 317–327.

Horowitz, E. and Sahni, S. (1976b): *Fundamentals of Data Structures*, Computer Science Press, Rockville, Maryland.

Horowitz, E. and Sahni, S. (1978): *Fundamentals of Computer Algorithm*, Computer Science Press, Rockville, Maryland.

Horowitz, E., Sahni, S. and Rajasekaran, S. (1998): *Computer Algorithms*, W. H. Freeman, New York.

Horton, J. D. (1987): A Polynomial-Time Algorithm to Find the Shortest Cycle Basis of a Graph, *SIAM Journal on Computing*, Vol. 16, No. 2, pp. 358–366.

Horvath, E. C., Lam, S. and Sethi, R. (1977): A Level Algorithm for Preemptive Scheduling, *Journal of the ACM*, Vol. 24, No. 1, pp. 32–43.

Hsiao, J. Y., Tang, C. Y. and Chang, R. S. (1993): The Summation and Bottleneck Minimization for Single Step Searching on Weighted Graphs, *Information Sciences*, Vol. 74, pp. 1–28.

Hsu, W. L. (1984): Approximation Algorithms for the Assembly Line Crew Scheduling Problem, *Mathematics of Operations Research*, Vol. 9, pp. 376–383.

Hsu, W. L. and Nemhauser, G. L. (1979): Easy and Hard Bottleneck Location Problems, *Discrete Applied Mathematics*, Vol. 1, No. 3, pp. 209–215.

Hu, T. C. and Shing, M. T. (1982): Computation of Matrix Chain Products Part I, *SIAM Journal on Computing*, Vol. 11, No. 2, pp. 362–373.

Hu, T. C. and Shing, M. T. (1984): Computation of Matrix Chain Products Part II, *SIAM Journal of Computing*, Vol. 13, No. 2, pp. 228–251.

Hu, T. C. and Tucker, A. C. (1971): Optimal Computer Search Trees and Variable-Length Alphabetical Codes, *SIAM Journal on Applied Mathematics*, Vol. 21, No. 4, pp. 514–532.

Hu, T. H., Tang, C. Y. and Lee, R. C. T. (1992): An Average Analysis of a Resolution Principle Algorithm in Mechanical Theorem Proving, *Annals of Mathematics and Artificial Intelligence*, Vol. 6, pp. 235–252.

Huang, S. H. S., Liu, H. and Viswanathan, V. (1994): Parallel Dynamic Programming, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No. 3, pp. 326–328.

Huang, X. and Waterman, M. S. (1992): Dynamic Programming Algorithms for Restriction Map Comparison, *Computational Applied Biology Science*, pp. 511–520.

Huddleston, S. and Mehlhorn, K. (1982): A New Data Structure for Representing Sorted Lists, *Acta Informatica*, Vol. 17, pp. 157–184.

Huffman, D. A. (1952): A Method for the Construction of Minimum-Redundancy Codes, *Proceedings of IRE*, Vol. 40, pp. 1098–1101.

Huo, D. and Chang, G. J. (1994): The Provided Minimization Problem in Tree, *SIAM Journal of Computing*, Vol. 23, No. 1, pp. 71–81.

Huyn, N., Dechter, R. and Pearl, J. (1980): Probabilistic Analysis of the Complexity of A*, *Artificial Intelligence*, Vol. 15, pp. 241–254.

Huynh, D. T. (1986): Some Observations about the Randomness of Hard Problems, *SIAM Journal on Computing*, Vol. 15, No. 4, pp. 1101–1105.

Hyafil, L. (1976): Bounds for Selection, *SIAM Journal on Computing*, Vol. 5, No. 1, pp. 109–114.

Ibaraki, T. (1977): The Power of Dominance Relations in Branch-and-Bound Algorithms, *Journal of the ACM*, Vol. 24, No. 2, pp. 264–279.

Ibaraki, T. and Nakamura, Y. (1994): A Dynamic Programming Method for Single Machine Scheduling, *European Journal of Operational Research*, Vol. 76, p. 72.

Ibarra, O. H. and Kim, C. E. (1975): Fast Approximation Algorithms for the Knapsack and the Sum of Subset Problems, *Journal of the ACM*, Vol. 22, pp. 463–468.

Imai, H. (1993): Geometric Algorithms for Linear Programming, *IEICE Transactions on Fundamentals of Electronics Communications and Computer Sciences*, Vol. E76-A, No. 3, pp. 259–264.

Imai, H., Kato, K. and Yamamoto, P. (1989): A Linear-Time Algorithm for Linear L1 Approximation of Points, *Algorithmica*, Vol. 4, No. 1, pp. 77–96.

Imai, H., Lee, D. T. and Yang, C. D. (1992): 1-Segment Center Problem, *ORSA Journal on Computing*, Vol. 4, No. 4, pp. 426–434.

Imase, M. and Waxman, B. M. (1991): Dynamic Steiner Tree Problem, *SIAM Journal on Discrete Mathematics*, Vol. 4, No. 3, pp. 369–384.

Irani, S., Shukla, S. and Gupta, R. (2003): Online Strategies for Dynamic Power Management in Systems with Multiple Power-Saving States, *ACM Transactions on Embedded Computing Systems (TECS)*, Vol. 2, pp. 325–346.

Italiano, G. F. (1986): Amortized Efficiency of a Path Retrieval Data Structure, *Theoretical Computer Science*, Vol. 48, No. 2, pp. 273–281.

Ivanov, A. G. (1984): Distinguishing an Approximate Word's Inclusion on Turing Machine in Real Time, *Izvestiia Academii Nauk USSSR, Series Math*, Vol. 48, pp. 520–568.

Iwamura, K. (1993): Discrete Decision Process Model Involves Greedy Algorithm Over Greedoid, *Journal of Information and Optimization Sciences*, Vol. 14, pp. 83–86.

Jadhavm, S. and Mukhopadhyay, A. (1993): Computing a Centerpoint of a Finite Planar Set of Points in Linear Time, *Proceedings of the 9th Annual Symposium on Computational Geometry*, ACM Press, San Diego, California, pp. 83–90.

Jain, K. and Vazirani, V. V. (2001): Approximation Algorithms for Metric Facility Location and *k*-Median Problems Using the Primal-Dual Schema and Lagrangian Relaxation, *Journal of the ACM*, Vol. 48, No. 2.

Janssen, J., Krizanc, D., Narayanan, L. and Shende, S. (2000): Distributed Online Frequency Assignment in Cellular Networks, *Journal of Algorithms*, Vol. 36, pp. 119–151.

Jayram, T., Kimbrel, T., Krauthgamer, R., Schieber, B. and Sviridenko, M. (2001): Online Server Allocation in a Server Farm via Benefit Task Systems, *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing*, ACM Press, Hersonissos, Greece, pp. 540–549.

Jerrum, M. (1985): The Complexity of Finding Minimum-Length Generator Sequences, *Theoretical Computer Science*, Vol. 36, pp. 265–289.

Jiang, T., Kearney, P. and Li, M. (1998): Orchestrating Quartets: Approximation and Data Correction, *Proceedings of the 39th IEEE Symposium on Foundations of Computer Science*, IEEE Press, Palo Alto, California, pp. 416–425.

Jiang, T., Kearney, P. and Li, M. (2001): A Polynomial Time Approximation Scheme For Inferring Evolutionary Trees from Quartet Topologies and Its Application, *SIAM Journal on Computing*, Vol. 30, No. 6, pp. 1942–1961.

Jiang, T., Lawler, E. L. and Wang, L. (1994): Aligning Sequences via an Evolutionary Tree: Complexity and Approximation, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing*, ACM Press, Montreal, Quebec, pp. 760–769.

Jiang, T. and Li, M. (1995): On the Approximation of Shortest Common Supersequences and Longest Common Subsequences, *SIAM Journal on Computing*, Vol. 24, No. 5, pp. 1122–1139.

Jiang, T., Wang, L. and Lawler, E. L. (1996): Approximation Algorithms for Tree Alignment with a Given Phylogeny, *Algorithmica*, Vol. 16, pp. 302–315.

John, J. W. (1988): A New Lower Bound for the Set-Partitioning Problem, *SIAM Journal on Computing*, Vol. 17, No. 4, pp. 640–647.

Johnson, D. S. (1973): Near-Optimal Bin Packing Algorithms, *MIT Report MAC TR–109*.

Johnson, D. S. (1974): Approximation Algorithms for Combinatorial Problems, *Journal of Computer and System Sciences*, Vol. 9, pp. 256–278.

Johnson, D. S., Demars, A., Ullman, J. D., Garey, M. R. and Graham, R. L. (1974): Worst-Case Performance Bounds for Simple One-Dimensional Packing Algorithms, *SIAM Journal on Computing*, Vol. 3, No. 4, pp. 299–325.

Johnson, D. S., Yanakakis, M. and Papadimitriou, C. H. (1988): On Generating All Maximal Independent Sets, *Information Processing Letters*, Vol. 27, No. 3, pp. 119–123.

Johnson, H. W. and Burrus, C. S. (1983): The Design of Optimal DFT Algorithms Using Dynamic Programming, *IEEE Transactions on Acoustics Speech and Signal Processing*, Vol. 31, No. 2, pp. 378–387.

Jonathan, T. (1989): Approximation Algorithms for the Shortest Common Superstring Problem, *Information and Computation*, Vol. 83, pp. 1–20.

Jorma, T. and Ukkonen, E. (1988): A Greedy Approximation Algorithm for Constructing Shortest Common Superstrings, *Theoretical Computer Science*, Vol. 57, pp. 131–145.

Juedes, D. W. and Lutz, J. H. (1995): The Complexity and Distribution of Hard Problems, *SIAM Journal of Computing*, Vol. 24, No. 2, pp. 279–295.

Kalyanasundaram, B. and Pruhs, K. (1993): On-Line Weighted Matching, *Journal of Algorithms*, Vol. 14, pp. 478–488.

Kamidoi, Y., Wakabayashi, S. and Yoshida, N. (2002): A Divide-and-Conquer Approach to the Minimum k-Way Cut Problem, *Algorithmica*, Vol. 32, pp. 262–276.

Kannan, R., Mount, J. and Tayur, S. (1995): A Randomized Algorithm to Optimize Over Certain Convex Sets, *Mathematical Operating Research*, Vol. 20, No. 3, pp. 529–549.

Kannan, S., Lawler, E. L. and Warnow, T. J. (1996): Determining the Evolutionary Tree Using Experiments, *Journal of Algorithms*, Vol. 21, pp. 26–50.

Kannan, S. and Warnow, T. (1994): Inferring Evolutionary History from DNA Sequences, *SIAM Journal on Computing*, Vol. 23, pp. 713–737.

Kannan, S. and Warnow, T. (1995): Tree Reconstruction from Partial Orders, *SIAM Journal on Computing*, Vol. 24, pp. 511–519.

Kantabutra, V. (1994): Linear-Time Near-Optimum-Length Triangulation Algorithm for Convex Polygons, *Journal of Computer and System Sciences*, Vol. 49, No. 2, pp. 325–333.

Kao, E. P. C. and Queyranne, M. (1982): On Dynamic Programming Methods for Assembly Line Balancing, *Operations Research*, Vol. 30, No. 2, pp. 375–390.

Kao, M. Y., Ma, Y., Sipser, M. and Yin, Y. (1998): Optimal Constructions of Hybrid Algorithms, *Journal of Algorithms*, Vol. 29, pp. 142–164.

Kao, M. Y. and Tate, S. R. (1991): Online Matching with Blocked Input, *Information Processing Letters*, Vol. 38, pp. 113–116.

Kaplan, H. and Shamir, R. (1994): On the Complexity of DNA Physical Mapping, *Advances in Applied Mathematics*, Vol. 15, pp. 251–261.

Karger, D. R., Klein, P. N. and Tarjan, R. E. (1995): Randomized Linear-Time Algorithm to Find Minimum Spanning Trees, *Journal of the Association for Computing Machinery*, Vol. 42, No. 2, pp. 321–328.

Karger, D. R., Phillips, S. and Torng, E. (1994): A Better Algorithm for an Ancient Scheduling Problem, *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, Arlington, Virginia, pp. 132–140.

Karger, D. R. and Stein, C. (1996): New Approach to the Minimum Cut Problem, *Journal of the Association for Computing Machinery*, Vol. 43, No. 4, pp. 601–640.

Karkkainen, J., Navarro, G. and Ukkonen, E. (2000): Approximate String Matching over Ziv-Lempel Compressed Text, *Lecture Notes in Computer Science*, Vol. 1848, pp. 195–209.

Karlin, A. R., Manasse, M. S., Rudolph, L. and Sleator, D. D. (1988): Competitive Snoopy Caching, *Algorithmica*, Vol. 3, No. 1, pp. 79–119.

Karoui, N. E. and Quenez, M. C. (1995): Dynamic Programming and Pricing of Contingent Claims in an Incomplete Market, *SIAM Journal on Control and Optimization*, 1995, pp. 27–66.

Karp, R. M. (1972): Reducibility among Combinatorial Problems, in R. Miller and J. Thatcher (Eds.), *Complexity of Computer Computations* (pp. 85–103), Plenum Press, New York.

Karp, R. M. (1986): Combinatorics; Complexity and Randomness, *Communications of the ACM*, Vol. 29, No. 2, 1986, pp. 98–109.

Karp, R. M. (1994): Probabilistic Recurrence Relations, *Journal of the Association for Computing Machinery*, Vol. 41, No. 6, 1994, pp. 1136–1150.

Karp, R. M., Montwani, R. and Raghavan, P. (1988): Deferred Data Structuring, *SIAM Journal on Computing*, Vol. 17, No. 5, 1988, pp. 883–902.

Karp, R. M. and Pearl, J. (1983): Searching for an Optimal Path in a Tree with Random Costs, *Artificial Intelligence*, Vol. 21, 1983, pp. 99–116.

Karp, R. M. and Rabin, M. O. (1987): Efficient Randomized Pattern-Matching Algorithms, *IBM Journal of Research and Development*, Vol. 31, 1987, pp. 249–260.

Karp, R. M., Vazirani, U. V. and Vazirani, V. V. (1990): An Optimal Algorithm for On-Line Bipartite Matching, *Proceedings of the 22nd ACM Symposium on Theory of Computing*, ACM Press, Baltimore, Maryland, pp. 352–358.

Kearney, P., Hayward, R. B. and Meijer, H. (1997): Inferring Evolutionary Trees from Ordinal Data, *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, SIAM, New Orleans, Louisiana, pp. 418–426.

Kececioglu, J. D. (1991): *Exact and approximate algorithms for sequence recognition problems in molecular biology*. Unpublished Ph.D. thesis, University of Arizona.

Kececioglu, J. D. and Myers, W. E. (1995a): Exact and Approximation Algorithms for the Sequence Reconstruction Problem, *Algorithmica*, Vol. 13, pp. 7–51.

Kececioglu, J. D. and Myers, W. E. (1995b): Combinatorial Algorithms for DNA Sequence Assembly, *Algorithmica*, Vol. 13, pp. 7–51.

Kececioglu, J. D. and Sankoff, D. (1993): Exact and Approximation Algorithms for the Inversion Distance between Two Chromosomes, *Lecture Notes in Computer Science*, Vol. 684, pp. 87–105.

Kececioglu, J. D. and Sankoff, D. (1995): Exact and Approximate Algorithms for Sorting by Reversals with Application to Genome Rearrangement, *Algorithmica*, Vol. 13, pp. 180–210.

Keogh, E., Chu, S., Hart, D. and Pazzani, M. (2001): An Online Algorithm for Segmenting Time Series, *IEEE Computer Science Press*, pp. 289–296.

Khachian, L. G. (1979): A Polynomial Algorithm for Linear Programming, *Doklady Akademii Nauk*, USSR, Vol. 244, No. 5, pp. 1093–1096. Translated in *Soviet Math. Doklady*, Vol. 20, pp. 191–194.

Khuller, S., Mitchell, S. and Vazirani, V. V. (1994): On-Line Algorithms for Weighted Bipartite Matching and Stable Marriage, *Theoretical Computer Science*, Vol. 127, No. 2, pp. 255–267.

Kilpelainen P. and Mannila H. (1995): Ordered and Unordered Tree Inclusion, *SIAM Journal on Computing*, Vol. 24, No. 2, pp. 340–356.

Kim, D. S., Yoo, K. H., Chwa, K. Y. and Shin, S. Y. (1998): Efficient Algorithms for Computing a Complete Visibility Region in Three-Dimensional Space, *Algorithmica*, Vol. 20, pp. 201–225.

Kimura, M. (1979): The Neutral Theory of Molecular Evolution, *Scientific American*, Vol. 241, pp. 98–126.

King, T. (1992): *Dynamic Data Structures: Theory and Applications*, Academic Press, London.

Kingston, J. H. (1986): The Amortized Complexity of Henriksen's Algorithm, *BIT*, Vol. 26, No. 2, pp. 156–163.

Kirousis, L. M. and Papadimitriou, C. H. (1988): The Complexity of Recognizing Polyhedral Scenes, *Journal of Computer and System Sciences*, Vol. 37, No. 1, pp. 14–38.

Kirpatrick, D. and Snoeyink, J. (1993): Tentative Prune-and-Search for Computing Fixed-Points with Applications to Geometric Computation, *Proceedings of the Ninth Annual Symposium on Computational Geometry*, ACM Press, San Diego, California, pp. 133–142.

Kirpatrick, D. and Snoeyink, J. (1995): Tentative Prune-and-Search for Computing Fixed-Points with Applications to Geometric Computation, *Fundamenta Informaticae*, Vol. 22, pp. 353–370.

Kirschenhofer, P., Prodinger, H. and Szpankowski, W. (1994): Digital Search Trees Again Revisited: The Internal Path Length Perspective, *SIAM Journal on Applied Mathematics*, Vol. 23, No. 3, pp. 598–616.

Klarlund, N. (1999): An *n* log *n* Algorithm for Online BDD Refinement, *Journal of Algorithms*, Vol. 32, pp. 133–154.

Klawe, M. M. (1985): A Tight Bound for Black and White Pebbles on the Pyramid, *Journal of the ACM*, Vol. 32, No. 1, pp. 218–228.

Kleffe, J. and Borodovsky, M. (1992): First and Second Moment of Counts of Words in Random Texts Generated by Markov Chains, *Computer Applications in the Biosciences*, Vol. 8, pp. 433–441.

Klein, C. M. (1995): A Submodular Approach to Discrete Dynamic Programming, *European Journal of Operational Research*, Vol. 80, pp. 145–155.

Klein, P. and Subramanian, S. (1997): A Randomized Parallel Algorithm for Single-Source Shortest Paths, *Journal of Algorithm*, Vol. 25, pp. 205–220.

Kleinberg, J. and Tardos, E. (2002): Approximation Algorithms for Classification Problems with Pairwise Relationships: Metric Labeling and Markov Random Fields, *Journal of the ACM*, Vol. 49, No. 5, pp. 616–639.

Knuth, D. E. (1969): The Art of Computer Programming, *Fundamental Algorithms*, Vol. 1, p. 634.

Knuth, D. E. (1971): Optimum Binary Search Trees, *Acta Informatica*, Vol. 1, pp. 14–25.

Knuth, D. E. (1973): *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, Mass.

Ko, M. T., Lee, R. C. T. and Chang, J. S. (1990): An Optimal Approximation Algorithm for the Rectilinear *m*-Center Problem, *Algorithmica*, Vol. 5, pp. 341–352.

Kolliopoulos, S. G. and Stein, C. (2002): Approximation Algorithms for Single-Source Unsplittable Flow, *SIAM Journal on Computing*, Vol. 31, No. 3, pp. 919–946.

Kolman, P. and Scheideler, C. (2001): Simple On-Line Algorithms for the Maximum Disjoint Paths Problem, *Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures*, ACM Press, Crete Island, Greece, pp. 38–47.

Kontogiannis, S. (2002): Lower Bounds and Competitive Algorithms for Online Scheduling of Unit-Size Tasks to Related Machines, *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, ACM Press, Montreal, Canada, pp. 124–133.

Koo, C. Y., Lam, T. W., Ngan, T. W., Sadakane, K. and To, K. K. (2003): On-line Scheduling with Tight Deadlines, *Theoretical Computer Science*, Vol. 295, 2003, pp. 1–12.

Korte, B. and Louasz, L. (1984): Greedoids: A Structural Framework for the Greedy Algorithms, in W. R. Pulleybland (Ed.), *Progress in Combinatorial Optimization* (pp. 221–243), Academic Press, London.

Kortsarz, G. and Peleg, D.(1995): Approximation Algorithms for Minimum-Time Broadcast, *SIAM Journal on Discrete Mathematics*, Vol. 8, pp. 407–421.

Kossmann, D., Ramsak, F. and Rost, S. (2002): Shooting Stars in the Sky: An Online Algorithm for Skyline Queries, *Proceedings of the 28th VLDB Conference*, VLDB Endowment, Hong Kong, pp. 275–286.

Kostreva, M. M. and Wiecek, M. M. (1993): Time Dependency in Multiple Objective Dynamic Programming, *Journal of Mathematical Analysis and Applications*, Vol. 173, pp. 289–307.

Kou, L., Markowsky, G. and Berman, L. (1981): A Fast Algorithm for Steiner Trees, *Acta Informatica*, Vol. 15, pp. 141–145.

Koutsoupias, E. and Nanavati, A. (2003): Online Matching Problem on a Line, *Lecture Notes in Computer Science*, Vol. 2909, pp. 179–191.

Koutsoupias, E. and Papadimitriou, C. H. (1995): On the *k*-Server Conjecture, *Journal of the ACM*, Vol. 42, No. 5, pp. 971–983.

Kozen, D. C. (1997): *The Design and Analysis of Algorithms*, Springer-Verlag, New York.

Kozhukhin, C. G. and Pevzner, P. A. (1994): Genome Inhomogeneity Is Determined Mainly by WW and SS Dinucleotides, *Computer Applications in the Biosciences*, pp. 145–151.

Krarup, J. and Pruzan, P. (1986): Assessment Approximate Algorithms: The Error Measures's Crucial Role, *BIT*, Vol. 26, No. 3, pp. 284–294.

Krivanek, M. and Moravek, J. (1986): NP-Hard Problems in Hierarchical-Tree Clustering, *Acta Informatica*, Vol. 23, No. 3, pp. 311–323.

Krogh, A., Brown, M., Mian, I. S., Sjolander, K. and Haussler, D. (1994): Hidden Markov Models in Computational Biology: Applications to Protein Modeling, *Journal of Molecular Biology*, Vol. 235, pp. 1501–1531.

Kronsjö, L. I. (1987): *Algorithms: Their Complexity and Efficiency*, John Wiley & Sons, New York.

Krumke, S. O., Marathe, M. V. and Ravi, S. S. (2001): Models and Approximation Algorithms for Channel Assignment in Radio Networks, *Wireless Networks*, Vol. 7, No. 6, pp. 575–584.

Kruskal, J. B. Jr. (1956): On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem, *Proceedings of the American Mathematical Society*, Vol. 7, No. 1, pp. 48–50.

Kryazhimskiy, A. V. and Savinov V. B. (1995): Travelling-Salesman Problem with Moving Objects, *Journal of Computer and System Sciences*, Vol. 33, No. 3, pp. 144–148.

Kucera, L. (1991): *Combinatorial Algorithms*, IOP Publishing, Philadelphia.

Kumar, S., Kiran, R. and Pandu, C. (1987): Linear Space Algorithms for the LCS Problem, *Acta Informatica*, Vol. 24, No. 3, pp. 353–362.

Kung, H. T., Luccio, F. and Preparata, F. P. (1975): On Finding the Maxima of a Set of Vectors, *Journal of the ACM*, Vol. 22, No. 4, pp. 469–476.

Kurtz, S. A. (1987): A Note on Randomized Polynomial Time, *SIAM Journal on Computing*, Vol. 16, No. 5, pp. 852–853.

Lai, T. W. and Wood, D. (1998): Adaptive Heuristics for Binary Search Trees and Constant Linkage Cost, *SIAM Journal on Applied Mathematics*, Vol. 27, No. 6, pp. 1564–1591.

Landau, G. M. and Schmidt, J. P. (1993): An Algorithm for Approximate Tandem Repeats, *Lecture Notes in Computer Science*, Vol. 684, pp. 120–133.

Landau, G. M. and Vishkin, U. (1989): Fast Parallel and Serial Approximate String Matching, *Journal of Algorithms*, Vol. 10, pp. 157–169.

Langston, M. A. (1982): Improved 0=1-Interchange Scheduling, *BIT*, Vol. 22, pp. 282–290.

Lapaugh, A. S. (1980): Algorithms for Integrated Circuit Layout: Analytic Approach, *Computer Science*, pp. 155–169.

Laquer, H. T. (1981): Asymptotic Limits for a Two-Dimensional Recursion, *Studies in Applied Mathematics*, pp. 271–277.

Larson, P. A. (1984): Analysis of Hashing with Chaining in the Prime Area, *Journal of Algorithms*, Vol. 5, pp. 36–47.

Lathrop, R. H. (1994): The Protein Threading Problem with Sequence Amino Acid Interaction Preferences Is NP-Complete (Prove NP-Complete Problem), *Protein Engineering*, Vol. 7, pp. 1059–1068.

Lau, H. T. (1991): *Algorithms on Graphs*, TAB Books, Blue Ridge Summit, Philadelphia.

Lawler, E. L. (1976): *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York.

Lawler, E. L., Lenstra, J. K., Rinnooy Kan, A. H. G. and Shmoys, D. B. (1985): *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, John Wiley & Sons, New York.

Lawler, E. L. and Moore, J. (1969): A Functional Equation and Its Application to Resource Allocation and Sequencing Problems, *Management Science*, Vol. 16, No. 1, pp. 77–84.

Lawler, E. L. and Wood, D. (1966): Branch-and-Bound Methods: A Survey, *Operations Research*, Vol. 14, pp. 699–719.

Lee, C. C. and Lee, D. T. (1985): A Simple On-Line Bin-Packing Algorithm, *Journal of the Association for Computing Machinery*, Vol. 32, No. 3, pp. 562–572.

Lee, C. T. and Sheu, C. Y. (1992): A Divide-and-Conquer Approach with Heuristics of Motion Planning for a Cartesian Manipulator, *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 22, No. 5, pp. 929–944.

Lee, D. T. (1982): On *k*-Nearest Neighbor Voronoi Diagrams in the Plane, *IEEE Transactions on Computers*, Vol. C–31, pp. 478–487.

Lee, D. T. and Lin, A. K. (1986): Computational Complexity of Art Gallery Problems, *IEEE Transactions on Information Theory*, Vol. IT–32, No. 2, pp. 276–282.

Lee, D. T. and Preparata, F. P. (1984): Computational Geometry: A Survey, *IEEE Transactions on Computers*, Vol. C–33, pp. 1072–1101.

Lee, J. (2003a): Online Deadline Scheduling: Team Adversary and Restart, *Lecture Notes in Computer Science*, Vol. 2909, pp. 206–213.

Lee, J. (2003b): Online Deadline Scheduling: Multiple Machines and Randomization, *Proceedings of the 15th Annual ACM Symposium on Parallel Algorithms and Architectures*, ACM Press, San Diego, California, pp. 19–23.

Leighton, T. and Rao, S. (1999): Multicommodity Max-Flow Min-Cut Theorems and Their Use in Designing Approximation Algorithms, *Journal of the ACM*, Vol. 46, No. 6, pp. 787–832.

Lent, J. and Mahmoud, H. M. (1996): On Tree-Growing Search Strategies, *The Annals of Applied Probability*, Vol. 6, No. 4, pp. 1284–1302.

Leonardi, S., Spaccamela, A. M., Presciutti, A. and Ros, A. (2001): On-Line Randomized Call Control Revisited, *SIAM Journal on Computing*, Vol. 31, pp. 86–112.

Leoncini, M., Manzini, G. and Margara, L. (1999): Parallel Complexity of Numerically Accurate Linear System Solvers, *SIAM Journal on Computing*, Vol. 28, No. 6, pp. 2030–2058.

Levcopoulos, C. and Lingas, A. (1987): On Approximation Behavior of the Greedy Triangulation for Convex Polygons, *Algorithmica*, Vol. 2, pp. 175–193.

Levin, L. A. (1986): Average Case Complete Problem, *SIAM Journal on Computing*, Vol. 15, No. 1, pp. 285–286.

Lew, W. and Mahmoud, H. M. (1992): The Joint Distribution of Elastic Buckets in Multiway Search Trees, *SIAM Journal on Applied Mathematics*, Vol. 23, No. 5, pp. 1050–1074.

Lewandowski, G., Condon, A. and Bach, E. (1996): Asynchronous Analysis of Parallel Dynamic Programming Algorithms, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, pp. 425–438.

Lewis, H. R. and Denenberg, L. (1991): *Data Structures and Their Algorithms*, Harper Collins, New York.

Liang, S. Y. (1985): *Parallel algorithm for a personnel assignment problem*. Unpublished Master's thesis, National Tsing Hua University, Hsinchu, Taiwan.

Liang, Y. D. (1994): On the Feedback Vertex Set Problem in Permutation Graphs, *Information Processing Letters*, Vol. 52, No. 3, pp. 123–129.

Liao, L. Z. and Shoemaker, C. A. (1991): Convergence in Unconstrained Discrete-time Differential Dynamic Programming, *IEEE Transactions Automatic Control*, Vol. 36, pp. 692–706.

Liaw, B. C. and Lee, R. C. T. (1994): Optimal Algorithm to Solve the Minimum Weakly Cooperative Guards Problem for 1-Spiral Polygons, *Information Processing Letters*, Vol. 52, No. 2, pp. 69–75.

Lin, C. K., Fan, K. C. and Lee, F. T. (1993): On-line Recognition by Deviation-Expansion Model and Dynamic Programming Matching, *Pattern Recognition*, Vol. 26, No. 2, pp. 259–268.

Lin, G., Chen, Z. Z., Jiang, T. and Wen, J. (2002): The Longest Common Subsequence Problem for Sequences with Nested Arc Annotations (Prove NP-Complete Problem), *Journal of Computer and System Sciences*, Vol. 65, pp. 465–480.

Lipton, R. J. (1995): Using DNA to Solve NP-Complete Problems, *Science*, Vol. 268, pp. 542–545.

Little, J. D. C., Murty, K. G., Sweeney, D. W. and Karel, C. (1963): An Algorithm for the Traveling Salesman Problem, *Operations Researc*h, Vol. 11, pp. 972–989.

Littman, M. L., Cassandra, A. R. and Kaelbling, L. P. (1996): An Algorithm for Probabilistic Planning: Efficient Dynamic-Programming Updates in Partially Observable Markov Decision Processes, *Artificial Intelligence*, Vol. 76, pp. 239–286.

Liu, C. L. (1985): *Elements of Discrete Mathematic*s, McGraw-Hill, New York.

Liu, J. (2002): On Adaptive Agentlets for Distributed Divide-and-Conquer: A Dynamical Systems Approach, *IEEE Transactions on Systems*, *Man and Cybernetics*, Vol. 32, No. 2, pp. 214–227.

Lo, V., Rajopadhye, S., Telle, J. A. and Zhong, X. (1996): Parallel Divide and Conquer on Meshes, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 7, No. 10, pp. 1049–1058.

Long, T. J. and Selman, A. L. (1986): Relativizing Complexity Classes with Sparse Oracles, *Journal of the ACM*, Vol. 33, No. 3, pp. 618–627.

Lopez, J. and Zapata, E. (1994): Unified Architecture for Divide and Conquer Based Tridiagonal System Solvers, *IEEE Transactions on Computers*, Vol. 43, No. 12, pp. 1413–1425.

Louchard, G., Szpankowski, W. and Tang, J. (1999): Average Profile of the Generalized Digital Search Tree and the Generalized Lempel—Ziv Algorithm, *SIAM Journal on Applied Mathematics*, Vol. 28, No. 3, pp. 904–934.

Lovasz, L., Naor, M., Newman, I. and Wigderson, A. (1995): Search Problems in the Decision Tree Model, *SIAM Journal on Applied Mathematics*, Vol. 8, No. 1, pp. 119–132.

Luby, M. (1986): A Simple Parallel Algorithm for the Maximal Independent Set Problem, *SIAM Journal on Computing*, Vol. 15, No. 4, pp. 1036–1053.

Lueker, G. (1998): Average-Case Analysis of Off-Line and On-Line Knapsack Problems, *Journal of Algorithms*, Vol. 29, pp. 277–305.

Lyngso, R. B. and Pedersen, C. N. S. (2000): Pseudoknots in RNA Secondary Structure (Prove NP-Complete Problem), *ACM*, pp. 201–209.

Ma, B., Li, M. and Zhang, L. (2000): From Gene Trees to Species Trees, *SIAM Journal on Applied Mathematics*, Vol. 30, No. 3, pp. 729–752.

Maes, M. (1990): On a Cyclic String-to-String Correction Problem, *Information Processing Letters*, Vol. 35, pp. 73–78.

Maffioli, F. (1986): Randomized Algorithm in Combinatorial Optimization: A Survey, *Discrete Applied Mathematics*, Vol. 14, No. 2, June, pp. 157–170.

Maggs, B. M. and Sitaraman, R. K. (1999): Simple Algorithms for Routing on Butterfly Networks with Bounded Queues, *SIAM Journal on Computing*, Vol. 28, pp. 984–1003.

Maier, D. (1978): The Complexity of Some Problems on Subsequences and Supersequences, *Journal of the ACM*, Vol. 25, pp. 322–336.

Maier, D. and Storer, J. A. (1978): A Note on the Complexity of the Superstring Problem, *Proceedings of the 12th Conference on Information Sciences and Systems (CISS)*, The Johns Hopkins University, Baltimore, Maryland, pp. 52–56.

Makinen, E. (1987): On Top-Down Splaying, *BIT*, Vol. 27, No. 3, pp. 330–339.

Manacher, G. (1975): A New Linear-Time "On-line" Algorithm for Finding the Smallest Initial Palindrome of a String, *Journal of the ACM*, Vol. 22, No. 3, pp. 346–351.

Manasse, M., McGeoch, L. and Sleator, D. (1990): Competitive Algorithms for Server Problems, *Journal of Algorithms*, Vol. 11, No. 2, pp. 208–230.

Manber, U. (1989): *Introduction to Algorithms: A Creative Approach*, Addison-Wesley, Reading, Mass.

Mandic, D. and Cichocki, A. (2003): An Online Algorithm for Blind Extraction of Sources with Different Dynamical Structures, *Proceedings of the 4th International Symposium on Independent Component Analysis and Blind Signal Separation (ICA2003)*, Nara, Japan, pp. 645–650.

Mandrioli, D. and Ghezzi, C. (1987): *Theoretical Foundations of Computer Science*, John Wiley & Sons, New York.

Maniezzo, V. (1998): Exact and Approximate Nondeterministic Tree-Search Procedures for the Quadratic Assignment Problem, *Research Report CSR 98–1*.

Mansour, Y. and Schieber, B. (1992): The Intractability of Bounded Protocols for On-line Sequence Transmission over Non-FIFO Channels, *Journal of the ACM*, Vol. 39, No. 4, pp. 783–799.

Marion, J. Y. (2003): Analysing the Implicit Complexity of Programs, *Information and Computation*, Vol. 183, No. 1, pp. 2–18.

Martello, S. and Toth, P. (1990): *Knapsack Problem Algorithms & Computer Implementations*, John Wiley & Sons, New York.

Martin, G. L. and Talley, J. (1995): Recognizing Handwritten Phrases from U. S. Census Forms by Combining Neural Networks and Dynamic Programming, *Journal of Artificial Neural Networks*, Vol. 2, pp. 167–193.

Martinez, C. and Roura, S. (2001): Optimal Sampling Strategies in Quicksort and Quickselect, *SIAM Journal on Computing*, Vol. 31, No. 3, pp. 683–705.

Matousek, J. (1991): Randomized Optimal Algorithm for Slope Selection, *Information Processing Letters*, Vol. 39, No. 4, pp. 183–187.

Matousek, J. (1995): On Enclosing *K* Points by a Circle, *Information Processing Letters*, Vol. 53, No. 4, pp. 217–221.

Matousek, J. (1996): Derandomization in Computational Geometry, *Journal of Algorithms*, Vol. 20, pp. 545–580.

Mauri, G., Pavesi, G. and Piccolboni, A. (1999): Approximation Algorithms for Protein Folding Prediction, *Proceedings of the 10th Annual Symposium on Discrete Algorithms*, SIAM, Baltimore, Maryland, pp. 945–946.

McDiarmid, C. (1988): Average-Case Lower Bounds for Searching, *SIAM Journal on Computing*, Vol. 17, No. 5, pp. 1044–1060.

McHugh, J. A. (1990): *Algorithmic Graph Theory*, Prentice-Hall, London.

Meacham, C. A. (1981): A Probability Measure for Character Compatibility, *Mathematical Biosciences*, Vol. 57, pp. 1–18.

Megiddo, N. (1983): Linear-Time Algorithm for Linear Programming in R3 and Related Problems, *SIAM Journal on Computing*, Vol. 12, No. 4, pp. 759–776.

Megiddo, N. (1984): Linear Programming in Linear Time When the Dimension Is Fixed, *Journal of the ACM*, Vol. 31, No. 1, pp. 114–127.

Megiddo, N. (1985): Note Partitioning with Two Lines in the Plane, *Journal of Algorithms*, Vol. 6, No. 3, pp. 430–433.

Megiddo, N. and Supowit, K. J. (1984): On the Complexity of Some Common Geometric Location Problems, *SIAM Journal on Computing*, Vol. 13, No. 1, pp. 182–196.

Megiddo, N. and Zemel, E. (1986): An $O(n \log n)$ Randomizing Algorithm for the Weighted Euclidean 1-Center Problem, *Journal of Algorithms*, Vol. 7, No. 3, pp. 358–368.

Megow, N. and Schulz, A. (2003): Scheduling to Minimize Average Completion Time Revisited: Deterministic On-Line Algorithms, *Lecture Notes in Computer Science*, Vol. 2909, pp. 227–234.

Mehlhorn, K. (1984): *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*, Springer-Verlag, Berlin.

Mehlhorn, K. (1984): *Data Structures and Algorithms 3: Multi-Dimensional Searching and Computational Geometry*, Springer-Verlag, Berlin.

Mehlhorn, K. (1987): *Data Structures & Algorithms: Sorting and Searching*, Springer-Verlag, New York.

Mehlhorn, K. (1988): A Faster Approximation Algorithm for the Steiner Problems in Graphs, *Information Processing Letters*, Vol. 27, No. 3, pp. 125–128.

Mehlhorn, K., Naher, S. and Alt, H. (1988): A Lower Bound on the Complexity of the Union-Split-Find Problem, *SIAM Journal on Computing*, Vol. 17, No. 6, pp. 1093–1102.

Mehlhorn, K. and Tsakalidis, A. (1986): An Amortized Analysis of Insertions into AVL-Trees, *SIAM Journal on Computing*, Vol. 15, No. 1, pp. 22–33.

Meijer, H. and Rappaport, D. (1992): Computing the Minimum Weight Triangulation of a Set of Linearly Ordered Points, *Information Processing Letters*, Vol. 42, No. 1, pp. 35–38.

Meleis, W. M. (2001): Dual-Issue Scheduling for Binary Trees with Spills and Pipelined Loads, *SIAM Journal on Applied Mathematics*, Vol. 30, No. 6, pp. 1921–1941.

Melnik, S. and Garcia-Molina, H. (2002): Divide-and-Conquer Algorithm for Computing Set Containment Joins, *Lecture Notes in Computer Science*, Vol. 2287, pp. 427–444.

Merlet, N. and Zerubia, J. (1996): New Prospects in Line Detection by Dynamic Programming, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 18, pp. 426–431.

Messinger, E., Rowe, A. and Henry, R. (1991): A Divide-and-Conquer Algorithm for the Automatic Layout of Large Directed Graphs, *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 21, No. 1, pp. 1–12.

Miller, G. L. and Teng, S. H. (1999): The Dynamic Parallel Complexity of Computational Circuits, *SIAM Journal on Computing*, Vol. 28, No. 5, pp. 1664–1688.

Minoux, M. (1986): *Mathematical Programming: Theory and Algorithms*, John-Wiley & Sons, New York.

Mitten, L. (1970): Branch-and-Bound Methods: General Formulation and Properties, *Operations Research*, Vol. 18, pp. 24–34.

Mohamed, M. and Gader, P. (1996): Handwritten Word Recognition Using Segmentation-Free Hidden Markov Modeling and Segmentation-Based Dynamic Programming Techniques, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 18, pp. 548–554.

Monien, B. and Sudborough, I. H. (1988): Min Cut Is NP-Complete for Edge Weighted Trees, *Theoretical Computer Science*, Vol. 58, No. 1–3, pp. 209–229.

Monier, L. (1980): Combinatorial Solutions of Multidimensional Divide-and-Conquer Recurrences, *Journal of Algorithms*, Vol. 1, pp. 69–74.

Moor, O. de (1994): Categories Relations and Dynamic Programming, *Mathematical Structures in Computer Science*, Vol. 4, pp. 33–69.

Moran, S. (1981): General Approximation Algorithms for Some Arithmetical Combinatorial Problems, *Theoretical Computer Science*, Vol. 14, pp. 289–303.

Moran, S., Snir, M. and Manber, U. (1985): Applications of Ramsey's Theorem to Decision Tree Complexity, *Journal of the ACM*, Vol. 32, pp. 938–949.

Moret, B. M. E. and Shapiro, H. D. (1991): *Algorithms from P to NP*, Benjamin Cummings, Redwood City, California.

Morin, T. and Marsten, R. E. (1976): Branch-and-Bound Strategies for Dynamic Programming, *Operations Research*, Vol. 24, pp. 611–627.

Motta, M. and Rampazzo, F. (1996): Dynamic Programming for Nonlinear Systems Driven by Ordinary and Impulsive Controls, *SIAM Journal on Control and Optimization*, Vol. 34, pp. 199–225.

Motwani, R. and Raghavan, P. (1995): *Randomized Algorithms*, Cambridge University Press, Cambridge, England.

Mulmuley, K. (1998): *Computational Geometry: An Introduction through Randomized Algorithms*, Prentice-Hall, Englewoods Cliffs, New Jersey.

Mulmuley, K., Vazirani, U. V. and Vazirani, V. V. (1987): Matching Is as Easy as Matrix Inversion, *Combinatorica*, Vol. 7, No. 1, pp. 105–113.

Murgolo, F. D. (1987): An Efficient Approximation Scheme for Variable-Sized Bin Packing, *SIAM Journal on Computing*, Vol. 16, No. 1, pp. 149–161.

Myers, E. and Miller, W. (1989): Approximate Matching of Regular Expression, *Bulletin of Mathematical Biology*, Vol. 51, pp. 5–37.

Myers, E. W. (1994): A Sublinear Algorithm for Approximate Keyword Searching, *Algorithmica*, Vol. 12, No. 4–5, pp. 345–374.

Myoupo, J. F. (1992): Synthesizing Linear Systolic Arrays for Dynamic Programming Problems, *Parallel Processing Letters*, Vol. 2, pp. 97–110.

Nakayama, H., Nishizeki, T. and Saito, N. (1985): Lower Bounds for Combinatorial Problems on Graphs, *Journal of Algorithms*, Vol. 6, pp. 393–399.

Naor, M. and Ruah, S. (2001): On the Decisional Complexity of Problems Over the Reals, *Information and Computation*, Vol. 167, No. 1, pp. 27–45.

Nau, D. S., Kumar, V. and Kanal, L. (1984): General Branch and Bound and Its Relation to A* and AO*, *Artificial Intelligence*, Vol. 23, pp. 29–58.

Neapolitan, R. E. and Naimipour, K. (1996): *Foundations of Algorithms*, D.C. Heath and Company, Lexington, Mass.

Neddleman, S. B. and Wunsch, C. D. (1970): A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins, *Journal of Molecular Biology*, Vol. 48, pp. 443–453.

Nemhauser, G. L. (1966): *Introduction to Dynamic Programming*, John Wiley & Sons, New York.

Nemhauser, G. L. and Ullman, Z. (1969) Discrete Dynamic Programming and Capital Allocation, *Management Science*, Vol. 15, No. 9, pp. 494–505.

Neogi, R. and Saha, A. (1995): Embedded Parallel Divide-and-Conquer Video Decompression Algorithm and Architecture for HDTV Applications, *IEEE Transactions on Consumer Electronics*, Vol. 41, No. 1, pp. 160–171.

Ney, H. (1984): The Use of a One-Stage Dynamic Programming Algorithm for Connected Word Recognition, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. 2, pp. 263–271.

Ney, H. (1991): Dynamic Programming Parsing for Context-Free Grammars in Continuous Speech Recognition, *IEEE Transactions on Signal Processing*, Vol. 39, pp. 336–340.

Nilsson, N. J. (1980): *Principles of Artificial Intelligence*, Tioga Publishing Company, Palo Alto, California.

Nishizeki, T., Asano, T. and Watanabe, T. (1983): An Approximation Algorithm for the Hamiltonian Walk Problem on a Maximal Planar Graph, *Discrete Applied Mathematics*, Vol. 5, No. 2, pp. 211–222.

Nishizeki, T. and Chiba, N. (1988): *Planar Graphs: Theory and Algorithms*, Elsevier, Amsterdam.

Novak, E. and Wozniakowski, H. (2000): Complexity of Linear Problems with a Fixed Output Basis, *Journal of Complexity*, Vol. 16, No. 1, pp. 333–362.

Nuyts, J., Suetens, P., Oosterlinck, A., Roo, M. De and Mortelmans, L. (1991): Delineation of ECT Images Using Global Constraints and Dynamic Programming, *IEEE Transactions on Medical Imaging*, Vol. 10, No. 4, pp. 489–498.

Ohno, S., Wolf, U. and Atkin, N. B. (1968): Evolution from Fish to Mammals by Gene Duplication, *Hereditas*, Vol. 59, pp. 708–713.

Ohta, Y. and Kanade, T. (1985): Stereo by Intra- and Inter-Scanline Search Using Dynamic Programming, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 7, pp. 139–154.

Oishi, Y. and Sugihara, K. (1995): Topology-Oriented Divide-and-Conquer Algorithm for Voronoi Diagrams, *Graphical Models and Image Processing*, Vol. 57, No. 4, pp. 303–314.

Omura, J. K. (1969): On the Viterbi Decoding Algorithm, *IEEE Transactions on Information Theory*, pp. 177–179.

O'Rourke, J. (1987): *Art Gallery Theorems and Algorithms*, Oxford University Press, Cambridge, England.

O'Rourke, J. (1998): *Computational Geometry in C*, Cambridge University Press, Cambridge, England.

Orponen, P. and Mannila, H. (1987): On Approximation Preserving Reductions: Complete Problems and Robust Measures, *Technical Report C–1987–28*.

Ouyang, Z. and Shahidehpour, S. M. (1992): Hybrid Artificial Neural Network-Dynamic Programming Approach to Unit Commitment, *IEEE Transactions on Power Systems*, Vol. 7, pp. 236–242.

Owolabi, O. and McGregor, D. R. (1988): Fast Approximate String Matching, *Software Practice and Experience*, Vol. 18, pp. 387–393.

Oza, N. and Russell, S. (2001): Experimental Comparisons of Online and Batch Versions of Bagging and Boosting, *Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ACM Press, San Francisco, California, pp. 359–364.

Ozden, M. (1988): A Solution Procedure for General Knapsack Problems with a Few Constraints, *Computers and Operations Research*, Vol. 15, No. 2, pp. 145–156.

Pach, J. (1993): *New Trends in Discrete and Computational Geometry*, Springer-Verlag, New York.

Pacholski, L., Szwast, W. and Tendera, L. (2000): Complexity Results for First-Order Two-Variable Logic with Counting, *SIAM Journal on Computing*, Vol. 29, No. 4, pp. 1083–1117.

Pandurangan, G. and Upfal, E. (2001): Can Entropy Characterize Performance of Online Algorithms? *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, SIAM, Washington, DC, pp. 727–734.

Papadimitriou, C. H. (1977): The Euclidean TSP is NP-Complete, *Theoretical Computer Science*, Vol. 4, pp. 237–244.

Papadimitriou, C. H. (1981): Worst-Case and Probabilistic Analysis of a Geometric Location Problem, *SIAM Journal on Computing*, Vol. 10, No. 3, pp. 542–557.

Papadimitriou, C. H. (1994): *Computational Complexity*, Addison-Wesley, Reading, Mass.

Papadimitriou, C. H. and Steiglitz, K. (1982): *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, New Jersey.

Papadimitriou, C. H. and Yannakakis, M. (1991a): Optimization; Approximation; and Complexity Classes, *Journal of Computer and System Sciences*, Vol. 43, pp. 425–440.

Papadimitriou, C. H. and Yannakakis, M. (1991b): Shortest Paths without a Map, *Theoretical Computer Science*, Vol. 84, pp. 127–150.

Papadimitriou, C. H. and Yannakakis, M. (1992): The Traveling Salesman Problem with Distances One and Two, *Mathematics of Operations Research*, Vol. 18, No. 1, pp. 1–11.

Papadopoulou, E. and Lee, D. T. (1998): A New Approach for the Geodesic Voronoi Diagram of Points in a Simple Polygon and Other Restricted Polygonal Domains, *Algorithmica*, Vol. 20, pp. 319–352.

Parida, L., Floratos, A. and Rigoutsos, I. (1999): An Approximation Algorithm for Alignment of Multiple Sequences Using Motif Discovery, *Journal of Combinatorial Optimization*, Vol. 3, No. 2–3, pp. 247–275.

Park, J. K. (1991): Special Case of the *n*-Vertex Traveling-Salesman Problem That Can Be Solved in *O*(*n*) Time, *Information Processing Letters*, Vol. 40, No. 5, pp. 247–254.

Parker, R. G. and Rardin, R. L. (1984): Guaranteed Performance Heuristic for the Bottleneck Traveling Salesperson Problem, *Operations Research Letters*, Vol. 2, No. 6, pp. 269–272.

Pearl, J. (1983): Knowledge Versus Search: A Quantitative Analysis Using A*, *Artificial Intelligence*, Vol. 20, pp. 1–13.

Pearson, W. R. and Miller, W. (1992): Dynamic Programming Algorithms for Biological Sequence Comparison, *Methods in Enzymology*, Vol. 210, pp. 575–601.

Pe'er, I. and Shamir, R. (1998): The Median Problems for Breakpoints are NP-Complete, *Electronic Colloquium on Computational Complexity*, Vol. 5, No. 71, pp. 1–15.

Pe'er, I. and Shamir, R. (2000): Approximation Algorithms for the Median Problem in the Breakpoint Model, in D. Sankoff and J. H. Nadeau (Eds.), *Comparative Genomics: Empirical and Analytical Approaches to Gene Order Dynamics, Map Alignment and the Evolution of Gene Families*, Kluwer Academic Press, Dordrecht, The Netherlands.

Peleg, D. and Rubinovich, V. (2000): A Near-Tight Lower Bound on the Time Complexity of Distributed Minimum-Weight Spanning Tree Construction, *SIAM Journal on Applied Mathematics*, Vol. 30, No. 5, pp. 1427–1442.

Peng, S., Stephens, A. B. and Yesha, Y. (1993): Algorithms for a Core and *k*-Tree Core of a Tree, *Journal of Algorithms*, Vol. 15, pp. 143–159.

Penny D., Hendy, M. D. and Steel, M. (1992): Progress with Methods for Constructing Evolutionary Trees, *Trends in Ecology and Evolution*, Vol. 7, No. 3, pp. 73–79.

Perl, Y. (1984): Optimum Split Trees, *Journal of Algorithms*, Vol. 5, pp. 367–374.

Peserico, E. (2003): Online Paging with Arbitrary Associativity, *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, SIAM, Baltimore, Maryland, pp. 555–564.

Petr, S. (1996): A Tight Analysis of the Greedy Algorithm for Set Cover, *ACM*, pp. 435–441.

Pevzner, P. A. (1992): Multiple Alignment; Communication Cost; and Graph Matching, *SIAM Journal on Applied Mathematics*, Vol. 52, No. 6, pp. 1763–1779.

Pevzner, P. A. (2000): *Computational Molecular Biology: An Algorithmic Approach*, The MIT Press, Boston.

Pevzner, P. A. and Waterman, M. S. (1995): Multiple Filtration and Approximate Pattern Matching, *Algorithmica*, Vol. 13, No. 1–2, pp. 135–154.

Pierce, N. A. and Winfree, E. (2002): Protein Design Is NP-Hard (Prove NP-Complete Problem), *Protein Engineering*, Vol. 15, No. 10, pp. 779–782.

Pittel, B. and Weishaar, R. (1997): On-Line Coloring of Sparse Random Graphs and Random Trees, *Journal of Algorithms*, Vol. 23, pp. 195–205.

Pohl, I. (1972): A Sorting Problem and Its Complexity, *Communications of the ACM*, Vol. 15, No. 6, pp. 462–463.

Ponzio, S. J., Radhakrishnan, J. and Venkatesh, S. (2001): The Communication Complexity of Pointer Chasing, *Journal of Computer and System Sciences*, Vol. 62, No. 2, pp. 323–355.

Preparata, F. P. and Hong, S. J. (1977): Convex Hulls of Finite Sets of Points in Two and Three Dimensions, *Communications of the ACM*, Vol. 2, No. 20, pp. 87–93.

Preparata, F. P. and Shamos, M. I. (1985): *Computational Geometry: An Introduction*, Springer-Verlag, New York.

Prim, R. C. (1957): Shortest Connection Networks and Some Generalizations, *Bell System Technical Journal*, pp. 1389–1401.

Promel, H. J. and Steger, A. (2000): A New Approximation Algorithm for the Steiner Tree Problem with Performance Ratio 5/3, *Journal of Algorithms*, Vol. 36, pp. 89–101.

Purdom, P. W. Jr. and Brown, C. A. (1985a): *The Analysis of Algorithms*, Holt, Rinehart and Winston, New York.

Purdom, P. W. Jr. and Brown, C. A. (1985b): The Pure Literal Rule and Polynomial Average Time, *SIAM Journal on Computing*, Vol. 14, No. 4, pp. 943–953.

Rabin, M. O. (1976): Probabilistic Algorithm. In J. F. Traub (Ed.), *Algorithms and Complexity: New Directions and Recent Results* (pp. 21–39), Academic Press, New York.

Raghavachari, B. and Veerasamy, J. (1999): A 3/2-Approximation Algorithm for the Mixed Postman Problem, *SIAM Journal on Discrete Mathematics*, Vol. 12, No. 4, pp. 425–433.

Raghavan, P. (1988): Probabilistic Construction of Deterministic Algorithms: Approximating Packing Integer Programs, *Journal of Computer and System Sciences*, Vol. 37, No. 2, pp. 130–143.

Raghavan, P. and Thompson C. (1987): Randomized Rounding: A Technique for Provably Good Algorithms and Algorithmic Proofs, *Combinatorica*, Vol. 7, No. 4, pp. 365–374.

Ramanan, P., Deogun, J. S. and Liu, C. L. (1984): A Personnel Assignment Problem, *Journal of Algorithms*, Vol. 5, No. 1, pp. 132–144.

Ramesh, H. (1995): On Traversing Layered Graphs On-Line, *Journal of Algorithms*, Vol. 18, pp. 480–512.

Rangan, C. P. (1983): On the Minimum Number of Additions Required to Compute a Quadratic Form, *Journal of Algorithms*, Vol. 4, pp. 282–285.

Reingold, E. M., Nievergelt, J. and Deo, N. (1977): *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, Englewood Cliffs, New Jersey.

Reingold, E. M. and Supowit, K. J. (1983): Probabilistic Analysis of Divide-and-Conquer Heuristics for Minimum Weighted Euclidean Matching, *Networks*, Vol. 13, No. 1, pp. 49–66.

Rival, I. and Zaguia, N. (1987): Greedy Linear Extensions with Constraints, *Discrete Mathematics*, Vol. 63, No. 2, pp. 249–260.

Rivas, E. and Eddy, S. R. (1999): A Dynamic Programming Algorithm for RNA Structure Prediction Including Pseudoknots, *Journal of Molecular Biology*, Vol. 285, pp. 2053–2068.

Rivest, L. R. (1995): *Game Tree Searching by Min/Max Approximation*, MIT Laboratory for Computer Science, Cambridge, Mass.

Robinson, D. F. and Foulds, L. R. (1981): Comparison of Phylogenetic Tree, *Mathematical Biosciences*, Vol. 53, pp. 131–147.

Robinson, J. A. (1965): Machine Oriented Logic Based on the Resolution Principle, *Journal of the ACM*, Vol. 12, No. 1, pp. 23–41.

Rosenkrantz, D. J., Stearns, R. E. and Lewis, P. M. (1977): An Analysis of Several Heuristics for the Traveling Salesman Problem, *SIAM Journal on Computing*, Vol. 6, pp. 563–581.

Rosenthal, A. (1982): Dynamic Programming Is Optimal for Nonserial Optimization Problems, *SIAM Journal on Computing*, Vol. 11, No. 1, pp. 47–59.

Rosler, U. (2001): On the Analysis of Stochastic Divide and Conquer Algorithms, *Algorithmica*, Vol. 29, pp. 238–261.

Rosler, U. and Ruschendorf, L. (2001): The Contraction Method for Recursive Algorithms, *Algorithmica*, Vol. 29, pp. 3–33.

Roura, S. (2001): Improved Master Theorems for Divide-and-Conquer Recurrences, *Journal of the ACM*, Vol. 48, No. 2, pp. 170–205.

Rzhetsky, A. and Nei, M. (1992): A Simple Method for Estimating and Testing Minimum-Evolution Tree, *Molecular Biology and Evolution*, Vol. 9, pp. 945–967.

Rzhetsky, A. and Nei, M. (1992): Statistical Properties of the Ordinary Least-Squares; Generalized Least-Squares; and Minimum-Evolution Methods of Phylogenetic Inference, *Journal of Molecular Evolution*, Vol. 35, pp. 367–375.

Sahni, S. (1976): Algorithm for Scheduling Independent Tasks, *Journal of the ACM*, Vol. 23, No. 1, pp. 116–127.

Sahni, S. (1977): General Techniques for Combinatorial Approximation, *Operations Research*, Vol. 25, pp. 920–936.

Sahni, S. and Gonzalez, T. (1976): P-Complete Approximation Problems, *Journal of the ACM*, Vol. 23, pp. 555–565.

Sahni, S. and Wu, S. Y. (1988): Two NP-Hard Interchangeable Terminal Problems, *IEEE Transactions on CAD*, Vol. 7, No. 4, pp. 467–471.

Sakoe, H. and Chiba, S. (1978): Dynamic Programming Algorithm Optimization for Spoken Word Recognition, *IEEE Transactions on Acoustics Speech and Signal Processing*, Vol. 27, pp. 43–49.

Santis, A. D. and Persiano, G. (1994): Tight Upper and Lower Bounds on the Path Length of Binary Trees, *SIAM Journal on Applied Mathematics*, Vol. 23, No. 1, pp. 12–24.

Sarrafzadeh, M. (1987): Channel-Routing Problem in the Knock-Knee Mode Is NP-Complete, *IEEE Transactions on CAD*, Vol. CAD–6, No. 4, pp. 503–506.

Schmidt, J. (1998): All Highest Scoring Paths in Weighted Grid Graphs and Their Application to Finding All Approximate Repeats in Strings, *SIAM Journal on Computing*, Vol. 27, No. 4, pp. 972–992.

Schwartz, E. S. (1964): An Optimal Encoding with Minimum Longest Code and Total Number of Digits, *Information and Control*, Vol. 7, No. 1, pp. 37–44.

Sedgewick, R. and Flajolet, D. (1996): *An Introduction to the Analysis of Algorithms*, Addison-Wesley, Reading, Mass.

Seiden, S. (1999): A Guessing Game and Randomized Online Algorithms, *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, ACM Press, Portland, Oregon, pp. 592–601.

Seiden, S. (2002): On the Online Bin Packing Problem, *Journal of the ACM*, Vol. 49, No. 5, September, pp. 640–671.

Sekhon, G. S. (1982): Dynamic Programming Interpretation of Construction-Type Plant Layout Algorithms and Some Results, *Computer Aided Design*, Vol. 14, No. 3, pp. 141–144.

Sen, S and Sherali, H. D. (1985): A Branch and Bound Algorithm for Extreme Point Mathematical Programming Problem, *Discrete Applied Mathematics*, Vol. 11, No. 3, pp. 265–280.

Setubal, J. and Meidanis, J. (1997): *Introduction to Computational Biology*, PWS Publishing, Boston, Mass.

Sgall, J. (1996): Randomized On-Line Scheduling of Parallel Jobs, *Journal of Algorithms*, Vol. 21, pp. 149–175.

Shaffer, C. A. (2001): *A Practical Introduction to Data Structures and Algorithm Analysis*, Prentice-Hall, Englewood Cliffs, New Jersey.

Shamos, M. I. (1978): *Computational geometry*. Unpublished Ph.D. dissertation, Yale University.

Shamos, M. I. and Hoey, D. (1975): Closest-Point Problems, *Proceedings of the Sixteenth Annual IEEE Symposium on Foundations of Computer Science*, IEEE Press, Washington, DC, pp. 151–162.

Shamos, M. I. and Hoey, D. (1976): Geometric Intersection Problems, *Proceedings of the Seventeenth Annual IEEE Symposium on Foundations of Computer Science*, IEEE Press, Washington, DC, pp. 208–215.

Shmueli, O. and Itai, A. (1987): Complexity of Views: Tree and Cyclic Schemas, *SIAM Journal on Computing*, Vol. 16, No. 1, pp. 17–37.

Shreesh, J., Asish, M. and Binay, B. (1996): An Optimal Algorithm for the Intersection Radius of a Set of Convex Polygons, *Journal of Algorithms*, Vol. 20, No. 2, pp. 244–267.

Simon, R. and Lee, R. C. T. (1971): On the Optimal Solutions to AND/OR Series-Parallel Graphs, *Journal of the ACM*, Vol. 18, No. 3, pp. 354–372.

Slavik, P. (1997): A Tight Analysis of the Greedy Algorithm for Set Cover, *Journal of Algorithms*, Vol. 25, pp. 237–254.

Sleator, D. D. and Tarjan, R. E. (1983): A Data Structure for Dynamic Trees, *Journal of Computer and System Sciences*, Vol. 26, No. 3, pp. 362–391.

Sleator, D. D. and Tarjan, R. E. (1985a): Amortized Efficiency of List Update and Paging Rules, *Communications of the ACM*, Vol. 28, No. 2, pp. 202–208.

Sleator, D. D. and Tarjan, R. E. (1985b): Self-Adjusting Binary Search Trees, *Journal of the ACM*, Vol. 32, No. 3, pp. 652–686.

Sleator, D. D. and Tarjan, R. E. (1986): Self-Adjusting Heaps, *SIAM Journal on Computing*, Vol. 15, No. 1, February, pp. 52–69.

Smith, D. (1984): Random Trees and the Analysis of Branch-and-Bound Procedures, *Journal of the ACM*, Vol. 31, No. 1, pp. 163–188.

Smith, J. D. (1989): *Design and Analysis of Algorithms*, PWS Publishing, Boston, Mass.

Snyder, E. E. and Stormo, G. D. (1993): Identification of Coding Regions in Genomic DNA Sequences: An Application of Dynamic Programming and Neural Networks, *Nucleic Acids Research*, Vol. 21, No. 3, pp. 607–613.

Solovay, R. and Strassen, V. (1977): A Fast Monte-Carlo Test for Primality, *SIAM Journal on Computing*, Vol. 6, No. 1, pp. 84–85.

Spirakis, P. (1988): Optimal Parallel Randomized Algorithm for Sparse Addition and Identication, *Information and Computation*, Vol. 76, No. 1, pp. 1–12.

Srimani, P. K. (1989): Probabilistic Analysis of Output Cost of a Heuristic Search Algorithm, *Information Sciences*, Vol. 47, pp. 53–62.

Srinivasan, A. (1999): Improved Approximation Guarantees for Packing and Covering Integer Programs, *SIAM Journal on Computing*, Vol. 29, pp. 648–670.

Srinivasan, A. and Teo, C. P. (2001): A Constant-Factor Approximation Algorithm for Packet Routing and Balancing Local vs. Global Criteria, *SIAM Journal on Computing*, Vol. 30, pp. 2051–2068.

Steel, M. A. (1992): The Complexity of Reconstructing Trees from Qualitative Characters and Subtrees, *Journal of Classification*, Vol. 9, pp. 91–116.

Steele, J. M. (1986): An Efron-Stein Inequality for Nonsymmetric Statistics, *Annals of Statistics*, Vol. 14, pp. 753–758.

Stewart, G. W. (1999): The QLP Approximation to the Singular Value Decomposition, *SIAM Journal on Scientific Computing*, Vol. 20, pp. 1336–1348.

Stoneking, M., Jorde, L. B., Bhatia, K. and Wilson, A. C. (1990): Geographic Variation in Human Mitochondrial DNA from Papua New Guinea, *Genetics*, Vol. 124, pp. 717–733.

Storer, J. A. (1977): NP-Completeness Results Concerning Data Compression (Prove NP-Complete Problem), *Technical Report 233*, Princeton University, Princeton, New Jersey.

Strassen, V. (1969): Gaussian Elimination Is Not Optimal, *Numerische Mathematik*, Vol. 13, pp. 354–356.

Stringer, C. B. and Andrews, P. (1988): Genetic and Fossil Evidence for the Origin of Modern Humans, *Science*, Vol. 239, pp. 1263–1268.

Sutton, R. S. (1990): Integrated Architectures for Learning, Planning and Reacting Based on Approximating Dynamic Programming, *Proceedings of the Seventh International Conference on Machine Learning*, Morgan Kaufmann Publishers Inc., San Francisco, California, pp. 216–224.

Sweedyk, E. S. (1995): *A 2 1/2 approximation algorithm for shortest common superstring*. Unpublished Ph.D. thesis, University of California.

Sykora, O. and Vrto, I. (1993): Edge Separators for Graphs of Bounded Genus with Applications, *Theoretical Computer Science*, Vol. 112, No. 2, pp. 419–429.

Szpankowski, W. (2001): *Average Case Analysis of Algorithms on Sequences*, John Wiley & Sons, New York.

Tamassia, R. (1996): On-line Planar Graph Embedding, *Journal of Algorithms*, Vol. 21, pp. 201–239.

Tang, C. Y., Buehrer, D. J. and Lee, R. C. T. (1985): On the Complexity of Some Multi-Attribute File Design Problems, *Information Systems*, Vol. 10, No. 1, pp. 21–25.

Tarhio, J. and Ukkonen, E. (1986): A Greedy Algorithm for Constructing Shortest Common Superstrings, *Lecture Notes in Computer Science*, Vol. 233, pp. 602–610.

Tarhio, J. and Ukkonen, E. (1988): A Greedy Approximation Algorithm for Constructing Shortest Common Superstrings, *Theoretical Computer Science*, Vol. 57, pp. 131–145.

Tarjan, R. E. (1983): Data Structures and Network Algorithms, *SIAM*, Vol. 29.

Tarjan, R. E. (1985): Amortized Computational Complexity, *SIAM Journal on Algebraic Discrete Methods*, Vol. 6, No. 2, pp. 306–318.

Tarjan, R. E. (1987): Algorithm Design, *Communications of the ACM*, Vol. 30, No. 3, pp. 204–213.

Tarjan, R. E. and Van Leeuwen, J. (1984): Worst Case Analysis of Set Union Algorithms, *Journal of the ACM*, Vol. 31, No. 2, pp. 245–281.

Tarjan, R. E. and Van Wyk, C. J. (1988): An $O(n \log \log n)$-Time Algorithm for Triangulating a Simple Polygon, *SIAM Journal on Computing*, Vol. 17, No. 1, pp. 143–178.

Tataru, D. (1992): *Viscosity Solutions for the Dynamic Programming Equations, Applied Mathematics and Optimization*, Vol. 25, pp. 109–126.

Tatman, J. A. and Shachter, R. D. (1990): Dynamic Programming and Influence Diagrams, *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 20, pp. 365–379.

Tatsuya, A. (2000): Dynamic Programming Algorithms for RNA Secondary Structure Prediction with Pseudoknots, *Discrete Applied Mathematics*, Vol. 104, pp. 45–62.

Teia, B. (1993): Lower Bound for Randomized List Update Algorithms, *Information Processing Letters*, Vol. 47, No. 1, pp. 5–9.

Teillaud, M. (1993): *Towards Dynamic Randomized Algorithms in Computational Geometry*, Springer-Verlag, New York.

Thomassen, C. (1997): On the Complexity of Finding a Minimum Cycle Cover of a Graph, *SIAM Journal on Computing*, Vol. 26, No. 3, pp. 675–677.

Thulasiraman, K. and Swamy, M. N. S. (1992): *Graphs: Theory and Algorithms*, John Wiley & Sons, New York.

Tidball, M. M. and Atman, E. (1996): Approximations in Dynamic Zero-Sum Games I, *SIAM Journal on Control and Optimization*, Vol. 34, No. 1, pp. 311–328.

Ting, H. F. and Yao, A. C. (1994): Randomized Algorithm for Finding Maximum with $O((\log n)^2)$, *Information Processing Letters*, Vol. 49, No. 1, pp. 39–43.

Tisseur, F. and Dongarra, J. (1999): A Parallel Divide and Conquer Algorithm for the Symmetric Eigenvalue Problem on Distributed Memory Architectures, *SIAM Journal on Scientific Computing*, Vol. 20, No. 6, pp. 2223–2236.

Tomasz, L. (1998): A Greedy Algorithm Estimating the Height of Random Trees, *SIAM Journal on Discrete Mathematics*, Vol. 11, pp. 318–329.

Tong, C. S. and Wong, M. (2002): Adaptive Approximate Nearest Neighbor Search for Fractal Image Compression, *IEEE Transactions on Image Processing*, Vol. 11, No. 6, pp. 605–615.

Traub, J. F. and Wozniakowski, H. (1984): On the Optimal Solution of Large Linear Systems, *Journal of the ACM*, Vol. 31, No. 3, pp. 545–549.

Trevisan, L. (2001): Non-Approximability Results for Optimization Problems on Bounded Degree Instances, *ACM*, pp. 453–461.

Tsai, C. J. and Katsaggelos, A. K. (1999): Dense Disparity Estimation with a Divide-and-Conquer Disparity Space Image Technique, *IEEE Transactions on Multimedia*, Vol. 1, No. 1, pp. 18–29.

Tsai, K. H. and Hsu, W. L. (1993): Fast Algorithms for the Dominating Set Problem on Permutation Graphs, *Algorithmica*, Vol. 9, No. 6, pp. 601–614.

Tsai, K. H. and Lee, D. T. (1997): K Best Cuts for Circular-Arc Graphs, *Algorithmica*, Vol. 18, pp. 198–216.

Tsai, Y. T., Lin, Y. T. and Hsu, F. R. (2002): The On-Line First-Fit Algorithm for Radio Frequency Assignment Problem, *Information Processing Letters*, Vol. 84, No. 4, pp. 195–199.

Tsai, Y. T. and Tang, C. Y. (1993): The Competitiveness of Randomized Algorithms for Online Steiner Tree and On-Line Spanning Tree Problems, *Information Processing Letters*, Vol. 48, pp. 177–182.

Tsai, Y. T., Tang, C. Y. and Chen, Y. Y. (1994): Average Performance of a Greedy Algorithm for On-Line Minimum Matching Problem on Euclidean Space, *Information Processing Letters*, Vol. 51, pp. 275–282.

Tsai, Y. T., Tang, C. Y. and Chen, Y. Y. (1996): An Average Case Analysis of a Greedy Algorithm for the On-Line Steiner Tree Problem, *Computers and Mathematics with Applications*, Vol. 31, No. 11, pp. 121–131.

Turner, J. S. (1989): Approximation Algorithms for the Shortest Common Superstring Problem, *Information and Computation*, Vol. 83, pp. 1–20.

Ukkonen, E. (1985a): Algorithms for Approximate String Matching, *Information and Control*, Vol. 64, pp. 10–118.

Ukkonen, E. (1985b): Finding Approximate Patterns in Strings, *Journal of Algorithms*, Vol. 6, pp. 132–137.

Ukkonen, E. (1990): A Linear Time Algorithms for Finding Approximate Shortest Common Superstrings, *Algorithmica*, Vol. 5, pp. 313–323.

Ukkonen, E. (1992): Approximate String-Matching with Q-Grams and Maximal Matches, *Theoretical Computer Science*, Vol. 92, pp. 191–211.

Unger, R. and Moult, J. (1993): Finding the Lowest Free Energy Conformation of a Protein Is an NP-Hard Problem: Proof and Implications (Prove NP-Complete Problem), *Bulletin of Mathematical Biology*, Vol. 55, pp. 1183–1198.

Uspensky, V. and Semenov, A. (1993): *Algorithms: Main Ideas and Applications*, Kluwer Press, Norwell, Mass.

Vaidya, P. M. (1988): Minimum Spanning Trees in *k*-Dimensional Space, *SIAM Journal on Computing*, Vol. 17, No. 3, pp. 572–582.

Valiant, L. G. and Vazirani, V. V. (1986): NP Is as Easy as Detecting Unique Solutions, *Theoretical Computer Science*, Vol. 47, No. 1, pp. 85–93.

Van Leeuwen, J. (1990): *Handbook of Theoretical Computer Science: Volume A: Algorithms and Complexity*, Elsevier, Amsterdam.

Vazirani, V. V. (2001): *Approximation Algorithms*, Springer-Verlag, New York.

Verma, R. M. (1997): General Techniques for Analyzing Recursive Algorithms with Applications, *SIAM Journal on Computing*, Vol. 26, No. 2, pp. 568–581.

Veroy, B. S. (1988): Average Complexity of Divide-and-Conquer Algorithms, *Information Processing Letters*, Vol. 29, No. 6, pp. 319–326.

Vintsyuk, T. K. (1968): Speech Discrimination by Dynamic Programming, *Cybernetics*, Vol. 4, No. 1, pp. 52–57.

Vishwanathan, S. (1992): Randomized Online Graph Coloring, *Journal of Algorithms*, Vol. 13, pp. 657–669.

Viterbi, A. J. (1967): Error Bounds for Convolutional Codes and an Asymptotically Optimal Decoding Algorithm, *IEEE Transactions on Information Theory*, pp. 260–269.

Vliet, A. (1992): An Improved Lower Bound for On-Line Bin Packing Algorithms, *Information Processing Letters*, Vol. 43, No. 5, pp. 277–284.

Von Haeselerm A., Blum, B., Simpson, L., Strum, N. and Waterman, M. S. (1992): Computer Methods for Locating Kinetoplastid Cryptogenes, *Nucleic Acids Research*, Vol. 20, pp. 2717–2724.

Voronoi, G. (1908): Nouvelles Applications des Parameters Continus a la Theorie des Formes Quadratiques. Deuxieme M'emoire: Recherches Sur les Parall'eloedres Primitifs, *J. Reine Angew. Math.*, Vol. 134, pp. 198–287.

Vyugin, M. V. and V'yugin, V. V. (2002): On Complexity of Easy Predictable Sequences, *Information and Computation*, Vol. 178, No. 1, pp. 241–252.

Wah. B. W. and Yu, C. F. (1985): Stochastic Modeling of Branch-and-Bound Algorithms with Best-First Search, *IEEE Transactions on Software Engineering*, Vol. SE–11, No. 9, pp. 922–934.

Walsh, T. R. (1984): How Evenly Should One Divide to Conquer Quickly? *Information Processing Letters*, Vol. 19, No. 4, pp. 203–208.

Wang, B. F. (1997): Tighter Bounds on the Solution of a Divide-and-Conquer Maximum Recurrence, *Journal of Algorithms*, Vol. 23, pp. 329–344.

Wang, B. F. (2000): Tight Bounds on the Solutions of Multidimensional Divide-and-Conquer Maximum Recurrences, *Theoretical Computer Science*, Vol. 242, pp. 377–401.

Wang, D. W. and Kuo, Y. S. (1988): A Study of Two Geometric Location Problems, *Information Processing Letters*, Vol. 28, No. 6, pp. 281–286.

Wang, J. S. and Lee, R. C. T. (1990): An Efficient Channel Routing Problem to Yield an Optimal Solution, *IEEE Transactions on Computers*, Vol. 39, No. 7, pp. 957–962.

Wang, J. T. L., Zhang, K., Jeong, K. and Shasha, D. (1994): A System for Approximate Tree Matching, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 6, No. 4, pp. 559–571, 1041–4347.

Wang, L. and Gusfield, D. (1997): Improved Approximation Algorithms for Tree Alignment, *Journal of Algorithms*, Vol. 25, No. 2, pp. 255–273.

Wang, L. and Jiang, T. (1994): On the Complexity of Multiple Sequence Alignment, *Journal of Computational Biology*, Vol. 1, No. 4, pp. 337–348.

Wang, L., Jiang, T. and Gusfield, D. (2000): A More Efficient Approximation Scheme for Tree Alignment, *SIAM Journal on Applied Mathematics*, Vol. 30, No. 1, pp. 283–299.

Wang, L., Jiang, T. and Lawler, E. L. (1996): Approximation Algorithms for Tree Alignment with a Given Phylogeny, *Algorithmica*, Vol. 16, pp. 302–315.

Wang, X., He, L., Tang, Y. and Wee, W. G. (2003): A Divide and Conquer Deformable Contour Method with a Model Based Searching Algorithm, *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 33, No. 5, pp. 738–751.

Wareham, H. T. (1995): A Simplified Proof of the NP- and MAX SNP-Hardness of Multiple Sequence Tree Alignments (Prove NP-Complete Problem), *Journal of Computational Biology*, Vol. 2, No. 4.

Waterman, M. S. (1995): *Introduction to Computational Biology: Maps, Sequences and Genomes*, Chapman & Hall/CRC, New York.

Waterman, M. S. and Smith, T. F. (1978): RNA Secondary Structure: A Complete Mathematical Analysis, *Mathematical Bioscience*, Vol. 42, pp. 257–266.

Waterman, M. S. and Smith, T. F. (1986): Rapid Dynamic Programming Algorithms for RNA Secondary Structure, *Advances in Applied Mathematics*, Vol. 7, pp. 455–464.

Waterman, M. S. and Vingron, M. (1994): Sequence Comparison Significance and Poisson Approximation, *Statistical Science*, Vol. 2, pp. 367–381.

Weide, B. (1977): A Survey of Analysis Techniques for Discrete Algorithms, *ACM Computing Surveys*, Vol. 9, No. 4, pp. 291–313.

Weiss, M. A. (1992): *Data Structures and Algorithm Analysis*, Benjamin Cummings, Redwood City, California.

Wenger, R. (1997): Randomized Quickhull, *Algorithmica*, Vol. 17, No. 3, pp. 322–329.

Westbrook, J. and Tarjan, R. E. (1989): Amortized Analysis of Algorithms for Set Union with Backtracking, *SIAM Journal on Computing*, Vol. 18, No. 1, pp. 1–11.

Wilf, H. S. (1986): *Algorithms & Complexity*, Prentice-Hall, Engelwood Cliffs, New Jersey.

Williams, J. W. J. (1964): Heapsort: Algorithm 232, *Communications of the ACM*, Vol. 7, pp. 347–348.

Wood, D. (1993): *Data Structures, Algorithms and Performance*, Addison-Wesley, Reading, Mass.

Wright, A. H. (1994): Approximate String Matching Using Withinword Parallelism, *Software: Practice and Experience*, Vol. 24, pp. 337–362.

Wu, B. Y., Lancia, G., Bafna, V., Chao, K. M., Ravi, R. and Tang, C. Y. (2000): A Polynomial-Time Approximation Scheme for Minimum Routing Cost Spanning Trees, *SIAM Journal on Computing*, Vol. 29, No. 3, pp. 761–778.

Wu, L. C. and Tang, C. Y. (1992): Solving the Satisability Problem by Using Randomized Approach, *Information Processing Letters*, Vol. 41, No. 4, pp. 187–190.

Wu, Q. S., Chao, K. M. and Lee, R. C. T. (1998): The NPO-Completeness of the Longest Hamiltonian Cycle Problem, *Information Processing Letters*, Vol. 65, pp. 119–123.

Wu, S. and Manber, U. (1992): Fast Text Searching Allowing Errors, *Communications of the ACM*, Vol. 35, pp. 83–90.

Wu, S. and Myers, G. (1996): A Subquadratic Algorithm for Approximate Limited Expression Matching, *Algorithmica*, Vol. 15, pp. 50–67.

Wu, T. (1996): A Segment-Based Dynamic Programming Algorithm for Predicting Gene Structure, *Journal of Computational Biology*, Vol. 3, pp. 375–394.

Wu, Y. F., Widmayer, P. and Wong, C. K. (1986): A Faster Approximation Algorithm for the Steiner Problem in Graphs, *Acta Informatica*, Vol. 23, No. 2, pp. 223–229.

Xu, S. (1990): *Dynamic programming algorithms for alignment hyperplanes*. Unpublished Master's thesis, University of Southern California.

Yagle, A. E. (1998): Divide-and-Conquer 2-D Phase Retrieval Using Subband Decomposition and Filter Banks, *IEEE Transactions on Signal Processing*, Vol. 46, No. 4, pp. 1152–1154.

Yang, C. I., Wang, J. S. and Lee, R. C. T. (1989): A Branch-and-Bound Algorithm to Solve the Equal-Execution-Time Job Scheduling Problem with Precedence Constraint and Prole, *Computers and Operations Research*, Vol. 16, No. 3, pp. 257–269.

Yannakakis, M. (1985): A Polynomial Algorithm for the Min-Cut Linear Arrangement of Trees, *Journal of the Association for Computing Machinery*, Vol. 32, No. 4, pp. 950–988.

Yannakakis, M. (1989): Embedding Planar Graphs in Four Pages, *Journal of Computer and System Sciences*, Vol. 38, No. 1, pp. 36–67.

Yao, A. C. (1981): Should Tables be Sorted, *Journal of the ACM*, Vol. 28, No. 3, pp. 615–628.

Yao, A. C. (1985): On the Complexity of Maintaining Partial Sums, *SIAM Journal on Computing*, Vol. 14, No. 2, pp. 277–288.

Yao, A. C. (1991): Lower Bounds to Randomized Algorithms for Graph Properties, *Journal of Computer and System Sciences*, Vol. 42, No. 3, pp. 267–287.

Ye, D. and Zhang, G. (2003): On-Line Extensible Bin Packing with Unequal Bin Sizes, *Lecture Notes in Computer Science*, Vol. 2909, pp. 235–247.

Yen, C. C. and Lee, R. C. T. (1990): The Weighted Perfect Domination Problem, *Information Processing Letters*, Vol. 35, pp. 295–299.

Yen, C. C. and Lee, R. C. T. (1994): Linear Time Algorithm to Solve the Weighted Perfect Domination Problem in Series-Parallel Graphs, *European Journal of Operational Research*, Vol. 73, No. 1, pp. 192–198.

Yen, C. K. and Tang, C. Y. (1995): An Optimal Algorithm for Solving the Searchlight Guarding Problem on Weighted Trees, *Information Sciences*, Vol. 87, pp. 79–105.

Yen, F. M. and Kuo, S. Y. (1997): Variable Ordering for Ordered Binary Decision Diagrams by a Divide-and-Conquer Approach, *IEE Proceedings: Computer and Digital Techniques*, Vol. 144, No. 5, pp. 261–266.

Yoo, J., Smith, K. F. and Gopalarkishnan, G. (1997): A Fast Parallel Squarer Based on Divide-and-Conquer, *IEEE Journal of Solid-State Circuits*, Vol. 32, No. 6, pp. 909–912.

Young, N. (2000): On-Line Paging Against Adversarially Biased Random Inputs, *Journal of Algorithms*, Vol. 37, pp. 218–235.

Younger, D. H. (1967): Recognition and Parsing of Context-Free Languages in Time n^3, *Information and Control*, Vol. 10, No. 2, pp. 189–208.

Zelikovsky, A. (1993): An 11/6 Approximation Algorithm for the Steiner Tree Problem in Graph, *Information Processing Letters*, Vol. 46, pp. 317–323.

Zemel, E. (1987): A Linear Randomized Algorithm for Searching Rank Functions, *Algorithmica*, Vol. 2, No. 1, pp. 81–90.

Zhang, K. and Jiang, T. (1994): Some Max SNP-Hard Results Concerning Unordered Labeled Trees, *Information Processing Letters*, Vol. 49, pp. 249–254.

Zhang, Z., Schwartz, S., Wagner, L. and Miller, W. (2003): A Greedy Algorithm for Aligning DNA Sequences, *Journal of Computational Biology*, Vol. 7, pp. 203–214.

Zuker, M. (1989): The Use of Dynamic Programming Algorithms in RNA Secondary Structure Prediction, in M. S. Waterman (Ed.), *Mathematical Methods for DNA Sequences* (pp. 159–185), CRC Press, Boca Raton, Florida.

Author Index

Subject Index

This index uses the following convention. Numbers are alphabetized as if spelled out; for example, "1-center problem" is indexed as if it were "one-center problem." The letters *f* and *t* followed by page numbers indicate figures and tables, respectively.