44

# Data Structures and Network Algorithms

ROBERT ENDRE TARJAN
Bell Laboratories
Murray Hill, New Jersey

CBMS-NSF
REGIONAL CONFERENCE SERIES
IN APPLIED MATHEMATICS

SPONSORED BY
CONFERENCE BOARD OF
THE MATHEMATICAL SCIENCES

SUPPORTED BY
NATIONAL SCIENCE
FOUNDATION

# CBMS-NSF REGIONAL CONFERENCE SERIES IN APPLIED MATHEMATICS

A series of lectures on topics of current research interest in applied mathematics under the direction of the Conference Board of the Mathematical Sciences, supported by the National Science Foundation and published by SIAM.

GARRETT BIRKHOFF, *The Numerical Solution of Elliptic Equations*
D. V. LINDLEY, *Bayesian Statistics, A Review*
R. S. VARGA, *Functional Analysis and Approximation Theory in Numerical Analysis*
R. R. BAHADUR, *Some Limit Theorems in Statistics*
PATRICK BILLINGSLEY, *Weak Convergence of Measures: Applications in Probability*
J. L. LIONS, *Some Aspects of the Optimal Control of Distributed Parameter Systems*
ROGER PENROSE, *Techniques of Differential Topology in Relativity*
HERMAN CHERNOFF, *Sequential Analysis and Optimal Design*
J. DURBIN, *Distribution Theory for Tests Based on the Sample Distribution Function*
SOL I. RUBINOW, *Mathematical Problems in the Biological Sciences*
P. D. LAX, *Hyperbolic Systems of Conservation Laws and the Mathematical Theory of Shock Waves*
I. J. SCHOENBERG, *Cardinal Spline Interpolation*
IVAN SINGER, *The Theory of Best Approximation and Functional Analysis*
WERNER C. RHEINBOLDT, *Methods of Solving Systems of Nonlinear Equations*
HANS F. WEINBERGER, *Variational Methods for Eigenvalue Approximation*
R. TYRRELL ROCKAFELLAR, *Conjugate Duality and Optimization*
SIR JAMES LIGHTHILL, *Mathematical Biofluiddynamics*
GERARD SALTON, *Theory of Indexing*
CATHLEEN S. MORAWETZ, *Notes on Time Decay and Scattering for Some Hyperbolic Problems*
F. HOPPENSTEADT, *Mathematical Theories of Populations: Demographics, Genetics and Epidemics*
RICHARD ASKEY, *Orthogonal Polynomials and Special Functions*
L. E. PAYNE, *Improperly Posed Problems in Partial Differential Equations*
S. ROSEN, *Lectures on the Measurement and Evaluation of the Performance of Computing Systems*
HERBERT B. KELLER, *Numerical Solution of Two Point Boundary Value Problems*
J. P. LASALLE, *The Stability of Dynamical Systems - Z. ARTSTEIN, Appendix A: Limiting Equations and Stability of Nonautonomous Ordinary Differential Equations*
D. GOTTLIEB AND S. A. ORSZAG, *Numerical Analysis of Spectral Methods: Theory and Applications*
PETER J. HUBER, *Robust Statistical Procedures*
HERBERT SOLOMON, *Geometric Probability*
FRED S. ROBERTS, *Graph Theory and Its Applications to Problems of Society*
JURIS HARTMANIS, *Feasible Computations and Provable Complexity Properties*
ZOHAR MANNA, *Lectures on the Logic of Computer Programming*
ELLIS L. JOHNSON, *Integer Programming: Facets, Subadditivity, and Duality for Group and Semi-Group Problems*
SHMUEL WINOGRAD, *Arithmetic Complexity of Computations*
J. F. C. KINGMAN, *Mathematics of Genetic Diversity*
MORTON E. GURTIN, *Topics in Finite Elasticity*
THOMAS G. KURTZ, *Approximation of Population Processes*

(continued on inside back cover)

Robert Endre Tarjan
Bell Laboratories
Murray Hill, New Jersey

# Data Structures and Network Algorithms

siam.
SOCIETY FOR INDUSTRIAL AND APPLIED MATHEMATICS
PHILADELPHIA

10 9



Library of Congress Catalog Card Number: 83-61374

ISBN 0-89871-187-8

*To Gail Maria Zawacki*

*This page intentionally left blank*

# Contents

# Preface

In the last fifteen years there has been an explosive growth in the field of combinatorial algorithms. Although much of the recent work is theoretical in nature, many newly discovered algorithms are quite practical. These algorithms depend not only on new results in combinatorics and especially in graph theory, but also on the development of new data structures and new techniques for analyzing algorithms. My purpose in this book is to reveal the interplay of these areas by explaining the most efficient known algorithms for a selection of combinatorial problems. The book covers four classical problems in network optimization, including a development of the data structures they use and an analysis of their running times. This material will be included in a more comprehensive two-volume work I am planning on data structures and graph algorithms.

My goal has been depth, precision and simplicity. I have tried to present the most advanced techniques now known in a way that makes them understandable and available for possible practical use. I hope to convey to the reader some appreciation of the depth and beauty of the field of graph algorithms, some knowledge of the best algorithms to solve the particular problems covered, and an understanding of how to implement these algorithms.

The book is based on lectures delivered at a CBMS Regional Conference at the Worcester Polytechnic Institute (WPI) in June, 1981. It also includes very recent unpublished work done jointly with Dan Sleator of Bell Laboratories. I would like to thank Paul Davis and the rest of the staff at WPI for their hard work in organizing and running the conference, all the participants for their interest and stimulation, and the National Science Foundation for financial support. My thanks also to Cindy Romeo and Marie Wenslau for the diligent and excellent job they did in preparing the manuscript, to Michael Garey for his penetrating criticism, and especially to Dan Sleator, with whom it has been a rare pleasure to work.

*This page intentionally left blank*

# CHAPTER 1

# Foundations

**1.1. Introduction.** In this book we shall examine efficient computer algorithms for four classical problems in network optimization. These algorithms combine results from two areas: data structures and algorithm analysis, and network optimization, which itself draws from operations research, computer science and graph theory. For the problems we consider, our aim is to provide an understanding of the most efficient known algorithms.

We shall assume some introductory knowledge of the areas we cover. There are several good books on data structures and algorithm analysis [1], [35], [36], [44], [49], [58] and several on graph algorithms and network optimization [8], [11], [21], [38], [39], [41], [50]; most of these touch on both topics. What we shall stress here is how the best algorithms arise from the interaction between these areas. Many presentations of network algorithms omit all but a superficial discussion of data structures, leaving a potential user of such algorithms with a nontrivial programming task. One of our goals is to present good algorithms in a way that makes them both easy to understand and easy to implement. But there is a deeper reason for our approach. A detailed consideration of computational complexity serves as a kind of "Occam's razor": the most efficient algorithms are generally those that compute exactly the information relevant to the problem situation. Thus the development of an especially efficient algorithm often gives us added insight into the problem we are considering, and the resultant algorithm is not only efficient but simple and elegant. Such algorithms are the kind we are after.

Of course, too much detail will obscure the most beautiful algorithm. We shall not develop FORTRAN programs here. Instead, we shall work at the level of simple operations on primitive mathematical objects, such as lists, trees and graphs. In §§1.3 through 1.5 we develop the necessary concepts and introduce our algorithmic notation. In Chapters 2 through 5 we use these ideas to develop four kinds of composite data structures that are useful in network optimization.

In Chapters 6 through 9, we combine these data structures with ideas from graph theory to obtain efficient algorithms for four network optimization tasks: finding minimum spanning trees, shortest paths, maximum flows, and maximum matchings. Not coincidentally, these are four of the five problems discussed by Klee in his excellent survey of network optimization [34]. Klee's fifth problem, the minimum tour problem, is one of the best known of the so-called "NP-complete" problems; as far as is known, it has no efficient algorithm. In §1.2, we shall review some of the concepts of computational complexity, to make precise the idea of an efficient algorithm and to provide a perspective on our results (see also [53], [55]).

We have chosen to formulate and solve network optimization problems in the setting of graph theory. Thus we shall omit almost all mention of two areas that

provide alternative approaches: matroid theory and linear programming. The books of Lawler [38] and Papadimitriou and Steiglitz [41] contain information on these topics and their connection to network optimization.

**1.2. Computational complexity.** In order to study the efficiency of algorithms, we need a model of computation. One possibility is to develop a denotational definition of complexity, as has been done for program semantics [19], but since this is a current research topic we shall proceed in the usual way and define complexity operationally. Historically the first machine model proposed was the *Turing machine* [56]. In its simplest form a Turing machine consists of a finite state control, a two-way infinite memory tape divided into squares, each of which can hold one of a finite number of symbols, and a read/write head. In one step the machine can read the contents of one tape square, write a new symbol in the square, move the head one square left or right, and change the state of the control.

The simplicity of Turing machines makes them very useful in high-level theoretical studies of computational complexity, but they are not realistic enough to allow accurate analysis of practical algorithms. For this purpose a better model is the *random-access machine* [1], [14]. A random-access machine consists of a finite program, a finite collection of registers, each of which can store a single integer or real number, and a memory consisting of an array of $n$ words, each of which has a unique address between 1 and $n$ (inclusive) and can hold a single integer or real number. In one step, a random-access machine can perform a single arithmetic or logical operation on the contents of specified registers, fetch into a specified register the contents of a word whose address is in a register, or store the contents of a register in a word whose address is in a register.

A similar but somewhat less powerful model is the *pointer machine* [35], [46], [54]. A pointer machine differs from a random-access machine in that its memory consists of an extendable collection of *nodes*, each divided into a fixed number of named *fields*. A field can hold a number or a pointer to a node. In order to fetch from or store into one of the fields in a node, the machine must have in a register a pointer to the node. Operations on register contents, fetching from or storing into node fields, and creating or destroying a node take constant time. In contrast to the case with random-access machines, address arithmetic is impossible on pointer machines, and algorithms that require such arithmetic, such as hashing [36], cannot be implemented on such machines. However, pointer machines make lower bound studies easier, and they provide a more realistic model for the kind of list-processing algorithms we shall study. A pointer machine can be simulated by a random-access machine in real time. (One operation on a pointer machine corresponds to a constant number of operations on a random-access machine.)

All three of these machine models share two properties: they are *sequential*, i.e., they carry out one step at a time, and *deterministic*, i.e., the future behavior of the machine is uniquely determined by its present configuration. Outside this section we shall not discuss parallel computation or nondeterminism, even though parallel algorithms are becoming more important because of the novel machine architectures made possible by very large scale integration (VLSI), and nondeterminism of

various kinds has its uses in both theory and practice [1], [19], [23]. One important research topic is to determine to what extent the ideas used in sequential, deterministic computation carry over to more general computational models.

An important caveat concerning random-access and pointer machines is that if the machine can manipulate numbers of arbitrary size in constant time, it can perform hidden parallel computation by encoding several numbers into one. There are two ways to prevent this. Instead of counting each operation as one step (the *uniform cost measure*), we can charge for an operation a time proportional to the number of bits needed to represent the operands (the *logarithmic cost measure*). Alternatively we can limit the size of the integers we allow to those representable in a constant times log $n$ bits, where $n$ is a measure of the input size, and restrict the operations we allow on real numbers. We shall generally use the latter approach; all our algorithms are implementable on a random-access or pointer machine with integers of size at most $n^c$ for some small constant $c$ with only comparison, addition, and sometimes multiplication of input values allowed as operations on real numbers, with no clever encoding.

Having picked a machine model, we must select a complexity measure. One possibility is to measure the complexity of an algorithm by the length of its program. This measure is *static*, i.e., independent of the input values. Program length is the relevant measure if an algorithm is only to be run once or a few times, and this measure has interesting theoretical uses [10], [37], [42], but for our purposes a better complexity measure is a *dynamic* one, such as running time or storage space as a function of input size. We shall use running time as our complexity measure; most of the algorithms we consider have a space bound that is a linear function of the input size.

In analyzing running times we shall ignore constant factors. This not only simplifies the analysis but allows us to ignore details of the machine model, thus giving us a complexity measure that is machine independent. As Fig. 1.1 illustrates, for large enough problem sizes the relative efficiencies of two algorithms depend on their running times as an asymptotic function of input size, independent of constant factors. Of course, what "large enough" means depends upon the situation; for some problems, such as matrix multiplication [15], the asymptotically most efficient known algorithms beat simpler methods only for astronomical problem sizes. The algorithms we shall consider are intended to be practical for moderate problem sizes. We shall use the following notation for asymptotic running times: If $f$ and $g$ are functions of nonnegative variables $n, m, \cdots$ we write "$f$ is $O(g)$" if there are positive constants $c_1$ and $c_2$ such that $f(n, m, \cdots) \leqq c_1 g(n, m, \cdots) + c_2$ for all values of $n, m, \cdots$. We write "$f$ is $\Omega(g)$" if $g$ is $O(f)$, and "$f$ is $\Theta(g)$" if $f$ is $O(g)$ and $\Omega(g)$.

We shall generally measure the running time of an algorithm as a function of the worst-case input data. Such an analysis provides a performance guarantee, but it may give an overly pessimistic estimate of the actual performance if the worst case occurs rarely. An alternative is an average-case analysis. The usual kind of averaging is over the possible inputs. However, such an analysis is generally much harder than worst-case analysis, and we must take care that our probability

| COMPLEXITY \ SIZE | 20 | 50 | 100 | 200 | 500 | 1000 |
|---|---|---|---|---|---|---|
| $1000n$ | .02 sec | .05 sec | .1 sec | .2 sec | .5 sec | 1 sec |
| $1000n \lg n$ | .09 sec | .3 sec | .6 sec | 1.5 sec | 4.5 sec | 10 sec |
| $100n^2$ | .04 sec | .25 sec | 1 sec | 4 sec | 25 sec | 2 min |
| $10n^3$ | .02 sec | 1 sec | 10 sec | 1 min | 21 min | 2.7 hr |
| $n^{\lg n}$ | .4 sec | 1.1 hr | 220 DAYS | 125 CENT | $5 \times 10^8$ CENT | |
| $2^{n/3}$ | .0001 sec | .1 sec | 2.7 hr | $3 \times 10^4$ CENT | | |
| $2^n$ | 1 sec | 35 YR | $3 \times 10^4$ CENT | | | |
| $3^n$ | 58 min | $2 \times 10^9$ CENT | | | | |

FIG. 1.1. *Running time estimates. One step takes one microsecond,* lgn *denotes* $\log_2 n$.

distribution accurately reflects reality. A more robust approach is to allow the algorithm to make probabilistic choices. Thus for worst-case input data we average over possible algorithms. For certain problem domains, such as table look-up [9], [57], string matching [31], and prime testing [3], [43], [48], such randomized algorithms are either simpler or faster than the best known deterministic algorithms. For the problems we shall consider, however, this is not the case.

A third kind of averaging is *amortization.* Amortization is appropriate in situations where particular algorithms are repeatedly applied, as occurs with operations on data structures. By averaging the time per operation over a worst-case sequence of operations, we sometimes can obtain an overall time bound much smaller than the worst-case time per operation multiplied by the number of operations. We shall use this idea repeatedly.

By an *efficient algorithm* we mean one whose worst-case running time is bounded by a polynomial function of the input size. We call a problem *tractable* if it has an efficient algorithm and *intractable* otherwise, denoting by P the set of tractable problems. Cobham [12] and Edmonds [20] independently introduced this idea. There are two reasons for its importance. As the problem size increases, polynomial-time algorithms become unusable gradually, whereas nonpolynomial-time algorithms have a problem size in the vicinity of which the algorithm rapidly becomes completely useless, and increasing by a constant factor the amount of time allowed or the machine speed doesn't help much. (See Fig. 1.2.) Furthermore, efficient algorithms usually correspond to some significant structure in the problem, whereas inefficient algorithms often amount to brute-force search, which is defeated by combinatorial explosion.

| TIME / COMPLEXITY | 1sec | $10^2$ sec (1.7 min) | $10^4$sec (2.7 hr) | $10^6$ sec (12 DAYS) | $10^8$sec (3 YEARS) | $10^{10}$ sec (3 CENT.) |
|---|---|---|---|---|---|---|
| 1000n | $10^3$ | $10^5$ | $10^7$ | $10^9$ | $10^{11}$ | $10^{13}$ |
| 1000n lgn | $1.4\times10^2$ | $7.7\times10^3$ | $5.2\times10^5$ | $3.9\times10^7$ | $3.1\times10^9$ | $2.6\times10^{11}$ |
| $100n^2$ | $10^2$ | $10^3$ | $10^4$ | $10^5$ | $10^6$ | $10^7$ |
| $10n^3$ | 46 | $2.1\times10^2$ | $10^3$ | $4.6\times10^3$ | $2.1\times10^4$ | $10^5$ |
| nlgn | 22 | 36 | 54 | 79 | 112 | 156 |
| $2^{n/3}$ | 59 | 79 | 99 | 119 | 139 | 159 |
| $2^n$ | 19 | 26 | 33 | 39 | 46 | 53 |
| $3^n$ | 12 | 16 | 20 | 25 | 29 | 33 |

FIG. 1.2. *Maximum size of a solvable problem. A factor of ten increase in machine speed corresponds to a factor of ten increase in time.*

Figure 1.3 illustrates what we call the "spectrum of computational complexity," a plot of problems versus the complexities of their fastest known algorithms. There are two regions, containing the tractable and intractable problems. At the top of the plot are the *undecidable* problems, those with *no* algorithms at all. Lower are the problems that do have algorithms but only inefficient ones, running in exponential or superexponential time. These intractable problems form the subject matter of *high-level complexity*. The emphasis in high-level complexity is on proving non-

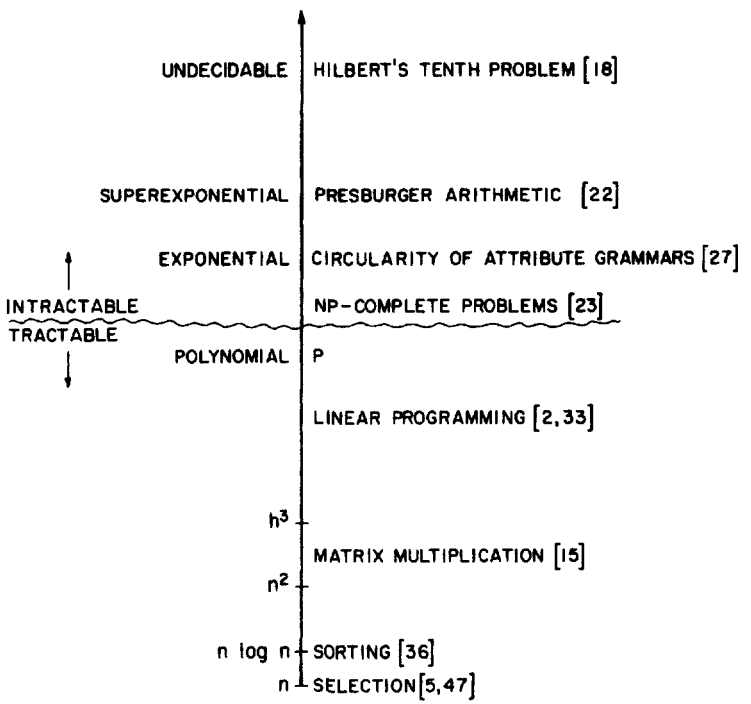



FIG. 1.3. *The spectrum of computational complexity.*

polynomial lower bounds on the time or space requirements of various problems. The machine model used is usually the Turing machine; the techniques used, simulation and diagonalization, derive from Godel's incompleteness proof [24], [40] and have their roots in classical self-reference paradoxes.

Most network optimization problems are much easier than any of the problems for which exponential lower bounds have been proved; they are in the class NP of problems solvable in polynomial time on a nondeterministic Turing machine. A more intuitive definition is that a problem is in NP if it can be phrased as a yes-no question such that if the answer is "yes" there is a polynomial-length proof of this. An example of a problem in NP is the *minimum tour problem:* given $n$ cities and pairwise distances between them, find a tour that passes through each city once, returns to the starting point, and has minimum total length. We can phrase this as a yes-no question by asking if there is a tour of length at most $x$; a "yes" answer can be verified by exhibiting an appropriate tour.

Among the problems in NP are those that are hardest in the sense that if one has a polynomial-time algorithm then so does every problem in NP. These are the NP-*complete* problems. Cook [13] formulated this notion and illustrated it with several NP-complete problems; Karp [29], [30] established its importance by compiling a list of important problems, including the minimum tour problem, that are NP-complete. This list has now grown into the hundreds; see Garey and Johnson's book on NP-completeness [23] and Johnson's column in the Journal of Algorithms [28]. The NP-complete problems lie on the boundary between intractable and tractable. Perhaps the foremost open problem in computational complexity is to determine whether P = NP; that is, whether or not the NP-complete problems have polynomial-time algorithms.

The problems we shall consider all have efficient algorithms and thus lie within the domain of *low-level complexity,* the bottom half of Fig. 1.3. For such problems lower bounds are almost nonexistent; the emphasis is on obtaining faster and faster algorithms and in the process developing data structures and algorithmic techniques of wide applicability. This is the domain in which we shall work.

Although the theory of computational complexity can give us important information about the practical behavior of algorithms, it is important to be aware of its limitations. An example that illustrates this is *linear programming,* the problem of maximizing a linear function of several variables constrained by a set of linear inequalities. Linear programming is the granddaddy of network optimization problems; indeed, all four of the problems we consider can be phrased as linear programming problems. Since 1947, an effective, but not efficient algorithm for this problem has been known, the *simplex method* [16]. On problems arising in practice, the simplex method runs in low-order polynomial time, but on carefully constructed worst-case examples the algorithm takes an exponential number of arithmetic operations. On the other hand, the newly discovered *ellipsoid method* [2], [33], which amounts to a very clever $n$-dimensional generalization of binary search, runs in polynomial time with respect to the logarithmic cost measure but performs very poorly in practice [17]. This paradoxical situation is not well understood but is perhaps partially explained by three observations: (i) hard problems for the simplex method seem to be relatively rare; (ii) the average-case running time of the ellipsoid

method seems not much better than that for its worst case; and (iii) the ellipsoid method needs to use very high precision arithmetic, the cost of which the logarithmic cost measure underestimates.

**1.3. Primitive data structures.** In addition to integers, real numbers and bits (a bit is either **true** or **false**), we shall regard certain more complicated objects as primitive. These are *intervals, lists, sets,* and *maps.* An *interval* $[j\,..\,k]$ is a sequence of integers $j, j+1, \cdots, k$. We extend the notation to represent *arithmetic progressions:* $[j, k\,..\,l]$ denotes the sequence $j, j+\Delta, j+2\Delta, \cdots, j+i\Delta$, where $\Delta = k - j$ and $i = \lfloor (l-j)/\Delta \rfloor$. (If $x$ is a real number, $\lfloor x \rfloor$ denotes the largest integer not greater than $x$ and $\lceil x \rceil$ denotes the smallest integer not less than $x$.) If $i < 0$, the progression is empty; if $j = k$, the progression is undefined. We use $\in$ to denote membership and $\notin$ to denote nonmembership in intervals, lists and sets; thus for instance $i \in [j\,..\,k]$ means $i$ is an integer such that $j \leq i \leq k$.

A *list* $q = [x_1, x_2, \cdots, x_n]$ is a sequence of arbitrary elements, some of which may be repeated. Element $x_1$ is the *head* of the list and $x_n$ is the *tail;* $x_1$ and $x_n$ are the *ends* of the list. We denote the *size* $n$ of the list by $|q|$. An *ordered pair* $[x_1, x_2]$ is a list of two elements; [ ] denotes the *empty list* of no elements. There are three fundamental operations on lists:

*Access.* Given a list $q = [x_1, x_2, \cdots, x_n]$ and an integer $i$, return the $i$th element $q(i) = x_i$ on the list. If $i \notin [1\,..\,n]$, $q(i)$ has the special value **null.**

*Sublist.* Given a list $q = [x_1, x_2, \cdots, x_n]$ and a pair of integers $i$ and $j$, return the list $q[i\,..\,j] = [x_i, x_{i+1}, \cdots, x_j]$. If $j$ is missing or greater than $n$ it has an implied value of $n$; similarly if $i$ is missing or less than one it has an implied value of 1. Thus for instance $q[3\,..\,] = [x_3, x_4, \cdots, x_n]$. We can extend this notation to denote sublists corresponding to arithmetic progressions.

*Concatenation.* Given two lists $q = [x_1, x_2, \cdots, x_n]$ and $r = [y_1, y_2, \cdots, y_m]$, return their *concatenation* $q \,\&\, r = [x_1, x_2, \cdots, x_n, y_1, y_2, \cdots, y_m]$.

We can represent arbitrary insertion and deletion in lists by appropriate combinations of sublist and concatenation. Especially important are the special cases of access, sublist and concatenation that manipulate the ends of a list:

*Access head.* Given a list $q$, return $q(1)$.
*Push.* Given a list $q$ and an element $x$, replace $q$ by $[x] \,\&\, q$.
*Pop.* Given a list $q$, replace $q$ by $q[2\,..\,]$.
*Access tail.* Given a list $q$, return $q(|q|)$.
*Inject.* Given a list $q$ and an element $x$, replace $q$ by $q \,\&\, [x]$.
*Eject.* Given a list $q$, replace $q$ by $q[\,..\,|q|-1]$.

A list on which the operations access head, push and pop are possible is a *stack.* With respect to insertion and deletion a stack functions in a last-in, first-out manner. A list on which access head, inject and pop are possible is a *queue.* A queue functions in a first-in, first-out manner. A list on which all six operations are possible is a *deque* (double-ended queue). If all operations but eject are possible the list is an *output-restricted deque.* (See Fig. 1.4.)

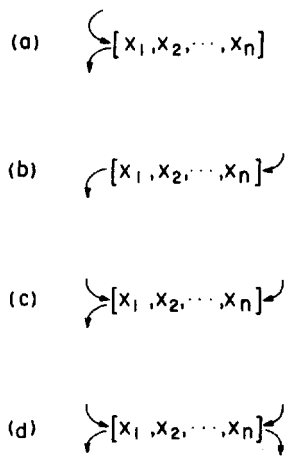



FIG. 1.4. *Types of lists.* (a) *Stack.* (b) *Queue.* (c) *Output-restricted deque.* (d) *Deque.*

A *set* $s = \{x_1, x_2, \cdots, x_n\}$ is a collection of distinct elements. Unlike a list, a set has no implied ordering of its elements. We extend the size notation to sets; thus $|s| = n$. We denote the empty set by $\{\ \}$. The important operations on sets are union $\cup$, intersection $\cap$, and difference $-$: if $s$ and $t$ are sets, $s - t$ is the set of all elements in $s$ but not in $t$. We extend difference to lists as follows: if $q$ is a list and $s$ a set, $q - s$ is the list formed from $q$ by deleting every copy of every element in $s$.

A *map* $f = \{[x_1, y_1], [x_2, y_2], \cdots, [x_n, y_n]\}$ is a set of ordered pairs no two having the same first coordinate (head). The *domain* of $f$ is the set of first coordinates, **domain** $(f) = \{x_1, x_2, \cdots, x_n\}$. The *range* of $f$ is the set of second coordinates (tails), **range** $(f) = \{y_1, y_2, \cdots, y_n\}$. We regard $f$ as a function from the domain to the range; the *value* $f(x_i)$ of $f$ at an element $x_i$ of the domain is the corresponding second coordinate $y_i$. If $x \notin$ **domain** $(f)$, $f(x) =$ **null**. The size $|f|$ of $f$ is the size of its domain. The important operations on functions are accessing and redefining function values. The assignment $f(x) := y$ deletes the pair $[x, f(x)]$ (if any) from $f$ and adds the pair $[x, y]$. The assignment $f(x) :=$ **null** merely deletes the pair $[x, f(x)]$ (if any) from $f$. We can regard a list $q$ as a map with domain $[1\ .\ .\ |q|]$.

There are several good ways to represent maps, sets, and lists using arrays and linked structures (collections of nodes interconnected by pointers). We can represent a map as an array of function values (if the domain is an interval or can be easily transformed into an interval or part of one) or as a node field (if the domain is a set of nodes). These representations correspond to the memory structures of random-access and pointer machines respectively; they allow accessing or redefining $f(x)$ given $x$ in $O(1)$ time. We shall use functional notation rather than dot notation to represent the values of node fields; depending upon the circumstances $f(x)$ may represent the value of map $f$ at $x$, the value stored in position $x$ of array $f$, the value of field $f$ in node $x$, or the value returned by the function $f$ when applied to $x$. These are all just alternative ways of representing functions. We shall use a small circle to denote function composition: $f \circ g$ denotes the function defined by $(f \circ g)(x) = f(g(x))$.

We can represent a set by using its characteristic function over some universe or by using one of the list representations discussed below and ignoring the induced order of the elements. If $s$ is a subset of a universe $U$, its characteristic function $\chi_s$ over $U$ is $\chi_s(x) =$ **true** if $x \in S$, **false** if $x \in U - S$. We call the value of $\chi_s(x)$ the *membership bit* of $x$ (with respect to $s$). A characteristic function allows testing for membership in $O(1)$ time and can be updated in $O(1)$ time under addition or deletion of a single element. We can define characteristic functions for lists in the same way. Often a characteristic function is useful in combination with another set or list representation. If we need to know the size of a set frequently, we can maintain the size as an integer; updating the size after a one-element addition or deletion takes $O(1)$ time.

We can represent a list either by an array or by a linked structure. The easiest kind of list to represent is a stack. We can store a stack $q$ in an array $aq$, maintaining the last filled position as an integer $k$. The correspondence between stack and array is $q(i) = aq(k + 1 - i)$; if $k = 0$ the stack is empty. With this representation each of the stack operations takes $O(1)$ time. In addition, we can access and even redefine arbitrary positions in the stack in $O(1)$ time. We can extend the representation to deques by keeping two integers $j$ and $k$ indicating the two ends of the deque and allowing the deque to "wrap around" from the back to the front of the array. (See Fig. 1.5.) The correspondence between deque and array is $q(i) = aq(((j + i - 2) \textbf{ mod } n) + 1)$, where $n$ is the size of the array and $x$ **mod** $y$ denotes the remainder of $x$ when divided by $y$. Each of the deque operations takes $O(1)$ time. If the elements of the list are nodes, it is sometimes useful to have a field in each node called a *list index* indicating the position of the node in the array. An array representation of a list is a good choice if we have a reasonably tight upper bound on the maximum size of the list and we do not need to perform many sublist and concatenate operations; such operations may require extensive copying.

There are many ways to represent a list as a linked structure. We shall consider eight, classified in three ways: as endogenous or exogenous, single or double and linear or circular. (See Fig. 1.6.) We call a linked data structure defining an arrangement of nodes *endogenous* if the pointers forming the "skeleton" of the structure are contained in the nodes themselves and *exogenous* if the skeleton is outside the nodes. In a single list, each node has a pointer to the next node on the list

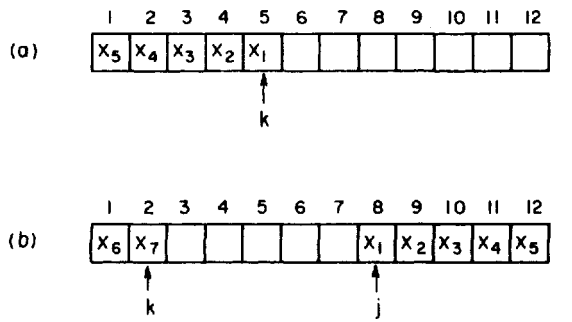



FIG. 1.5. *Array representation of lists.* (a) *Stack.* (b) *Deque that has wrapped around the array.*

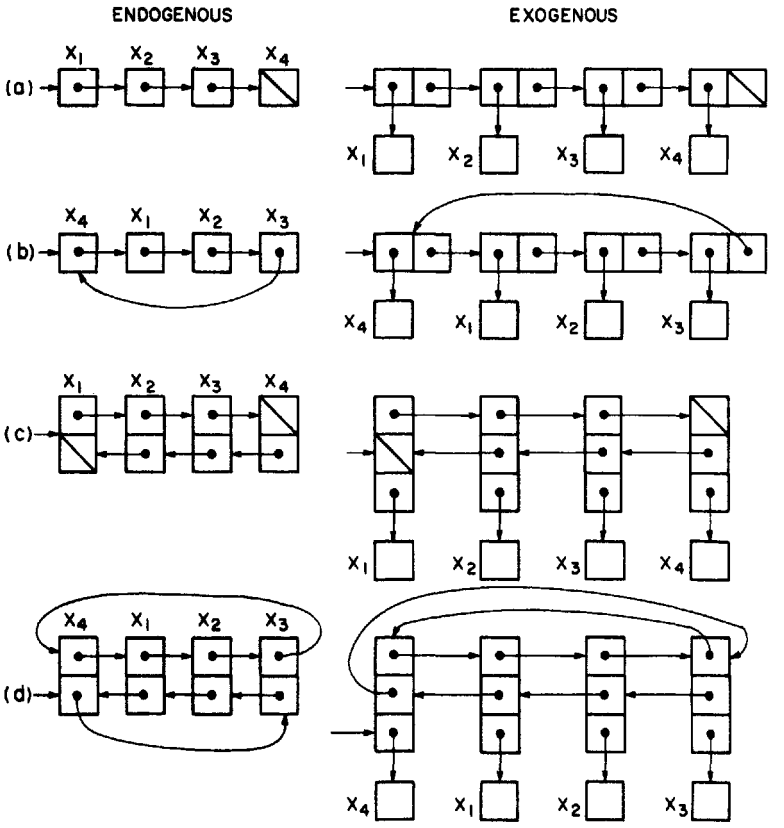



FIG. 1.6. *Linked representations of lists. Missing pointers are* **null.** (a) *Single linear.* (b) *Single circular.* (c) *Double linear.* (d) *Double circular.*

(its *successor*); in a double list, each node also has a pointer to the previous node (its *predecessor*). In a linear list, the successor of the last node is **null,** as is the predecessor of the first node; in a circular list, the successor of the last node is the first node and the predecessor of the first node is the last node. We access a linear list by means of a pointer to its head, a circular list by means of a pointer to its tail.

Figure 1.7 indicates the power of these representations. A single linear list suffices to represent a stack so that each access head, push, or pop operation takes $O(1)$ time. A single circular list suffices for an output-restricted deque and also allows concatenation in $O(1)$ time if we allow the concatenation to destroy its inputs. (All our uses of concatenation will allow this.) Single linking allows insertion of a new element after a specified one or deletion of the element after a specified one in $O(1)$ time; to have this capability if the list is exogenous we must store in each list element an inverse pointer indicating its position in the list. Single linking also allows scanning the elements of a list in order in $O(1)$ time per element scanned.

Double linking allows inserting a new element before a given one or deleting any element. It also allows scanning in reverse order. A double circular list suffices to represent a deque.

| | SINGLE | | DOUBLE | |
|---|---|---|---|---|
| | LINEAR | CIRCULAR | LINEAR | CIRCULAR |
| ACCESS HEAD | YES | YES | YES | YES |
| PUSH | YES | YES | YES | YES |
| POP | YES | YES | YES | YES |
| ACCESS TAIL | NO | YES | NO | YES |
| INJECT | NO | YES | NO | YES |
| EJECT | NO | NO | NO | YES |
| INSERT AFTER | YES(a) | YES(a) | YES(a) | YES(a) |
| INSERT BEFORE | NO | NO | YES(a) | YES(a) |
| DELETE AFTER | YES(a) | YES(a) | YES(a) | YES(a) |
| DELETE | NO | NO | YES(a) | YES(a) |
| CONCATENATE | NO | YES | NO | YES |
| REVERSE | NO | NO | NO | YES(b) |
| FORWARD SCAN | YES | YES | YES | YES |
| BACKWARD SCAN | NO | NO | YES | YES |

FIG. 1.7. *The power of list representations. "Yes" denotes an* $O(1)$*–time operation (*$O(1)$ *time per element for forward and backward scanning).* (a) *If the representation is exogenous, insertion and deletion other than at the ends of the list require the position of the element. Inverse pointers furnish this information.* (b) *Reversal requires a modified representation. (See Fig.* 1.8.)

Endogenous structures are more space-efficient than exogenous ones, but they require that a given element be in only one or a fixed number of structures at a time. The array representation of a list can be regarded as an exogenous structure.

Some variations on these representations are possible. Instead of using circular linking, we can use linear linking but maintain a pointer to the tail as well as to the head of a list. Sometimes it is useful to make the head of a list a special dummy node called a *header;* this eliminates the need to treat the empty list as a special case.

Sometimes we need to be able to *reverse* a list, i.e. replace $q = [x_1, x_2, \cdots, x_n]$ by **reverse** $(q) = [x_n, x_{n-1}, \cdots, x_1]$. To allow fast reversal we represent a list by a double circular list accessed by a pointer to the tail, with the following modification: each node except the tail contains two pointers, to its predecessor and successor, but in no specified order; only for the tail is the order known. (See Fig. 1.8.) Since any access to the list is through the tail, we can establish the identity of predecessors and successors as nodes are accessed in $O(1)$ time per node. This representation allows all the deque operations, concatenation and reversal to be performed in $O(1)$ time per operation.

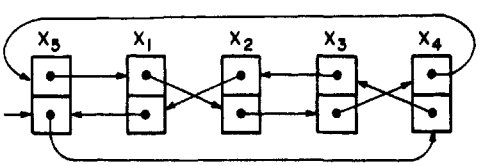



FIG. 1.8. *Endogenous representation of a reversible list.*

**1.4. Algorithmic notation.** To express an algorithm we shall use either a step-by-step description or a program written in an Algol-like language. Our language combines Dijkstra's guarded command language [19] and SETL [32]. We use ":=" to denote assignment and ";" as a statement separator. We allow sequential and parallel assignment: "$x_1 := x_2 := \cdots := x_n := expression$" assigns the value of the expression to $x_n$, $x_{n-1}, \cdots, x_1$; "$x_1, x_2, \cdots, x_n := exp_1, exp_2, \cdots, exp_n$" simultaneously assigns the value of $exp_i$ to $x_i$, for $i \in [1 \,..\, n]$. The double arrow "↔" denotes swapping: "$x \leftrightarrow y$" is equivalent to "$x, y := y, x$."

We use three control structures: Dijkstra's **if** $\cdots$ **fi** and **do** $\cdots$ **od**, and a **for** $\cdots$ **rof** statement.

The form of an **if** statement is:

**if** *condition*$_1$ → *statement list*$_1$ | *condition*$_2$ → *statement list*$_2$ | $\cdots$ | *condition*$_n$ → *statement list*$_n$ **fi.**

The effect of this statement is to cause the conditions to be evaluated and the statement list for the first **true** condition to be executed; if none of the conditions is **true** none of the statement lists is executed. We use a similar syntax for defining conditional expressions: **if** *condition*$_1$ → *exp*$_1$ | $\cdots$ | *condition*$_n$ → *exp*$_n$ **fi** evaluates to *exp*$_i$ if *condition*$_i$ is the first **true** condition. (Dijkstra allows nondeterminism in **if** statements; all our programs are written to be correct for Dijkstra's semantics.)

The form of a **do** statement is:

**do** *condition*$_1$ → *statement list*$_1$ | *condition*$_2$ → *statement list*$_2$ | $\cdots$ | *condition*$_n$ → *statement list*$_n$ **od.**

The effect of this statement is similar to that of an **if** except that after the execution of a statement list the conditions are reevaluated, the appropriate statement list is executed, and this is repeated until all conditions evaluate to **false.**

The form of a **for** statement is:

**for** *iterator* → *statement list* **rof.**

This statement causes the statement list to be evaluated once for each value of the iterator. An iterator has the form $x \in s$, where $x$ is a variable and $s$ is an interval, arithmetic progression, list, or set; the statement list is executed $|s|$ times, once for each element $x$ in $s$. If $s$ is a list, successive values of $x$ are in list order; similarly if $s$ is an interval or arithmetic progression. If $s$ is a set, successive values of $x$ are in unpredictable order. We allow the following abbreviations: "$x = j \,..\, k$" is equivalent to $x \in [j \,..\, k]$, "$x = j, k \,..\, l$" is equivalent to "$x \in [j, k \,..\, l]$," and "**for** $x \in s$: *condition*$_1$ → *statement list*$_1$ | $\cdots$ | *condition*$_n$ → *statement list*$_n$ **rof**" is equivalent to "**for** $x \in s$ → **if** *condition*$_1$ → *statement list*$_1$ | $\cdots$ | *condition*$_n$ → *statement list*$_n$ **fi rof.**"

We allow procedures, functions (procedures that return a nonbit result) and predicates (procedures that return a bit result). The **return** statement halts execution of a procedure and returns execution to the calling procedure; **return** *expression* returns the value of the expression from a function or predicate. Parameters are called by value unless otherwise indicated; the other options are **result** (call by result) and **modifies** (call by value and result). (When a parameter is a set, list, or

similar structure, we assume that what is passed is a pointer to an appropriate representation of the structure.) The syntax of procedure definitions is

**procedure** *name* (*parameter list*); *statement list* **end** for a procedure,
*type* **function** *name* (*parameter list*); *statement list* **end** for a function, and
**predicate** *name* (*parameter list*); *statement list* **end** for a predicate.

We allow declaration of local variables within procedures. Procedure parameters and declared variables have a specified type, such as **integer, real, bit, map, list, set,** or a user-defined type. We shall be somewhat loose with types; in particular we shall not specify a mechanism for declaring new types and we shall ignore the issue of type checking.

We regard **null** as a node capable of having fields. We assume the existence of certain built-in functions. In particular, **create** *type* returns a new node of the specified type. Function **min** *s* returns the smallest element in a set or list *s* of numbers; **min** *s* **by** *key* returns the element in *s* of minimum *key*, where *s* is a set or list of nodes and *key* is a field or function. Function **max** is similar. Function **sort** *s* returns a sorted list of the elements in a set or list *s* of numbers; **sort** *s* **by** *key* returns a list of the elements in *s* sorted by *key*, where *s* is a set or list of nodes and *key* is a field or function.

As an example of the use of our notation we shall develop and analyze a procedure that implements **sort** *s*. For descriptive purposes we assume *s* is a list. Our algorithm is called list merge sort [36]; it sorts *s* by repeatedly merging sorted sublists. The program merge (*s*, *t*), defined below, returns the sorted list formed by merging sorted lists *s* and *t*:

```
list function merge (list s, t);
  return if s = [ ] → t
         | t = [ ] → s
         | s ≠ [ ] and t ≠ [ ] and s(1) ≤ t(1) →
                [s(1)] & merge (s[2 . . ], t)
         | s ≠ [ ] and t ≠ [ ] and s(1) > t(1) →
                [t(1)] & merge (s, t[2 . . ])
         fi
end merge;
```

This program merges *s* and *t* by scanning the elements in the two lists in nondecreasing order; it takes $O(|s| + |t|)$ time. To sort a list *s*, we make each of its elements into a single-element list, place all the sublists in a queue, and repeat the following step until the queue contains only one list, which we return:

MERGE STEP. Remove the first two lists from the front of the queue, merge them, and add the result to the rear of the queue.

The following program implements this method:

```
list function sort (list s);
  list queue;
  queue := [ ];
```

```
    for x ∈ s → queue := queue & [[x]] rof;
    do |queue| ≥ 2 → queue := queue[3 . . ] & merge (queue (1), queue (2)) od;
    return if queue = [ ] → [ ] | queue ≠ [ ] → queue (1) fi
end;
```

Each pass through the queue takes $O(|s|)$ time and reduces the number of lists on the queue by almost a factor of two, from $|queue|$ to $\lceil |queue|/2 \rceil$. Thus there are $O(\log |s|)$[1] passes and the total time to sort is $O(|s| \log |s|)$, which is minimum to within a constant factor for sorting by comparison [36]. If this method is implemented iteratively instead of recursively, it is an efficient, practical way to sort lists. A further improvement in efficiency can be obtained by initially breaking $s$ into sorted sublists instead of singletons: if $s = [x_1, \cdots, x_n]$, we split $s$ between each pair of elements $x_i, x_{i+1}$ such that $x_i > x_{i+1}$. This method is called natural list merge sort [36].

**1.5. Trees and graphs.** The main objects of our study are trees and graphs. Our definitions are more or less standard; for further information see any good text on graph theory [4], [6], [7], [25], [26]. A graph $G = [V, E]$ consists of a *vertex set* $V$ and an *edge set* $E$. Either $G$ is *undirected*, in which case every edge is an unordered pair of distinct vertices, or $G$ is *directed*, in which case every edge is an ordered pair of distinct vertices. In order to avoid repeating definitions, we shall denote by $(v, w)$ either an undirected edge $\{v, w\}$ or a directed edge $[v, w]$, using the context to resolve the ambiguity. We do not allow loops (edges of the form $(v, v)$) or multiple edges, although all our algorithms extend easily to handle such edges. If $\{v, w\}$ is an undirected edge, $v$ and $w$ are *adjacent*. A directed edge $[v, w]$ *leaves* or *exits* $v$ and *enters* $w$; the edge is *out of* $v$ and *into* $w$. If $(v, w)$ is any edge, $v$ and $w$ are its *ends*; $(v, w)$ is *incident* to $v$ and $w$, and $v$ and $w$ are incident to $(v, w)$. We extend the definition of incidence to sets of vertices as follows: If $S$ is a set of vertices, an edge is incident to $S$ if exactly one of its ends is in $S$. A graph is *bipartite* if there is a subset $S$ of the vertices such that every edge is incident to $S$. (Every edge has one end in $S$ and one end in $V - S$.) If $v$ is a vertex in an undirected graph, its *degree* is the number of adjacent vertices. If $v$ is a vertex in a directed graph, its *in-degree* is the number of edges $[u, v]$ and its *out-degree* is the number of edges $[v, w]$.

If $G$ is a directed graph, we can convert it to an undirected graph called the *undirected version* of $G$ by replacing each edge $[v, w]$ by $\{v, w\}$ and removing duplicate edges. Conversely, we obtain the *directed version* of an undirected graph $G$ by replacing every edge $\{v, w\}$ by the pair of edges $[v, w]$ and $[w, v]$. If $G_1 = [V_1, E_1]$ and $G_2 = [V_2, E_2]$ are graphs, both undirected or both directed, $G_1$ is a *subgraph* of $G_2$ if $V_1 \subseteq V_2$ and $E_1 \subseteq E_2$. $G_1$ is a *spanning subgraph* of $G_2$ if $V_1 = V_2$. $G_1$ is the *subgraph* of $G_2$ *induced by the vertex set* $V_1$ if $E_1$ contains every edge $(v, w) \in E_2$ such that $\{v, w\} \subseteq V_1$. $G_1$ is the *subgraph* of $G_2$ *induced by the edge set* $E_1$ if $V_1$ contains exactly the ends of the edges in $E_1$. If $G = [V, E]$ is a graph and $S$ is a subset of the vertices, the *condensation* of $G$ with respect to $S$ is the graph formed by

[1] We shall use lg $n$ to denote the binary logarithm of $n$. In situations where the base of the algorithm is irrelevant, as inside "$O$," we use log $n$.

condensing $S$ to a single vertex; i.e., G is the graph with vertex set $V - S \cup \{x\}$, where $x$ is a new vertex, and edge set $\{(v', w') \mid v' \neq w' \text{ and } (v, w) \in E\}$, where $v' = v$ if $v \notin S$, $v' = x$ if $v \in S$.

A *path* in a graph *from* vertex $v_1$ *to* vertex $v_k$ is a list of vertices $[v_1, v_2, \cdots, v_k]$ such that $(v_i, v_{i+1})$ is an edge for $i \in [1 \,..\, k-1]$. The path *contains* vertex $v_i$ for $i \in [1 \,..\, k]$ and edge $(v_i, v_{i+1})$ for $i \in [1 \,..\, k-1]$ and *avoids* all other vertices and edges. Vertices $v_1$ and $v_k$ are the *ends* of the path. The path is *simple* if all its vertices are distinct. If the graph is directed, the path is a *cycle* if $k > 1$ and $v_1 = v_k$, and a *simple cycle* if in addition $v_1, v_2, \cdots, v_{k-1}$ are distinct. If the graph is undirected, the path is a cycle if $k > 1$, $v_1 = v_k$ and no edge is repeated, and a simple cycle if in addition $v_1, v_2, \cdots, v_{k-1}$ are distinct. A graph without cycles is *acyclic*. If there is a path from a vertex $v$ to a vertex $w$ then $w$ is *reachable* from $v$.

An undirected graph $G$ is *connected* if every vertex is reachable from every other vertex and *disconnected* otherwise. The maximal connected subgraphs of $G$ are its *connected components;* they partition the vertices of $G$. We extend this definition to directed graphs as follows: If $G$ is directed, its connected components are the subgraphs induced by the vertex sets of the connected components of the undirected version of $G$.

When analyzing graph algorithms we shall use $n$ to denote the number of vertices and $m$ to denote the number of edges. In an undirected graph $m \leq n(n-1)/2$; in a directed graph $m \leq n(n-1)$. A graph is *dense* if $m$ is large compared to $n$ and *sparse* otherwise; the exact meaning of these notions depends upon the context. We shall assume that $n$ and $m$ are positive and $m = \Omega(n)$; thus $n + m = O(m)$. (If $m < n/2$ the graph is disconnected, and we can apply our graph algorithms to the individual connected components.)

We shall generally represent a graph by the set of its vertices and for each vertex one or two sets of incident edges. If the graph is directed, we use the sets $out(v) = \{[v, w] \in E\}$ and possibly $in(v) = \{[u, v] \in E\}$ for $v \in V$. If the graph is undirected, we use $edges(v) = \{\{v, w\} \in E\}$ for $v \in V$. Alternatively, we can represent an undirected graph by a representation of its directed version. We can also represent a graph by using an $n \times n$ *adjacency matrix* $A$ defined by $A(v, w) =$ **true** if $(v, w)$ is an edge, **false** otherwise. Unfortunately, storing such a matrix takes $\Omega(n^2)$ space and using it to solve essentially any nontrivial graph problem takes $\Omega(n^2)$ time [45], which is excessive for sparse graphs.

A *free tree* $T$ is an undirected graph that is connected and acyclic. A free tree of $n$ vertices contains $n-1$ edges and has a unique simple path from any vertex to any other. When discussing trees we shall restrict our attention to simple paths.

A *rooted tree* is a free tree $T$ with a distinguished vertex $r$, called the *root*. If $v$ and $w$ are vertices such that $v$ is on the path from $r$ to $w$, $v$ is an *ancestor* of $w$ and $w$ is a *descendant* of $v$. If in addition $v \neq w$, $v$ is a *proper ancestor of* $w$ and $w$ is a *proper descendant* of $v$. If $v$ is a proper ancestor of $w$ and $v$ and $w$ are adjacent, $v$ is the *parent* of $w$ and $w$ is a *child* of $v$. Every vertex $v$ except the root has a unique parent, generally denoted by $p(v)$, and zero or more children; the root has no parent and zero or more children. We denote by $p^2(v), p^3(v), \cdots$ the grandparent, greatgrandparent, $\cdots$ of $v$. A vertex with no children is a *leaf*. When appropriate we shall regard the edges of a rooted tree as directed, either from child to parent or from

parent to child. We can represent a rooted tree by storing with each vertex its parent or its set of children or (redundantly) both.

We define the *depth* of a vertex $v$ in a rooted tree recursively by *depth* $(v) = 0$ if $v$ is the root, *depth* $(v)$ = *depth* $(p(v)) + 1$ otherwise. Similarly we define the *height* of a vertex $v$ by *height* $(v) = 0$ if $v$ is a leaf, *height* $(v) = \max \{\text{height } (w) \mid w \text{ is a child of } v\} + 1$ otherwise. The *subtree rooted at* vertex $v$ is the rooted tree consisting of the subgraph induced by the descendants of $v$, with root $v$. The *nearest common ancestor* of two vertices $v$ and $w$ is the deepest vertex that is an ancestor of both.

A *tree traversal* is the process of visiting each of the vertices in a rooted tree exactly once. There are several systematic orders in which we can visit the vertices. The following recursive procedure defines *preorder* and *postorder*. If we execute *traverse* $(r)$, where $r$ is the tree root, the procedure applies an arbitrary procedure previsit to the vertices in preorder and an arbitrary procedure postvisit to the vertices in postorder. (See Fig. 1.9.)

```
procedure traverse (vertex v);
   previsit(v);
   for w ∈ children(v) → traverse(w) rof;
   postvisit(v)
end traverse;
```

In preorder, parents are visited before children; in postorder the reverse is true. Another useful ordering is *breadth-first order*, obtained by visiting the root and then repeating the following step until all vertices are visited: visit an unvisited child of the least recently visited vertex with an unvisited child. We can implement a breadth-first traversal by storing the visited vertices (or their unvisited children) on a queue. Each of these three kinds of traversal takes $O(n)$ time if the tree is represented by sets of children; in each case the exact ordering obtained depends upon the order in which the children of each vertex are selected.

A *full binary tree* is a rooted tree in which each vertex $v$ has either two children, its *left child left* $(v)$ and its *right child right* $(v)$, or no children. A vertex with two children is *internal;* a vertex with no children is *external*. If $v$ is an internal vertex, its *left subtree* is the subtree rooted at its left child and its *right subtree* is the

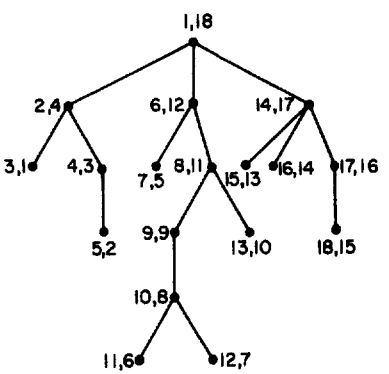



FIG. 1.9. *Tree traversal. First number at a vertex is preorder, second is postorder.*

subtree rooted at its right child. A *binary tree* is obtained from a full binary tree by discarding all the external nodes; in a binary tree each node has a left and right child but either or both may be missing (which we generally denote by **null**.) On binary trees we define preorder, postorder, and another ordering, *inorder* or *symmetric order*, recursively as follows:

```
procedure traverse (vertex v);
    previsit (v);
    if left (v) ≠ null → traverse (left (v));
    invisit (v);
    if right (v) ≠ null → traverse (right (v));
    postvisit (v)
end traverse;
```

We shall use trees extensively as data structures. When doing so we shall call the tree vertices *nodes;* we generally think of them as being nodes in the memory of a pointer machine.

A *forest* is a vertex-disjoint collection of trees. A *spanning tree* of a graph $G$ is a spanning subgraph of $G$ that is a tree (free if G is undirected, rooted with edges directed from parent to child if $G$ is directed).

The idea of a tree traversal extends to graphs. If $G$ is a graph and $s$ is an arbitrary *start vertex*, we carry out a *search* of $G$ starting from $s$ by visiting $s$ and then repeating the following step until there is no unexamined edge $(v, w)$ such that $v$ has been visited:

SEARCH STEP. Select an unexamined edge $(v, w)$ such that $v$ has been visited and examine it, visiting $w$ if $w$ is unvisited.

Such a search visits each vertex reachable from $s$ exactly once and examines exactly once each edge $(v, w)$ such that $v$ is reachable from $s$. The search also generates a spanning tree of the subgraph induced by the vertices reachable from $s$, defined by the set of edges $(v, w)$ such that examination of $(v, w)$ causes $w$ to be visited.

The order of edge examination defines the kind of search. In a *depth-first search*, we always select an edge $(v, w)$ such that $v$ was visited most recently. In a *breadth-first search*, we always select an edge $(v, w)$ such that $v$ was visited least recently. (See Fig. 1.10.)

Both depth-first and breadth-first searches take $O(m)$ time if implemented properly. We can implement depth-first search by using a stack to store eligible unexamined edges. Equivalently, we can use recursion, generalizing the program for preorder and postorder tree traversal. If $G$ is a directed graph represented by *out* sets, the procedure call dfs ($s$) will carry out a depth-first search of $G$ starting from vertex $s$, where dfs is defined as follows:

```
procedure dfs (vertex v);
    previsit (v);
    for [v, w] ∈ out (v):not visited (w) → dfs (w) rof;
    postvisit (v)
end dfs;
```

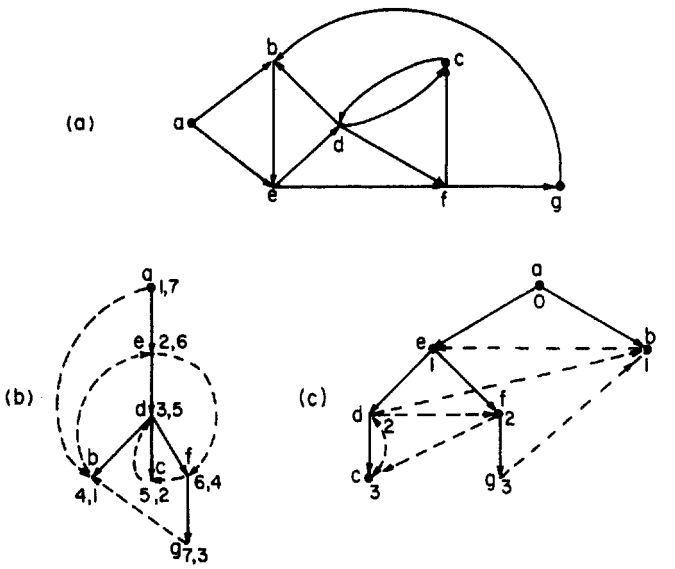



FIG 1.10. *Graph search.* (a) *Graph.* (b) *Depth-first search. Edges of spanning tree are solid, other edges dashed. Vertices are numbered in preorder and postorder. If edges* [*b, e*] *and* [*c, d*] *are deleted, graph becomes acyclic, and postorder is a reverse topological order.* (c) *Breadth-first search. Vertex numbers are levels.*

To be correct, dfs requires that all vertices be unvisited initially and that previsit mark as visited each vertex it visits. The vertices are visited in preorder by previsit and in postorder by postvisit with respect to the spanning tree defined by the search. A similar procedure will search undirected graphs.

We can implement breadth-first search by using a queue to store either eligible unexamined edges or visited vertices. The following program uses the latter method:

```
procedure bfs (vertex s);
  vertex v; list queue;
  queue := [s];
  do queue ≠ [ ] →
    v := queue (1); queue := queue [2 . .];
    bfvisit (v);
    for [v, w] ∈ out (v):not visited (w) and w ∉ queue →
      queue := queue & [w]
    rof
  od
end bfs;
```

As with dfs, the correctness of bfs requires that all vertices be unvisited initially and that bfvisit mark vertices visited. A similar procedure will search undirected graphs.

Both depth-first and breadth-first searches have many applications [51], [52], [53]; we close this chapter with one application of each. Suppose $G$ is a directed graph. A *topological ordering* of $G$ is a total ordering of its vertices such that if $[v, w]$ is an edge, $v$ is ordered before $w$. $G$ has a topological ordering if and only if it is acyclic. Knuth [35] has given an $O(m)$-time algorithm to find such an ordering that works by repeatedly deleting a vertex of in-degree zero. An alternative $O(m)$-time method is to carry out a depth-first search and order the vertices in decreasing order as they are postvisited [52]. (If not all vertices are reachable from the original start vertex we repeatedly search from a new unvisited start vertex until all vertices are visited.) To prove the correctness of this method it suffices to note that during the running of the algorithm there is a path of vertices in decreasing postorder from the start vertex to the current vertex; this path contains all visited vertices greater in postorder than the current vertex.

We can use breadth-first search to compute distances from the start vertex, measured by the number of edges on a path. Suppose we define $level\ (s) = 0$ and carry out a breadth-first search starting from $s$, assigning $level\ (w) = level\ (v) + 1$ when we examine an edge $[v, w]$ such that $w$ is unvisited. Then every edge $[v, w]$ will satisfy $level\ (w) \leq level\ (v) + 1$, and $level\ (v)$ for any vertex will be the length of a shortest path from $s$ to $v$, if we define the length of a path to be the number of edges it contains. To prove this it suffices to note that vertices are removed from the queue in nondecreasing order by level.

## References


[1] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] B. ASPVALL AND R. E. STONE, *Khachiyan's linear programming algorithm*, J. Algorithms, 1 (1980), pp. 1–13.

[3] A. O. L. ATKIN AND R. G. LARSON, *On a primality test of Solovay and Strassen*, SIAM J. Comput., 11 (1982), pp. 789–791.

[4] C. BERGE, *Graphs and Hypergraphs*, North-Holland, Amsterdam, 1973.

[5] M. BLUM, R. W. FLOYD, V. R. PRATT, R. L. RIVEST AND R. E. TARJAN, *Time bounds for selection*, J. Comput. System Sci., 7 (1973), pp. 448–461.

[6] J. A. BONDY AND U. S. R. MURTY, *Graph Theory with Applications*, North-Holland, New York, 1976.

[7] R. G. BUSACKER AND T. L. SAATY, *Finite Graphs and Networks: An Introduction with Applications*, McGraw-Hill, New York, 1965.

[8] B. CARRÉ, *Graphs and Networks*, Clarendon Press, Oxford, 1979.

[9] J. L. CARTER AND M. N. WEGMAN, *Universal classes of hash functions*, J. Comput. System Sci., 18 (1979), pp. 143–154.

[10] G. J. CHAITIN, *On the length of programs for computing finite binary sequences*, J. Assoc. Comput. Mach., 13 (1966), pp. 547–569.

[11] N. CHRISTOFIDES, *Graph Theory: An Algorithmic Approach*, Academic Press, New York, 1975.

[12] A. COBHAM, *The intrinsic computational difficulty of functions*, in Proc. 1964 International Congress for Logic Methodology and Philosophy of Science, Y. Bar-Hillel, ed., North-Holland, Amsterdam, 1964, pp. 24–30.

[13] S. A. COOK, *The complexity of theorem-proving procedures*, Proc. Third ACM Symposium on Theory of Computing, 1971, pp. 151–158.

[14] S. A. COOK AND R. A. RECKHOW, *Time-bounded random access machines*, J. Comput. System Sci., 7 (1973), pp. 354–375.

[15] D. COPPERSMITH AND S. WINOGRAD, *On the asymptotic complexity of matrix multiplication*, SIAM J. Comput., 11 (1982), pp. 472–492.
[16] G. B. DANTZIG, *Linear Programming and Extensions*, Princeton Univ. Press, Princeton, NJ, 1963.
[17] ——, *Comments on Khachian's algorithm for linear programming*, Tech. Rep. SOR 79-22, Dept. Operations Research, Stanford Univ., Stanford, CA, 1979.
[18] M. DAVIS, Y. MATIJASEVIC AND J. ROBINSON, *Hilbert's tenth problem. Diophantine equations: positive aspects of a negative solution*, in Mathematical Developments Arising from Hilbert Problems, American Mathematical Society, Providence, RI, 1976, pp. 323–378.
[19] E. W. DIJKSTRA, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.
[20] J. EDMONDS, *Paths, trees, and flowers*, Canad. J. Math., 17 (1965), pp. 449–467.
[21] S. EVEN, *Graph Algorithms*, Computer Science Press, Potomac, MD, 1979.
[22] M. J. FISCHER AND M. O. RABIN, *Super-exponential complexity of Presburger arithmetic*, in Complexity of Computation, R. M. Karp, ed., American Mathematical Society, Providence, RI, 1974, pp. 27–41.
[23] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractibility: A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, 1979.
[24] K. GÖDEL, *Über formal unentscheidbare Sätze der Principia Mathematica und verwandten Systeme* I, Monatsch. Math. und Phys., 38 (1931), pp. 173–198.
[25] F. HARARY, *Graph Theory*, Addison-Wesley, Reading, MA, 1969.
[26] F. HARARY, R. Z. NORMAN AND D. CARTWRIGHT, *Structural Models: An Introduction to the Theory of Directed Graphs*, John Wiley, New York, 1965.
[27] M. JAZAYERI, W. F. OGDEN AND W. C. ROUNDS, *The intrinsically exponential complexity of the circularity problem for attribute grammars*, Comm. ACM, 18 (1975), pp. 697–706.
[28] D. S. JOHNSON, *The NP-completeness column: An ongoing guide*, J. Algorithms, 4 (1981), pp. 393–405.
[29] R. M. KARP, *Reducibility among combinatorial problems*, in Complexity of Computer Computations, R. E. Miller, ed., Plenum Press, New York, 1972, pp. 85–103.
[30] ———, *On the complexity of combinatorial problems*, Networks, 5 (1975), pp. 45–68.
[31] R. M. KARP AND M. O. RABIN, *Efficient randomized pattern-matching algorithms*, J. Assoc. Comput. Mach., to appear.
[32] K. KENNEDY AND J. SCHWARTZ, *An introduction to the set theoretical language* SETL, Comput. Math. Appl, 1 (1975), pp. 97–119.
[33] L. G. KHACHIAN, *A polynomial algorithm for linear programming*, Soviet Math. Dokl., 20 (1979), pp. 191–194.
[34] V. KLEE, *Combinatorial optimization: What is the state of the art*, Math. Oper. Res., 5 (1980), pp. 1–26.
[35] D. E. KNUTH, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 2nd ed., Addison-Wesley, Reading, MA, 1973.
[36] ———, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.
[37] A. KOLMOGOROV, *Three approaches to the quantitative definition of information*, Problems Inform. Transmission, 1 (1965), pp. 1–7.
[38] E. L. LAWLER, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart, and Winston, New York, 1976.
[39] E. MINIEKA, *Optimization Algorithms for Networks and Graphs*, Marcel Dekker, New York, 1978.
[40] E. NAGEL AND J. R. NEWMAN, *Gödel's Proof*, New York Univ. Press, New York, 1958.
[41] C. H. PAPADIMITRIOU AND K. STEIGLITZ, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ, 1982.
[42] W. J. PAUL, J. I. SEIFERAS AND J. SIMON, *An information-theoretic approach to time bounds for on-line computation*, J. Comput. System Sci., 23 (1981), pp. 108–126.
[43] M. O. RABIN, *Probabilistic algorithms*, in Algorithms and Complexity: New Directions and Recent Results, J. F. Traub, ed., Academic Press, New York, 1976, pp. 21–39.

[44] E. M. REINGOLD, J. NIEVERGELT AND N. DEO, *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, Englewood Cliffs, NJ, 1977.

[45] R. L. RIVEST AND J. VUILLEMIN, *On recognizing graph properties from adjacency matrices*, Theoret. Comput. Sci., 3 (1976), pp. 371–384.

[46] A. SCHÖNHAGE, *Storage modification machines*, Siam J. Comput., 9 (1980), pp. 490–508.

[47] A. SCHÖNHAGE, M. PATERSON AND N. PIPPENGER, *Finding the median*, J. Comput. System Sci., 13 (1975), pp. 184–199.

[48] R. SOLOVAY AND V. STRASSEN, *A fast Monte-Carlo test for primality*, SIAM J. Comput., 6 (1977), pp. 84–85.

[49] T. A. STANDISH, *Data Structure Techniques*, Addison-Wesley, Reading, MA, 1980.

[50] M. N. S. SWAMY AND K. THULASIRAMAN, *Graphs, Networks, and Algorithms*, John Wiley, New York, 1981.

[51] R. E. TARJAN, *Depth-first search and linear graph algorithms*, SIAM J. Comput., 1 (1972), pp. 146–160.

[52] ———, *Finding dominators in directed graphs*, SIAM J. Comput., 3 (1974), pp. 62–89.

[53] ———, *Complexity of combinatorial algorithms*, SIAM Rev., 20 (1978), pp. 457–491.

[54] ———, *A class of algorithms which require nonlinear time to maintain disjoint sets*, J. Comput. System Sci., 18 (1979), pp. 110–127.

[55] ———, *Recent developments in the complexity of combinatorial algorithms*, in Proc. Fifth IBM Symposium on Mathematical Foundations of Computer Science, IBM Japan, Tokyo, 1980, pp. 1–28.

[56] A. M. TURING, *On computable numbers, with an application to the Entscheidungs problem*, Proc. London Math. Soc., 2–42 (1936), pp. 230–265. Correction, ibid., 2–43, pp. 544–546.

[57] M. N. WEGMAN AND J. L. CARTER, *New hash functions and their use in authentication and set equality*, J. Comput. System Sci., 22 (1981), pp. 265–279.

[58] N. WIRTH, *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

*This page intentionally left blank*

# CHAPTER 2

# Disjoint Sets

**2.1. Disjoint sets and compressed trees.** We begin our study of network algorithms with an algorithm that is easy, indeed almost trivial, to implement, but whose analysis 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 the three set operations as follows:

makeset $(x)$: Create a new set containing the single element $x$, previously in no set.

find $(x)$: Return the canonical element of the set containing element $x$.

link $(x, y)$: Form a new set that is the union of the two sets whose canonical elements are $x$ and $y$, destroying the two old sets. Select and return a canonical element for the new set. This operation assumes that $x \neq y$.

To solve this problem we use a data structure proposed by Galler and Fischer [6]. We represent each set by a rooted tree. The nodes of the tree are the elements of the set; the canonical element is the root of the tree. Each node $x$ has a pointer $p(x)$ to 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 Fig. 2.1.)

This naive algorithm is not very efficient, requiring $O(n)$ time per find in the worst case, where $n$ is the total number of elements (makeset operations). 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 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 Fig. 2.2.) Path compression, invented by McIlroy and Morris [2], increases the time of a single find by a constant factor but saves enough time in later finds to more than pay for itself.

The second heuristic, called *union by rank*, keeps the trees shallow by using a freedom implicit in the implementation of link. With each node $x$ we store a nonnegative integer *rank* $(x)$ that is an upper bound on the height of $x$. When carrying out makeset $(x)$, we define *rank* $(x)$ to be 0. To carry out link $(x, y)$, we compare *rank* $(x)$ and *rank* $(y)$. If *rank* $(x) <$ *rank* $(y)$, we make $x$ point to $y$ and

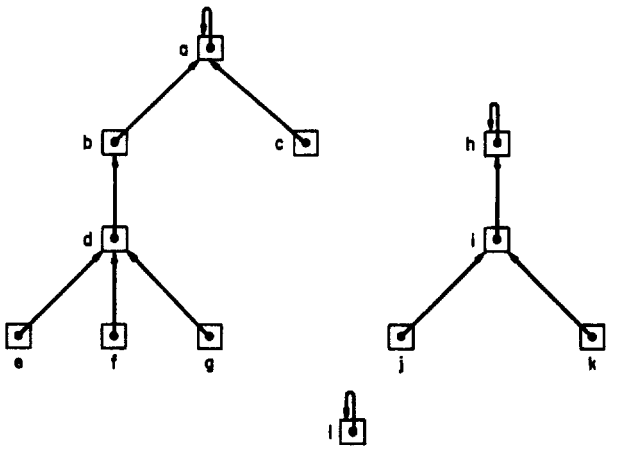



FIG. 2.1. *Representation of sets* $\{a, b, c, d, e, f, g\}$, $\{h, i, j, k\}$, $\{l\}$. *Squares denote nodes. Operation* find ($f$) *returns* $a$, link ($a$, $h$) *makes a point to* $h$.

return $y$ as the canonical element. If *rank* $(x) >$ *rank* $(y)$, we make $y$ point to $x$ and return $x$. Finally, if *rank* $(x) =$ *rank* $(y)$ we make $x$ point to $y$, increase *rank* $(y)$ by one, and return $y$. (See Fig. 2.3.) Union by rank, invented by Tarjan [11], is a variant of the *union by size* heuristic proposed by Galler and Fischer [6].

The following programs implement the three set operations using these heuristics:

```
procedure makeset (element x);
    p(x) := x; rank (x) := 0
end makeset;

element function find (element x);
    if x ≠ p(x) → p(x) := find (p(x)) fi;
    return p(x)
end find;

element function link (element x, y);
    if rank (x) > rank (y) → x ↔ y
      | rank (x) = rank (y) → rank (y) := rank (y) + 1
    fi;
    p(x) := y;
    return y
end link;
```

**2.2. An amortized upper bound for path compression.** Our goal now is to analyze the running time of an intermixed sequence of the three set operations. We shall use $m$ to denote the number of operations and $n$ to denote the number of elements; thus

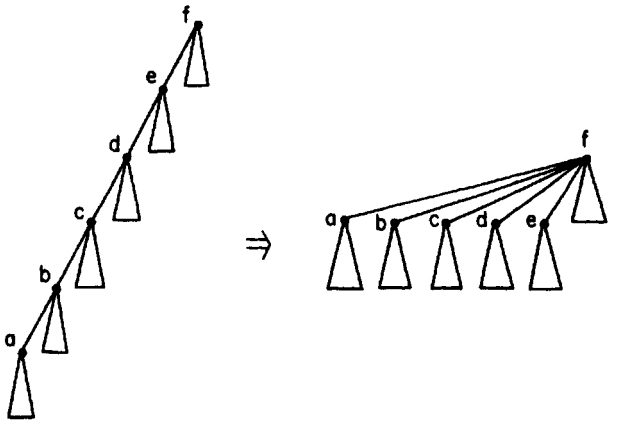



FIG. 2.2. *Compression of the path* [*a*, *b*, *c*, *d*, *e*, *f*]. *Triangles denote subtrees.*

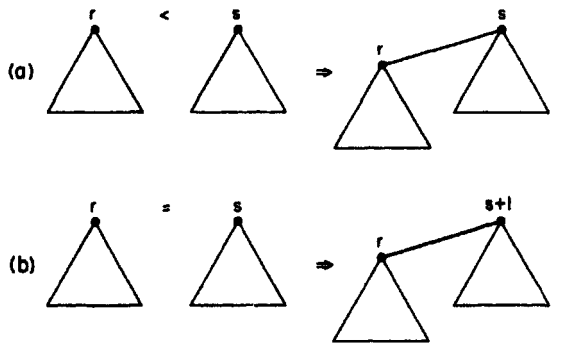



FIG. 2.3. *Linking by rank.* (a) *Roots of unequal rank. Smaller ranked root points to larger.* (b) *Roots of equal rank. Root of new tree increases by one in rank.*

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. Fischer [4] derived an upper bound of $O(m \log \log n)$. Hopcroft and Ullman [7] improved the bound to $O(m \lg^* n)$ where $\lg^* n$ is the iterated logarithm, defined by $\lg^{(0)} n = n$, $\lg^{(i)} n = \lg^{(i-1)} \lg n$ for $i \geq 1$, and $\lg^* n = \min \{i \mid \lg^{(i)} n \leq 1\}$. Tarjan [8] obtained the actual worst-case bound, $\Theta(m\alpha(m, n))$, where $\alpha(m, n)$ is a functional inverse of Ackerman's function [1]. For $i, j \geq 1$ we define Ackerman's function $A(i, j)$ by

$$A(1, j) = 2^j \quad \text{for } j \geq 1,$$

$$A(i, 1) = A(i - 1, 2) \quad \text{for } i \geq 2,$$

$$A(i, j) = A(i - 1, A(i, j - 1)) \quad \text{for } i, j \geq 2.$$

We define the inverse function $\alpha(m, n)$ for $m \geq n \geq 1$ by

$$\alpha(m, n) = \min \{i \geq 1 \mid A(i, \lfloor m/n \rfloor) > \lg n\}.$$

The most important property of $A(i, j)$ is its explosive growth. In the usual definition, $A(1, j) = j + 1$ and the explosion does not occur quite so soon. However, this change only adds a constant to the inverse function $\alpha$, which grows very slowly. With our definition, $A(3, 1) = 16$; thus $\alpha(m, n) \leq 3$ for $n < 2^{16} = 65{,}536$. $A(4, 1) = A(2, 16)$, which is very large. Thus for all practical purposes $\alpha(m, n)$ is a constant not larger than four. For fixed $n$, $\alpha(m, n)$ decreases as $m/n$ increases. In particular, let $a(i, n) = \min \{j \geq 1 \mid A(i, j) > \lg n\}$. Then $\lfloor m/n \rfloor \geq a(i, n)$ implies $\alpha(m, n) \leq i$. For instance, $\lfloor m/n \rfloor \geq 1 + \lg \lg n$ implies $\alpha(m, n) \leq 1$, and $\lfloor m/n \rfloor \geq \lg^* n$ implies $\alpha(m, n) \leq 2$.

We shall derive an upper bound of $O(m\alpha(m, n))$ on the running time of the disjoint set union algorithm by using Tarjan's *multiple partition* method [8], [11]. We begin by noting some simple but crucial properties of ranks.

LEMMA 2.1. *If $x$ is any node, rank $(x) \leq$ rank $(p(x))$, with the inequality strict if $p(x) \neq x$. The value of rank $(x)$ is initially zero and increases as time passes until $p(x)$ is assigned a value other than $x$; subsequently rank $(x)$ does not change. The value of rank $(p(x))$ is a nondecreasing function of time.*

*Proof.* Immediate by induction on time using the implementations of makeset, find and link. □

LEMMA 2.2. *The number of nodes in a tree with root $x$ is at least $2^{rank(x)}$.*

*Proof.* By induction on the number of links. The lemma is true before the first link. Consider an operation link $(x, y)$ before which the lemma holds and let *rank* denote the rank function just before the link. If *rank* $(x) <$ *rank* $(y)$, the tree formed by the link has root $y$, with unchanged rank, and contains more nodes than the old tree with root $y$; thus the lemma holds after the link. The case *rank* $(x) <$ *rank* $(y)$ is symmetric. Finally, if *rank* $(x) =$ *rank* $(y)$, the tree formed by the link contains at least $2^{rank(x)} + 2^{rank(y)} = 2^{rank(y)+1}$ nodes. Since the rank of its root, $y$, is now *rank* $(y) + 1$, the lemma holds after the link. □

LEMMA 2.3. *For any integer $k \geq 0$, the number of nodes of rank $k$ is at most $n/2^k$. In particular, every node has rank at most* lg $n$.

*Proof.* Fix $k$. When a node $x$ is assigned a rank of $k$, label by $x$ all the nodes contained in the tree with root $x$. By Lemma 2.2 at least $2^k$ nodes are so labeled. By Lemma 2.1 if the root of the tree containing $x$ changes, the rank of the new root is at least $k + 1$. Thus no node can be labeled twice. Since there are $n$ nodes there are at most $n$ labels, at least $2^k$ for each node of rank $k$, which means that at most $n/2^k$ nodes are ever assigned rank $k$ by the algorithm. □

A single makeset or link operation takes $O(1)$ time. A single find takes $O(\log n)$ time, since by Lemma 2.1 the node ranks strictly increase along the find path and by Lemma 2.2 no node has rank exceeding lg $n$. Thus we obtain an $O(m \log n)$ bound on the time required for a sequence of $m$ set operations. The $O(\log n)$ bound on find is valid whether or not we use path compression; it depends only on union by rank.

We can obtain a much better overall bound on the algorithm with path compression by amortizing, that is, by averaging over time. To make the analysis as

concrete as possible we introduce the concept of *credits* and *debits*. One credit will pay for a constant amount of computing. To perform a set operation we are given a certain number of credits to spend. If we complete the operation before running out of credits we can save the unused credits for future operations. If we run out of credits before completing an operation we can borrow credits by creating credit-debit pairs and spending the created credits; the corresponding debits remain in existence to account for our borrowing. If desired we can use surplus credits to pay off existing debits one for one. With this accounting scheme the total time for a sequence of set operations is proportional to the total number of credits allocated for the operations plus the number of debits remaining when all the operations are complete.

In using this scheme to analyze the set union algorithm we shall not keep track of surplus credits. Thus we never pay off debits; instead we store them in the compressed trees. It is important to remember that the credits and debits are only an analytical tool; the actual implementations of the set operations need not and should not refer to them.

In order to carry out the analysis we need a few more concepts. We define a *partitioning function* $B(i, j)$ using Ackerman's function $A(i, j)$ as follows:

$$B(0, j) = j \quad \text{for } j \geq 0,$$

$$B(i, 0) = 0 \quad \text{for } i \in [1 \,..\, \alpha(m, n) + 1],$$

$$B(i, j) = A(i, j) \quad \text{for } i \in [1 \,..\, \alpha(m, n)], \quad j \geq 1,$$

$$B(\alpha(m, n) + 1, 1) = A(\alpha(m, n), \lfloor m/n \rfloor).$$

For each level $i \in [0 \,..\, \alpha(m, n) + 1]$, we use $B(i, j)$ to define a partition of the integers $k \in [0 \,..\, \lg n]$ into *blocks* given by

$$block\ (i, j) = [B(i, j) \,..\, B(i, j + 1) - 1] \cap [0 \,..\, \lg n] \quad \text{for } j \geq 0.$$

(See Fig. 2.4.) Every level-zero block is a singleton ($block\ (0, j) = \{j\}$). As the level increases, the partition becomes coarser and coarser, until at level $\alpha(m, n) + 1$ there is only one block ($block\ (\alpha(m, n) + 1, 0) = [0 \,..\, \lg n]$ since $B(\alpha(m, n) + 1, 1) = A(\alpha(m, n), \lfloor m/n \rfloor) > \lg n$ by the definition of $\alpha$). As a measure of the coarsening we define $b_{ij}$ to be the number of level-$(i - 1)$ blocks whose intersection with $block\ (i, j)$ is nonempty.

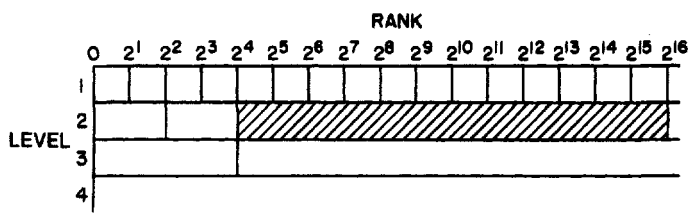



FIG. 2.4. *Multiple partition for analysis of set union algorithm. Level zero is omitted and a logarithmic scale is used. Shaded area is block* $(2, 2) = [2^4 \,..\, 2^{16} - 1]$.

As an aid in the credit accounting we define the *level* of a node $x$ to be the minimum level $i$ such that *rank* $(x)$ and *rank* $(p(x))$ are in a common block of the level $i$ partition. As long as $x = p(x)$, *level* $(x) = 0$. When $p(x)$ is assigned a value other than $x$, *level* $(x)$ becomes positive, since then *rank* $(x) <$ *rank* $(p(x))$. Subsequently *rank* $(x)$ remains fixed but *rank* $(p(x))$ increases; as it does, *level* $(x)$ increases, up to a maximum of $\alpha(m, n) + 1$.

To pay for the set operations we allocate one credit to each makeset, one credit to each link, and $\alpha(m, n) + 2$ credits to each find. The credit allocated to a makeset or link completely pays for the corresponding operation. To analyze the find operations, let us consider a find that starts at a node $x_0$ and follows the path $x_0, x_1 = p(x_0), \cdots, x_l = p(x_{l-1})$, where $p(x_l) = x_l$. To pay for the find we need one credit per node on the find path. For each value of $i$ in the interval $[0 \,..\, \alpha(m, n) + 1]$, we assign one of the credits allocated to the find to the *last* node of level $i$ on the path. At every node on the path not receiving a credit, we create a credit-debit pair. We can now pay for the find, leaving a debit on every node that is not last in its level on the path.

The total number of credits allocated to find operations is $m(\alpha(m, n) + 2)$. It remains for us to count the debits remaining after every find is carried out. A node receives no debit until its level is at least one and thus its rank is fixed. Consider a typical node $x$ and a level $i \geq 1$. We shall bound the number of debits assigned to $x$ while *level* $(x) = i$. Consider a find path $x_0, x_1, \cdots, x_l$ that causes $x = x_k$ to receive a debit while on level $i$. Then *rank* $(x)$ and *rank* $(p(x))$ = *rank* $(x_{k+1})$ are in different level-$(i-1)$ blocks just before the find, and since $x$ is not the last node of level $i$ on the path, *rank* $(x_{k+1})$ and *rank* $(x_l)$ are also in different level-$(i-1)$ blocks just before the find. After the find, $p(x) = x_l$; thus compressing the find path causes *rank* $(p(x))$ to move from one level-$(i-1)$ block, say *block* $(i-1, j')$, to another, say *block* $(i-1, j'')$, where $j'' > j'$. This means that $x$ can receive at most $b_{ij} - 1$ debits while on level $i$, where $j$ is the index such that *rank* $(x) \in$ *block* $(i, j)$: When the level of $x$ first becomes $i$, *rank* $(x)$ and *rank* $(p(x))$ are in different level-$(i-1)$ blocks. Each debit subsequently placed on $x$ causes *rank* $(p(x))$ to move to a new level-$(i-1)$ block. After this happens $b_{ij} - 1$ times, *rank* $(x)$ and *rank* $(p(x))$ are in different level-$i$ blocks.

Consider the situation after all the set operations have been performed. Let $n_{ij}$ be the number of nodes with rank in *block* $(i, j)$. The argument above implies that the total number of debits in existence is at most

$$\sum_{i=1}^{\alpha(m,n)+1} \sum_{j \geq 0} n_{ij} (b_{ij} - 1).$$

We have the following estimates on $n_{ij}$ and $b_{ij}$:

$$n_{i0} \leq n \quad \text{for } i \in [1 \,..\, \alpha(m, n) + 1].$$

$$n_{ij} \leq \sum_{k=B(i,j)}^{B(i,j+1)-1} n/2^k \quad \text{by Lemma 2}$$

$$\leq n/2^{B(i,j)-1} = n/2^{A(i,j)-1} \quad \text{for } i \in [1 \,..\, \alpha(m, n)], \quad j \geq 1.$$

$b_{i0} = 2 \quad \text{for } i \in [1 \,..\, \alpha(m, n)]$

since $A(1, 1) = 2$ and $A(i, 1) = A(i - 1, 2)$ for $i \geq 2$ implies

$$block(i, 0) \subseteq [0 \,..\, A(i, 1) - 1] = [0 \,..\, A(i - 1, 2) - 1].$$

$b_{ij} \leq A(i, j) \quad \text{for } i \in [1 \,..\, \alpha(m, n)], \quad j \geq 1$

since

$$block(i, j) \subseteq [A(i, j) \,..\, A(i, j + 1) - 1] \subseteq [0 \,..\, A(i - 1, A(i, j)) - 1].$$

$b_{\alpha(m,n)+1,0} \leq \lfloor m/n \rfloor$

since

$$block(\alpha(m, n) + 1, 0) \subseteq [0 \,..\, A(\alpha(m, n), \lfloor m/n \rfloor) - 1].$$

Substituting these estimates into the sum, we obtain the following upper bound on the number of debits:

$$\sum_{i=1}^{\alpha(m,n)+1} \sum_{j \geq 0} n_{ij}(b_{ij} - 1) \leq \alpha(m, n)n + m + n \sum_{i=1}^{\alpha(m,n)} \sum_{j \geq 1} A(i, j)/2^{A(i,j)-1}$$

$$\leq \alpha(m, n)n + m + n \sum_{i \geq 1} (A(i, 1) + 1)/2^{A(i,1)-2}$$

$$\leq \alpha(m, n)n + m + n \sum_{k \geq 2} (k + 1)/2^{k-2} = (\alpha(m, n) + 8)n + m.$$

Thus we have:

THEOREM 2.1. *The set union algorithm with path compression and union by rank runs in $O(m\alpha(m, n))$ time.*

**2.3. Remarks.** Path compression has the practical and esthetic disadvantage that it requires two passes over the find path, one to find the tree root and another to perform the compression. One may ask whether there is any efficient one-pass variant of compression. Tarjan and van Leeuwen [11] have studied a number of one-pass variants, some of which run in $O(m\alpha(m, n))$ time when combined with union by rank. The most intriguing, seemingly practical one is *path halving*: when traversing a find path, make every other node on the path point to its grandparent. (See Fig. 2.5.)

The following program implements path halving:

```
element function find (element x);
  do p(p(x)) ≠ p(x) → x := p(x) := p(p(x)) od;
  return p(x)
end find;
```

Another important question is whether the bound in Theorem 1 is tight; that is, are there sequences of set operations that actually take $\Omega(m\alpha(m, n))$ time? The answer is yes, as proved by Tarjan [8]. Indeed, the $\Omega(m\alpha(m, n))$-time lower bound

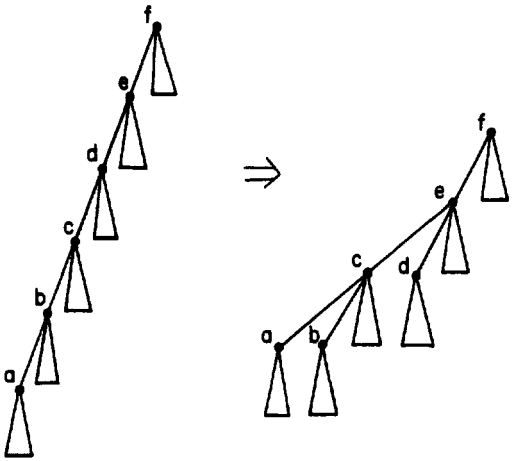



FIG. 2.5. *Halving a path.*

extends to a very general class of pointer manipulation methods for maintaining disjoint sets [3], [9]. However, this result does not rule out the possibility of a linear-time algorithm that uses the extra power of random access. For the restricted case of set union in which the pattern of link operations is known ahead of time, Gabow and Tarjan [5] have recently discovered a linear-time algorithm that combines path compression on large sets with table look-up on small sets. One can use this algorithm in many, but not all, of the common situations requiring disjoint set union. (In particular, we shall use disjoint set union in Chapter 6 in a setting where Gabow and Tarjan's algorithm does not apply.)

Path compression applies to many problems other than set union. The most general result is by Tarjan [10], who has defined a generic problem requiring the maintenance of a function defined on paths in trees that can be solved in $O(m\alpha(m, n))$ time using this technique.

## References

[1] W. ACKERMANN, *Zum Hilbertschen Aufbau der reellen Zahlen*, Math Ann., 99 (1928), pp. 118–133.

[2] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[3] L. BANACHOWSKI, *A complement to Tarjan's result about the lower bound on the complexity of the set union problem*, Inform. Process. Lett., 11 (1980), pp. 59–65.

[4] M. J. FISCHER, *Efficiency of equivalence algorithms*, in Complexity of Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 153–168.

[5] H. GABOW AND R. E. TARJAN, *A linear-time algorithm for a special case of disjoint set union*, Proc. Fifteenth Annual ACM Symposium on Theory of Computing, 1983, pp. 246–251.

[6] B. A. GALLER AND M. J. FISCHER, *An improved equivalence algorithm*, Comm. ACM, 7 (1964), pp. 301–303.

[7] J. E. HOPCROFT AND J. D. ULLMAN, *Set-merging algorithms*, SIAM J. Comput., 2 (1973), pp. 294–303.

[8] R. E. TARJAN, *Efficiency of a good but not linear set union algorithm*, J. Assoc. Comput. Mach., 22 (1975), pp. 215–225.

[9] ———, *A class of algorithms which require nonlinear time to maintain disjoint sets*, J. Comput. System Sci., 18 (1979), pp. 110–127.

[10] ———, *Applications of path compression on balanced trees*, J. Assoc. Comput. Mach., 26 (1979), pp. 690–715.

[11] R. E. TARJAN AND J. VAN LEEUWEN, *Worst-case analysis of set union algorithms*, J. Assoc. Comput. Mach, to appear.

*This page intentionally left blank*

# CHAPTER 3

# Heaps

**3.1. Heaps and heap-ordered trees.** Our use of trees as data structures in Chapter 2 was especially simple, since we needed only parent pointers to represent the trees, and the position of items within each tree was completely unspecified, depending only on the sequence of set operations. However, in many uses of trees, the items in the tree nodes have associated real-valued keys, and the position of items within the tree depends on key order. In this and the next chapter we shall study the two major examples of this phenomenon.

A *heap* is an abstract data structure consisting of a collection of items, each with an associated real-valued *key*. Two operations are possible on a heap:

insert ($i$, $h$): Insert item $i$ into heap $h$, not previously containing $i$.
deletemin ($h$): Delete and return an item of minimum key from heap $h$; if $h$ is empty return **null**.

The following operation creates a new heap:

makeheap ($s$): Construct and return a new heap whose items are the elements in set $s$.

In addition to these three heap operations, we sometimes allow several others:

findmin ($h$): Return but do not delete an item of minimum key from heap $h$; if $h$ is empty return **null**.
delete ($i$, $h$): Delete item $i$ from heap $h$.
meld ($h_1$, $h_2$): Return the heap formed by combining disjoint heaps $h_1$ and $h_2$. This operation destroys $h_1$ and $h_2$.

Williams [10], the originator of the term "heap," meant by it the specific concrete data structure that we call a $d$-heap in §2. Knuth [8] used the term "priority queue" to denote a heap. Aho, Hopcroft and Ullman [1] used "priority queue" for an unmeldable heap and "mergeable heap" for a meldable heap.

To implement a heap we can use a *heap-ordered tree*. (See Fig. 3.1.) Each tree node contains one item, with the items arranged in heap order: if $x$ and $p(x)$ are a node and its parent, respectively, then the key of the item in $p(x)$ is no greater than the key of the item in $x$. Thus the root of the tree contains an item of minimum key, and we can carry out findmin in $O(1)$ time by accessing the root. The time bounds of the other operations depend on the tree structure.

We shall study two heap implementations that use this idea. The *d-heaps* described in §3.2 are appropriate when only one or a few unmeldable heaps are needed. The *leftist heaps* of §3.3 are appropriate when several meldable heaps are needed.

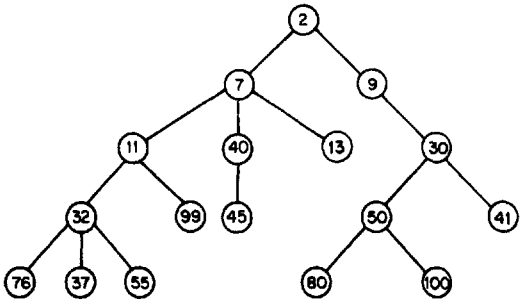



FIG. 3.1. *A heap-ordered tree. Numbers in nodes are keys. (To simplify this and subsequent figures, we treat the keys as if they themselves are the items.)*

**3.2. *d*-heaps.** Suppose the tree representing a heap is exogenous; that is, the heap items and the tree nodes are distinct. This allows us to restore heap order after an update by moving the items among the nodes. In particular, we can insert a new item $i$ as follows. To make room for $i$, we add a new vacant node $x$ to the tree; the parent of $x$ can be arbitrary, but $x$ must have no children. Storing $i$ in $x$ will violate heap order if the parent $p(x)$ of $x$ contains an item whose key exceeds that of $i$, but we can remedy this by carrying out a *sift-up:* While $p(x)$ is defined and contains an item whose key exceeds $key(i)$, we store in $x$ the item previously in $p(x)$, replace the

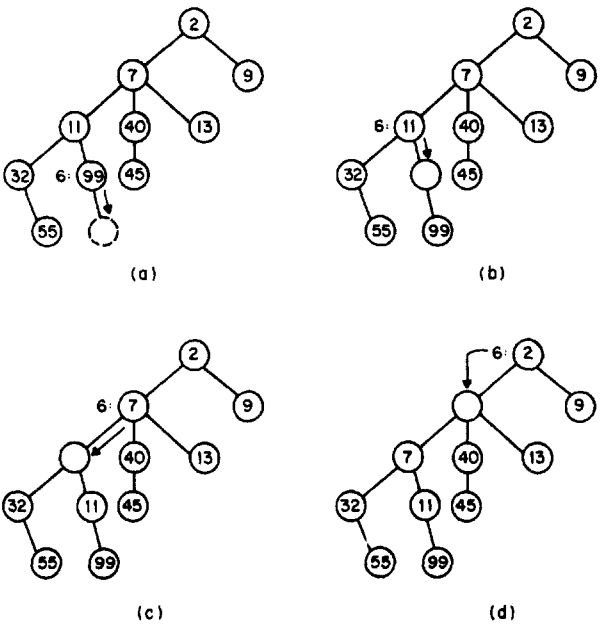



FIG. 3.2. *Insertion of key 6 using sift-up.* (a) *Creation of a new leaf. Since* $6 < 99$, 99 *moves into the new node.* (b) *Since* $6 < 11$, 11 *moves into the vacant node.* (c) *Since* $6 < 7$, 7 *moves into the vacant node.* (d) *Since* $6 \geq 2$, 6 *moves into the vacant node and the sift-up stops.*

vacant node $x$ by $p(x)$, and repeat. When the sifting stops, we store $i$ in $x$. (See Fig. 3.2.)

Deletion is a little more complicated than insertion. To delete an item $i$, we begin by finding a node $y$ with no children. We remove the item, say $j$, from $y$ and delete $y$ from the tree. If $i = j$ we are done. Otherwise we remove $i$ from the node, say $x$, containing it, and attempt to replace it by $j$. If $key\ (j) \leq key\ (i)$, we reinsert $j$ by carrying out a sift-up starting from $x$. If $key\ (j) > key\ (i)$, we reinsert $j$ by carrying out a *sift-down* starting from $x$: While $key\ (j)$ exceeds the key of some child of $x$, we choose a child $c$ of $x$ containing an item of minimum key, store in $x$ the item in $c$, replace $x$ by $c$, and repeat. (See Fig. 3.3.) When the sifting stops, we store $j$ in $x$.

When deleting an item, the easiest way to obtain a node $y$ with no children is to choose the most-recently added node not yet deleted. If we always use this rule, the tree nodes behave like a stack (last-in, first-out) with respect to addition and deletion.

The running times of the heap operations depend upon the structure of the tree, which we must still define. The time for a sift-up is proportional to the depth of the node at which the sift-up starts. The time for a sift-down is proportional to the total number of children of the nodes made vacant during the sift-down. A type of tree that has small depth, few children of nodes on a path and a uniform structure is the

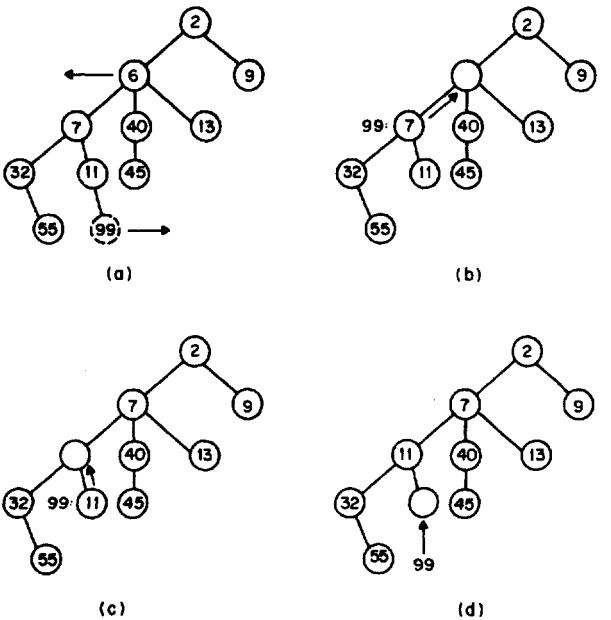



FIG. 3.3. *Deletion of key* 6 *using sift-down. The last-created node contains key* 99. (a) *Destruction of the last leaf. Key* 99 *must be reinserted.* (b) *The smallest key among the children of the vacant node is* 7. *Since* 99 > 7, 7 *moves into the vacant node.* (c) *The smallest key among the children of the vacant node is* 11. *Since* 99 > 11, 11 *moves into the vacant node.* (d) *Since the vacant node has no children,* 99 *moves in and the sift-down stops.*

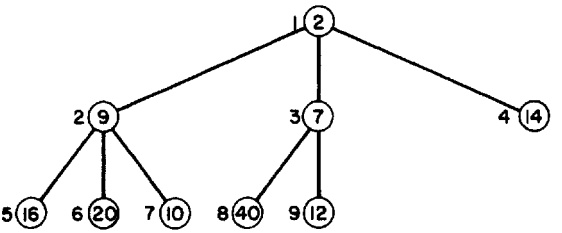



FIG. 3.4. *A 3-heap with nodes numbered in breadth-first order. Next node to be added is* 10, *a child of* 3.

*complete d-ary tree*: each node has at most $d$ children and nodes are added in breadth-first order. Thus we define a *d-heap* to be a complete $d$-ary tree containing one item per node arranged in heap order. (See Fig. 3.4.)

The $d$-heap operations have running times of $O(1)$ for findmin, $O(\log_d n)$ for insert, and $O(d \log_d n)$ for delete and deletemin, where $n$ is the number of items in the heap, since a complete $d$-ary tree has depth $\log_d n + O(1)$. The parameter $d$ allows us to choose the data structure to fit the relative frequencies of the operations; as the proportion of deletions decreases, we can increase $d$, saving time on insertions. We shall use this capability in Chapter 7 to speed up a shortest-path algorithm.

The structure of a $d$-heap is so regular that we need no explicit pointers to represent it. If we number the nodes of a complete $d$-ary tree from one to $n$ in breadth-first order and identify nodes by number, then the parent of node $x$ is $\lceil (x-1)/d \rceil$ and the children of node $x$ are the integers in the interval $[d(x-1)+2 \,..\, \min\{dx+1, n\}]$. (See Fig. 3.4.) Thus we can represent each node by an integer and the entire heap by a map $h$ from $\{1 \,..\, n\}$ onto the items. The following programs implement all the $d$-heap operations except makeheap, which we shall consider later.

```
item function findmin (heap h)
   return if h = { } → null | h ≠ { } → h(1) fi
end findmin;

procedure insert (item i, modifies heap h);
1. siftup (i, |h| + 1, h)
end insert;

procedure delete (item i, modifies heap h);
   item j;
   j := h(|h|);
2. h(|h|) := null;
   if i ≠ j and key (j) ≤ key (i) → siftup (j, h⁻¹(i), h)
     | i ≠ j and key (j) > key (i) → siftdown (j, h⁻¹(i), h)
   fi
end delete;
```

```
item function deletemin (modifies heap h);
  if h = { } → return null
    | h ≠ { } →
    item i;
    i := h(1);
    delete (h(1), h);
    return i
  fi
end deletemin;

procedure siftup (item i, integer x, modifies heap h);
  integer p;
  p := ⌈(x − 1)/d⌉;
  do p ≠ 0 and key (h(p)) > key (i) →
    h(x), x, p := h(p), p, ⌈(p − 1)/d⌉ od;
3. h(x) := i
end siftup;

procedure siftdown (item i, integer x, modifies heap h);
  integer c;
  c := minchild (x, h);
  do c ≠ 0 and key (h(c)) < key (i) →
                    h(x), x, c := h(c), c, minchild (c, h) od;
    h(x) := i
end siftdown;

integer function minchild (integer x, heap h);
  return if d·(x − 1) + 2 > |h| → 0
4. |d·(x − 1) + 2 ≤ |h| → min {d · (x − 1) + 2 .. min {d · x + 1, |h|}} by key ∘ h
  fi
end minchild;
```

*Notes.* See §1.3 for a discussion of our notation for maps. The complicated expression in line 4 selects from among the set of integers from $d(x-1)+2$ to $\min\{dx+1, |h|\}$, i.e. the children of $x$, an integer $c$ such that $key(h(c))$ is minimum, i.e. a node containing an item of minimum key. □

The best way to implement the map representing a $d$-heap is as an array of positions from one to the maximum possible heap size. We must also store an integer giving the size of the heap and, if arbitrary deletion is necessary, every item $i$ must have a heap index giving its position $h^{-1}(i)$ in the heap. If arbitrary deletion is unnecessary we need not store heap indices, provided we customize deletemin to call siftdown directly. (A call to delete from deletemin will never result in a sift-up.)

We have written siftup and siftdown so that they can be used as heap operations to restore heap order after changing the key of an item. If the key of item $i$ in heap $h$ decreases, the call siftup $(i, h^{-1}(i), h)$ will restore heap order; if the key increases, siftdown $(i, h^{-1}(i), h)$ will restore heap order. We shall use this capability in Chapter 7.

The last heap operation we must implement is makeheap. We can initialize a $d$-heap by performing $n$ insertions, but then the time bound is $O(n \log_d n)$. A better method is to build a complete $d$-ary tree of the items in arbitrary order and then execute siftdown $(h(x), x, h)$ for $x = n, n-1, \cdots, 1$, where $h$ is the map defining the item positions. The running time of this method is bounded by a constant times the sum

$$\sum_{i=0}^{\infty} \frac{n(i+1)}{d^i} = O(n),$$

since there are at most $n/d^i$ nodes of height $i$ in a complete $d$-ary tree of $n$ nodes.

The following program implements makeheap:

```
heap function makeheap (set s);
    map h;
    h: = { };
    for i ∈ s → h(|h| + 1) := i rof;
    for x = |s|, |s| − 1 . . 1 → siftdown (h(x), x, h) rof;
    return h
end makeheap;
```

We close this section with some history and a remark. Williams, building on the earlier TREESORT algorithm of Floyd, invented 2-heaps and discovered how to store them as arrays [6], [10]. Johnson [7] suggested the generalization to $d > 2$. An analysis of the constant factor involved in the timing of the heap operations suggests that the choice $d = 3$ or 4 dominates the choice $d = 2$ in all circumstances, although this requires experimental confirmation.

**3.3. Leftist heaps.** The $d$-heaps of §3.2 are not easy to meld. In this section we shall study leftist heaps, which provide an alternative to $d$-heaps when melding is necessary. Knuth [8] coined the term "leftist" for his version of a data structure invented by Crane [5].

If $x$ is a node in a full binary tree, we define the *rank* of $x$ to be the minimum length of a path from $x$ to an external node. That is, *rank* $(x) = 0$ if $x$ is an external node, *rank* $(x) = 1 + \min \{$*rank* (*left* $(x))$, *rank* (*right* $(x))\}$ if $x$ is an internal node. A full binary tree is *leftist* if *rank* (*left* $(x)) \geq$ *rank* (*right* $(x))$ for every internal

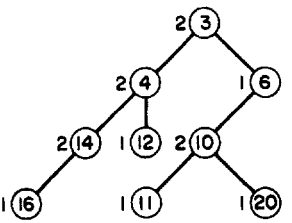



FIG. 3.5. *A leftist heap. External nodes are omitted. Numbers near nodes are ranks.*

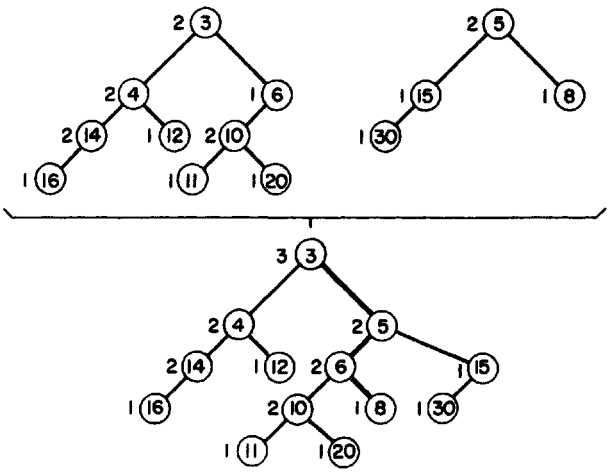



FIG. 3.6. *Merging two leftist heaps. Merged path is marked by heavy lines. To maintain the leftist property, children of 5 are swapped.*

node $x$. In a leftist tree the right path is a shortest path from the root to an external node. It is easy to prove by induction that this path has length at most $lg\ n$.

A *leftist heap* is a leftist tree containing one item per internal node, with items arranged in heap order. (See Fig. 3.5.) We shall treat leftist heaps as being endogenous; that is, the items themselves are the tree nodes. Since the external nodes contain no information, we shall assume that every external node is **null**. To save tests in our programs we assume that *rank* (**null**) is initialized to zero. To store a leftist heap, we need two pointers and one integer for each internal node, giving its left and right children and its rank. To identify the heap, we use a pointer to its root.

The fundamental operation on leftist heaps is melding. To meld two heaps, we merge their right paths, arranging nodes in nondecreasing order by key. Then we recompute the ranks of the nodes on the merged path and make the tree leftist by swapping left and right children as necessary. (See Fig. 3.6.) The entire meld takes $O(\log n)$ time, where $n$ is the number of nodes in the two heaps.

To insert an item into a leftist heap, we make the item into a one-node heap and meld it with the existing heap. To delete a minimum item, we remove the root and meld its left and right subtrees. Both these operations take $O(\log n)$ time. The following programs implement findmin, meld, insert, and deletemin on leftist heaps:

```
item function findmin (heap h);
   return h
end findmin;

heap function meld (heap h1, h2);
   return if h1 = null → h2 | h2 = null → h1
          | h1 ≠ null and h2 ≠ null → mesh (h1, h2) fi
end meld;
```

```
heap function mesh (heap h1, h2);
  if key (h1) > key (h2) → h1 ↔ h2 fi;
  right (h1) := if right (h1) = null → h2
                | right (h1) ≠ null → mesh (right (h1), h2) fi;
  if rank (left (h1)) < rank (right (h1)) → left (h1) ↔ right (h1) fi;
  rank (h1) := rank (right (h1)) + 1;
  return h1
end mesh;
```

*Note.* The meld program is merely a driver to deal efficiently with the special case of a **null** input; the mesh program performs the actual melding. □

```
procedure insert (item i, modifies heap h);
  left (i), right (i), rank (i) := null, null; 1;
  h := meld (i, h)
end insert;

item function deletemin (modifies heap h);
  item i;
  i := h;
  h := meld (left (h), right (h));
  return i;
end deletemin;
```

Before studying heap initialization and arbitrary deletion, let us consider two other useful heap operations:

listmin ($x$, $h$): Return a list containing all items in heap $h$ with key not exceeding real number $x$.

heapify ($q$): Return a heap formed by melding all the heaps on the list $q$. This operation assumes that the heaps on $q$ are disjoint and destroys both $q$ and the heaps on it.

The heap order of leftist heaps allows us to carry out listmin in time proportional to the number of elements listed: we perform a preorder traversal starting from the root of the heap, listing every encountered item with key not exceeding $x$ and immediately retreating from each item with key exceeding $x$. The same method works on $d$-heaps but takes $O(dk)$ rather than $O(k)$ time, where $k$ is the size of the output list. The following program implements listmin on leftist heaps:

```
list function listmin (real x, heap h);
  return if h = null or key (h) > x → [ ]
          | h ≠ null and key (h) ≤ x → [h] & listmin (left (h))
                                           & listmin (right (h))
          fi
end listmin;
```

To carry out heapify, we treat the input list as a queue. We repeat the following step until only one heap remains, which we return: Remove the first two heaps from

the queue, meld them, and add the new heap to the end of the queue. The following program implements this method:

```
heap function heapify (list q);
  do |q| ≥ 2 → q := q[3 . .] & meld (q(1), q(2)) od;
  return if q = [ ] → null | q ≠ [ ] → q(1) fi
end heapify;
```

In order to analyze the running time of heapify, let us consider one pass through the queue. Let $k$ be the number of heaps on the queue and $n$ the total number of items they contain. After $\lceil k/2 \rceil$ melds, every heap has been melded with another, leaving at most $\lfloor k/2 \rfloor$ heaps. The total time for these melds is

$$O\left(\sum_{i=1}^{\lfloor k/2 \rfloor} \max\{1, \log n_i\}\right),$$

where $n_i$ is the number of items in the $i$th heap remaining after the pass. We have $0 \le n_i \le n$ and $\sum_{i=1}^{\lfloor k/2 \rfloor} n_i = n$. These constraints imply that the time for one pass through the queue is $O(k \max\{1, \log (n/k)\})$. The time for the entire heapify is

$$O\left(\sum_{i=0}^{\lfloor \lg k \rfloor} \frac{k}{2^i} \max\left\{1, \log \frac{n2^i}{k}\right\}\right) = O\left(k \max\left\{1, \log \frac{n}{k}\right\}\right),$$

where $k$ is the number of original heaps and $n$ is the total number of items they contain.

We can make a leftist heap of $n$ items in $O(n \log n)$ time by repeated insertion, but heapify gives a better method: We make each item into a one-item heap and apply heapify to a list of these heaps. In this case $k = n$ and the running time is $O(n)$. The following program implements makeheap using this method:

```
heap function makeheap (set s);
  list q;
  q := [ ];
  for i ∈ s → left (i), right (i), rank (i), q := null, null, 1, q & [i] rof;
  return heapify (q)
end makeheap;
```

The last heap operation is delete. It is possible to delete an arbitrary item from a leftist heap in $O(\log n)$ time if we add parent pointers to the tree representation. A better method for our purposes is to use *lazy deletion*, as proposed by Cheriton and Tarjan [4]: To delete an item, we merely mark it deleted; we carry out the actual deletion during a subsequent findmin or deletemin. To carry out findmin, we perform a preorder traversal of the tree, making a list of each nondeleted node all of whose proper ancestors are marked deleted; then we heapify the subtrees whose roots are on the list and return the root of the single tree resulting. Implementation of deletemin is similar. With this method we can also if we wish perform *lazy melding*: To meld two heaps we create a dummy node whose children are the roots of the two heaps to be melded. During findmin and deletemin we treat dummy nodes as if they were marked deleted. The following programs implement lazy melding

and findmin using this method. (We leave as an exercise implementing delete and deletemin and modifying insert and makeheap to mark newly inserted items nondeleted.) The program lazymeld marks dummy nodes by giving them a key of minus infinity.

```
heap function lazymeld (heap h1, h2);
  if h1 = null → return h2 | h2 = null → return h1
    | h1 ≠ null and h2 ≠ null →
      item i;
      i := create item;
      if rank (h1) < rank (h2) → h1 ↔ h2 fi;
      left (i), right (i), key (i), rank (i) := h1, h2, −∞, rank (h2) + 1;
      return i
  fi
end lazymeld;

item function findmin (modifies heap h);
  h := heapify (purge (h));
  return h
end findmin;

list function purge (heap h);
  return if h = null → [ ]
          | h ≠ null and key (h) > −∞ and not deleted (h) → [h]
          | h ≠ null and (key (h) = −∞ or deleted (h)) →
            purge (left (h)) & purge (right (h))
         fi
end purge;
```

*Note.* The purge program makes a list of all nondummy, nondeleted nodes in $h$ all of whose proper ancestors are dummy or deleted. The heapify program must use the original version of meld rather than lazymeld. □

With lazy melding and lazy deletion, the time for a meld or deletion is $O(1)$; the time for a findmin is $O(k \max\{1, \log(n/(k+1))\})$, where $k$ is the number of dummy and deleted items discarded from the heap. Lazy deletion is especially useful when there is a way to mark deleted items implicitly. We shall see an example of this in Chapter 6.

**3.4. Remarks.** There are several other heap operations that can be added to our repertoire with little loss in efficiency. One such operation is addtokeys $(x, h)$, which adds real number $x$ to the key of every item in heap $h$. To implement this operation we change our heap representation so that, instead of storing a key with each item, we store a key difference:

$$\Delta key(x) = \begin{cases} key(x) & \text{if } x \text{ is a tree root,} \\ key(x) - key(p(x)) & \text{if } p(x) \text{ is the parent of } x. \end{cases}$$

With this representation we can evaluate the key of an item $x$ by summing key differences along the path from $x$ to the tree root. The operation addtokeys $(x, h)$ takes $O(1)$ time: we add $x$ to the key difference of the tree root. The asymptotic running times of the other heap operations are unaffected by this change in the data structure. We shall see another use of storing differences in the next chapter. Cheriton and Tarjan [4] first proposed using key differences in heaps, borrowing the idea from an algorithm of Aho, Hopcroft and Ullman [2] for computing depths in trees.

Although leftist trees give a simple and efficient way to represent meldable heaps, almost any class of balanced trees will do almost as well; all we need is a definition of balance that allows rapid melding of two balanced trees containing items arranged in heap order. Empirical and theoretical evidence gathered by Brown [3] suggests that the class of "binomial trees" [9] gives the fastest implementation of meldable heaps when constant factors are taken into account. However, on binomial heaps the heap operations are harder to describe and implement than on leftist heaps.

## References


[1] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] ———, *On finding lowest common ancestors in trees*, SIAM J. Comput., 5 (1975), pp. 115–132.

[3] M. R. BROWN, *Implementation and analysis of binomial queue algorithms*, SIAM J. Comput., 7 (1978), pp. 298–319.

[4] D. CHERITON AND R. E. TARJAN, *Finding minimum spanning trees*, SIAM J. Comput., 5 (1976), pp. 724–742.

[5] C. A. CRANE, *Linear lists and priority queues as balanced binary trees*, Tech. Rep. STAN-CS-72-259, Computer Science Dept., Stanford Univ., Stanford, CA, 1972.

[6] R. W. FLOYD, *Algorithm* 245: *Treesort* 3, Comm. ACM, 7 (1964), p. 701.

[7] D. B. JOHNSON, *Priority queues with update and finding minimum spanning trees*, Inform. Process. Lett., 4 (1975), pp. 53–57.

[8] D. E. KNUTH, *The Art of Computer Programming, Vol.* 3: *Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[9] J. VUILLEMIN, *A data structure for manipulating priority queues*, Comm. ACM, 21 (1978), pp. 309–314.

[10] J. W. J. WILLIAMS, *Algorithm* 232: *Heapsort*, Comm. ACM, 7 (1964), pp. 347–348.

*This page intentionally left blank*

# CHAPTER 4

# Search Trees

**4.1. Sorted sets and binary search trees.** Heap order is not the only way to arrange items in a tree. Symmetric order, which we shall study in this chapter, is perhaps even more useful. We consider the problem of maintaining one or more sets of items under the following operations, where each item has a distinct key chosen from a totally ordered universe:

access $(k, s)$: Return the item in set $s$ with key $k$; if no item in $s$ has key $k$, return **null.**
insert $(i, s)$: Insert item $i$ into set $s$, not previously containing $i$.
delete $(i, s)$: Delete item $i$ from set $s$.

We can regard each set as totally ordered by key; thus we shall call this problem the *sorted set problem.* The following operation creates a new sorted set.

makesortedset: Return a new, empty sorted set.

We shall also need two more drastic update operations on sorted sets:

join $(s_1, i, s_2)$: Return the sorted set formed by combining disjoint sets $s_1$, $\{i\}$, and $s_2$. This operation destroys $s_1$ and $s_2$ and assumes that every item in $s_1$ has key less than *key* $(i)$ and every item in $s_2$ has key greater than *key* $(i)$.
split $(i, s)$: Split sorted set $s$, containing item $i$, into three sets: $s_1$, containing all items with key less than *key* $(i)$; $\{i\}$; and $s_2$, containing all items with key greater than *key* $(i)$. Return the pair $[s_1, s_2]$. This operation destroys $s$.

We shall assume that every item is in at most one sorted set. This means that the parameter $s$ in delete and split is redundant, and when convenient or necessary we shall omit it.

We can represent a sorted set by a full binary tree containing one item per internal node, with items arranged in symmetric order: if node $x$ contains an item with key $k$, then every item in the left subtree of $x$ has key less than $k$, and every item in the right subtree of $x$ has key greater than $k$. (See Fig. 4.1.) We call this data structure a *binary search tree.*

We shall regard binary search trees as endogenous (the items themselves are the tree nodes) and every external node as **null.** With each node $x$ we store three pointers: *left* $(x)$, *right* $(x)$, and $p(x)$, to the left child, right child, and parent of $x$, respectively. (We can omit parent pointers at some loss in convenience, as we shall discuss later.) We identify a binary search tree by a pointer to its root; a **null** pointer denotes an empty tree.

To access the item with key $k$ in a binary search tree, we carry out a search by beginning at the root $x$ and repeating the following step until *key* $(x) = k$ or

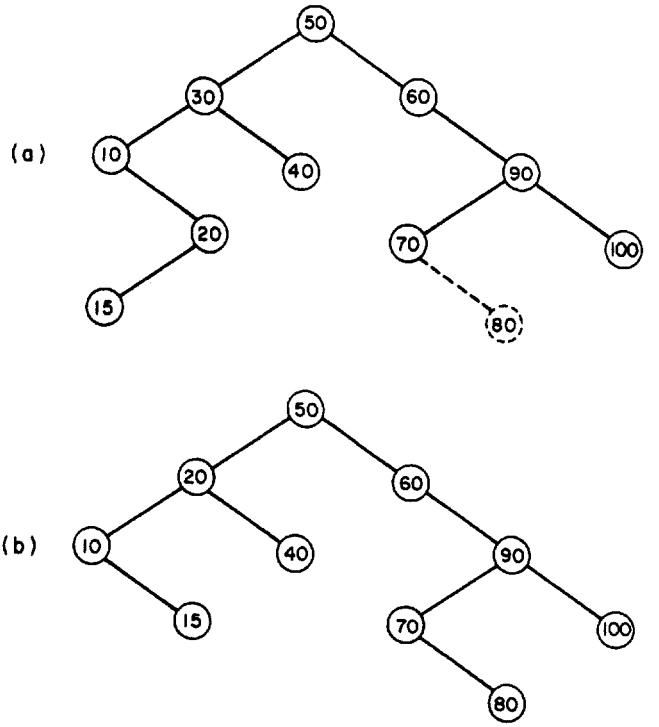



FIG. 4.1. *Insertion and deletion in a binary search tree. External nodes are omitted.*(a) *Insertion of item with key* 80. (b) *Deletion of item with key* 30. *It is replaced by its predecessor, with key* 20, *which is in turn replaced by its left child, with key* 15.

$x$ = **null:** If *key* $(x) > k$, replace $x$ by *left* $(x)$; otherwise (*key* $(x) < k$) replace $x$ by *right* $(x)$. When the search stops, we return $x$. To insert an item $i$, we perform a similar search, using $k = key\ (i)$ and maintaining a "trailing pointer" to the parent of the current node $x$. The search will stop at a (**null**) external node, which we replace by $i$. (See Fig. 4.1.)

In search trees, as in heaps, deletion is more complicated than insertion. To delete an item $i$, we test whether $i$ has a **null** child. If not, we swap $i$ with its symmetric-order predecessor $j$, found by following right pointers from *left* $(i)$ until reaching a node with a **null** right child. Although $i$ may now be out of order, it has at least one **null** child. To delete $i$ we replace it by its non-**null** child, if any, or by **null.** (See Fig. 4.1.)

A join of two search trees is easy: We perform join $(s_1, i, s_2)$ by making the roots of the trees representing $s_1$ and $s_2$ the left and right children of $i$, respectively; then we return $i$. Splitting a search tree requires a sequence of joins. (See Fig. 4.2.) To split a search tree at a node $i$, we traverse the path from $i$ to the root of the tree, deleting all edges on the path and incident to it. This breaks the original tree into a collection of subtrees, each of which is either a node on the path or has a root that is a child of a node on the path. We join the subtrees containing items smaller than $i$ to

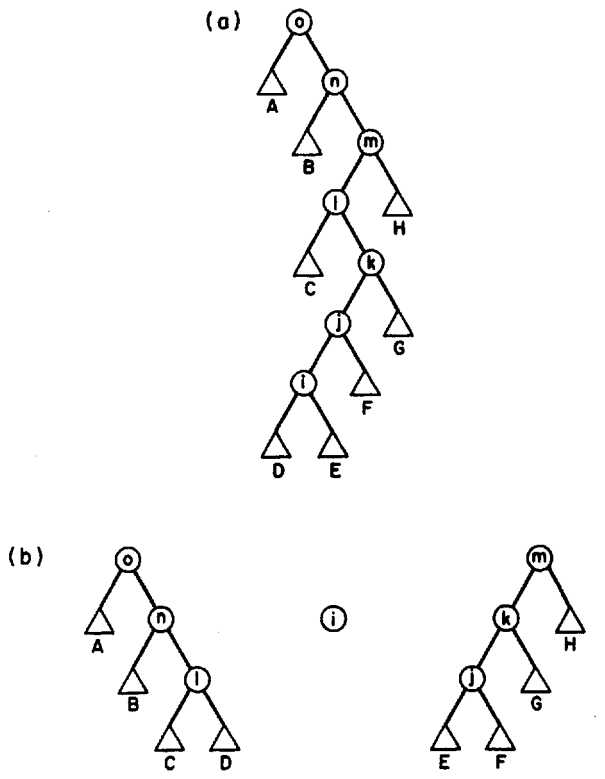



FIG. 4.2. *Splitting a binary search tree.* (a) *Original tree. Triangles denote subtrees.* (b) *Result of splitting at node i.*

form a left tree and the subtrees containing items larger than $i$ to form a right tree. A more precise description of the process is as follows.

To carry out split $(i, s)$, we initialize the current node $x$, the previous node $y$, the left tree $s_1$ and the right tree $s_2$, to be the parent of $i$, $i$ itself, the left subtree of $i$ and the right subtree of $i$, respectively. Then we repeat the following step until $x$ is **null:** If $y$ is the left child of $x$, simultaneously replace $x$, $y$ and $s_2$ by $p(x)$, $x$ and the join of $s_2$, $x$ and the right subtree of $x$; otherwise ($y$ is the right child of $x$) simultaneously replace $x$, $y$ and $s_1$ by $p(x)$, $x$ and the join of the left subtree of $x$, $x$ and $s_1$.

With a representation of sorted sets by binary search trees, accessing an item $i$ takes time proportional to the depth of $i$ in its search tree. Inserting an item $i$ takes time proportional to the depth of $i$ after the insertion. Deleting an item $i$ takes time proportional to the depth of $i$ if $i$ has a **null** child or to the depth of its predecessor if not. A join takes $O(1)$ time. Splitting at an item $i$ takes time proportional to the depth of $i$. A binary search tree of $n$ items can have depth $n - 1$ (if it consists of a path of $n$ internal nodes); thus all the operations except join have a worst-case running time of $O(n)$. In §4.2 and 4.3 we shall discuss two ways of controlling the tree structure to reduce this bound.

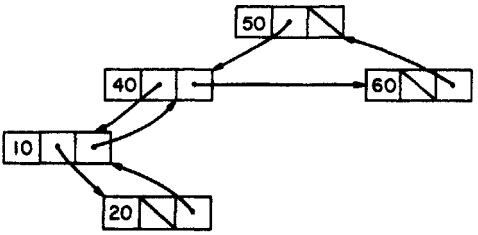



FIG. 4.3. *Representation of parents and children in a binary tree using two pointers per node. Empty fields contain* **null** *pointers.*

The only operations in which we have used parent pointers are deletion and splitting. If we have access to the root of the tree in which such an operation is to take place, we can perform the operation in a top-down fashion without using parent pointers, saving space amounting to one pointer per node in our data structure. A way to save the same amount of space without sacrificing parent pointers is to use the tree representation shown in Fig. 4.3, in which each node points to its left child, or to its right child if the left child is **null**, and a left child points to its right sibling, or to its parent if it has no right sibling. Accessing the parent of a node or either of its children requires following at most two pointers; thus this representation trades time for space.

**4.2. Balanced binary trees.** The standard way to make search tree operations efficient in the worst case is to impose a *balance condition* that forces the depth of an $n$-node tree to be $O(\log n)$. This requires rebalancing the tree after (or during) each update operation. Kinds of balanced search trees for which this is possible include height-balanced binary trees [1], [6], [10], weight-balanced binary trees [12], $B$-trees and their variants [4], [8], [9], [11] (which are not binary but have variable node degree) and many others. In this section we shall study one kind of balanced search tree that stands out as possessing most of the desirable features of all the others. We call these trees balanced binary trees.

A *balanced binary tree* is a full binary tree each of whose nodes $x$ has an integer rank, denoted by *rank* $(x)$, such that the ranks have the following properties (see Fig. 4.4):

(i) If $x$ is any node with a parent, *rank* $(x) \leq$ *rank* $(p(x)) \leq$ *rank* $(x) + 1$.

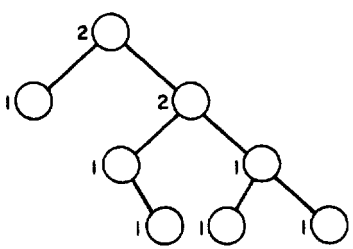



FIG. 4.4. *A balanced binary tree. Numbers next to nodes are ranks.*

(ii) If $x$ is any node with a grandparent, *rank* $(x) <$ *rank* $(p^2(x))$.

(iii) If $x$ is an external node, *rank* $(x) = 0$ and *rank* $(p(x)) = 1$ if $x$ has a parent.

Bayer [3] invented balanced binary trees, calling them "symmetric binary $B$-trees." Guibas and Sedgewick [7] studied the properties of these and related kinds of trees, collectively calling them "red-black trees." Olivié [13], [14] rediscovered balanced binary trees using a completely different definition; he called them "half-balanced trees."

Let us call a node $x$ *black* if *rank* $(p(x)) =$ *rank* $(x) + 1$ or $p(x)$ is undefined and red if *rank* $(p(x)) =$ *rank* $(x)$. By (i), every node is either red or black. By (ii), every red node has a black parent. By (iii), every external node is black, and all paths from a given node to an external node contain the same number of black nodes. We can use these properties of red and black nodes as an alternative definition of balanced binary trees. To represent a balanced binary tree, we store with each internal node a bit that indicates its color.

If in a balanced binary tree we condense every red node into its parent, we obtain a 2, 4 *tree:* every internal node has two, three, or four children and all external nodes have the same depth. Conversely, we can convert any 2, 4 tree into a balanced binary tree by binarizing every node. (See Fig. 4.5.) This correspondence is not one-to-one, since there are two representations of a 3-node, but it does give a third way to define balanced binary trees. A fourth way is Olivié's definition: a binary tree is balanced if and only if it is *half-balanced:* for every node $x$, the length of the longest path from $x$ to an external node is at most twice the length of the shortest path from $x$ to an external node [13].

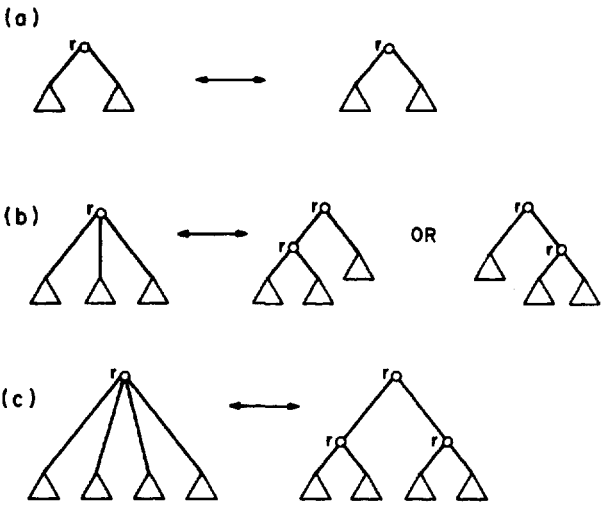



FIG. 4.5. *Correspondence between 2, 4 and balanced binary trees. Letters next to nodes denote height in 2, 4 tree, rank in balanced binary tree. Triangles denote subtrees with roots of height or rank* $r - 1$. (a) *2-node.* (b) *3-node. There are two binary forms.* (c) *4-node.*

LEMMA 4.1. *A node of rank k has height at most 2k and has at least* $2^{k+1}-1$ *descendants. Thus a balanced binary tree with n internal nodes has depth at most* $2\lfloor \lg (n+1) \rfloor$.

*Proof.* The first part of the lemma, which implies the second part, is immediate by induction on $k$. □

Lemma 4.1 implies that the access time in a balanced binary search tree is $O(\log n)$. We can also rebalance such a tree after an insertion or deletion in $O(\log n)$ time. The atomic operations necessary for rebalancing after an insertion are *promotion,* which increases the rank of a node by one (called "color flip" by Guibas and Sedgewick), *single rotation* and *double rotation* (which consists of two single rotations). Each such operation takes $O(1)$ time.

Figure 4.6 illustrates these operations. When a single or double rotation is performed, we refer to the root of the affected subtree as the node at which the rotation takes place. Single and double rotations preserve symmetric order, which is what makes them so useful for rebalancing.

The effect of an insertion is to replace a node of rank zero (a **null** external node) by a node of rank one (the inserted node). This may violate property (ii) by creating a red node $x$ with a red parent. To rebalance the tree, we test whether the (black) grandparent $p^2(x)$ of $x$ has two red children. If so, we promote $p^2(x)$, replace $x$ by $p^2(x)$, and test for a new violation of (ii). If not ($p^2(x)$ has a black child), we perform a single or double rotation as appropriate, which completes the rebalancing. (See Fig. 4.7.) With this method, rebalancing after an insertion requires a sequence of promotions followed by at most two single rotations, for a total of $O(\log n)$ time.

To rebalance after a deletion, we use *demotion,* which decreases the rank of a node by one, in place of promotion. (See Fig. 4.6.) The effect of a deletion is to

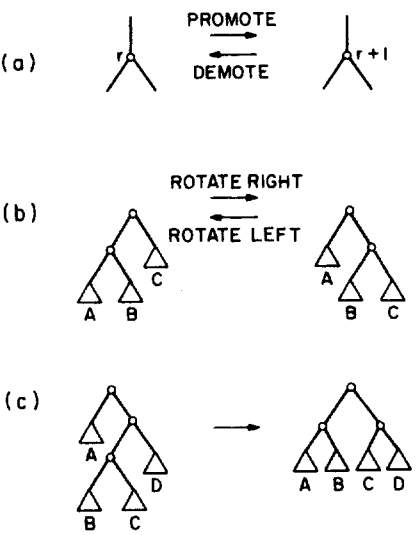



FIG. 4.6. *Atomic operations for insertion and deletion.* (a) *Promotion/demotion. Circle denotes a node of rank indicated.* (b) *Single rotation. Triangles denote subtrees.* (c) *Double rotation. There is a symmetric variant.*

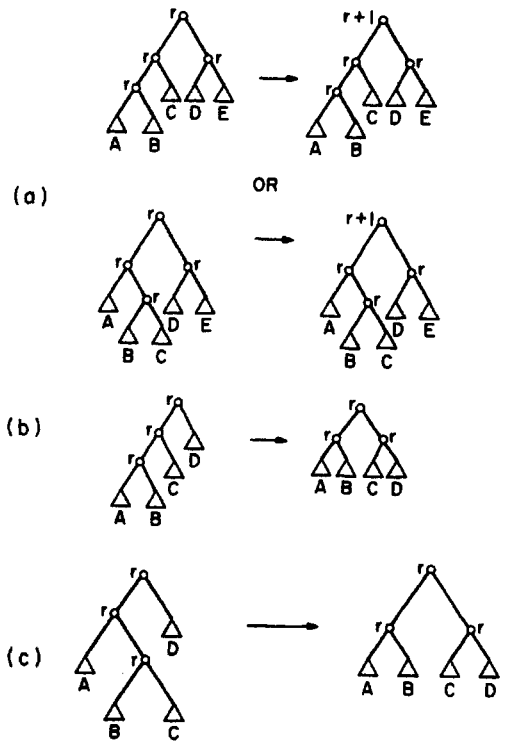



FIG. 4.7. *Insertion steps. Each case has a symmetric variant (not shown). Circles denote nodes of indicated ranks. Triangles denote subtrees with black roots. The current node x is the parent of the root of subtree B.* (a) *Promotion.* (b) *Single rotation.* (c) *Double rotation.*

replace a node in the tree by another node $x$ of one lower rank. The rank of $x$ may be two less than the rank of its parent, which violates property (i). To rebalance the tree, we apply the appropriate case below (see Fig. 4.8).

*Case* 1. *The sibling y of x is black.*

*Subcase* 1a. *Both children of y are black.* Demote $p(x)$, replace $x$ by $p(x)$, and test for a new violation of (i).

*Subcase* 1b. *The child of y farthest from x is red.* Perform a single rotation at $p(x)$ and stop.

*Subcase* 1c. *The child of y nearest x is red and its sibling is black.* Perform a double rotation at $p(x)$ and stop.

*Case* 2. *The sibling y of x is red.* Both children of $y$ are black by (i). Perform a single rotation at $p(x)$ and proceed as in Case 1. (After the rotation the new sibling of $x$ is black.) When applying Case 1 in this situation, Subcase 1a cannot cause a new violation of (i); thus all subcases are terminal.

With this method, rebalancing after a deletion requires a sequence of demotions followed by at most three single rotations, for a total of $O(\log n)$ time. It is also possible to rebalance in a top-down fashion during an insertion or deletion; this takes $O(\log n)$ rather than $O(1)$ rotations but still takes $O(\log n)$ time [7].

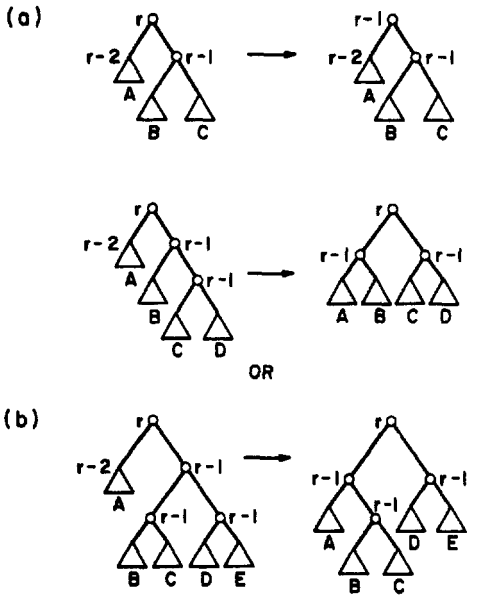



FIG. 4.8. *Deletion steps. Each case has a symmetric variant (not shown). The current node x is the root of subtree A. Roots of lettered subtrees other than A are black.* (a) *Subcase* 1a. *Demotion.* (b) *Subcase* 1b. *Single rotation.*

To join balanced binary trees it is useful to store with each tree root its rank; then while descending through a tree we can compute the rank of each node visited in $O(1)$ time per node. To perform join $(s_1, i, s_2)$, we compare *rank* $(s_1)$ with *rank* $(s_2)$. If *rank* $(s_1) \geq$ *rank* $(s_2)$, we follow right pointers from $s_1$ until reaching a node $x$ with *rank* $(x) =$ *rank* $(s_2)$. We replace $x$ and its subtree by $i$, making $x$ the left child

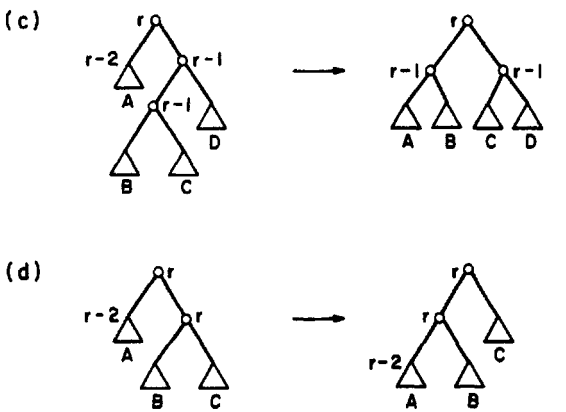



FIG. 4.8 (continued). (c) *Subcase* 1c. *Double rotation.* (d) *Case* 2. *Single rotation to produce terminating instance of Case* 1. *Demotion of bottom node of rank r does not cause a new violation of property* (*i*).

and $s_2$ the right child of $i$. We define the rank of $i$ to be $rank\ (x) + 1$, thus making $x$ and $s_2$ black. Then we rebalance as in an insertion, starting from $i$. The case $rank\ (s_1) < rank\ (s_2)$ is symmetric. Such a join takes $O(|\ rank\ (s_1) - rank\ (s_2)\ |) = O(\log n)$ time and produces a tree whose root has rank $\max\{rank\ (s_1), rank\ (s_2)\}$ or $\max\{\ rank\ (s_1, rank\ (s_2)\} + 1$; in the latter case both children of the root are black.

To split a balanced binary tree at an item $i$, we perform a sequence of joins exactly as described in §4.1, using the method above for each join. The time bounds for the joins that construct the left tree form a telescoping series summing to $O(\log n)$. This is also true for the joins that construct the right tree. More precisely, the split algorithm maintains the following invariant, where $x$, $y$, $s_1$ and $s_2$ are the current node, the previous node and the roots of the left and right trees, respectively (see the description of split in §1): If $y$ was black in the unsplit tree, then $\max\{rank\ (s_1), rank\ (s_2)\} \leqq rank\ (x)$, and the time spent on the split so far is $O(rank\ (s_1) + rank\ (s_2) + rank\ (x))$. Thus a split takes $O(\log n)$ time.

In addition to having an $O(\log n)$ time bound for each of the sorted set operations, balanced binary trees have other useful properties. All the operations have alternative top-down implementations that make concurrent search and updating easy [7]. Bottom-up rebalancing after an insertion or a deletion, in addition to needing only $O(1)$ rotations, takes $O(1)$ total time if the time is amortized over a sequence of updates [8], [9], [11]. Furthermore, time-consuming insertions and deletions are rare; in particular, operations that take time $t$ occur with a frequency that is an exponentially decreasing function of $t$ [8], [9]. This makes balanced binary trees ideal for use as "finger" search trees, in which we maintain one or more pointers to search starting points; the time for an access, insertion, or deletion is $O(\log d)$, where $d$ is the number of items between the search starting point and the accessed item [8], [9], [11].

**4.3. Self-adjusting binary trees.** There is another way to obtain $O(\log n)$ operation times in binary search trees. If we are willing to settle for an amortized rather than worst-case time bound per operation, we do not need to maintain any explicit balance condition. Instead, we adjust the tree every time we do an access or update, using a method that depends only on the structure of the access path. Allen and Munro [2] and Bitner [5] studied such restructuring heuristics for binary trees, but none of their methods has an $O(\log n)$ amortized time bound; they are all $O(n)$. Sleator and Tarjan [15] recently discovered a heuristic that does give an $O(\log n)$ amortized time bound, which we shall study here.

The restructuring heuristic proposed by Sleator and Tarjan is the *splay* operation. To splay a binary tree at an internal node $x$, we begin at $x$ and traverse the path to the root, performing a single rotation at each node. We perform the rotations in pairs, in an order that depends on the structure of the tree. The splay consists of repeating the following *splay step* until $p(x)$ is undefined (see Fig. 4.9): If $x$ has a parent but no grandparent, we rotate at $p(x)$. If $x$ has a grandparent and $x$ and $p(x)$ are both left or both right children, we rotate at $p^2(x)$ and then at $p(x)$. If $x$ has a grandparent and $x$ is a left and $p(x)$ a right child, or vice versa, we rotate at $p(x)$

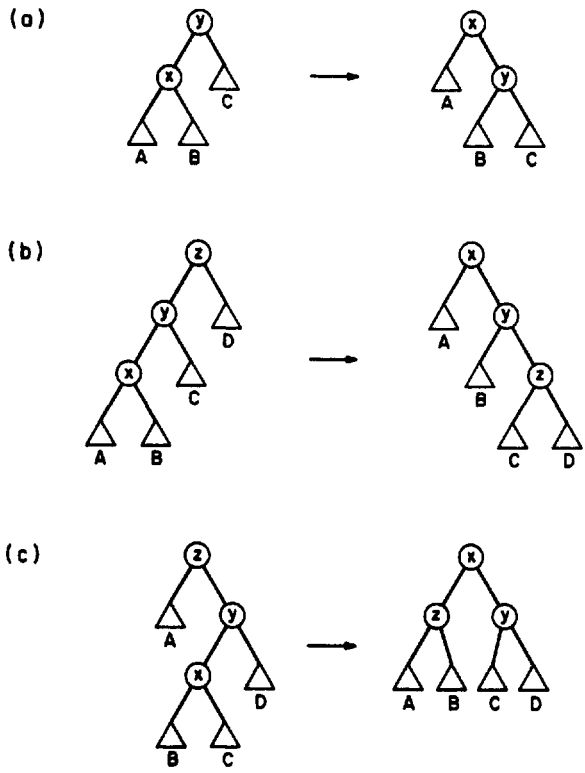



FIG. 4.9. *Cases of splay step. Each case has a symmetric variant (not shown).* (a) *Terminating single rotation.* (b) *Two single rotations.* (c) *Double rotation.*

and then at the new parent of $x$ (the old grandparent of $x$). The overall effect of the splay is to move $x$ to the root of the tree while rearranging the rest of the original path to $x$. (See Fig. 4.10.)

To make binary search trees self-adjusting, we perform a splay during each access or update operation other than a join. More precisely, after accessing or inserting an item $i$ we splay at $i$. After deleting an item $i$ we splay at its parent just before the deletion (this is the original parent of $i$'s predecessor if $i$ and the predecessor were swapped). Before splitting at an item $i$ we splay at $i$. This makes the split easy, since it moves $i$ to the root: we merely return the left and right subtrees of the root. In each case the time for the operation is proportional to the length of the path along which the splay proceeds, which we call the *splay path*. Although we have described splay as a bottom-up operation, an appropriate variant will work top-down [15]. Thus for example when accessing or inserting an item $i$ we can splay at $i$ while we search for $i$.

Splaying is reminiscent of path compression and even more of path halving (see Chapter 2) in the way it changes the tree structure. Although the techniques that Tarjan and van Leeuwen [17] used to analyze path halving also apply to splaying,

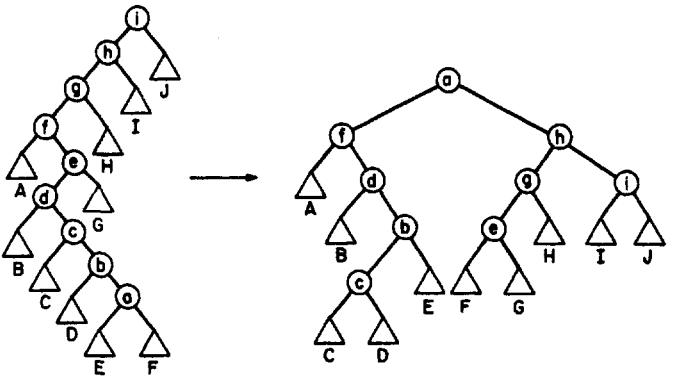



FIG. 4.10. *Splay at node a.*

another method gives a simpler analysis. In the remainder of this section we shall derive Sleator and Tarjan's $O(m \log n)$ bound on the total time required for a sequence of $m$ sorted set operations beginning with no sorted sets, where $n$ is the number of insert and join operations (items).

We use a credit-debit accounting scheme (see Chapter 2), keeping track only of credits. We assign to each item $i$ an *individual weight* $iw(i)$. The individual weights are arbitrary positive real numbers, whose value we shall choose later. We define the *total weight* $tw(x)$ of a node $x$ in a binary search tree to be the sum of the individual weights of all descendants of $x$, including $x$ itself. Finally, we define the *rank* of a node $x$ to be $rank(x) = \lfloor \lg tw(x) \rfloor$. We maintain the following *credit invariant:* any internal node $x$ holds $rank(x)$ credits.

LEMMA 4.2. *Splaying a tree with root $v$ at a node $x$ while maintaining the credit invariant takes* $3(rank(v) - rank(x)) + 1$ *credits.*

*Proof.* Consider a splay step involving the nodes $x$, $y = p(x)$, and $z = p^2(x)$, where $p$ denotes the parent function before the step. Let rank and rank', tw and tw' denote the rank and total weight functions before and after the step respectively. To this step we allocate $3(rank'(x) - rank(x))$ credits and one additional credit if this is the last step. There are three cases (see Fig. 4.9).

*Case* 1. *Node $z$ is undefined.* This is the last step of the splay, and the extra credit allocated to the step pays for it. We have $rank'(x) = rank(y)$. Thus the number of credits that must be added to the tree to maintain the invariant is $rank'(y) - rank(x) \leq rank'(x) - rank(x)$, which is one third of the remaining credits on hand.

*Case* 2. *Node $z$ is defined and $x$ and $y$ are both left or both right children.* We have $rank'(x) = rank(z)$. The number of credits that must be added to the tree to maintain the invariant is $rank'(y) + rank'(z) - rank(y) - rank(x) \leq 2(rank'(x) - rank(x))$, which is two thirds of the credits on hand. If $rank'(x) > rank(x)$, there is at least one extra credit on hand to pay for the step. Otherwise, $rank'(x) = rank(x) = rank(y) = rank(z)$. Furthermore $tw(x) + tw'(z) \leq tw'(x)$, which implies that $rank'(z) < rank'(x)$. (If

$rank'(z) = rank'(x)$, $tw'(x) > 2^{rank(x)} + 2^{rank'(z)} \geq 2^{rank'(x)+1}$, a contradiction.) Maintaining the invariant thus frees $rank(y) + rank(x) - rank'(y) - rank'(z) \geq 1$ credits, one of which will pay for the step.

*Case* 3. *Node z is defined and x is a left and y is a right child, or vice versa.* The credit accounting in this case is the same as in Case 2, with the following exception: If $rank'(x) = rank(x)$, we have $\min\{rank'(y), rank'(z)\} < rank'(x)$ since $tw'(y) + tw'(z) \leq tw'(x)$, and maintaining the invariant frees $rank(y) + rank(x) - rank'(y) - rank'(z) = 2\ rank'(x) - rank'(y) - rank'(z) \geq 1$ credit to pay for the step.

Summing over all steps of a splay, we find that the total number of credits used is $3(rank'(x) - rank(x)) + 1 = 3(rank(v) - rank(x)) + 1$, where *rank* and *rank'* denote the rank function before and after the entire splay. □

In order to complete the analysis, we must consider the effects of insertion, deletion, join and split on the ranks of nodes. To simplify matters, let us define the individual weight of every item to be 1. Then every node has a rank in the range $[0\,..\,\lfloor \lg n \rfloor]$, and Lemma 2 gives a bound of $3\lfloor \lg n \rfloor + 1$ credits for splaying. Inserting a new item $i$ without splaying takes at most $\lfloor \lg n \rfloor + 1$ credits, since only nodes along the path from $i$ to the root of the tree gain in total weight, and of these only the ones with an old total weight of $2^k - 1$ for some $k \in [0\,..\,\lfloor \lg n \rfloor]$ gain (by one) in rank. Deleting an item $i$ without splaying causes either a net decrease or no change in the number of credits in the tree. Joining two trees requires placing at most $\lfloor \lg n \rfloor$ credits on the new root, and splitting a tree frees the credits (if any) on the old root. Thus we have the following theorem:

THEOREM 4.1. *The total time required for a sequence of m sorted set operations using self-adjusting trees, starting with no sorted sets, is* $O(m \log n)$, *where n is the number of insert and join operations (items).*

A more refined analysis using weights other than one shows that self-adjusting search trees are as efficient in an amortized sense as static optimum search trees, to within a constant factor [15]. Self-adjusting trees have other remarkable properties [15]. In the next chapter we shall study a problem in which the use of self-adjusting search trees gives an asymptotically more efficient algorithm than seems possible with any kind of balanced search trees.

## References

[1] G. M. ADEL'SON-VEL'SKII AND E. M. LANDIS, *An algorithm for the organization of information*, Soviet Math. Dokl., 3 (1962), pp. 1259–1262.

[2] B. ALLEN AND I. MUNRO, *Self-organizing search trees*, J. Assoc. Comput. Mach. 25 (1978), pp. 526–535.

[3] R. BAYER, *Symmetric binary B-trees: Data structure and maintenance algorithms*, Acta Inform., 1 (1972), pp. 290–306.

[4] R. BAYER AND E. MCCREIGHT, *Organization of large ordered indexes*, Acta Inform., 1 (1972), pp. 173–189.

[5] J. R. BITNER, *Heuristics that dynamically organize data structures*, SIAM J. Comput., 8 (1979), pp. 82–110.

[6] C. A. CRANE, *Linear lists and priority queues as balanced binary trees*, Tech. Rep. STAN-CS-72-259, Computer Science Dept., Stanford Univ., Stanford, CA, 1972.

[7] L. J. GUIBAS AND R. SEDGEWICK, *A dichromatic framework for balanced trees*, in Proc. Nineteenth Annual IEEE Symposium on Foundations of Computer Science, 1978, pp. 8–21.

[8] S. HUDDLESTON AND K. MEHLHORN, *A new data structure for representing sorted lists*, Acta Inform., 17 (1982), pp. 157–184.

[9] ——, *Robust balancing in B-trees*, in Lecture Notes in Computer Science 104, Springer-Verlag, 1981, pp. 234–244.

[10] D. E. KNUTH, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[11] D. MAIER AND S. C. SALVETER, *Hysterical B-trees*, Inform. Process Lett., 12 (1981), pp. 199–202.

[12] J. NIEVERGELT AND E. M. REINGOLD, *Binary search trees of bounded balance*, SIAM J. Comput., 2 (1973), pp. 33–43.

[13] H. OLIVIÉ, *A new class of balanced search trees: Half-balanced binary search trees*, Tech. Rep. 80-02, Interstedelijke Industriele Hogeschool Antwerpen-Mechelen, Antwerp, 1980.

[14] ——, *On $\alpha$-balanced binary search trees*, in Lecture Notes in Computer Science 104, Springer-Verlag, Berlin, 1981, pp. 98–108.

[15] D. D. SLEATOR AND R. E. TARJAN, *Self-adjusting binary trees*, Proc. Fifteenth Annual ACM Symposium on Theory of Computing, 1983, pp. 235–245.

[16] R. E. TARJAN, *Updating a balanced search tree in $O(1)$ rotations*, Inform. Proc. Lett., to appear.

[17] R. E. TARJAN AND J. VAN LEEUWEN, *Worst-case analysis of set union algorithms*, J. Assoc. Comput. Mach., to appear.

*This page intentionally left blank*

# CHAPTER 5

# Linking and Cutting Trees

**5.1. The problem of linking and cutting trees.** The trees we have studied in Chapters 2, 3 and 4 generally occur indirectly in network algorithms, as concrete representations of abstract data structures. Trees also occur more directly in such algorithms, either as part of the problem statement or because they express the behavior of the algorithm. In this chapter we shall study a generic tree manipulation problem that occurs in many network algorithms, including a maximum flow algorithm that we shall study in Chapter 8.

The problem is to maintain a collection of vertex-disjoint rooted trees that change over time as edges are added or deleted. More precisely, we wish to perform the following operations on such trees, each of whose vertices has a real-valued cost (see Fig. 5.1):

maketree ($v$): Create a new tree containing the single vertex $v$, previously in no tree, with cost zero.

findroot ($v$): Return the root of the tree containing vertex $v$.

findcost ($v$): Return the pair $[w, x]$ where $x$ is the minimum cost of a vertex on the tree path from $v$ to findroot ($v$) and $w$ is the last vertex (closest to the root) on this path of cost $x$.

addcost ($v, x$): Add real number $x$ to the cost of every vertex on the tree path from $v$ to findroot ($v$).

link ($v,w$): Combine the trees containing vertices $v$ and $w$ by adding the edge $[v, w]$. (We regard tree edges as directed from child to parent.) This operation assumes that $v$ and $w$ are in different trees and $v$ is a root.

cut ($v$): Divide the tree containing vertex $v$ into two trees by deleting the edge out of $v$. This operation assumes that $v$ is not a tree root.

In discussing this problem we shall use $m$ to denote the number of operations and $n$ to denote the number of vertices (maketree operations). One way to solve this problem is to store with each vertex its parent and its cost. With this representation each maketree, link, or cut operation takes $O(1)$ time, and each findroot, findcost, or addcost operation takes time proportional to the depth of the input vertex, which is $O(n)$.

By representing the structure of the trees implicitly, we can reduce the time for findroot, findcost and addcost to $O(\log n)$, amortized over a sequence of operations, while increasing the time for link and cut to $O(\log n)$. In the next two sections we shall study a data structure developed by Sleator and Tarjan [4] that has these bounds.

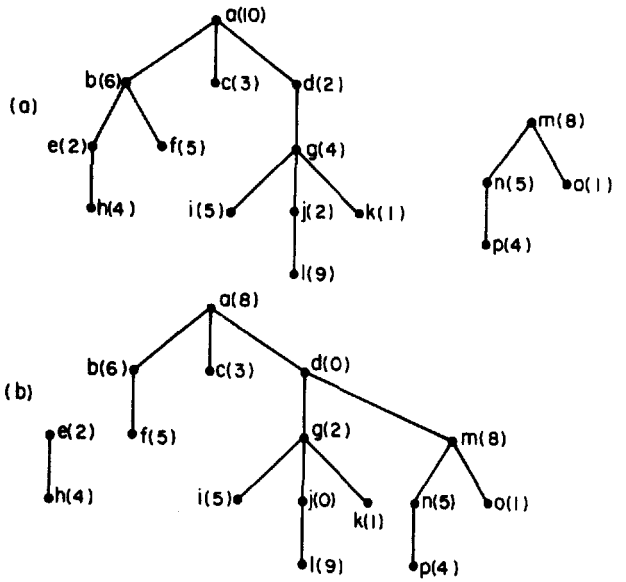



FIG. 5.1. *Tree operations.* (a) *Two trees. Numbers in parentheses are vertex costs. Operation* findroot $(j)$ *returns* $a$, findcost $(j)$ *returns* $[d, 2]$. (b) *After* addcost $(j, -2)$, link $(m, d)$, cut $(e)$.

**5.2. Representing trees as sets of paths.** There are no standard data structures that directly solve the problem of linking and cutting trees. However, there do exist methods for the special case in which each tree is a path. In this section we shall assume the existence of such a special-case method and use it to solve the general problem.

Suppose we have a way to carry out the following operations on vertex-disjoint paths, each of whose vertices has a real-valued cost:

makepath $(v)$: Create a new path containing the single vertex $v$, previously on no path, with cost zero.

findpath $(v)$: Return the path containing vertex $v$.

findtail $(p)$: Return the tail (last vertex) of path $p$.

findcost $(p)$: Return the pair $[w, x]$, where $x$ is the minimum cost of a vertex on path $p$ and $w$ is the last vertex on $p$ of cost $x$.

addpathcost $(p, x)$: Add real number $x$ to the cost of every vertex on path $p$.

join $(p, v, q)$: Combine path $p$, the one-vertex path containing only $v$, and path $q$ into a single path by adding an edge from the tail of $p$ to $v$ and an edge from $v$ to the head (first vertex) of $q$. Return the new path. This operation allows either $p$ or $q$ to be empty, in which case the corresponding edge is not added.

split $(v)$: Divide the path containing vertex $v$ into at most three paths by deleting the edges incident to $v$. Return the pair $[p, q]$, where $p$ is the part of the original path before but not including $v$, and $q$ is the part after but not including $v$. If $v$ is the head of the original path, $p$ is empty; if $v$ is the tail, $q$ is empty.

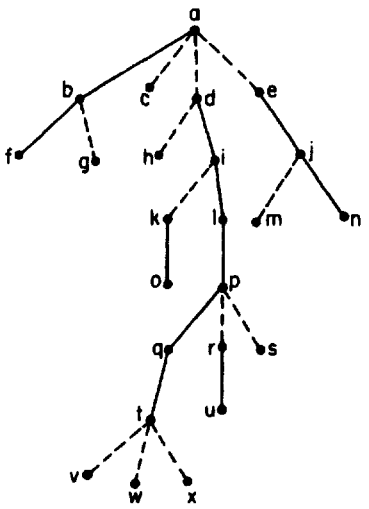



FIG. 5.2. *A tree partitioned into solid paths. Path* [*t, q, p, l, i, d*] *has head t and tail d.*

By using these path operations as primitives, we can solve the problem of linking and cutting trees. We partition each tree into a set of vertex-disjoint paths and carry out each tree operation by means of one or more path operations. To define the paths, we partition the edges of each tree into two kinds, *solid* and *dashed*, so that at most one solid edge enters any vertex. (See Fig. 5.2.) The solid edges partition the tree into vertex-disjoint *solid paths*.

To represent the dashed edges, we store with each solid path $p$ its *successor*, defined to be the vertex entered by the dashed edge leaving the tail of $p$; if the tail of

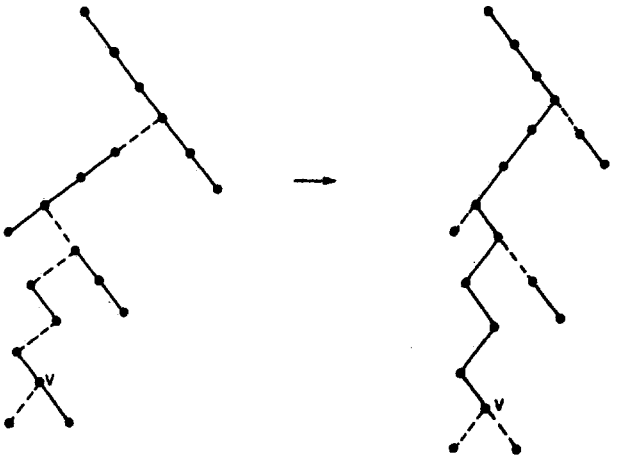



FIG. 5.3. *Exposing a vertex v.*

$p$ is a tree root, its successor is **null**. To carry out the tree operations we need the following composite operation on paths (see Fig. 5.3):

expose ($v$): Make the tree path from $v$ to findroot ($v$) solid by converting dashed edges along the path to solid and solid edges incident to the path to dashed. Return the resulting solid path.

Our implementation of the dynamic tree operations identifies each path by a vertex on it and uses **null** to identify the empty path. The following programs implement the six tree operations with the help of expose:

```
procedure maketree (vertex v);
   makepath (v);
   successor (v) := null
end maketree;

vertex function findroot (vertex v);
   return findtail (expose (v))
end findroot;

list function findcost (vertex v);
   return findpathcost (expose (v))
end findcost;

procedure addcost (vertex v, real x);
   addpathcost (expose (v), x)
end addcost;

procedure link (vertex v, w);
1. successor (join (null, expose (v), expose (w))) := null
end link;

procedure cut (vertex v);
   path p, q;
2. expose (v);
3. [p, q] := split (v);
   successor (v) := successor (q) := null
end cut;
```

*Notes.* The expose in line 1 of link makes $v$ into a one-vertex solid path. After line 2 in cut, vertex $v$ has no entering solid edge; thus the path $p$ returned by the split in line 3 is empty. Since in cut vertex $v$ is not a tree root, the path $q$ returned in line 3 is nonempty. □

The following program implements expose:

```
path function expose (vertex v);
   path p, q, r; vertex w;
   p := null;
   do v ≠ null →
      w := successor (findpath (v));
```

```
    1. [q, r] := split (v);
       if q ≠ null → successor (q) := v fi;
    2. p := join (p, v, r);
       v := w
    od;
    successor (p) := null
end expose;
```

*Note.* The solid edge from $v$ to the head of $r$ broken by the split in line 1 is restored by the join in line 2. Each iteration of the **do** loop converts to solid the edge from the tail of $p$ to $v$ (if $p \neq$ **null**) and to dashed the edge from the tail of $q$ to $v$ (if $q \neq$ **null**). □

We shall call each iteration of the **do** loop in expose a *splice*. Each tree operation takes $O(1)$ path operations and at most two expose operations. Each splice within an expose takes $O(1)$ path operations. At this level of detail, the only remaining task is to count the number of splices per expose. In the remainder of this section we shall derive an $O(m \log n)$ bound on the number of splices caused by a sequence of $m$ tree operations. This bound implies that there are $O(\log n)$ splices per expose amortized over the sequence.

To carry out the proof we need one new concept. We define the *size* of a vertex $v$ to be the number of descendants of $v$, including $v$ itself. We define an edge from $v$ to its parent $w$ to be *heavy* if $2 \cdot size\ (v) > size\ (w)$ and *light* otherwise. The following result is obvious.

LEMMA 5.1. *If $v$ is any vertex, there is at most one heavy edge entering $v$ and there are at most $\lfloor \lg n \rfloor$ light edges on the tree path from $v$ to* findroot $(v)$.

To bound the number of splices, we consider their effect on the number of tree edges that are both solid and heavy. There are no such edges (indeed, there are no edges at all) initially, and there are at most $n - 1$ such edges after all the tree operations. Consider exposing a vertex. During the expose, each splice except the first converts a dashed edge to solid. At most $\lfloor \lg n \rfloor$ of these splices convert a light dashed edge to solid. Each splice converting a heavy dashed edge to solid increases the number of heavy solid edges by one, since the edge it makes dashed, if any, is light. Thus $m$ tree operations cause a total of $m\ (\lfloor \lg n \rfloor + 1)$ splices plus at most one for each heavy solid edge created by the tree operations.

To bound the number of heavy solid edges created, we note that all but $n - 1$ of them must be destroyed. Each expose destroys at most $\lfloor \lg n \rfloor + 1$ heavy solid edges, at most one for each splice that does not increase the number of heavy solid edges. Links and cuts can also destroy heavy solid edges by changing node sizes.

An operation link $(v, w)$ increases the size of all nodes on the tree path from $w$ to findroot $(w)$, possibly converting edges on this path from light to heavy and edges incident to this path from heavy to light. After the operation expose $(w)$ in the implementation of link, the edges incident to the path are dashed, and adding the edge $[v, w]$ thus converts no solid edges from heavy to light.

An operation cut $(v)$ decreases the size of all nodes on the tree path from the parent of $v$ to findroot $(v)$, converting at most $\lfloor \lg n \rfloor$ heavy edges to light. The cut

also deletes a solid edge that may be heavy. Thus a cut destroys at most $\lfloor \lg n \rfloor + 1$ heavy solid edges, not including those destroyed by the expose that begins the cut.

Combining our estimates, we find that at most $(3m/2)(\lfloor \lg n \rfloor + 1)$ heavy solid edges are destroyed by $m$ tree operations, since there are at most $m/2$ cuts (an edge deleted by a cut must have been added by a previous link). Thus the total number of splices is at most $(5m/2)(\lfloor \lg n \rfloor + 1) + n - 1$, and we have the following theorem:

THEOREM 5.1. *A sequence of $m$ tree operations including $n$ maketree operations requires $O(m)$ path operations and in addition at most $m$ exposes. The exposes require $O(m \log n)$ splices, each of which takes $O(1)$ path operations.*

**5.3. Representing paths as binary trees,** To complete our solution to the problem of linking and cutting trees, we need a way to represent solid paths. We shall represent each solid path by a binary tree whose nodes in symmetric order are the vertices on the path. (See Fig. 5.4.) Each node $x$ contains three pointers: *parent* $(x)$, *left* $(x)$ and *right* $(x)$, to its parent, left child and right child, respectively. We use the root of the tree to identify the path; thus the root contains a pointer to the successor of the path. To distinguish between the trees defined by the link and cut operations and the trees representing solid paths, we shall call the latter *solid trees.*

To represent vertex costs, we use two real-valued fields per node. For any vertex $x$, let *cost* $(x)$ be the cost of $x$, and let *mincost* $(x)$ be the minimum cost of a descendant of $x$ in its solid tree. With $x$ we store $\Delta cost\ (x)$ and $\Delta min\ (x)$, defined as follows (see Fig. 5.4):

$$\Delta cost\ (x) = cost\ (x) - mincost\ (x),$$

$$\Delta min\ (x) = \begin{cases} mincost\ (x) & \text{if } x \text{ is a solid tree root,} \\ mincost\ (x) - mincost\ (parent\ (x)) & \text{if } x \text{ is not a solid tree root.} \end{cases}$$

Note that $\Delta cost\ (x) \geq 0$ for any vertex $x$ and $\Delta min\ (x) \geq 0$ for any nonroot vertex $x$. Given $\Delta cost$ and $\Delta min$, we can compute *mincost* $(x)$ for any vertex $x$ by

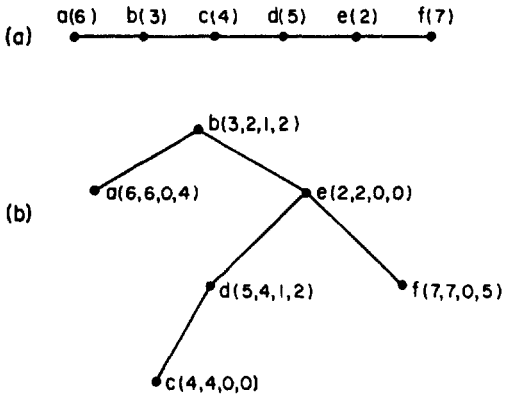



FIG. 5.4. *Representation of a path.* (a) *Path with head a and tail f.* (b) *Binary tree representing the path. Numbers labeling nodes are cost, mincost, Δcost, and Δmin, respectively.*

summing $\Delta min$ along the solid tree path from the root to $x$ and compute $cost\ (x)$ as $mincost\ (x) + \Delta cost\ (x)$.

With this representation, we can perform a single or double rotation in $O(1)$ time. We can also perform in $O(1)$ time two operations that assemble and disassemble solid trees:

assemble $(u, v, w)$: Given the roots $u$, $v$, $w$ of three solid trees, such that the tree with root $v$ contains only one node, combine the trees into a single tree with root $v$ and left and right subtrees the trees rooted at $u$ and $w$, respectively. Return $v$.

disassemble $(v)$: Given the root $v$ of a solid tree, break the tree into three parts, a tree containing only $v$ and the left and right subtrees of $v$. Return the pair consisting of the roots of the left and right subtrees.

*Note.* The effect of a rotation depends on whether or not the rotation takes place at the root of a solid tree. In particular, if we rotate at a root we must move the successor pointer for the corresponding solid path from the old root to the new root. □

Since assembly, disassembly and rotation are $O(1)$-time operations, we can use any of the kinds of binary search trees discussed in Chapter 4 to represent solid paths. We perform the six path operations as follows:

makepath $(v)$: Construct a binary tree of one node, $v$, with $\Delta min\ (v) = \Delta cost\ (v) = 0$.

findpath $(v)$: In the solid tree containing $v$, follow parent pointers from $v$ until reaching a node with no parent; return this node.

findtail $(p)$: Node $p$ is the root of a solid tree. Initialize vertex $v$ to be $p$ and repeatedly replace $v$ by $right\ (v)$ until $right\ (v) =$ **null**. Then return $v$.

findpathcost $(p)$: Initialize vertex $w$ to be $p$ and repeat the following step until $\Delta cost\ (w) = 0$ and either $right\ (w) =$ **null** or $\Delta min\ (right\ (w)) > 0$: If $right\ (w) \neq$ **null** and $\Delta min\ (right\ (w)) = 0$, replace $w$ by $right\ (w)$; otherwise if $\Delta cost\ (w) > 0$ replace $w$ by $left\ (w)$. (In the latter case $\Delta min\ (left\ (w)) = 0$.) Once the computation of $w$ is completed, return $[w, \Delta min\ (p)]$.

addpathcost $(p, x)$: Add $x$ to $\Delta min\ (p)$.

join $(p, v, q)$: Join the solid trees with roots $p$, $v$ and $q$ using any of the methods discussed in Chapter 4 for joining search trees.

split $(v)$: Split the solid tree containing node $v$ using any of the methods discussed in Chapter 4 for splitting search trees.

Both makepath and addpathcost take $O(1)$ time. The operations findpath, findtail and findpathcost are essentially the same as an access operation in a search tree (although findpath proceeds bottom-up instead of top-down), and each such operation takes time proportional to the depth of the bottommost accessed node. Both join and split are equivalent to the corresponding operations on search trees. If we use balanced binary trees to represent the solid paths, the time per path operation is $O(\log n)$, and by Theorem 5.1 a sequence of $m$ tree operations selected from maketree, findroot, findcost, addcost, link and cut takes $O(m\ (\log n)^2)$ time. The ideas behind this result are due to Galil and Naamad [2] and Shiloach [3].

Since the *successor* field is only needed for tree roots and the *parent* field is only needed for nonroots, we can save space in the implementation of the tree operations by storing parents and successors in a single field, if we store with each vertex a bit indicating whether it is a solid tree root. Another way to think of the data structure representing a tree $T$ is as a *virtual tree* $T'$: T' has the same vertex set as $T$ and has a parent function *parent'* defined as follows (see Fig. 5.5):

$$parent'(v) = \begin{cases} parent(v) & \text{if } v \text{ is not the root of a solid tree (we call } [v, parent'(v)] \text{ a } \textit{solid virtual edge}), \\ successor(v) & \text{if } v \text{ is a solid tree root (we call } [v, parent'(v)] \text{ a } \textit{dashed virtual edge}). \end{cases}$$

That is, $T'$ consists of the solid trees representing the solid paths of $T$ connected by edges corresponding to the dashed edges of $T$. To distinguish between a virtual tree $T'$ and the tree $T$ it represents, we shall call the latter an *actual tree*.

We can improve the $O(m(\log n)^2)$ time bound for $m$ tree operations by a factor of $\log n$ if we use self-adjusting trees rather than balanced trees to represent the solid paths. This also has the paradoxical effect of simplifying the implementation. Let

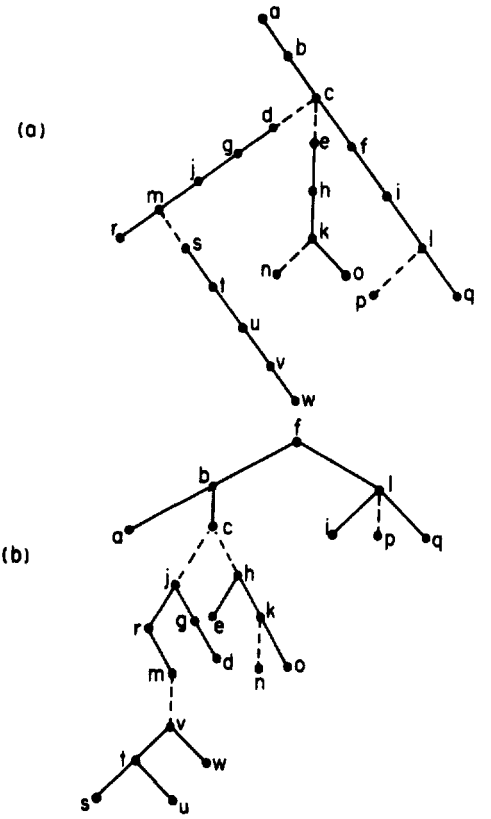



FIG. 5.5. *An actual tree and its virtual tree.* (a) *Actual tree.* (b) *Virtual tree.*

splay $(x)$ be the operation of splaying a solid tree at a node $x$. Recall that splay $(x)$ moves $x$ to the root of the tree by making single rotations. The following programs implement the path operations:

```
procedure makepath (vertex v);
  parent (v) := left (v) := right (v) := null;
  Δcost (v) := Δmin (v) := 0
end makepath;

vertex function findpath (vertex v);
  splay (v);
  return v
end findpath;

list function findpathcost (path p);
  do right(p) ≠ null and Δmin (right(p)) = 0 → p := right(p)
  |(right (p) = null or Δmin (right (p)) > 0) and Δcost (p) > 0 → p := left (p)
  od;
  splay (p);
  return [p, Δmin (p)]
end findpathcost;

vertex function findtail (path p):
  do right (p) ≠ null → p := right (p) od;
  splay (p);
  return p
end findtail;

procedure addpathcost (path p, real x);
  Δmin (p) := Δmin (p) + x
end addpathcost;

procedure join (path p, v, q);
  assemble (p, v, q)
end join;

list function split (vertex v);
  splay (v);
  return disassemble (v)
end split;
```

To analyze the running time of this implementation, we use the analysis of splaying given in §4.3. Recall that we have defined the size of a vertex $v$ in an actual tree $T$ to be the number of descendants of $v$, including $v$ itself. We define the *individual weight* of a vertex $v$ as follows:

$$iw(v) = \begin{cases} size(v) & \text{if } v \text{ has no entering solid edge,} \\ size(v) - size(u) & \text{if } [u, v] \text{ is a solid edge.} \end{cases}$$

We define the total weight $tw(v)$ of a vertex $v$ to be the sum of the individual weights of the descendants of $v$ in the solid tree containing $v$. If $v$ is any vertex in a tree, $tw(v)$ is the number of descendants of $v$ in the virtual tree containing it. Also $1 \leq tw(v) \leq n$ for all vertices $v$.

Let the rank of a vertex $v$ be $rank(v) = \lfloor \lg tw(v) \rfloor$. Suppose we use the accounting scheme of §4.3 to keep track of the time needed for operations on solid trees. Recall that we must keep $rank(v)$ credits on each vertex $v$ and that $3(rank(u) - rank(v)) + 1$ credits suffice to maintain this invariant and to pay for a splay at a vertex $v$ in a solid tree with root $u$. Since $0 \leq rank(v) \leq \lg n$ for any vertex $v$, this means that any solid path operation takes $O(\log n)$ credits.

The heart of the matter is the analysis of expose. We shall prove that a single expose takes $O(\log n)$ credits plus $O(1)$ credits per splice. Consider a typical splice. Let $v$ be the current vertex, $u$ the root of the solid tree containing $v$, and $w$ the parent of $u$ in the virtual tree. Referring to the program implementing expose, the operation findpath $(v)$ that begins the splice takes $3(rank(u) - rank(v)) + 1$ credits and makes $v$ the root of its solid tree. After this the operations split $(v)$ and join $(p, v, r)$ take $O(1)$ time and do not change $tw(v)$, since $tw(v)$ is the number of descendants of $v$ in the virtual tree, and the effect of the join and split on the virtual tree is merely to change from dashed to solid or vice versa at most two virtual edges entering $v$. (See Fig. 5.6.) Thus one additional credit pays for the split, the join and

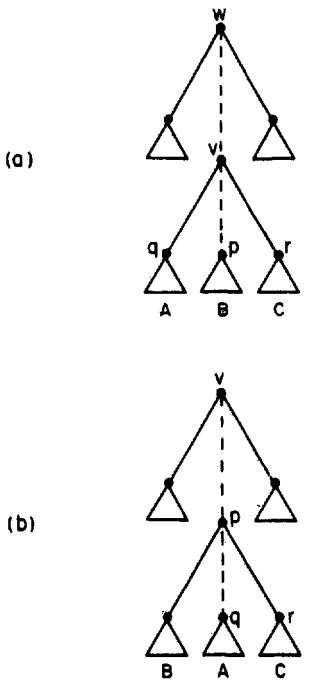



FIG. 5.6. *Effect of a splice on a virtual tree.* (a) *Relevant part of virtual tree after operation* findpath $(v)$. *Triangles denote solid subtrees. Vertex $v$ is now the root of its solid tree.* (b) *After completing the splice.*

the rest of the splice. No ancestor of $w$ in the virtual tree has its individual or total weight affected by the splice; thus no credits are needed to maintain the credit invariant on other solid trees. Vertex $w$, which is the next current vertex, has rank at least as great as that of $u$.

Summing over all splices during an expose, we find the total number of credits needed is 3 ($rank\ (u) - rank\ (v)$) plus two per splice, where $v$ is the vertex exposed and $u$ is the root of the virtual tree containing $v$. This bound is $O(\log n)$ plus two per splice. By Theorem 5.1 the total number of credits for all $m$ exposes caused by $m$ tree operations is $O(m \log n)$.

We must also account for the fact that link and cut change the total weights of vertices and thus affect the credit invariant. Consider an operation link $(v, w)$. This operation joins the empty path, the one-vertex path formed by exposing $v$, and the path formed by exposing $w$ into a single solid path. The only effect on total weight is to increase the total weight of $v$, requiring that we place $O(\log n)$ additional credits on $v$. Consider the operation cut $(v)$. After the operation expose $(v)$, the remaining effect of the cut is to delete the solid edge leaving $v$, which changes no individual weights and only decreases the total weights of some of the ancestors of $v$ in the actual tree originally containing it. This only releases credits.

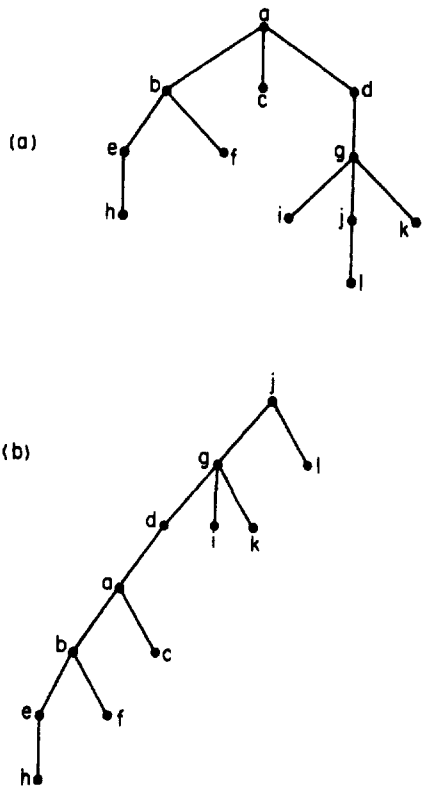



FIG. 5.7. *Everting a tree.* (a) *A rooted tree.* (b) *After the operation evert* ($j$).

Combining these estimates we have the following theorem:

THEOREM 5.2. *If we use self-adjusting binary trees to represent solid paths, a sequence of $m$ tree operations including $n$ maketree operations requires $O(m \log n)$* time.

**5.4. Remarks.** We can add other operations to our repertoire of tree operations without sacrificing the bound of $O(\log n)$ amortized time per operation. Perhaps the most interesting is evert ($v$), which turns the tree containing vertex $v$ "inside out" by making $v$ the root. (See Fig. 5.7.) When this operation is included the data structure is powerful enough to handle problems requiring linking and cutting of free (unrooted) trees; we root each tree at an arbitrary vertex and reroot as necessary using evert. We can associate costs with the edges of the trees rather than with the vertices. If worst-case rather than amortized running time is important, we can modify the data structure so that each tree operation takes $O(\log n)$ time in the worst case; the resulting structure uses biased search trees [1] in place of self-adjusting trees and is more complicated than the one presented here. Details of all these results may be found in Sleator and Tarjan [4].

## References

[1] S. W. BENT, D. D. SLEATOR AND R. E. TARJAN, *Biased search trees*, SIAM J. Comput., submitted.

[2] Z. GALIL AND A. NAAMAD, *An $O(E V \log^2 V)$ algorithm for the maximal flow problem*, J. Comput. System Sci., 21 (1980), pp. 203–217.

[3] Y. SHILOACH, *An $O(n \cdot I \log^2 I)$ maximum-flow algorithm*, Tech. Rep. STAN-CS-78-802, Computer Science Dept., Stanford Univ., Stanford, CA, 1978.

[4] D. D. SLEATOR AND R. E. TARJAN, *A data structure for dynamic trees*, J. Comput. System Sci., 24 (1983), to appear; also in Proc. 13th Annual ACM Symposium on Theory of Computing, 1981, pp. 114–122.

# CHAPTER 6

# Minimum Spanning Trees

**6.1. The greedy method.** In the last half of this book we shall use the data structures developed in the first half to solve four classical problems in network optimization. A *network* is a graph, either undirected or directed, each of whose edges has an associated real number. The object of a network optimization problem is to find a subgraph of a given network that has certain specified properties and that minimizes (or maximizes) some function of the edge numbers. In this chapter we shall study one of the simplest problems of network optimization, the *minimum spanning tree problem:* given a connected undirected graph each of whose edges has a real-valued *cost*, find a spanning tree of the graph whose total edge cost is minimum. We shall denote the cost of an edge $e = \{v, w\}$ by *cost* $(e)$ or, to avoid extra brackets, by *cost* $(v, w)$.

To solve this problem we use a simple incremental technique called the *greedy method:* we build up a minimum spanning tree edge by edge, including appropriate small edges and excluding appropriate large ones, until at last we have a minimum spanning tree. We shall formulate the greedy method in a way general enough to include all known efficient algorithms for the problem; this leaves us room to fill in the details of the method to make it as efficient as possible.

To make the greedy method concrete, we shall describe it as an edge-coloring process. Initially all edges of the graph are uncolored. We color one edge at a time either blue (accepted) or red (rejected). To color edges we use two rules that maintain the following *color invariant:*

> There is a minimum spanning tree containing all of the blue edges and none of the red ones.

The color invariant implies that when all the edges are colored, the blue ones form a minimum spanning tree.

To formulate the coloring rules we need the notion of a cut. A *cut* in a graph $G = [V, E]$ is a partition of the vertex set $V$ into two parts, $X$ and $\bar{X} = V - X$. An edge *crosses* the cut if it is incident to $X$ (one end is in $X$ and the other in $\bar{X}$). We shall sometimes regard a cut as consisting of the edges crossing it, but it is important to remember that technically a cut is a vertex partition.

The greedy method uses two coloring rules:

*Blue rule.* Select a cut that no blue edges cross. Among the uncolored edges crossing the cut, select one of minimum cost and color it blue.

*Red rule.* Select a simple cycle containing no red edges. Among the uncolored edges on the cycle, select one of maximum cost and color it red.

The method is nondeterministic: we are free to apply either rule at any time, in arbitrary order, until all edges are colored.

THEOREM 6.1. *The greedy method colors all the edges of any connected graph and maintains the color invariant.*

*Proof.* First we prove that the greedy method maintains the color invariant. Initially no edges are colored, and any minimum spanning tree satisfies the invariant; there is at least one minimum spanning tree since the graph is connected. Suppose the invariant is true before an application of the blue rule. Let $e$ be the edge colored blue by the rule and let $T$ be a minimum spanning tree satisfying the invariant before $e$ is colored. If $e$ is in $T$, then $T$ satisfies the invariant after $e$ is colored. If $e$ is not in $T$, consider the cut $X, \bar{X}$ to which the blue rule is applied to color $e$. There is a path in $T$ joining the ends of $e$, and at least one edge on this path, say $e'$, crosses the cut. By the invariant no edge of $T$ is red, and the blue rule implies that $e'$ is uncolored and $cost\ (e') \geq cost\ (e)$. Adding $e$ to $T$ and deleting $e'$ produces a minimum spanning tree $T'$ that satisfies the invariant after $e$ is colored.

The following similar argument shows that the red rule maintains the invariant. Let $e$ be an edge colored red by the rule and let $T$ be a minimum spanning tree satisfying the invariant before $e$ is colored. If $e$ is not in $T$, then $T$ satisfies the invariant after $e$ is colored. Suppose $e$ is in $T$. Deleting $e$ from $T$ divides $T$ into two trees that partition the vertices of $G$; $e$ has one end in each tree. The cycle to which the red rule was applied to color $e$ must have at least one other edge, say $e'$, with an end in each tree. Since $e'$ is not in $T$, the invariant and the red rule imply that $e'$ is uncolored and $cost\ (e') \leqq cost\ (e)$. Adding $e'$ to $T$ and deleting $e$ produces a new minimum spanning tree $T'$ that satisfies the invariant after $e$ is colored. We conclude that the greedy method maintains the color invariant.

Now suppose that the method stops early; that is, there is some uncolored edge $e$ but no rule applies. By the invariant, the blue edges form a forest, consisting of a set of trees that we shall call *blue trees*. If both ends of $e$ are in the same blue tree, the red rule applies to the cycle formed by $e$ and the path of blue edges joining the ends of $e$. If the ends of $e$ are in different blue trees, the blue rule applies to the cut one of whose parts is the vertex set of a blue tree containing one end of $e$. Thus an uncolored edge guarantees the applicability of some rule, and the greedy method colors all the edges. □

The greedy method applies to a wide variety of problems besides the minimum spanning tree problem. The proper general setting of the method is matroid theory [19]. In the special case of finding minimum spanning trees, especially efficient implementations are possible. In §6.2 we shall examine three well-known minimum spanning tree algorithms that are versions of the greedy method. In §6.3 we shall develop a relatively new algorithm that is the fastest yet known for sparse graphs.

**6.2. Three classical algorithms.** The minimum spanning tree problem seems to be the first network optimization problem ever studied; its history dates at least to 1926. Graham and Hell [14] have written an excellent historical survey of the problem. Borůvka [4] produced the first fully realized minimum spanning tree algorithm. The same algorithm was rediscovered by Choquet [7], Lukaszewicz et al. [20] and Sollin [1].

To understand Borůvka's algorithm, we need to study the behavior of the greedy

method. Recall our definition in the proof of Theorem 6.1 that a *blue tree* is a tree in the forest defined by the set of blue edges. Initially there are $n$ blue trees, each consisting of one vertex and no edges. Coloring an edge blue combines two blue trees into one. The greedy method combines blue trees two at a time until finally only one blue tree, a minimum spanning tree, remains.

Borůvka's algorithm begins with the initial set of $n$ blue trees and repeats the following step until there is only one blue tree (see Fig. 6.1).

COLORING STEP (Borůvka). For every blue tree $T$, select a minimum-cost edge incident to $T$. Color all selected edges blue.

Since every edge has two ends, an edge can be selected twice in the same execution of the coloring step, once for each end. Such an edge is only colored once. Borůvka's algorithm is guaranteed to work correctly only if all the edge costs are distinct, in which case the edges can be ordered so that the blue rule will color them one at a time. If the edge costs are nondistinct, we can break ties by assigning numbers to the edges (or vertices) and ordering edges lexicographically by cost and number (or cost and end numbers). Borůvka's algorithm, though old, is closely related to the new fast algorithm we shall study in §6.3. There are several ways to implement the algorithm, which we shall not discuss here except to note that the method is well suited for parallel computation.

The second and most recently discovered of the classical algorithms is that of

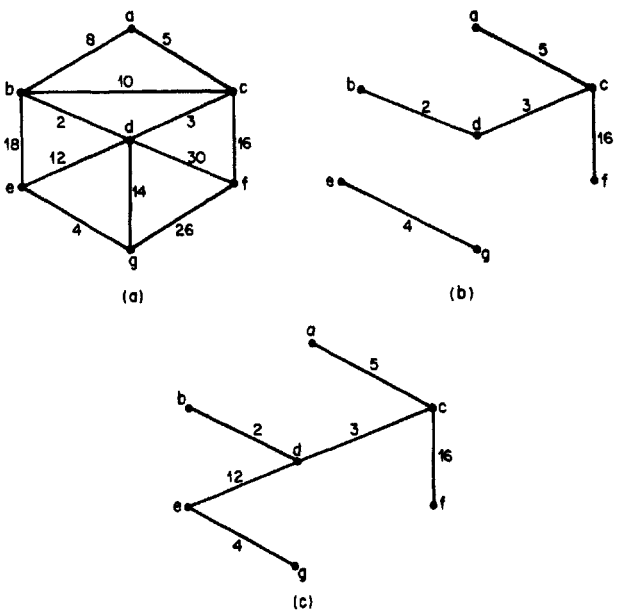



FIG. 6.1. *Execution of Borůvka's algorithm.* (a) *Input graph.* (b) *First step. Vertex a selects edge* $\{a, c\}$, *b and d select* $\{b, d\}$, *c selects* $\{c, d\}$, *e and g select* $\{e, g\}$ *and f selects* $\{c, f\}$. (c) *Second step. Both blue trees select* $\{d, e\}$.

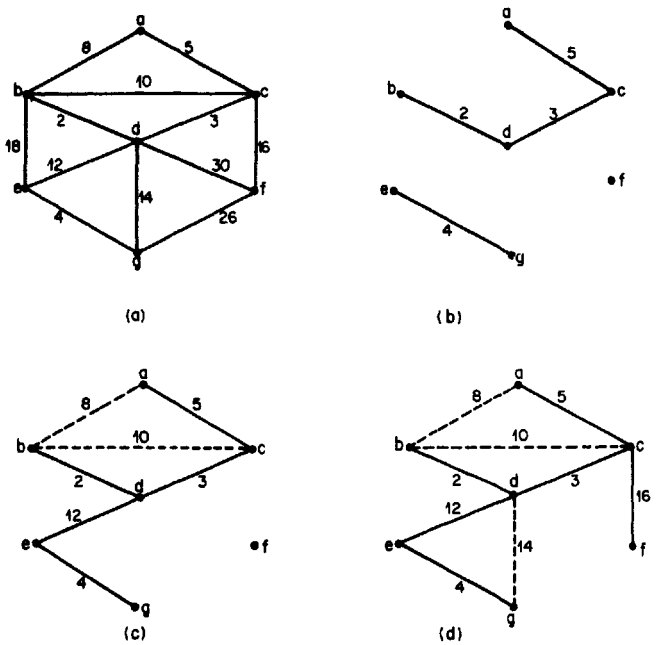



FIG. 6.2. *Execution of Kruskal's algorithm. Red edges are dashed.* (a) *Input graph.* (b) *After the first four steps. Edges* $\{b, d\}$, $\{c, d\}$, $\{e, g\}$, *and* $\{a, c\}$ *are colored blue.* (c) *After the next three steps. Edges* $\{a, b\}$ *and* $\{b, c\}$ *are colored red;* $\{d, e\}$ *is colored blue.* (d) *After two more steps. Edge* $\{d, g\}$ *is colored red and* $\{c, f\}$ *is colored blue.*

Kruskal [18]. Kruskal's algorithm consists of applying the following step to the edges in nondecreasing order by cost (see Fig. 6.2):

COLORING STEP (Kruskal). If the current edge $e$ has both ends in the same blue tree, color it red; otherwise color it blue.

To implement Kruskal's algorithm, we must solve two problems: ordering the edges by cost and representing the blue trees so that we can test whether two vertices are in the same blue tree. The simplest way to solve the former problem is to sort the edges by cost in a preprocessing step. The latter problem is a version of the disjoint set union problem discussed in Chapter 2, and we can use the operations of makeset, find and link to maintain the vertex sets of the blue trees.

The following program implements Kruskal's algorithm using these ideas. The program accepts as input sets of the vertices and edges in the graph and returns a set of the edges in a minimum spanning tree.

```
set function minspantree (set vertices, edges);
  set blue;
  blue := { };
  edges := sort edges by cost;
  for v ∈ vertices → makeset (v) rof;
```

```
    for {v, w} ∈ edges: find (v) ≠ find (w) →
        link (find (v), find (w)); blue := blue ∪ {{v, w}}
    rof;
    return blue
end minspantree;
```

The sorting step in the implementation requires $O(m \log n)$ time. Finding a minimum spanning tree given a sorted edge list requires $O(m, \alpha(m, n))$ time by the results of Chapter 2. Thus the total running time is $O(m \log n)$. This algorithm is best in situations where the edges are given in sorted order or can be sorted fast, for instance when the costs are small integers and radix sorting can be used. In such cases the running time is $O(m\alpha(m, n))$.

The third classical minimum spanning tree algorithm was discovered by Jarník [16] and rediscovered by Prim [22] and Dijkstra [9]; it is commonly known as Prim's algorithm. The algorithm uses an arbitrary starting vertex $s$ and consists of repeating the following step $n - 1$ times (see Fig. 6.3):

COLORING STEP (Prim). Let $T$ be the blue tree containing $s$. Select a minimum-cost edge incident to $T$ and color it blue.

Prim's algorithm is a kind of "shortest-first" search. (We shall study another kind of shortest-first search in Chapter 7.) Since the algorithm builds only one nontrivial

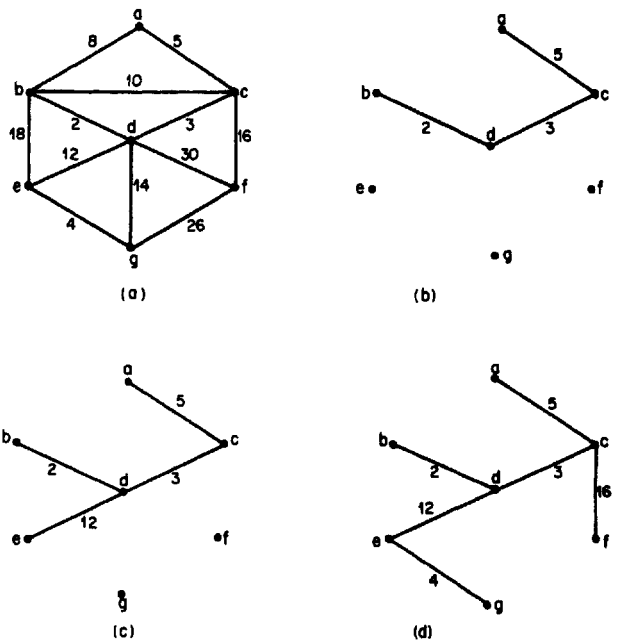



FIG. 6.3. *Execution of Prim's algorithm.* (a) *Input graph. Vertex a is the start vertex.* (b) *After three steps. Edges* $\{a, c\}$, $\{c, d\}$, *and* $\{d, b\}$ *become blue.* (c) *After four steps. Edge* $\{d, e\}$ *becomes blue.* (d) *After six steps. Edges* $\{e, g\}$ *and* $\{c, f\}$ *become blue.*

blue tree, we can implement the method without using a data structure for disjoint set union. We need only a single heap.

Although Jarník gave no detailed implementation, both Prim and Dijkstra did. Their idea is as follows: Let $T$ be the blue tree containing $s$. We say $v$ *borders* $T$ if $v$ is not in $T$ but some edge is incident to both $v$ and $T$. With each vertex $v$ bordering $T$ we associate a *light blue* edge $e$ that is a minimum-cost edge incident to $v$ and $T$. (Dijkstra [10] calls these edges "ultraviolet.") The light blue edges are candidates to become blue; the blue and light blue edges together form a tree spanning $T$ and its bordering vertices. To carry out a coloring step, we select a light blue edge of minimum cost and color it blue. This adds a new vertex, say $v$, to $T$. We complete the step by examining each edge $\{v, w\}$ incident to $v$. If $w$ is not in $T$ and has no incident light blue edge, we color $\{v, w\}$ light blue. If $w$ is not in $T$ but $w$ has an incident light blue edge, say $e$, of greater cost than $\{v, w\}$, we color $e$ red and color $\{v, w\}$ light blue. (See Fig. 6.4.)

Prim's algorithm implemented as described above has a running time of $O(n^2)$, $O(n)$ per coloring step. We can make the method faster on sparse graphs by maintaining a heap (see Chapter 3) of the vertices bordering $T$ and using the appropriate heap operations to carry out each coloring step. The key of a vertex in the heap is the cost of the incident light blue edge.

The program below implements this method. As input, the program needs a list of the vertices in the graph and the start vertex. The program assumes that, for each

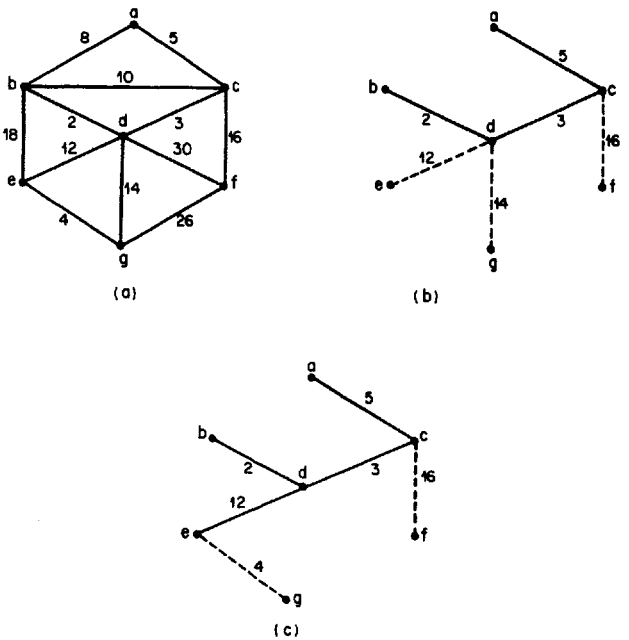



FIG. 6.4. *Efficient implementation of Prim's algorithm. Light blue edges are dashed.* (a) *Input graph. Vertex a is the start vertex.* (b) *After three steps.* (c) *After four steps. Edge* $\{d, e\}$ *becomes blue,* $\{e, g\}$ *becomes light blue, and* $\{d, g\}$ *and* $\{b, e\}$ *become red.*

vertex $v$, *edges* $(v)$ is the set of edges incident to $v$. For each vertex $v$ it maintains an incident edge *blue* $(v)$ and a real number *key* $(v)$ defined as follows: If $v$ is neither in $T$ nor borders $T$, *blue* $(v)$ is undefined and *key* $(v)$ is infinity. If $v$ borders $T$, *blue* $(v)$ is the light blue edge incident to $v$ and *key* $(v)$ is the cost of the edge. If $v$ is in $T$, *blue* $(v)$ is the blue edge whose coloring caused $v$ to be added to $T$ and key $(v)$ is minus infinity. When the program halts, the edges *blue* $(v)$ for $v \neq s$ form a minimum spanning tree.

The program is designed to use a $d$-heap (see §3.2) as proposed by Johnson [17]. In particular it uses the operation siftup $(v)$ to restore the heap after decreasing the key of vertex $v$ (by changing its incident light blue edge). The program must also be able to test vertices for heap membership. The *key* fields can be used for this purpose, since if the code below is written appropriately, a vertex $v$ is in the heap if and only if *key* $(v)$ is finite. The heap indices (see § 3.2) also provides this information.

```
procedure minspantree (set vertices, vertex s);
  vertex v; heap h;
  for v ⊂ vertices → key (v) := ∞ rof;
  h := makeheap ({ });
  v := s;
  do v ≠ null →
    key (v) := −∞;
    for {v, w} ⊂ edges (v): cost (v, w) < key (w) →
      key (w) := cost (v, w);
      blue (w) := {v, w};
      if w ∉ h → insert (w, h) | w ⊂ h → siftup (w, h⁻¹ (w), h) fi
    rof;
    v := deletemin (h)
  od
end minspantree;
```

*Remark.* It is possible to use a real-valued function to define *key*, thus saving the space of one field per vertex. □

The running time of this implementation is dominated by the heap operations, of which there are $n - 1$ deletemin operations, $n - 1$ insert operations, and at most $m - n + 1$ siftup operations. By the analysis of §3.2, the total running time is $O(n\,d \log_d n + m \log_d n)$. If we choose $d = \lceil 2 + m/n \rceil$, we obtain a time bound of $O(m \log_{(2+m/n)} n)$. If $m = \Omega(n^{1+\varepsilon})$ for some positive $\varepsilon$, the running time is $O(m/\varepsilon)$. Thus Prim's algorithm with Johnson's implementation is well suited for dense graphs, and the method is asymptotically worse than Kruskal's only if the edges are presorted.

**6.3. The round robin algorithm.** All three of the algorithms presented in §6.2 use mainly the blue rule. (It is also possible to find minimum spanning trees using mainly the red rule, but such methods seem to be less efficient.) The most general blue rule algorithm consists of beginning with the initial blue trees and repeating the following step $n - 1$ times:

COLORING STEP (round robin). Select a blue tree. Find a minimum-cost edge incident to this tree and color it blue.

Kruskal's algorithm builds blue trees in an irregular fashion dictated by the edge costs, Prim's algorithm builds only one nontrivial blue tree, and Borůvka's algorithm builds blue trees uniformly throughout the graph. By judiciously implementing a Borůvka-like algorithm using the appropriate data structures, we can obtain a method that is faster on sparse graphs than is any of the classical algorithms. Yao [29] was the first to propose such a method. His algorithm runs in $O(m \log \log n)$ time but needs a linear-time selection algorithm [2], [24] and is thus not very practical. We shall describe a similar but more practical $O(m \log \log n)$-time algorithm proposed by Cheriton and Tarjan [6].

To implement the general blue rule algorithm, we need two data structures for each blue tree. First, we need a way to represent the set of vertices in each tree. Second, we need a heap of the edges with at least one end in the tree that are candidates for becoming blue; the cost of an edge is its key in the heap. To represent the vertex sets, we use the data structure of Chapter 2, as in our implementation of Kruskal's algorithm. To represent the edge heaps, we use the leftist heaps of §3.3 with lazy melding and lazy deletion; we declare an edge "deleted" if its ends are in the same blue tree. We never explicitly mark edges deleted; instead, they are deleted implicitly when blue trees are combined.

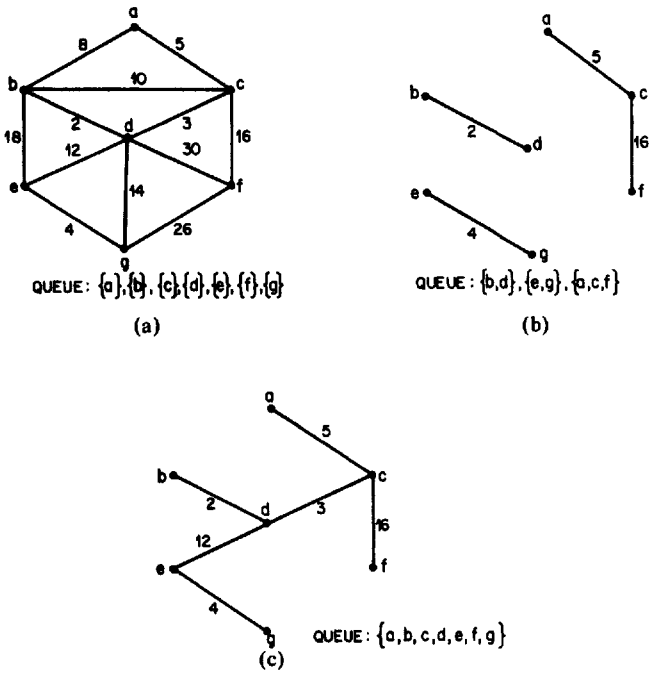



FIG. 6.5. *Execution of the round-robin algorithm.* (a) *Input graph. The queue contains the vertex sets of the blue trees.* (b) *After the initial pass through the queue. Edges* $\{a, c\}$, $\{b, d\}$, $\{e, g\}$, *and* $\{c, f\}$ *become blue.* (c) *After another pass. Edges* $\{c, d\}$ *and* $\{d, e\}$ *become blue.*

We also need a way to select blue trees for application of the blue rule. For this purpose we use a queue containing all the blue trees. To carry out a coloring step, we remove the first tree, say $T_1$, from the front of the queue and perform a findmin on its heap. We color blue the edge, say $e$, returned by the findmin. We remove from the queue the other tree, say $T_2$, incident to $e$. We update the vertex sets and edge heaps to reflect the combining of $T_1$ and $T_2$ and add the new blue tree to the rear of the queue. (See Fig. 6.5.) We call this method the *round robin algorithm*.

The following program implements this method. Input to the program is the set of vertices in the graph and, for each vertex $v$, the set *edges* $(v)$ of incident edges. To represent each blue tree, the program uses its canonical vertex as defined by the disjoint set union algorithm (see §2.1). The queue for selecting blue trees consists of a list of canonical vertices. For each canonical vertex $v$, $h(v)$ is the edge heap of the blue tree containing $v$. The predicate deleted $(e)$ returns **true** if the ends of $e$ are in the same blue tree. The program returns a set of the edges in a minimum spanning tree.

```
set function minspantree (set vertices);
   set blue;
   map h;
   list queue;
   vertex v, w;
   blue := { };
   queue := [ ]
   for v ∈ vertices → makeset (v); h(v) := makeheap (edges (v));
                      queue := queue & [v] rof;
   do |queue| > 1 →
      {v, w} := findmin (h(queue (1)));
      blue := blue ∪ {{v, w}};
      queue := queue − {find (v), find (w)};
      h(link (find (v), find (w))) := meld (h(find (v)), h(find (w)));
      queue := queue & [find (v)]
   od;
   return blue
end minspantree;

predicate deleted (edge {v, w});
   return find (v) = find (w)
end deleted;

real function key (edge e);
   return cost (e)
end key;
```

*Notes.* Every edge $\{v, w\}$ is initially in two heaps, $h(v)$ and $h(w)$. When such an edge is colored blue, the corresponding link operation automatically deletes both copies from the heap associated with the new blue tree. Since arbitrary elements must be deleted from the queue (by the assignment "*queue* := *queue* − {find $(v)$, find $(w)$}"), it is best to implement the queue as a doubly linked list or as an array of

vertices, each with an index giving its position in the array. (See §1.3.) Then the time for a deletion is $O(1)$. □

Let us analyze the running time of this algorithm. The running time is $O(m)$ not counting set and heap operations. The time for all the makeset and link operations is $O(n)$. If we assume that the time per find is $O(1)$, then the heap operations dominate the running time. We shall justify this assumption later.

To simplify the discussion we shall speak of the blue trees themselves rather than the canonical vertices as being on the queue. For $i = 1, 2, \cdots, n$ we define $T_i$ to be the blue tree selected during the $i$th coloring step, $m_i$ to be the number of edges in the heap associated with $T_i$ when $T_i$ is selected, and $k_i$ to be the number of edges purged from this heap during the $i$th findmin operation. We divide the execution of the algorithm into *passes* as follows. Pass zero consists of the selection and processing of blue trees initially on the queue. For $j > 0$, pass $j$ consists of the selection and processing of the trees added to but not deleted from the queue during pass $j - 1$.

LEMMA 6.1. *A blue tree on the queue during pass $j$ contains at least $2^j$ vertices. Thus after at most $\lfloor \lg n \rfloor$ passes there is a single blue tree, and the algorithm stops.*

*Proof.* Immediate by induction on $j$, since for $j > 0$ a blue tree on the queue during pass $j$ consists of a combination of two or more blue trees on the queue during pass $j - 1$. □

LEMMA 6.2.

$$\sum_{i=1}^{n-1} m_i \leq (2m + n - 1)\lfloor \lg n \rfloor .$$

*Proof.* Any two blue trees selected and processed during the same pass are vertex disjoint, since they are on the queue simultaneously. Thus the total size of all heaps processed during a single pass is at most $2m + n - 1$: each actual edge occurs in at most two heaps, and there are at most $n - 1$ dummy edges corresponding to lazy melds. (See §3.3.) The lemma follows from Lemma 6.1. □

Now we can bound the time for the heap operations. The time for all the makeheap operations is $O(m)$. The time for melds is $O(1)$ per meld, totalling $O(n)$ for all $n - 1$ melds. The time for the $i$th findmin is $O(k_i \max\{1, \log m_i/(k_i + 1)\})$. To estimate the total time for findmin operations, we divide them into two types: the $i$th findmin is *small* if $k_i \leq m_i/(\lg n)^2 - 1$ and *large* otherwise. The total time for small findmin operations is

$$O\left(\sum_{i=1}^{n-1} \frac{m_i}{(\log n)^2} \log m_i\right) = O\left(\sum_{i=1}^{n-1} \frac{m_i}{\log n}\right) = O(m)$$

by Lemma 6.2. The total time for large findmins is

$$O\left(\sum_{i=1}^{n-1} k_i \log \frac{m_i}{(m_i/(\log n)^2)}\right) = O\left(\sum_{i=1}^{n-1} k_i \log \log n\right) = O(m \log \log n),$$

since $\sum_{i=1}^{n-1} k_i \leq 2m + n - 1$, including the $n - 1$ dummy edges created by the melds.

THEOREM 6.2. *The round robin algorithm, implemented using leftist trees with lazy melding and lazy deletion, runs in $O(m \log \log n)$ time.*

*Proof.* The analysis above gives a time bound of $O(m \log \log n)$ with the finds counted at $O(1)$ time per find. This means that there are $O(m \log \log n)$ finds, which take $O((m \log \log n)\ \alpha(m \log \log n, n)) = O(m \log \log n)$ time (see §2.2). This gives the theorem. □

On sparse graphs, the round robin algorithm is asymptotically faster than any of the classical algorithms. On dense graphs it is slower by a factor of $O(\log \log n)$ than Prim's algorithm, although in practice the $O(m \log \log n)$ time bound is likely to be overly pessimistic. As a matter of theoretical interest, we can improve the running time of the round robin algorithm to $O(m \log \log_{(2+m/n)} n)$ time by "condensing" the graph at appropriate intervals, discarding from the edge heaps every edge with both ends in the same blue tree and all but a minimum cost edge between each pair of blue trees [6], [27]. The round robin algorithm with condensing is asymptotically as fast as any known minimum spanning tree algorithm, for any graph density.

**6.4. Remarks.** Not surprisingly, there are many results on special cases and variants of the minimum spanning tree problem. We conclude this chapter by mentioning some of the more interesting of such results. Other results may be found in the survey by Maffioli [21].

*An on-line algorithm.* Suppose we are presented with the edges of the graph one at a time in arbitrary order. We can build a minimum spanning tree on-line, as follows. We maintain a set of blue trees. To process an edge $e$, we color it blue. If this forms a cycle of blue edges, we discard a maximum-cost blue edge on the cycle. Using the data structures of Chapters 2 and 5 (the latter augmented to allow everting a tree), we can implement this algorithm to run on $O(m \log n)$ time.

*Alternative cost functions.* Minimum spanning trees are minimum with respect to any symmetric nondecreasing function of the edge costs [15].

*Verification, sensitivity analysis, and updating.* Three related problems are solvable in $O(m\alpha(m, n))$ time. Given a spanning tree, we can test whether it is minimum [26]. Given a minimum spanning tree, we can test by how much the cost of each edge can be increased or decreased without affecting the minimality of the tree [28]. Given a minimum spanning tree, we can find for each tree edge $e$ a minimum-cost substitute edge $e'$ such that, if $e$ is deleted from the graph, replacing it in the old minimum spanning tree by $e'$ produces a minimum spanning tree in the new graph [26].

*Linear-time special cases.* Let $S$ be a class of graphs closed under condensation of an edge and such that every graph in $S$ has $O(n)$ edges, where the constant depends on $S$ but not on the graph. Then there is an $O(n)$-time algorithm to find minimum spanning trees for graphs in $S$. As examples of this result, we can find a minimum spanning tree in a planar graph in $O(n)$ time [6] and we can in $O(n)$ time update a minimum spanning tree if a new vertex and incident edges are added to a graph whose minimum spanning tree is known [23].

*Degree constraints.* Suppose we wish to find a minimum spanning tree subject to a degree constraint at one or more vertices, i.e. at certain vertices $v$, the tree can have no more than $d(v)$ incident edges. With a degree constraint at only one vertex, the problem is linear-time reducible to the unconstrained problem [12]. With degree constraints at independent (pairwise nonadjacent) vertices, the problem is a special case of weighted matroid intersection [19] and is solvable in polynomial time. With degree constraints at all vertices the problem is NP-complete [13].

*Directed minimum spanning trees.* Given a directed graph with edge costs and a distinguished root $r$, there is an $O(\min\{m \log n, n^2\})$-time algorithm to find a minimum spanning tree rooted at $r$. The algorithm was discovered independently by Chu and Liu [8], Edmonds [11] and Bock [3]; Tarjan [25] gave an implementation with the claimed running time (see also [5]).

## References


[1] C. Berge and A. Ghouila-Houri, *Programming, Games, and Transportation Networks*, John Wiley, New York, 1965.

[2] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest and R. E. Tarjan, *Time bounds for selection*, J. Comput. System Sci., 7 (1973), pp. 448–461.

[3] F. Bock, *An algorithm to construct a minimum directed spanning tree in a directed network*, in Developments in Operations Research, Gordon and Breach, New York, 1971, pp. 29–44.

[4] O. Borůvka, *O jistém problému minimálním*, Práca Moravské Přírodovědecké Společnosti, 3 (1926), pp. 37–58. (In Czech.)

[5] P. M. Camerini, L. Fratta and F. Maffioli, *A note on finding optimum branchings*, Networks, 9 (1979), pp. 309–312.

[6] D. Cheriton and R. E. Tarjan, *Finding minimum spanning trees*, SIAM J. Comput., 5 (1976), pp. 724–742.

[7] G. Choquet, *Etude de certains réseaux de routes*, C. R. Acad. Sci. Paris, 206 (1938), pp. 310–313.

[8] Y. J. Chu and T. H. Liu, *On the shortest arborescence of a directed graph*, Sci. Sinica, 14 (1965), pp. 1396–1400.

[9] E. W. Dijkstra, *A note on two problems in connexion with graphs*, Numer. Math., 1 (1959), pp. 269–271.

[10] ———, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[11] J. Edmonds, *Optimum branchings*, J. Res. Nat. Bur. Standards Sect. B, 71 (1967), pp. 233–240.

[12] H. N. Gabow and R. E. Tarjan, *Efficient algorithms for a family of matroid intersection problems*, J. Algorithms, to appear.

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

[14] R. L. Graham and P. Hell, *On the history of the minimum spanning tree problem*, Ann. History of Computing, to appear.

[15] T. C. Hu, *The maximum capacity route problem*, Oper. Res., 9 (1961), pp. 898–900.

[16] V. Jarník, *O jistém problému minimálním*, Práca Moravské Přírodovědecké Společnosti, 6 (1930), pp. 57–63. (In Czech.)

[17] D. B. Johnson, *Priority queues with update and finding minimum spanning trees*, Inform. Process. Lett., 4 (1975), pp. 53–57.

[18] J. B. Kruskal, *On the shortest spanning subtree of a graph and the traveling salesman problem*, Proc. Amer. Math. Soc., 7 (1956), pp. 48–50.

[19] E. L. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York, 1976.

[20] L. Lukaszewicz, K. Florek, J. Perkal, H. Steinhaus and S. Zubrzycki, *Sur la liaison et la division des points d'un ensemble fini*, Colloq. Math., 2 (1951), pp. 282–285.

[21] F. MAFFIOLI, *Complexity of optimum undirected tree problems: A survey of recent results*, in Analysis and Design of Algorithms in Combinatorial Optimization, International Center for Mechanical Sciences, Courses and Lectures 266, Springer-Verlag, New York, 1981.

[22] R. C. PRIM, *Shortest connection networks and some generalizations*, Bell System Tech. J., 36 (1957), pp. 1389–1401.

[23] P. M. SPIRA AND A. PAN, *On finding and updating spanning trees and shortest paths*, SIAM J. Comput., 4 (1975), pp. 375–380.

[24] A. SCHÖNHAGE, M. PATERSON AND N. PIPPENGER, *Finding the median*, J. Comput. System Sci., 13 (1976), pp. 184–199.

[25] R. E. TARJAN, *Finding optimum branchings*, Networks, 7 (1977), pp. 25–35.

[26] ———, *Applications of path compression on balanced trees*, J. Assoc. Comput. Mach., 26 (1979), pp. 690–715.

[27] ———, *Minimum spanning trees*, Technical Memorandum, Bell Laboratories, Murray Hill, NJ, 1981.

[28] ———, *Sensitivity analysis of minimum spanning trees and shortest path trees*, Inform. Process. Lett., 14 (1982), pp. 30–33.

[29] A. YAO, *An $O|E|\log\log|V|)$ algorithm for finding minimum spanning trees*, Inform. Process. Lett., 4 (1975), pp. 21–23.

*This page intentionally left blank*

# CHAPTER 7

# Shortest Paths

**7.1. Shortest-path trees and labeling and scanning.** Another important network optimization problem is that of finding shortest paths. Let $G$ be a directed graph whose edges have real-valued (possibly negative) *lengths*. We shall denote the length of an edge $[v, w]$ by *length* $(v, w)$. The *length* of a *path* $p$, denoted by length $(p)$, is the sum of the lengths of the edges on $p$. A *shortest path* from a vertex $s$ to a vertex $t$ is a path from $s$ to $t$ whose length is minimum. The *shortest-path problem* is to find a shortest path from $s$ to $t$ for each member $[s, t]$ of a given collection of vertex pairs. The paper of Dreyfus [7] is a good survey of early work on this problem. We shall consider four versions of the problem:

*The single pair problem.* Find a shortest path from a given source $s$ to a given sink $t$.

*The single source problem.* Given a source $s$, find a shortest path from $s$ to $v$ for every vertex $v$.

*The single sink problem.* Given a sink $t$, find a shortest path from $v$ to $t$ for every vertex $v$.

*The all pairs problem.* For every pair of vertices $s$ and $t$, find a shortest path from $s$ to $t$.

The single source and single sink problems are directional duals of each other: reversing the edges of $G$ converts one to the other. All known methods of solving the single pair problem at least partially solve a single source or single sink problem. One way to solve the all pairs problem is to solve $n$ single source problems. For these reasons the single source problem is fundamental, and we shall concentrate on it, although in §7.3 we shall study the all pairs problem.

In order to devise algorithms for finding shortest paths, we need to know some of their properties. The following theorem characterizes the presence and form of shortest paths:

THEOREM 7.1. *Let $s$ and $t$ be vertices such that $t$ is reachable from $s$. There is a shortest path from $s$ to $t$ if and only if no path from $s$ to $t$ contains a cycle of negative length. (We call such a cycle a negative cycle.) If there is any shortest path from $s$ to $t$, there is one that is simple.*

*Proof.* If some path from $s$ to $t$ contains a negative cycle, we can produce an arbitrarily short path by repeating the cycle enough times. If no path from $s$ to $t$ contains a negative cycle, we can make any path from $s$ to $t$ simple without increasing its length by deleting cycles. □

In light of Theorem 7.1, we can regard the goal of a shortest-path problem as the exhibition of either appropriate shortest paths or a negative cycle. (An alternative in the presence of negative cycles is to ask for shortest *simple* paths, but finding such a path for a specified vertex pair is NP-complete [16].)

A compact way to represent the set of shortest paths for a single source $s$ is to use a *shortest-path tree.* This is a spanning tree rooted at $s$ each of whose paths is a shortest path in $G$. (See Fig. 7.1.)

THEOREM 7.2. *$G$ contains shortest paths from a source to all other vertices if and only if $G$ contains a shortest-path tree rooted at $s$.*

We shall prove Theorem 7.2 later, by giving an algorithm that finds either a shortest-path tree or a negative cycle. The heart of the algorithm is the following characterization of shortest-path trees. If $T$ is a spanning tree with root $s$ and $v$ is any vertex, we define *distance* $(v)$ to be the length of the path in $T$ from $s$ to $v$.

THEOREM 7.3. *$T$ is a shortest-path tree if and only if, for every edge $[v, w]$, distance $(v)$ + length $(v, w) \geq$ distance $(w)$.*

*Proof.* Let $T$ be a spanning tree with root $s$. If there is an edge $[v, w]$ such that *distance* $(v)$ + *length* $(v, w)$ < *distance* $(w)$, then the path in $T$ from $s$ to $w$ is not shortest. Conversely, suppose *distance* $(v)$ + *length* $(v, w) \geq$ *distance* $(w)$ for every edge $[v, w]$. Let $p$ be any path from $s$ to any vertex $v$. A proof by induction on the number of edges in $p$ shows that *length* $(p) \geq$ *distance* $(v)$. Hence $T$ is a shortest path tree. □

Theorem 7.3 allows us to test in $O(m)$ time whether a given spanning tree is a shortest-path tree: We compute *distance* $(v)$ for every vertex $v$ by processing the vertices in preorder, and then we test the distance inequality for each edge. The

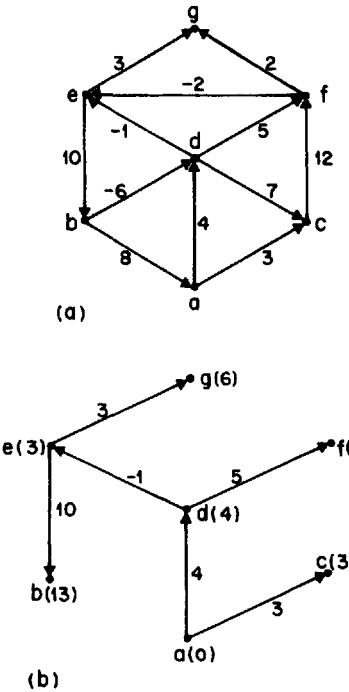



FIG. 7.1. *Shortest-path tree for a graph.* (a) *Graph.* (b) *Shortest-path tree rooted at vertex a. Distances from a are in parentheses.*

theorem also suggests a way to construct shortest-path trees by iterative improvement. For each vertex $v$, we maintain a tentative distance *dist* $(v)$ from $s$ to $v$ and a tentative parent $p(v)$ in the shortest-path tree. We initialize *dist* $(s) = 0$, *dist* $(v) = \infty$ for $v \neq s$, $p(v) =$ **null** for all $v$, and repeat the following step until *dist* $(v)$ + *length* $(v, w) \geq$ *dist* $(w)$ for every edge $[v, w]$:

LABELING STEP (Ford). Select an edge $[v, w]$ such that *dist* $(v)$ + *length* $(v, w) <$ *dist* $(w)$. Replace *dist* $(w)$ by *dist* $(v)$ + *length* $(v, w)$ and $p(w)$ by $v$.

Ford [12], [13], discovered this *labeling method*. Its correctness proof is surprisingly long but not difficult. We need two lemmas about tentative distances.

LEMMA 7.1. *The labeling method maintains the invariant that if dist* $(v)$ *is finite, there is a path from s to v of length dist* $(v)$.

*Proof.* By induction on the number of labeling steps. □

LEMMA 7.2. *If p is any path from s to any vertex v, then length* $(p) \geq$ *dist* $(v)$ *when Ford's method halts.*

*Proof.* By induction on the number of edges in $p$. □

THEOREM 7.4. *When the labeling method halts, dist* $(v)$ *is the length of a shortest path from s to v if v is reachable from s and dist* $(v) = \infty$ *otherwise. If there is a negative cycle reachable from s, the method never halts.*

*Proof.* Immediate from Theorem 7.1 and Lemmas 7.1 and 7.2. □

Proving that the method computes a shortest path tree takes three lemmas about tentative parents.

LEMMA 7.3. *The labeling method maintains the invariant that if* $p(v) \neq$ **null**, *dist* $(p(v))$ + *length* $(p(v), v) \leq$ *dist* $(v)$, *with equality when the method halts.*

*Proof.* By induction on the number of labeling steps. □

LEMMA 7.4. *The labeling method maintains the invariant that either the edges* $[p(v), v]$ *for v such that* $p(v) \neq$ **null** *form a tree rooted at s spanning the vertices v such that dist* $(v) < \infty$, *or there is a vertex v* such that $p^k(v) = v$ *for some k.*

*Proof.* A vertex $v \neq s$ has $p(v) \neq$ **null** if and only if *dist* $(v) < \infty$. If *dist* $(v) < \infty$ and $p(v) \neq$ **null** then *dist* $(p(v)) < \infty$. Thus if we start at any vertex $v$ such that *dist* $(v) < \infty$ and follow tentative parent pointers, we either reach $s$ and are unable to continue ($p(s) =$ **null**) or we repeat a vertex. This gives the lemma. □

LEMMA 7.5. *If at some time during the labeling method* $p^k(v) = v$ *for some vertex v and integer k, then the corresponding cycle in G is negative.*

*Proof.* Suppose applying the labeling step to edge $[x, y]$ creates a cycle of parent pointers. Consider the situation that just before the step. We have $p^k(x) = y$ for some $k$. Summing the inequalities given by Lemma 7.3 for $v = p^i(x)$, $i \in [0 \,..\, k-1]$ and the inequality *dist* $(x)$ + *length* $(x, y) <$ *dist* $(y)$, we have

$$dist\,(x) + \sum_{i=1}^{k} dist\,(p^i(x)) + length\,(x, y) + \sum_{i=0}^{k-1} length\,(p^{i+1}(x), p^i(x)) < dist\,(y) + \sum_{i=0}^{k-1} dist\,(p^i(x)).$$

Thus *length* $(x, y) + \sum_{i=0}^{k-1} length\,(p^{i+1}(x), p^i(x)) < 0$, which means that the cycle

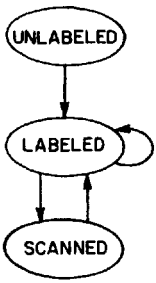



FIG. 7.2. *Possible vertex transitions in the labeling and scanning method.*

of parent pointers created by the labeling step corresponds to a negative cycle of $G$. □

THEOREM 7.5. *When the labeling method halts, the parent pointers define a shortest-path tree for the subgraph of G induced by the vertices reachable from s.*

*Proof.* Immediate from Theorem 7.4 and Lemmas 7.3, 7.4, and 7.5. □

We have not yet discussed termination. By Theorem 7.4, the labeling method fails to halt if $G$ contains a negative cycle reachable from $s$. If there is no such negative cycle, the method will halt, but it may require an exponential number of labeling steps. (An upper bound of $2^m - 1$ steps is not too hard to obtain [28].) Instead of verifying that the general algorithm halts, we shall study efficient refinements of it.

Our first refinement eliminates unnecessary edge examinations. We call this the *labeling and scanning method* [17]. The method maintains a partititon of the

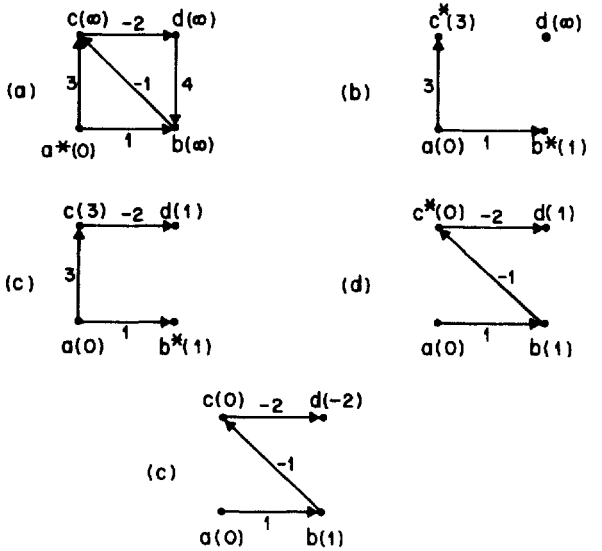



FIG. 7.3. *The labeling and scanning method. Labeled vertices are starred.* (a) *Input graph with initial distances.* (b) *After scanning vertex a.* (c) *After scanning vertices c and d.* (d) *After scanning vertex b.* (e) *After rescanning vertices c and d, the method stops.*

vertices into three states: *unlabeled, labeled* and *scanned*. The labeled and scanned vertices are exactly those with finite tentative distance. Figure 7.2 shows the transitions a vertex can undergo. Initially $s$ is labeled and every other vertex is unlabeled. The method consists of repeating the following step until there are no labeled vertices (see Fig. 7.3).

SCANNING STEP. Select a labeled vertex $v$ and *scan* it, thereby converting it to the scanned state, by applying the labeling step to each edge $[v, w]$ such that $dist\ (v) + length\ (v, w) < dist\ (w)$, thereby converting $w$ to the labeled state.

The labeling and scanning method, like the labeling method, is inefficient in general, but we can make it efficient by choosing a suitable scanning order. In the next section we shall study three scanning orders appropriate to three different situations.

**7.2. Efficient scanning orders.** Let us first consider the case of an acylic graph. If $G$ is acyclic, an appropriate scanning order is *topological*: We order the vertices reachable from $s$ so that if $[v, w]$ is an edge, $v$ appears before $w$ in the order (see §1.5), and we apply the scanning step once to each vertex in order. (See Fig. 7.4.) Because the scanning order is topological, once a vertex is scanned it never returns to the labeled state, and one scan of each reachable vertex suffices. The running time of the algorithm is $O(m)$, including the time for the topological ordering. We can also use topological scanning to compute longest paths in an acyclic graph: negating edge lengths converts longest to shortest paths and vice versa. Solving the longest path problem on acyclic graphs is an important part of "PERT" analysis [22].

One scan per vertex also suffices if $G$ has no edge of negative length. Dijkstra [6] proposed the appropriate scanning order, *shortest first*: among labeled vertices, always scan one whose tentative distance is minimum. (See Fig. 7.5.)

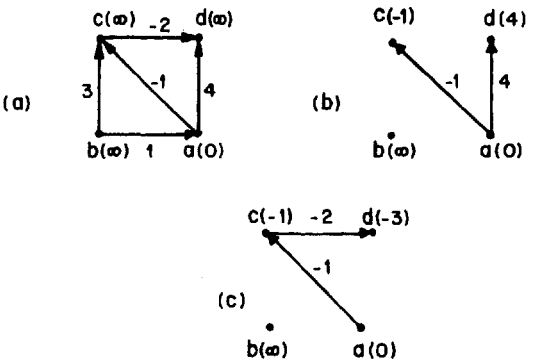



FIG. 7.4. *Topological scanning.* (a) *Input graph with initial distances. Vertex a is start vertex. Vertex b is not reachable from a. Scanning order is a, c, d.* (b) *After scanning vertex a.* (c) *After scanning vertices c and d.*

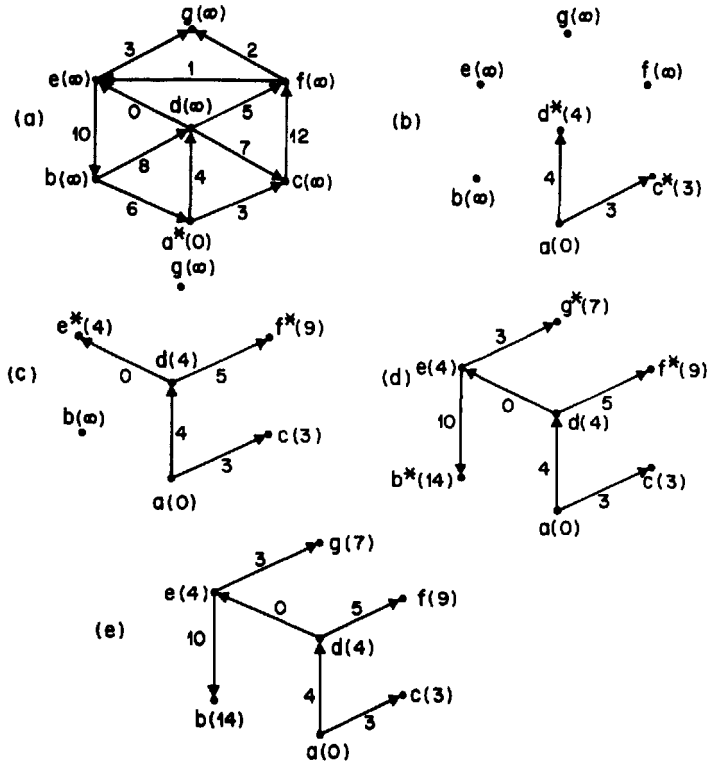



FIG. 7.5. *Dijkstra's algorithm. Labeled vertices are starred.* (a) *Input graph with initial distances.* (b) *After scanning vertex a.* (c) *After scanning vertices c and d.* (d) *After scanning vertex e.* (e) *After scanning vertices g, f and b, the method stops.*

THEOREM 7.6. *Suppose every edge has nonnegative length. If scanning is shortest first, then once a vertex is scanned, dist* $(v)$ *is the length of a shortest path from s to v.*

*Proof.* Consider scanning a vertex $v$. Just before $v$ is scanned, any labeled vertex $w$ satisfies $dist\ (w) \geq dist\ (v)$. An induction on the number of scanning steps using the nonnegativity of edge lengths shows that any vertex $w$ that becomes labeled during or after the scanning of $v$ also satisfies $dist\ (w) \geq dist\ (v)$. This means that vertices are scanned in nondecreasing order by distance from $s$ and that a vertex, once scanned, cannot become labeled. □

Dijkstra's shortest-path algorithm is almost identical to Prim's minimum spanning tree algorithm (see §6.2); indeed, Dijkstra discussed both algorithms in the same paper [6]. The program below, which implements Dijkstra's algorithm, differs only slightly from our implementation of Prim's algorithm. As suggested by Johnson [18], we store labeled vertices in a $d$-heap, using siftup to restore heap order after the key of a vertex is decreased. The major difference between the algorithms is in the definition of keys; to compute shortest paths we define the key of a vertex to

be its tentative distance. Input to the program is the vertex set, the source $s$ and, for each vertex $v$, the set $out\ (v)$ of its outgoing edges.

```
procedure shortestpathtree (set vertices, vertex s);
  vertex v; heap h;
  for v ∈ vertices → dist (v) := ∞; p(v) := null rof;
  dist (s) := 0;
  h := makeheap ({ });
  v := s;
  do v ≠ null →
    for [v, w] ∈ out (v):dist (v) + length (v, w) < dist (w) →
      dist (w) := dist (v) + length (v, w);
      p(w) := v;
      if w ∉ h → insert (w, h) | w ∈ h → siftup (w, h⁻¹(w), h) fi
    rof;
    v := deletemin (h)
  od
end shortestpathtree;

real function key (vertex v);
  return dist (v)
end key;
```

If we use a $d$-heap with $d = \lceil 2 + m/n \rceil$, the running time of this implementation is $O(m \log_{(2+m/n)} n)$; the analysis is the same as that at the end of §6.2. If instead we represent the heap by an unordered set, as Dijkstra did in his original formulation of the algorithm, the running time is $O(n^2)$ independent of the graph's density. If the edge lengths are small integers, we can use an array to represent the heap and obtain a running time of $O(m + d)$ and a space bound of $O(m + l)$ where $l$ is the length of the longest edge and $d$ is the maximum distance from the source to any vertex [4], [29].

Dijkstra's algorithm can be used in a bidirectional fashion to solve the single-pair shortest-path problem on graphs with nonnegative edge lengths [24], [26]. An appropriate variant of the algorithm solves the single source *bottleneck path problem:* for each vertex $v$, find a path from $s$ to $t$ that minimizes the length of the longest edge [8], [9]. For this problem the algorithm works even if some lengths are negative. Knuth [21] has generalized the algorithm to the computation of minimum-cost derivations in context-free languages under a rather general definition of cost.

The last case of the single source problem that we shall consider is the general case. In this case a good scanning order is *breadth-first:* Among labeled vertices, scan the one least recently labeled. The idea behind this method was discovered by Moore [23] and independently by Bellman [1]. To implement breadth-first scanning, we represent the set of labeled vertices as a queue [17]. Such an implementation raises the question of what to do with a labeled vertex that is relabeled before it is scanned. A strict interpretation of breadth-first scanning

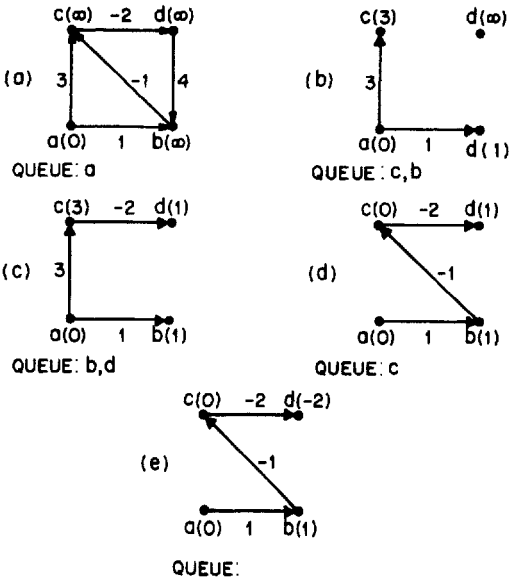



FIG. 7.6. *Breadth-first scanning.* (a) *Input graph with initial queue of labeled vertices.* (b) *After scanning vertex a.* (c) *After scanning vertex c.* (d) *After scanning vertices b and d.* (e) *After rescanning vertices c and d, the method stops.*

requires that we move such a vertex to the rear of the queue. We shall use a looser interpretation, leaving such a vertex in its current position on the queue. The following program implements this version of breadth-first scanning (see Fig. 7.6):

```
procedure shortestpathtree (set vertices, vertex s);
  vertex v; list queue;
  for v ∈ vertices → dist (v) := ∞; p(v) := null rof;
  dist (s) := 0;
  queue := [s];
  do queue ≠ [ ] →
    v := queue (1); queue := queue [2 . . ];
    for [v, w] ∈ out (v):dist (v) + length (v, w) < dist (w) →
      dist (w) := dist (v) + length (v, w);
      p(w) := v;
      if w ∉ queue → queue := queue & [w]fi
    rof
  od
end shortestpathtree;
```

*Note.* This program must be able to test vertices for queue membership. A membership bit for each vertex suffices for this purpose; testing or setting such a bit takes $O(1)$ time (see §1.3). □

To analyze the running time of this method, we divide its execution into *passes* in

a way similar to that used to analyze the round robin minimum spanning tree algorithm (see §6.3). Pass zero consists of the initial scanning of the source $s$. For $j > 0$, pass $j$ consists of the next scanning of all the vertices on the queue at the end of pass $j - 1$. Each pass requires $O(m)$ time, since during a pass each vertex is scanned at most once. The following theorem bounds the number of passes.

THEOREM 7.7. *If there is no negative cycle reachable from $s$, then breadth-first scanning runs in $O(nm)$ time, stopping by the end of pass $n - 1$. Otherwise the method never halts.*

*Proof.* We can prove by induction on $k$ that if there is a shortest path from $s$ to $v$ containing $k$ edges, then by the beginning of pass $k$ *dist* $(v)$ will be equal to the length of this path. The theorem then follows from Lemma 7.1 and Theorem 7.1. □

Theorem 7.7 implies Theorem 7.2, on the existence of shortest-path trees. To make breadth-first scanning robust, we must make sure it halts even in the presence of negative cycles. The easiest way to do this is to count passes. We add two variables to the program, an integer *pass*, to count passes, and a vertex *last*, indicating the last vertex to be scanned during the current pass. Initially *pass* = 0 and *last* = $s$. After vertex *last* is scanned, we add one to *pass* and replace *last* by the last vertex on the queue. If *pass* attains a value of $n$ with the queue nonempty, we terminate the algorithm and announce the presence of a negative cycle. With pass counting, breadth-first scanning runs in $O(nm)$ time whether or not there are negative cycles. Using the next lemma we can locate a negative cycle if one exists.

LEMMA 7.6. *If the queue is nonempty at the end of pass $n - 1$, then $p^k(v) = v$ for some vertex $v$ and integer $k$, and by Lemma 7.5 the corresponding cycle in $G$ is negative.*

*Proof.* Suppose we run the method until a vertex, say $w$, is scanned in pass $n$. Define the *pass* of a vertex $v$ to be the maximum $j$ such that $v$ is scanned during pass $j$. If *pass* $(v)$ is defined and positive, then $p(v)$ and *pass* $(p(v))$ are defined and *pass* $(p(v)) \geq$ *pass* $(v) - 1$. We have *pass* $(w) = n$. Following parent pointers from $w$ we must eventually repeat a vertex, since there are only $n$ vertices and the pass decreases by at most one with each step. □

Pass counting works best when the problem graph is unlikely to contain a negative cycle. If a negative cycle is likely, it may be better to stop the algorithm as soon as the parent pointers define a cycle. There are two ways to do this: when processing an edge $[v, w]$ such that *dist* $(v)$ + *length* $(v, w) <$ *dist* $(w)$, we can either look for $w$ among the ancestors of $v$ in the tentative shortest-path tree, or we can look for $v$ among the descendants of $w$ in the tree. The former method is easy to implement using parent pointers but increases the worst-case running time to $O(n^2m)$ [28]. The latter method requires storing extra information about the tree (a list of the vertices in preorder will do) but if carefully implemented preserves the $O(nm)$ running time [28].

Some experimental evidence [5] suggests that a hybrid scanning order [25] combining breadth-first and depth-first scanning (scan the most recently labeled vertex) performs better than breadth-first scanning in practice, although the worst case running time of the hybrid method is exponential [20]. The best choice of

method in practice depends on the graph structure and the nature of the edge lengths and is generally not clear cut.

**7.3. All pairs.** We conclude this chapter with two algorithms for the all pairs problem. The first, suited for sparse graphs, combines breadth-first scanning and Dijkstra's algorithm. The second, suited for dense graphs, uses dynamic programming.

If the problem graph $G$ has nonnegative edge lengths, we can solve the all pairs problem in $O(n\,m\log_{(2+m/n)}n)$ time by using $n$ iterations of Dijkstra's single source algorithm, one for each possible source. Even if there are negative-length edges, we can obtain the same time bound by making all the edge lengths nonnegative in a preprocessing step. Edmonds and Karp [9] defined the appropriate transformation. First we add to $G$ a new vertex $s$ and a zero-length edge $[s, v]$ for every vertex $v$ in $G$. Then for each vertex $v$ we compute the length *distance* $(v)$ of a shortest path from $s$ to $v$ in the augmented graph. Using breadth-first scanning this takes $O(nm)$ time. Finally, we define a new length for each edge $[v, w]$ by $length'(v, w) = length\,(v, w) + distance\,(v) - distance\,(w)$.

THEOREM 7.8. *For any edge* $[v, w]$, $length'\,(v, w) \geq 0$. *For every path* $p$ *from a vertex* $s$ *to a vertex* $t$, $length'\,(p) = length\,(p) + distance\,(s) - distance\,(t)$.

*Proof.* Theorem 7.3 implies the first part of the theorem. The second part is immediate by induction on the number of edges in $p$. □

By Theorem 7.8, the edge length transformation makes all edge lengths nonnegative and preserves shortest paths, since it transforms the lengths of all paths from a given vertex $s$ to a given vertex $t$ by the same amount. Thus we can find shortest paths for all pairs with respect to the new lengths using $n$ iterations of Dijkstra's algorithm, and we can apply the inverse transformation to find shortest distances with respect to the original lengths. With this method the time to solve the all pairs problem is $O(n\,m\log_{(2+m/n)}n)$, including pre- and postprocessing. On moderately dense graphs ($m = \Omega(n^{1+\varepsilon})$), the time is $O(nm/\varepsilon)$.

Another way to solve the all pairs problem is to use dynamic programming. Floyd [11] obtained such a method by generalizing the transitive closure algorithm of Warshall [30] (see also [10]). Dantzig [3] proposed a variant of Floyd's method. Both methods run in $O(n^3)$ time.

Floyd's algorithm maintains a tentative distance $dist\,(v, w)$ for every pair of vertices $v$ and $w$. The algorithm processes the vertices one at a time, maintaining the invariant that $dist\,(v, w)$ is the length of a shortest path from $v$ to $w$ that contains only processed vertices as intermediate vertices. Initially $dist\,(v, w)$ equals $length\,(v, w)$ if $[v, w]$ is an edge, zero if $v = w$, infinity otherwise. The algorithm consists of repeating the following step for each vertex $u$:

LABELING STEP (Floyd). If $dist\,(u, u) < 0$, abort: there is a negative cycle. Otherwise, for each pair $v, w$ such that $dist\,(v, w) > dist\,(v, u) + dist\,(u, w)$, replace $dist\,(v, w)$ by $dist\,(v, u) + dist\,(u, w)$.

As stated, Floyd's algorithm computes only shortest distances, but we can easily augment it to compute shortest paths. For each pair of vertices $v, w$, we maintain a

tentative parent $p(v, w)$ of $w$ in a shortest-path tree rooted at $v$. Initially $p(v, w)$ is $v$ if $[v, w]$ is an edge, **null** otherwise. When replacing *dist* $(v, w)$ by *dist* $(v, u)$ + *dist* $(u, w)$, we also replace $p(v, w)$ by $p(u, w)$.

THEOREM 7.9. *Floyd's algorithm, augmented as described, computes shortest distances for all pairs and shortest-path trees for all sources. If the algorithm aborts because dist* $(u, u) < 0$ *for some vertex u, there is a negative cycle* $v_1, v_2, \cdots, v_k$ *given by* $v_1 = v_k = u, v_{i-1} = p(u, v_i)$ *for* $i \in [2 \,..\, k]$.

*Proof.* Suppose that during the algorithm we maintain a path *path* $(v, w)$ for each pair of vertices such that *dist* $(v, w)$ is finite, defined by *path* $(v, w) = [v, w]$ if $[v, w]$ is an edge, *path* $(v, w) = [v]$ if $w = v$, *path* $(v, w)$ is undefined otherwise. When replacing *dist* $(v, w)$ by *dist* $(v, u)$ + *dist* $(u, w)$, we also replace *path* $(v, w)$ by $(path(v, u) - \{u\})$ & $path(u, w)$. The algorithm maintains the following invariants:

(i) *path* $(v, w)$ has length *dist* $(v, w)$;

(ii) *path* $(v, w)$ is a shortest path from $v$ to $w$ containing only processed vertices as intermediate vertices;

(iii) *path* $(v, w)$ is a simple path if $v \neq w$, a simple cycle if $v = w$;

(iv) *path* $(v, w) = path\,(v, p(v, w))$ & $[w]$.

We can verify the invariants by induction on the number of labeling steps, using the (crucial) fact that processing a vertex $u$ affects *dist* $(v, w)$ only if $u \notin \{v, w\}$. The theorem follows. □

Floyd's algorithm is so simple that on dense graphs it is likely to be faster by a constant factor than $n$ iterations of Dijkstra's algorithm, but it has disadvantages not suffered by the latter method: it needs $\Omega(n^2)$ storage, and it does not become appreciably faster as the graph becomes sparser. Floyd's algorithm is a special case of Jordan elimination [2]. The related method of Gaussian elimination [2], [14], which takes greater advantage of sparsity, can also be used to find shortest paths. Gaussian and Jordan elimination have many other applications, including solving systems of linear equations [2], [14], converting a finite automaton into a regular expression [19], and doing global flow analysis of computer programs [27]. All these problems can be treated together in an appropriate general setting [2], [27].

Fredman [15] discovered yet another algorithm for the all pairs problem. Fredman's algorithm runs in $O(n^3(\log\log n/\log n)^{1/3})$ time, beating $O(n^3)$ methods on dense graphs, but it seems to be too complicated to be practical.

## References


[1] R. E. BELLMAN, *On a routing problem*, Quart. Appl. Math., 16 (1958), pp. 87–90.

[2] B. CARRÉ, *Graphs and Networks*, Clarendon Press, Oxford, 1979.

[3] G. B. DANTZIG, *All shortest routes in a graph*, in Theory of Graphs, International Symposium, Gordon and Breach, New York, 1967, pp. 91–92.

[4] R. B. DIAL, *Algorithm* 360: *Shortest path forest with topological ordering*, Comm. ACM, 12 (1969), pp. 632–633.

[5] R. DIAL, F. GLOVER, D. KARNEY AND D. KLINGMAN, *A computational analysis of alternative algorithms for finding shortest path trees*, Networks, 9 (1979), pp. 215–248.

[6] E. W. DIJKSTRA, *A note on two problems in connexion with graphs*, Numer. Math., 1 (1959), pp. 269–271.

[7] S. E. DREYFUS, *An appraisal of some shortest-path algorithms*, Oper. Res., 17 (1969), pp. 395–412.

[8] J. EDMONDS AND D. R. FULKERSON, *Bottleneck extrema*, Memorandum RM-5375-PR, RAND Corp., Santa Monica, CA, 1968.

[9] J. EDMONDS AND R. M. KARP, *Theoretical improvements in algorithmic efficiency for network flow problems*, J. Assoc. Comput. Mach., 19 (1972), pp. 248–264.

[10] R. W. FLOYD, *Algorithm 96: Ancestor*, Comm. ACM, 5 (1962), pp. 344–345.

[11] ———, *Algorithm 97: Shortest path*, Comm. ACM, 5 (1962), pp. 345.

[12] L. R. FORD, JR., *Network flow theory*, Paper P-923, RAND Corp., Santa Monica, CA, 1956.

[13] L. R. FORD, JR. AND D. R. FULKERSON, *Flows in Networks*, Princeton Univ. Press, Princeton, NJ, 1962.

[14] G. E. FORSYTHE AND C. B. MOLER, *Computer Solution of Linear Algebraic Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1967.

[15] M. L. FREDMAN, *New bounds on the complexity of the shortest path problem*, SIAM J. Comput., 5 (1976), pp. 83–89.

[16] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, 1979.

[17] J. GILSINN AND C. WITZGALL, *A performance comparison of labeling algorithms for calculating shortest path trees*, National Bureau of Standards Technical Note 772, U.S. Dept. of Commerce, Washington, DC, 1973.

[18] D. B. JOHNSON, *Efficient algorithms for shortest paths in sparse networks*, J. Assoc. Comput. Mach., 24 (1977), pp. 1–13.

[19] S. C. KLEENE, *Representation of events in nerve nets and finite automata*, in Automata Studies, C. E. Shannon and J. McCarthy, eds., Princeton Univ. Press, Princeton, NJ, 1956, pp. 3–40.

[20] A. KERSHENBAUM, *A note on finding shortest path trees*, Networks, 11 (1981), pp. 399–400.

[21] D. E. KNUTH, *A generalization of Dijkstra's algorithm*, Inform. Process. Lett., 6 (1977), pp. 1–5.

[22] E. L. LAWLER, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York, 1976.

[23] E. F. MOORE, *The shortest path through a maze*, in Proc. International Symposium on the Theory of Switching, Part II, April 2–5, 1957, The Annals of the Computation Laboratory of Harvard University, 30, Harvard Univ. Press, Cambridge, MA, 1959.

[24] T. A. J. NICHOLSON, *Finding the shortest route between two points in a network*, Computer J., 9 (1966), pp. 275–280.

[25] U. PAPE, *Implementation and efficiency of Moore-algorithms for the shortest route problem*, Math. Programming, 7 (1974), pp. 212–222.

[26] I. POHL, *Bi-directional search*, in Machine Intelligence, Vol. 6, B. Meltzer and D. Michie, eds., Edinburgh Univ. Press, Edinburgh, 1971, pp. 124–140.

[27] R. E. TARJAN, *A unified approach to path problems*, J. Assoc. Comput. Mach., 28 (1981), pp. 577–593.

[28] ———, *Shortest paths*, Technical Memorandum, Bell Laboratories, Murray Hill, NJ, 1981.

[29] R. A. WAGNER, *A shortest path algorithm for edge-sparse graphs*, J. Assoc. Comput. Mach., 23 (1976), pp. 50–57.

[30] S. WARSHALL, *A theorem on Boolean matrices*, J. Assoc. Comput. Mach., 9 (1962), 11–12.

# CHAPTER 8

# Network Flows

**8.1. Flows, cuts and augmenting paths.** A group of network optimization problems with widespread and diverse applications are the *network flow problems*. Let $G = [V, E]$ be a directed graph with two distinguished vertices, a *source* $s$ and a *sink* $t$, and a positive capacity $cap\ (v, w)$ on every edge $[v, w]$. For convenience we define $cap\ (v, w) = 0$ if $[v, w]$ is not an edge. A *flow* on $G$ is a real-valued function $f$ on vertex pairs having the following three properties:

(i) *Skew symmetry.* $f(v, w) = -f(w, v)$. If $f(v, w) > 0$, we say there is a flow from $v$ to $w$.

(ii) *Capacity constraint.* $f(v, w) \leqq cap\ (v, w)$. If $[v, w]$ is an edge such that $f(v, w) = cap\ (v, w)$, we say the flow *saturates* $[v, w]$.

(iii) *Flow conservation.* For every vertex $v$ other than $s$ and $t$, $\sum_w f(v, w) = 0$.

The *value* $|f|$ of a flow $f$ is the net flow out of the source, $\sum_v f(s, v)$. The *maximum flow problem* is that of finding a flow of maximum value, called a *maximum flow*. This problem has a rich and elegant theory and many applications both in operations research and in combinatorics [9], [16], [19]. A series of faster and faster algorithms have been devised for the problem. (See Fig. 8.1.) The theory of network flows is an outgrowth of linear programming [16], [18] originally developed by Ford and Fulkerson, who wrote a classic book on the subject [9]. We shall begin our study by reviewing their basic results.

As in the minimum spanning tree problem, a key concept is that of a *cut*. In considering network flows, we define a cut $X, \bar{X}$ to be a partition of the vertex set $V$ into two parts $X$ and $\bar{X} = V - X$ such that $X$ contains $s$ and $\bar{X}$ contains $t$. The *capacity* of a cut $X, \bar{X}$ is $cap\ (X, \bar{X}) = \sum_{v \in X,\, w \in \bar{X}} cap\ (v, w)$. A cut of minimum capacity is a *minimum cut*. If $f$ is a flow and $X, \bar{X}$ is a cut, the *flow across the cut* is $f(X, \bar{X}) = \sum_{v \in X, w \in \bar{X}} f(v, w)$.

LEMMA 8.1. *For any flow $f$, the flow across any cut $X, \bar{X}$ is equal to the flow value.*

*Proof.*

$$f(X, \bar{X}) = \sum_{\substack{v \in X \\ w \in \bar{X}}} f(v, w) = \sum_{\substack{v \in X \\ w}} f(v, w) - \sum_{\substack{v \in X \\ w \in X}} f(v, w) = |f| - 0 = |f|,$$

since $\sum_{v \in X, w} f(v, w) = |f|$ by flow conservation and $\sum_{v \in X, w \in X} f(v, w) = 0$ by skew symmetry. □

By the capacity constraint, the flow across any cut cannot exceed the capacity of the cut. Thus the value of a maximum flow is no greater than the capacity of a minimum cut. The *max-flow min-cut theorem* states that these two numbers are equal. To prove this theorem, we need several concepts. The *residual capacity* for a

| DATE | DISCOVERER | RUNNING TIME |
|---|---|---|
| 1956 | FORD AND FULKERSON | — |
| 1969 | EDMONDS AND KARP | $O(nm^2)$ |
| 1970 | DINIC | $O(n^2 m)$ |
| 1974 | KARZANOV | $O(n^3)$ |
| 1978 | MALHOTRA, ET. AL. | $O(n^3)$ |
| 1977 | CHERKASKY | $O(n^2 m^{1/2})$ |
| 1978 | GALIL | $O(n^{5/3} m^{2/3})$ |
| 1979 | GALIL AND NAAMAD; SHILOACH | $O(nm(\log n)^2)$ |
| 1980 | SLEATOR AND TARJAN | $O(nm \log n)$ |

FIG. 8.1. *History of maximum flow algorithms.*

flow $f$ is the function on vertex pairs given by $res\ (v, w) = cap\ (v, w) - f(v, w)$. We can push up to $res\ (v, w)$ additional units of flow from $v$ to $w$ by increasing $f(v, w)$ and correspondingly decreasing $f(w, v)$. The *residual graph* $R$ for a flow $f$ is the graph with vertex set $V$, source $s$, sink $t$, and an edge $[v, w]$ of capacity $res\ (v, w)$ for every pair $v, w$ such that $res\ (v, w) > 0$. (See Fig. 8.2.) An *augmenting path* for $f$ is a path $p$ from $s$ to $t$ in $R$. The *residual capacity* of $p$, denoted by $res\ (p)$, is the minimum value of $res\ (v, w)$ for $[v, w]$ an edge of $p$. We can increase the value of $f$

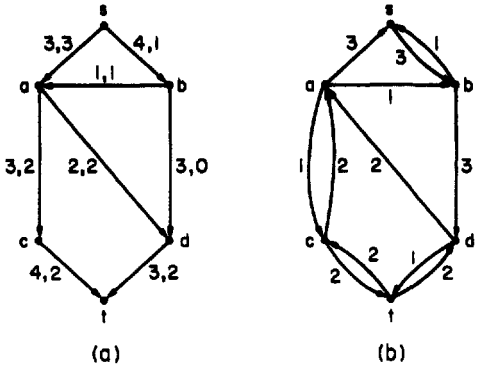



FIG. 8.2. *Residual graph for a flow.* (a) *Graph with flow. First number on an edge is its capacity, second is its flow. Edges* $[s, a]$ *and* $[a, d]$ *are saturated.* (b) *Residual graph. Path* $[s, b, d, a, c, t]$ *is augmenting, of residual capacity* 1.

by any amount $\Delta$ up to *res* $(p)$ by increasing the flow on every edge ot $p$ by $\Delta$. The following lemma gives a more quantitative version of this observation.

*Note.* Whenever we change $f(v, w)$ we must change $f(w, v)$ by a corresponding amount to maintain skew symmetry; we shall generally omit explicit mention of this. □

LEMMA 8.2. *Let $f$ be any flow and $f^*$ a maximum flow on $G$. If $R$ is the residual graph for $f$, then the value of a maximum flow on $R$ is $|f^*| - |f|$.*

*Proof.* For any flow $f'$ on $R$, define $f + f'$ by $(f + f')(v, w) = f(v, w) + f'(v, w)$. Then $f + f'$ is a flow on $G$ of value $|f| + |f'|$, which implies $|f'| \leq |f^*| - |f|$. Similarly the function $f^* - f$ defined by $(f^* - f)(v, w) = f^*(v, w) - f(v, w)$ is a flow on $R$ of value $|f^*| - |f|$ and is thus a maximum flow on $R$. □

THEOREM 8.1 (max-flow min-cut theorem) [5], [8]. *The following conditions are equivalent:*

(i) *$f$ is a maximum flow;*

(ii) *there is no augmenting path for $f$;*

(iii) *$|f| = cap\,(X, \bar{X})$ for some cut $X, \bar{X}$.*

*Proof.* (i) *implies* (ii). If there is an augmenting path $p$ for $f$ then we can increase the flow by increasing the flow along $p$.

(ii) *implies* (iii). Suppose there is no augmenting path for $f$. Let $X$ be the set of vertices reachable from $s$ in the residual graph $R$ for $G$ and let $\bar{X} = V - X$. Then $X$, $\bar{X}$ is a cut, and

$$|f| = \sum_{\substack{v \in X \\ w \in \bar{X}}} f(v, w) = \sum_{\substack{v \in X \\ w \in \bar{X}}} cap\,(v, w) = cap\,(X, \bar{X}),$$

since $v \in X$, $w \in \bar{X}$ implies $[v, w]$ is not an edge of $R$, i.e., $f(v, w) = \text{cap}\,(v, w)$.

(iii) *implies* (i). Since $|f| \leq cap\,(X, \bar{X})$ for any flow $f$ and any cut $X, \bar{X}$, $|f| = cap\,(X, \bar{X})$ implies $f$ is a maximum flow and $X, \bar{X}$ is a minimum cut. □

Theorem 8.1 gives a way to construct a maximum flow by iterative improvement, the *augmenting path method* of Ford and Fulkerson: Begin with a flow of zero on all edges (which we call the *zero flow*), and repeat the following step until obtaining a flow without an augmenting path:

AUGMENTING STEP (Ford and Fulkerson). Find an augmenting path $p$ for the current flow. Increase the value of the flow by pushing *res* $(p)$ units of flow along $p$.

Knowing a maximum flow, we can compute a minimum cut as described in the proof of Theorem 8.1, in $O(m)$ time.

Suppose the edge capacities are integers. Then the augmenting path method increases the flow value by at least one with each augmentation, and thus computes a maximum flow $f^*$ in at most $|f^*|$ augmenting steps. Furthermore, $f^*(v, w)$ is an integer for every $v, w$ (we call a flow with this property *integral*). Hence we have the following theorem:

THEOREM 8.2 (integrality theorem). *If all capacities are integers, there is an integral maximum flow.*

Unfortunately, if the capacities are large integers the value of a maximum flow may be large, and the augmenting path method may make many augmentations. (See Fig. 8.3.) Furthermore, if the capacities are irrational the method may not halt, and although successive flow values converge they need not converge to the value of a maximum flow [9]. Thus if the method is to be efficient we must select augmenting paths carefully. The next lemma suggests that this may be possible.

LEMMA 8.3. *Starting from the zero flow, there is a way to construct a maximum flow in at most m steps, each of which increases the flow along a single path in the original graph (by an amount that is not necessarily maximum; see Fig. 8.4).*

*Proof.* Let $f^*$ be a maximum flow. Let $G^*$ be the subgraph of $G$ induced by the edges $[v, w]$ such that $f^*(v, w) > 0$. Initialize $i$ to one and repeat the following step until $t$ is not reachable from $s$ in $G^*$:

PATHFINDING STEP. Find a path $p_i$ from $s$ to $t$ in $G^*$. Let $\Delta_i$ be the minimum of $f^*(v, w)$ for $[v, w]$ an edge of $p_i$. For every edge $[v, w]$ on $p_i$, decrease $f^*(v, w)$ by $\Delta_i$ and delete $[v, w]$ from $G^*$ if its flow is now zero. Increase $i$ by one.

Each pathfinding step deletes at least one edge from $G^*$; thus this algorithm halts after at most $m$ steps, having reduced $f^*$ to a flow of value zero. (There may still be cycles of flow.) Beginning with the zero flow and successively pushing $\Delta_1$ units of flow along $p_1$, $\Delta_2$ units along $p_2$, · · · produces a maximum flow in at most $m$ steps. □

A natural way to select augmenting paths is to always augment along a path of maximum residual capacity, as suggested by Edmonds and Karp [4]. Lemmas 8.2 and 8.3 allow us to analyze this method, which we call *maximum capacity augmentation.*

THEOREM 8.3. *Maximum capacity augmentation produces successive flow values that converge to the value of a maximum flow* [20]. *If the capacities are integers the method finds a maximum flow in O(m* log *c) augmenting steps where c is the maximum edge capacity.*

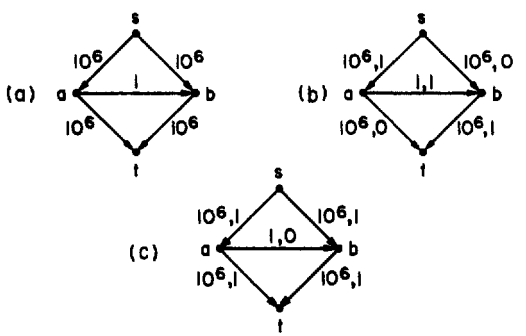



FIG. 8.3. *A bad graph for the augmenting path method.* (a) *Input graph.* (b) *After augmentation along* $[s, a, b, t]$. (c) *After augmentation along* $[s, b, a, t]$. *After* $2 \times 10^6$ *augmentations, alternately along* $[s, a, b, t]$ *and* $[s, b, a, t]$, *the flow is maximum.*

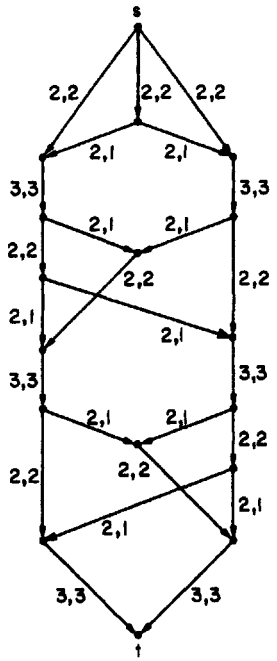



FIG. 8.4. *A maximum flow of value six. Although all capacities are at least two, no single path carries two units of flow.*

*Proof.* Let $f$ be any flow, and $f^*$, a maximum flow. By Lemma 8.2 there is a flow $f'$ on the residual graph $R$ of $f$ with value $|f^*| - |f|$. By the proof of Lemma 8.3, there are at most $m$ augmenting paths whose residual capacities sum to at least $|f'|$; thus the maximum capacity augmenting path has residual capacity at least $(|f^*| - |f|)/m$.

Consider a sequence of $2m$ consecutive maximum-capacity augmentations, starting with flow $f$. At least one of these must augment the flow by an amount $(|f^*| - |f|)/(2m)$ or less. Thus after $2m$ or fewer augmentations, the capacity of a maximum-capacity augmenting path is reduced by a factor of two. Since this capacity is initially at most $c$ and is at least one unless the flow is maximum, after $O(m \log c)$ maximum-capacity augmentations the flow must be maximum. □

Finding a maximum capacity augmenting path is a version of the bottleneck path problem mentioned in §7.2. (We take lengths equal to the negatives of the capacities.) We can find such a path using Dijkstra's algorithm suitably modified. This method takes $O(m \log_{(2+m/n)} n)$ time to find an augmenting path, and the total time to find a maximum flow is $O(m^2 (\log_{(2+m/n)} n)(\log c))$ if the capacities are integers. This bound is polynomial in $n$, $m$, and the number of bits needed to represent the capacities, but is still not entirely satisfactory; we would like a bound which is polynomial in just $n$ and $m$.

We obtain an algorithm with such a bound by using another way of selecting augmenting paths, also suggested by Edmonds and Karp [4]: Always choose a

shortest augmenting path, where we measure the length of a path by the number of edges it contains. This method is most efficient if we augment along paths of the same length simultaneously, as suggested by Dinic [3], who obtained his result independently. In the next two sections we shall develop this method.

**8.2. Augmenting by blocking flows.** In order to understand Dinic's algorithm we need two new concepts. A flow $f$ is a *blocking flow* if every path from $s$ to $t$ contains a saturated edge. The value of a blocking flow cannot be increased by pushing additional flow along any path in $G$, although it may be possible to increase the flow value by rerouting, i.e. decreasing the flow on some edges and increasing it on others. Let $R$ be the residual graph for a flow $f$. The *level* of a vertex $v$ is the length of the shortest path from $s$ to $v$ in $R$. The *level graph* $L$ for $f$ is the subgraph of $R$ containing only the vertices reachable from $s$ and only the edges $[v, w]$ such that $level(w) = level(v) + 1$. $L$ contains every shortest augmenting path and is constructible in $O(m)$ time by breadth-first search. (See §1.5.)

Dinic's algorithm consists of beginning with the zero flow and repeating the following step until $t$ is not in the level graph for the current flow (see Fig. 8.5):

BLOCKING STEP (Dinic). Find a blocking flow $f'$ on the level graph for the current flow $f$. Replace $f$ by the flow $f + f'$ defined by $(f + f')(v, w) = f(v, w) + f'(v, w)$.

THEOREM 8.4. *Dinic's algorithm halts after at most* $n - 1$ *blocking steps.*

*Proof.* Consider a blocking step. Let $f$ be the current flow, $R$ its residual graph, $L$ its level graph, $R'$ the residual graph after the step, and *level* and *level'* the level functions for $R$ and $R'$, respectively. Each edge $[v, w]$ in $R$ has $level(w) \leq level(v) + 1$. Each edge in $R'$ is either an edge in $R$ or is the reverse of an edge in $L$. Thus each edge $[v, w]$ in $R'$ has $level(w) \leq level(v) + 1$. This means $level'(t) \geq level(t)$. Suppose $level'(t) = level(t)$. Let $p$ be any shortest path from $s$ to $t$ in $R$. Then $level(w) = level(v) + 1$ for every edge $[v, w]$ on $p$, which means every such edge is in $L$. This contradicts the fact that at least one such edge is saturated by the blocking flow found on $L$ and does not appear in $R'$. Hence $level'(t) > level(t)$.

Since the level of $t$ is at least one, at most $n - 1$, and increases by at least one or becomes undefined with each blocking step, the number of steps is at most $n - 1$. □

On certain kinds of networks, Dinic's algorithm is even more efficient than indicated by Theorem 8.4. A *unit network* is a network in which all edge capacities are integers and each vertex $v$ other than $s$ and $t$ has either a single entering edge, of capacity one, or a single outgoing edge, of capacity one.

THEOREM 8.5 [7]. *On a unit network, Dinic's algorithm halts after at most* $2\lceil\sqrt{n-2}\rceil$ *blocking steps.*

*Proof.* Consider a blocking step. Let $f$ be the current flow, $f^*$ a maximum flow, and $R$ the residual graph for $f$. Then $f^* - f$ is a flow on $R$. Since $f$ is integral, $R$ is a unit network, and $f^* - f$ is zero or one on every edge. We can partition the edges on which $f^* - f$ is one into a collection of $|f^*| - |f|$ paths from $s$ to $t$ and possibly

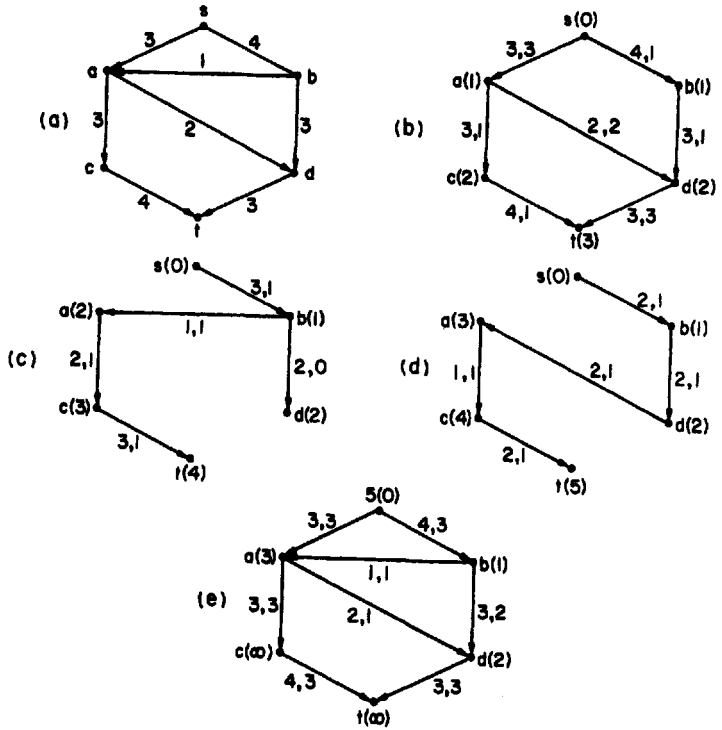



FIG. 8.5. *Dinic's maximum flow algorithm applied to the graph in Fig.* 8.2. (a) *Input graph.* (b) *First level graph with blocking flow. Levels of vertices are in parentheses.* (c) *Second level graph with blocking flow.* (d) *Third level graph with blocking flow.* (e) *Final flow. Minimum cut is* $\{s, a, b, d\}$, $\{c, t\}$.

some cycles. Since $R$ is unit, any vertex other than $s$ and $t$ is on at most one of the paths, which implies that $f$ has an augmenting path of length at most $(n-2)/(|f^*| - |f|) + 1$.

After $\lceil \sqrt{n-2} \rceil$ blocking steps, the shortest augmenting path contains at least $\sqrt{n-2} + 1$ edges by Theorem 8.4. Thus $\sqrt{n-2} + 1 \leq (n-2)/(|f^*| - |f|) + 1$, i.e. $|f^*| - |f| \leq \sqrt{n-2}$. After at most $\lfloor \sqrt{n-2} \rfloor$ additional blocking steps, the current flow is maximum. □

A similar argument shows that if all edge capacities are one, Dinic's algorithm halts after $O(\min\{n^{2/3}, m^{1/2}\})$ blocking steps [7].

Dinic's algorithm takes $O(km)$ time plus the time to find blocking flows on $k$ acyclic graphs, where $k$ is the number of blocking steps. In the next section we shall discuss various ways to find blocking flows. We conclude this section with Edmonds and Karp's analysis of Ford and Fulkerson's augmenting path method when augmentation is along shortest paths.

THEOREM 8.6. *If augmentation is along shortest paths, the augmenting path method halts after* $(n-1)m$ *augmenting steps and runs in* $O(nm^2)$ *time.*

*Proof.* A proof like that of Theorem 8.4 shows that the distance from $s$ to $t$ in the residual graph never decreases, and that after at most $m$ augmentations along shortest paths of the same length, the distance from $s$ to $t$ increases by at least one. Thus there are at most $(n-1)m$ augmenting steps. Finding a shortest augmenting path takes $O(m)$ time by breadth-first search, giving a total time bound of $O(nm^2)$. □

**8.3. Finding blocking flows.** Suppose $G$ is an acyclic network on which we wish to find a blocking flow. We shall describe four methods for finding such a flow, each of which gives a corresponding maximum flow algorithm. The simplest way to find a blocking flow is the method used by Dinic: Find a path from $s$ to $t$, push enough flow along it to saturate an edge, delete all newly saturated edges, and repeat until $t$ is not reachable from $s$. To find each path we use depth-first search. The following steps define this method more formally. We begin with the zero flow, go to *initialize*, and proceed as directed; $p$ is a path along which flow can be pushed from $s$ to the current vertex $v$.

*Initialize.* Let $p = [s]$ and $v = s$. Go to *advance*.

*Advance.* If there is no edge out of $v$, go to *retreat*. Otherwise, let $[v, w]$ be an edge out of $v$. Replace $p$ by $p \,\&\, [w]$ and $v$ by $w$. If $w \neq t$ repeat *advance;* if $w = t$ go to *augment*.

*Augment.* Let $\Delta$ be the minimum of $cap(v, w) - f(v, w)$ for $[v, w]$ an edge of $p$. Add $\Delta$ to the flow of every edge on $p$, delete from $G$ all newly saturated edges, and go to *initialize*.

*Retreat.* If $v = s$ halt. Otherwise, let $[u, v]$ be the last edge on $p$. Delete $v$ from $p$ and $[u, v]$ from $G$, replace $v$ by $u$, and go to *advance*.

THEOREM 8.7. *Dinic's algorithm correctly finds a blocking flow in $O(nm)$ time and a maximum flow in $O(n^2m)$ time.*

*Proof.* The algorithm deletes an edge $[v, w]$ from $G$ only if $[v, w]$ is saturated or every path from $w$ to $t$ contains a saturated edge. It follows that the algorithm constructs a blocking flow. Each *initialize*, *advance*, or *retreat* step takes $O(1)$ time; each *augment* takes $O(n)$ time. Since each augment or retreat deletes an edge, there are at most $m$ such steps. There are at most $m + 1$ initialize steps since each but the first follows an *advance*. At most $n - 1$ advance steps precede an *augment* or *retreat;* thus there are at most $(n-1)m$ advance steps. Combining these estimates, we see that the algorithm finds a blocking flow in $O(nm)$ time. The $O(n^2m)$ time bound for a maximum flow follows from Theorem 8.4. □

THEOREM 8.8 [7]. *On a unit network, Dinic's algorithm finds a blocking flow in $O(m)$ time and a maximum flow in $O(\sqrt{n}\, m)$ time.*

*Proof.* An *augment* step takes $O(k)$ time, where $k$ is the length of the augmenting path $p$. On a unit network, such a step saturates and deletes at least $(k-1)/2$ edges. Thus the total time for *augment* steps is $O(m)$. An edge added to $p$ by an *advance* is either deleted from $p$ by a *retreat* or has its flow increased by an *augment;* thus there are $O(m)$ *advance* steps. Combining estimates, we get a total

time of $O(m)$ to find a blocking flow. Theorem 8.5 gives the $O(\sqrt{n}\, m)$ time bound for a maximum flow. □

On a network all of whose edge capacities are one, Dinic's algorithm finds a maximum flow in $O(\min\{n^{2/3}m, m^{3/2}\})$ time [7].

On general networks Dinic's blocking flow method saturates only one edge at a time in the worst case, spending $O(n)$ time per edge saturated. On dense graphs there are faster methods that, in effect, saturate one vertex at a time and have an $O(n^2)$ running time. Karzanov [14] discovered the first such method. His original algorithm, as well as more recent variants [2], [10], [22], are rather complicated, but there is a simpler version [25] that we shall describe here. We call this the *wave method*. To present the method we need the concept of a preflow.

A *preflow* $f$ is a skew symmetric function on vertex pairs that satisfies the capacity constraint and has a nonnegative net flow $\Delta f(v)$ into every vertex $v$ other than $s$ and $t$, where we define $\Delta f(v) = \sum_u f(u, v)$. A vertex $v$ is *balanced* if $\Delta f(v) = 0$ and *unbalanced* if $\Delta f(v) > 0$. The preflow is *blocking* if it saturates an edge on every path from $s$ to $t$. The wave method finds a blocking preflow and gradually converts it into a blocking flow by balancing vertices, in successive forward and backward passes over the graph.

Each vertex is in one of two states: *unblocked* or *blocked*. An unblocked vertex can become blocked but not vice versa. We balance an unbalanced vertex $v$ by increasing the outgoing flow if $v$ is unblocked and decreasing the incoming flow if $v$ is blocked. More precisely, we balance an unblocked vertex $v$ by repeating the following step until $\Delta f(v) = 0$ (the balancing succeeds) or there is no unsaturated edge $[v, w]$ such that $w$ is unblocked (the balancing fails):

INCREASING STEP. Let $[v, w]$ be an unsaturated edge such that $w$ is unblocked. Increase $f(v, w)$ by $\min\{cap(v, w) - f(v, w), \Delta f(v)\}$.

We balance a blocked vertex $v$ by repeating the following step until $\Delta f(v) = 0$ (such a balancing always succeeds):

DECREASING STEP. Let $[u, v]$ be an edge of positive flow. Decrease $f(u, v)$ by $\min\{f(u, v), \Delta f(v)\}$.

To find a blocking flow, we begin with the preflow that saturates every edge out of $s$ and is zero on all other edges, make $s$ blocked and every other vertex unblocked, and repeat *increase flow* followed by *decrease flow* until there are no unbalanced vertices. (See Fig. 8.6.)

*Increase flow.* Scan the vertices other than $s$ and $t$ in topological order (see Chapter 1), balancing each vertex $v$ that is unbalanced and unblocked when it is scanned; if the balancing fails, make $v$ blocked.

*Decrease flow.* Scan the vertices other than $s$ and $t$ in reverse topological order, balancing each vertex that is unbalanced and blocked when it is scanned.

THEOREM 8.9. *The wave algorithm correctly computes a blocking flow in $O(n^2)$ time and a maximum flow in $O(n^3)$ time.*

*Proof.* The method maintains the invariant that if $v$ is blocked, every path from $v$

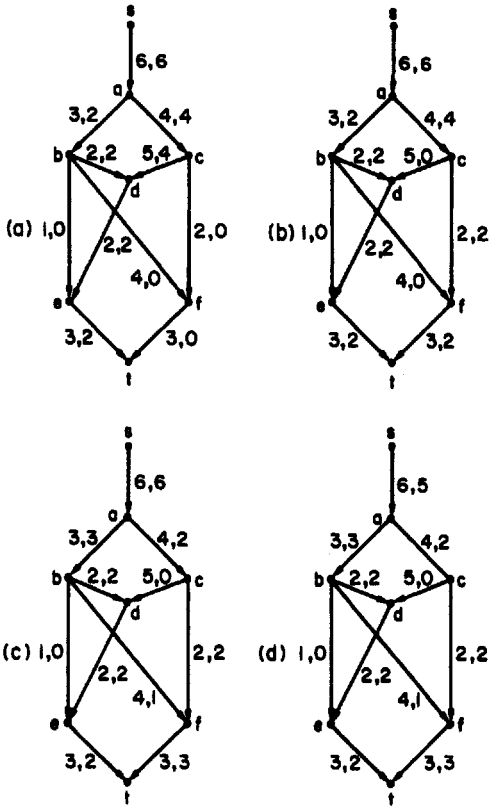



FIG. 8.6. *The wave method of computing a blocking flow.* (a) *After first flow increase. Vertex d is blocked.* (b) *After second flow increase. Vertex c is blocked.* (c) *After third flow increase. Vertex a is blocked.* (d) *Final blocking flow.*

to $t$ contains a saturated edge. Since $s$ is blocked initially, every preflow constructed by the algorithm is blocking. Scanning in topological order during *increase flow* guarantees that after such a step there are no unblocked, unbalanced vertices. Similarly each vertex blocked before a decrease flow step is balanced after the step and remains balanced during the next increase flow step, if any. Thus each increase flow step except the last blocks at least one vertex, and the method halts after at most $n - 1$ iterations of *increase flow* and *decrease flow,* having balanced all vertices except $s$ and $t$ and thus having produced a blocking flow.

There are at most $(n - 2)(n - 1)$ balancings. The flow on an edge $[v, w]$ first increases (while $w$ is unblocked), then decreases (while $w$ is blocked). Each increasing step either saturates an edge or terminates a balancing; each decreasing step either decreases the flow on an edge to zero or terminates a balancing. Thus there are at most $2m + (n - 2)(n - 1)$ increasing and decreasing steps.

To implement the method efficiently we maintain for each vertex $v$ the value of $\Delta f(v)$ and a bit indicating whether $v$ is unblocked or blocked. To balance an

unblocked vertex $v$, we examine the edges out of $v$, beginning with the last edge previously examined, and increase the flow on each edge to which the increasing step applies, until $\Delta f(v) = 0$ or we run out of edges (the balancing fails). Balancing a blocked vertex is similar. With such an implementation the method takes $O(n^2)$ time to find a blocking flow, including the time to topologically order the vertices. Theorem 8.4 gives the $O(n^3)$ time bound for finding a maximum flow. □

When using the wave algorithm to find a maximum flow, we can use the layered structure of the level graphs to find each blocking flow in $O(m + k)$ time, where $k$ is the number of balancings, eliminating the $O(n^2)$ overhead for scanning balanced vertices [25]. This may give an improvement in practice, though the time bound is still $O(n^2)$ in the worst case.

Malhotra, Kumar and Maheshwari [17] suggested another $O(n^2)$-time blocking flow method that is conceptually very simple. Initially we delete from $G$ every vertex and edge not on a path from $s$ to $t$. We maintain for each vertex $v$ the *potential throughput* of $v$, defined by

$$thruput\ (v) = \min\left\{\sum_{[u,v]\in E}(cap\ (u, v) - f(u, v)),\ \sum_{[v,w]\in E}(cap\ (v,w) - f(v, w))\right\}.$$

(To define *thruput* $(s)$ and *thruput* $(t)$, we assume the existence of a dummy edge of infinite capacity from $t$ to $s$.) To find a blocking flow we repeat the following step until $t$ is not reachable from $s$:

SATURATING STEP. Let $v$ be a vertex of minimum potential throughput. Send *thruput* $(v)$ units of flow forward from $v$ to $t$ by scanning the vertices in topological order and backward from $v$ to $s$ by scanning the vertices in reverse topological order. Update all throughputs, delete all newly saturated edges from $G$ (this includes either all edges entering or all edges leaving $v$) and delete all vertices and edges not on a path from $s$ to $t$.

Although this method is simple, it has two drawbacks. When actually implemented, it is at least as complicated as the wave method. Furthermore, it preferentially sends flow through narrow bottlenecks, which may cause it to perform many more augmentations than necessary. For these reasons we prefer the wave method.

A fourth way to find a blocking flow is to saturate one edge at a time as in Dinic's method, but to reduce the time per edge saturation by using an appropriate data structure to keep track of the flow. Galil and Naamad [11] and Shiloach [21] discovered a method of this kind that runs in $O(m(\log n)^2)$ time. Sleator and Tarjan [23], [24] improved the bound to $O(m \log n)$, inventing the data structure of Chapter 5 for this purpose. We conclude this section by describing their method.

Recall that the data structure of Chapter 5 allows us to represent a collection of vertex-disjoint rooted trees, each of whose vertices has a real-valued cost, under the following operations, each of which takes $O(\log n)$ amortized time:

maketree $(v)$: Create a new tree containing the single vertex $v$, of cost zero.
findroot $(v)$: Return the root of the tree containing vertex $v$.

findcost ($v$): Return the pair $[w, x]$, where $x$ is the minimum cost of a vertex on the tree path from $v$ to findroot ($v$) and $w$ is the last vertex on this path of cost $x$.

addcost ($v, x$): Add $x$ to the cost of every vertex on the tree path from $v$ to findroot ($v$).

link ($v, w$): Combine the two trees containing vertices $v$ and $w$ by adding the edge $[v, w]$ ($v$ must be a root).

cut ($v$): Divide the tree containing vertex $v$ into two trees by deleting the edge out of $v$ ($v$ must be a nonroot).

To find a blocking flow, we maintain for each vertex $v$ a current edge $[v, p(v)]$ on which it may be possible to increase the flow. These edges define a collection of trees. (Some vertices may not have a current edge.) The cost of a vertex $v$ is $cap(v, p(v)) - f(v, p(v))$ if $v$ is not a tree root, *huge* if $v$ is a tree root, where *huge* is a constant chosen larger than the sum of all the edge capacities. The following steps are a reformulation of Dinic's algorithm using the five tree operations. We find a blocking flow by first executing maketree ($v$) followed by addcost ($v$, *huge*) for all vertices, then going to *advance* and proceeding as directed.

*Advance.* Let $v$ = findroot ($s$). If there is no edge out of $v$, go to *retreat*. Otherwise, let $[v, w]$ be an edge out of $v$. Perform addcost ($v$, $cap(v, w)$ − *huge*) followed by link ($v, w$). Define $p(v)$ to be $w$. If $w \neq t$, repeat *advance;* if $w = t$, go to *augment*.

*Augment.* Let $[v, \Delta]$ = findcost ($s$). Perform addcost ($s, -\Delta$). Go to *delete*.

*Delete.* Perform cut ($v$) followed by addcost ($v$, *huge*). Define $f(v, p(v)) = cap(v, p(v))$. Delete $[v, p(v)]$ from the graph. Let $[v, \Delta]$ = findcost ($s$). If $\Delta = 0$, repeat *delete;* otherwise go to *advance*.

*Retreat.* If $v = s$ halt. Otherwise, for every edge $[u, v]$, delete $[u, v]$ from the graph and, if $p(u) \neq v$, define $f(u, v) = 0$; if $p(u) = v$, perform cut ($u$), let $[u, \Delta]$ = findcost ($u$), perform addcost ($u$, *huge* − $\Delta$), and define $f(u, v) =$ cap $(u, v) - \Delta$. After deleting all edges $[u, v]$, go to *advance*.

Once the algorithm halts, we use cut, findcost, and addcost as in *retreat* to find the flow on every remaining edge.

THEOREM 8.10. *The Sleator–Tarjan algorithm correctly finds a blocking flow in* $O(m \log n)$ *time and a maximum flow in* $O(nm \log n)$ *time.*

*Proof.* The correctness of the method is immediate. There are $O(m)$ tree operations, giving a time bound of $O(m \log n)$ to find a blocking flow. The time bound for a maximum flow follows from Theorem 8.4. □

It is intriguing to contemplate the possibility of implementing the wave method using the data structure for cutting and linking trees, thereby obtaining an algorithm as fast as any known method on both sparse and dense graphs. We leave this as an open problem; we conjecture that a time bound of $O(m \log (n^2/m))$ for finding a blocking flow can be obtained in this way.

**8.4. Minimum cost flows.** The idea of augmenting paths extends to a more general network flow problem. Let $G$ be a network such that each edge $[v, w]$ has a

cost per unit of flow, *cost* $(v, w)$, in addition to a capacity. For simplicity we shall assume that the costs are skew symmetric, i.e. *cost* $(v, w) = -$ *cost* $(w, v)$. (One can obtain the effect of allowing costs that are not skew symmetric by introducing multiple edges.) The *cost* of a flow $f$ is *cost* $(f) = \sum_{f(v,w)>0}$ *cost* $(v, w)f(v, w) = \sum_{v,w}$ *cost* $(v, w)f(v, w)/2$. A flow is *minimum cost* if among all flows of the same value it has minimum cost. The *minimum cost flow problem* is that of finding a maximum flow of minimum cost.

We define the *cost* of a path to be the sum of its edge costs and the *residual graph* R for a flow $f$ exactly as we did in §8.1, with the extension that *cost* $(v, w)$ is the same on $R$ as on $G$. The following two theorems justify two related algorithms for solving the minimum cost flow problem.

THEOREM 8.11. *A flow $f$ is minimum cost if and only if its residual graph $R$ has no negative cost cycle.*

*Proof.* If $R$ contains a negative cost cycle, then we can reduce the cost of $f$ without changing its value by pushing flow around the cycle. Conversely, suppose $f$ does not have minimum cost. Let $f^*$ be a minimum cost flow of the same value as $f$. Then $f^* - f$ is a flow on $R$ of value zero and negative cost. The flow $f^* - f$ can be partitioned into a sum of flows on cycles in a way analogous to that used in the proof of Lemma 8.3. At least one of these cycles must have negative cost. □

THEOREM 8.12 [1], [12], [13]. *If $f$ is a minimum-cost flow, then any flow obtained from* f *by augmenting along an augmenting path of minimum cost is also a minimum-cost flow.*

*Proof.* Let $p$ be an augmenting path of minimum cost for flow $f$ and let $f'$ be obtained from $f$ by augmenting along $p$. Suppose $f'$ is not minimum. Let $R$ be the residual graph for $f$. By Theorem 8.11 there is a negative cost cycle, say $c$, in the residual graph for $f'$. Cycle $c$ consists of edges in $R$ and one or more edges whose reversals are on $p$. Let $p \oplus c$ be the set of edges on $p$ or on $c$ except for those occurring on $p$ and reversed on $c$. The cost of $p \oplus c$ is *cost* $(p)$ + *cost* $(c) <$ *cost* $(p)$. Furthermore, $p \oplus c$ can be partitioned into a path from $s$ to $t$ and a collection of cycles. By Theorem 8.11 all the cycles must have nonnegative cost; thus the path is an augmenting path of cost less than $p$. This contradiction implies the theorem. □

Theorem 8.11 justifies the *cost reduction method* of finding a minimum cost maximum flow [15]: We begin with a maximum flow, push as much flow as possible along a negative cost cycle in the residual graph, and repeat until there are no negative cycles in the residual graph. Theorem 8.12 justifies an alternative method that works if $G$ has no cycles of negative cost: we find a maximum flow by the augmenting path method, always augmenting along a minimum cost path. (The lack of negative cycles means that the zero flow is minimum cost.) Both of these algorithms are quite practical; unfortunately, neither runs in polynomial time even for integer capacities [27]. For general costs the cost reduction algorithm need not even terminate [26], although for general costs and capacities minimum cost augmentation will terminate if we break ties among augmenting paths of the same cost by choosing a path of fewest edges [4].

Let us consider the case of integer capacities. There are a finite number of

integral flows, and since the cost reduction method maintains integrality it will eventually halt with a minimum cost maximum flow. This means that the integrality theorem (Theorem 8.2) holds for minimum cost flows. If $G$ contains no negative cost cycles then minimum cost augmentation will compute a minimum cost maximum flow $f^*$ in at most $|f^*|$ augmentations. Minimum cost augmentation has another useful property:

LEMMA 8.4. *With minimum cost augmentation, successive augmenting paths have nondecreasing cost.*

*Proof.* Let $f$ be a minimum cost flow and $R$, its residual graph. For any vertex $v$, let $cost\ (v)$ be the minimum cost of a path in $R$ from $s$ to $v$. Then for any edge $[v, w]$ in $R$, $cost\ (v) + cost\ (v, w) \geq cost\ (w)$, with equality if $[v, w]$ is on a minimum cost path from $s$ to $w$. If we augment along a minimum cost path, the only edges in the new residual graph $R'$ that are not in $R$ are of the form $[w, v]$ with $[v, w]$ in $R$ and $cost\ (w, v) = -cost\ (v, w) = cost\ (v) - cost\ (w)$. A proof by induction on the number of edges on a minimum cost path from $s$ to $v$ shows that $cost'\ (v) \geq cost\ (v)$, where $cost'$ is the cost function on $R'$. □

We can compute successive minimum cost augmenting paths using any single source shortest-path algorithm (see Chapter 7). By transforming the edge costs after each augmentation, we can keep the costs nonnegative and thus use Dijkstra's algorithm to find augmenting paths. Edmonds and Karp [4] defined the appropriate transformation, which we used in §7.3 to help solve the all pairs shortest-path problem.

To find a minimum cost augmenting path, we find a shortest-path tree rooted at $s$, defining the length of an edge to be its cost. We use the path in the tree from $s$ to $t$ as our augmenting path. After the augmentation, we redefine the cost of an edge $[v, w]$ in $G$ to be $cost'\ (v, w) = cost\ (v, w) + cost\ (v) - cost\ (w)$, where $cost\ (v)$ is the cost of a minimum cost path from $s$ to $v$. The new edge costs are nonnegative. Furthermore, if $[v, w]$ is an edge on the augmenting path, both $[v, w]$ and $[w, v]$ (which will appear in the new residential graph) have a transformed cost of zero. This method gives us the following theorem:

THEOREM 8.13. *On a graph $G$ with integer capacities and no negative cycles, minimum cost augmentation will find a minimum cost maximum flow $f^*$ in $O(nm + m|f^*|\log_{(2+m/n)}n)$ time. If $G$ is acyclic or has nonnegative edge costs, the time bound is $O(m|f^*|\log_{(2+m/n)}n)$.*

*Proof.* Using Dijkstra's algorithm to find augmenting paths and transforming the costs after each augmentation, the time for the second and successive augmentations is $O(m \log_{(2+m/n)}n)$ per augmentation. Since the initial edge costs are not necessarily nonnegative, we must use breadth-first scanning, with an $O(nm)$ running time, for the first augmentation. We can reduce the time for the first augmentation to $O(m \log_{(2+m/n)}n)$ using Dijkstra's algorithm if the initial costs are nonnegative, or to $O(m)$ using topological scanning if $G$ is acyclic. □

There is still much to be learned about the minimum cost flow problem. Edmonds and Karp [4] developed a "scaling" method that finds a minimum cost maximum flow $f^*$ in a time bound polynomial in $n$, $m$, and $\log|f^*|$, assuming integer capacities, but this method has not received much attention in practice. The

important question of whether there is an algorithm with a running time polynomial in $n$ and $m$ for general capacities and costs is open. Further information on network flows can be found in the books of Lawler [16] and Papadimitriou and Steiglitz [18].

## References


[1] R. G. BUSACKER AND P. J. GOWEN, *A procedure for determining a family of minimal-cost network flow patterns*, Technical Paper 15, O.R.O., 1961.

[2] R. V. CHERKASKY, *Algorithm of construction of maximal flow in networks with complexity of $O(V^2\sqrt{E})$ operations*, Mathematical Methods of Solution of Economical Problems, 7 (1977), pp. 112–125. (In Russian.)

[3] E. A. DINIC, *Algorithm for solution of a problem of maximum flow in a network with power estimation*, Soviet Math. Dokl., 11 (1970), pp. 1277–1280.

[4] J. EDMONDS AND R. M. KARP, *Theoretical improvements in algorithmic efficiency for network flow problems*, J. Assoc. Comput. Mach, 19 (1972), pp. 248–264.

[5] P. ELIAS, A. FEINSTEIN AND C. E. SHANNON, *Note on maximum flow through a network*, IRE Trans. Inform. Theory, IT-2 (1956), pp. 117–119.

[6] S. EVEN, *Graph Algorithms*, Computer Science Press, Potomac, MD, 1979.

[7] S. EVEN AND R. E. TARJAN, *Network flow and testing graph connectivity*, SIAM J. Comput., 4 (1975), pp. 507–518.

[8] L. R. FORD, JR. AND D. R. FULKERSON, *Maximal flow through a network*, Canad. J. Math., 8 (1956), pp. 399–404.

[9] ———, *Flows in Networks*, Princeton Univ. Press, Princeton, NJ, 1962.

[10] Z. GALIL, *An $O(V^{5/3}E^{2/3})$ algorithm for the maximal flow problem*, Acta Inform., 14(1980), pp. 221–242.

[11] Z. GALIL AND A. NAAMAD, *An $O(EV\log^2 V)$ algorithm for the maximal flow problem*, J. Comput. System Sci., 21 (1980), pp. 203–217.

[12] M. IRI, *A new method of solving transportation-network problems*, J. Oper. Res. Soc. Japan, 3 (1960), pp. 27–87.

[13] W. S. JEWELL, *Optimal flow through networks*, Interim Technical Report 8, Massachusetts Institute of Technology, Cambridge, MA, 1958.

[14] A. V. KARZANOV, *Determining the maximal flow in a network by the method of preflows*, Soviet Math. Dokl., 15 (1974), pp. 434–437.

[15] M. KLEIN, *A primal method for minimal cost flows*, Management Sci., 14 (1967), pp. 205–220.

[16] E. L. LAWLER, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart, and Winston, New York, 1976.

[17] V. M. MALHOTRA, M. P. KUMAR AND S. N. MAHESHWARI, *An $O(|V|^3)$ algorithm for finding maximum flows in networks*, Inform. Process Lett., 7 (1978), pp. 277–278.

[18] C. H. PAPADIMITRIOU AND K. STEIGLITZ, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[19] J.-C. PICARD AND M. QUEYRANNE, *Selected applications of maximum flows and minimum cuts in networks*, Rapport Technique EP-79-R-35, Département de Génie Industriel, Ecole Polytechnique de Montréal, Montreal, 1979.

[20] M. QUEYRANNE, *Theoretical efficiency of the algorithm 'capacity' for the maximum flow problem*, Math. Oper. Res., 5 (1980), pp. 258–266.

[21] Y. SHILOACH, *An $O(n \cdot I\log^2 I)$ maximum-flow algorithm*, Tech. Rep. STAN-CS-78-802, Computer Science Dept., Stanford Univ., Stanford, CA, 1978.

[22] Y. SHILOACH AND U. VISHKIN, *An $O(n^2\log n)$ parallel max-flow algorithm*, J. Algorithms, 3 (1982), pp. 128–146.

[23] D. D. SLEATOR, *An $O(nm\log n)$ algorithm for maximum network flow*, Tech. Rep. STAN-CS-80-831, Computer Science Dept., Stanford Univ., Stanford, CA, 1980.

[24] D. D. SLEATOR AND R. E. TARJAN, *A data structure for dynamic trees*, J. Comput. System Sci., to

appear; also in Proc. 13th Annual ACM Symposium on Theory of Computing, 1981, pp. 114–122.

[25] R. E. TARJAN, *A simple version of Karzanov's blocking flow algorithm*, to appear.

[26] N. ZADEH, *More pathological examples for network flow problems*, Math. Programming, 5 (1973), pp. 217–224.

[27] ———, *A bad network problem for the simplex method and other minimum cost flow algorithms*, Math. Programming, 5 (1973), pp. 255–266.

# CHAPTER 9

# Matchings

**9.1. Bipartite matchings and network flows.** The last network optimization problem we shall study has close connections with the maximum flow problem. Let $G = [V, E]$ be an undirected graph each of whose edges has a real-valued *weight*, denoted by *weight* $(v, w)$. A *matching* $M$ on $G$ is a set of edges no two of which have a common vertex. The *size* $|M|$ of $M$ is the number of edges it contains; the *weight* of $M$ is the sum of its edge weights. The *maximum matching problem* is that of finding a matching of maximum size or weight. We shall distinguish four versions of this problem, depending on whether we want to maximize size or weight (unweighted versus weighted matching) and whether $G$ is bipartite or not (bipartite versus nonbipartite matching). The weighted bipartite matching problem is commonly called the *assignment problem;* one application is the assignment of people to tasks, where the weight of an edge $\{x, y\}$ represents the benefit of assigning person $x$ to task $y$.

Bipartite matching problems can be viewed as a special case of network flow problems [8]. Suppose $G$ is bipartite, with a vertex partition $X, Y$ such that every edge has one end in $X$ and the other in $Y$. We shall denote a typical edge by $\{x, y\}$ with $x \in X$ and $y \in Y$. Let $s$ and $t$ be two new vertices. Construct a graph $G'$ with vertex set $V \cup \{s, t\}$, source $s$, sink $t$, and capacity-one edges $[s, x]$ of cost zero for every $x \subset X$, $[y, t]$ of cost zero for every $y \subset Y$, and $[x, y]$ of cost $-weight\ (x, y)$ for every $\{x, y\} \subset E$. (See Fig. 9.1.) $G'$ is a unit network as defined in Chapter 8.

An integral flow $f$ on $G'$ defines a matching on $G$ of size $|f|$ and weight $-cost(f)$ given by the set of edges $\{x\ y\}$ such that $[x, y]$ has flow one. Conversely a matching $M$ on $G$ defines a flow of value $|M|$ and cost $-weight(M)$ that is one on each path $[s, x]$, $[x, y]$, $[y, t]$ such that $\{x, y\} \in M$. This means that we can solve a matching problem on $G$ by solving a flow problem on $G'$.

Suppose we want a maximum size matching. Any integral maximum flow on $G'$ gives a maximum size matching on $G$. We can find such a flow in $O(\sqrt{n}\, m)$ time using Dinic's algorithm, since $G'$ is unit (see Theorem 8.8). Thus we have an $O(\sqrt{n}\, m)$-time algorithm for unweighted bipartite matching. This algorithm can be translated into the terminology of alternating paths (which we shall develop in the next section), and it was originally discovered in this form by Hopcroft and Karp [13]. Even and Tarjan [7] noted the connection with Dinic's algorithm.

Suppose we want a maximum weight matching. Since $G'$ is acyclic, it has no negative cost cycles, and we can apply minimum cost augmentation to $G'$ (see §8.4). Starting with the zero flow, this method will produce a sequence of at most $n/2$ minimum cost flows of increasing value, the last of which is a minimum cost maximum flow. Because successive augmenting paths have nondecreasing cost (Lemma 8.4), if we stop the algorithm just after the last augmentation along a path

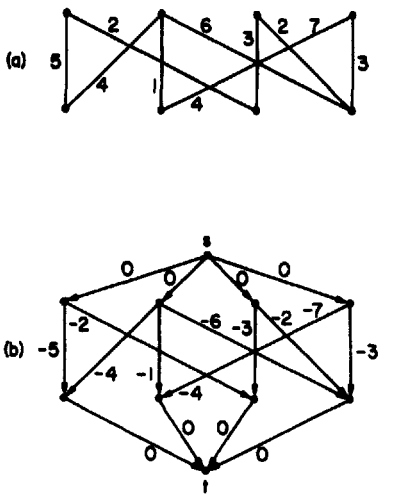



FIG. 9.1. *Transformation of a bipartite matching problem to a network flow problem.* (a) *Bipartite graph defining a weighted matching problem.* (b) *Corresponding network. Numbers on edges are costs; all capacities are one.*

of negative cost we will have a flow corresponding to a maximum weight matching. As implemented in §8.4, minimum cost augmentation solves the weighted bipartite matching problem in $O(n\,m\log_{(2+m/n)}n)$ time. This method, too, can be translated into the terminology of alternating paths, and it was discovered in this form by Kuhn [17], who named it the *Hungarian method* in recognition of Konig [15], [16] and Egervary's [5] work on maximum matching, which among other results produced the *Konig–Egervary theorem.* This theorem is the special case of the max-flow min-cut theorem for unweighted bipartite matching: the maximum size of a bipartite matching is equal to the minimum size of a vertex set containing at least one vertex of every edge.

**9.2. Alternating paths.** Nonbipartite matching is more complicated than bipartite matching. The idea of augmenting paths carries over from network flow theory, but to get efficient algorithms we need another idea, contributed by Edmonds in a paper with a flowery title [3]. In this section we shall develop the properties of augmenting paths in the setting of matching theory.

Let $M$ be a matching. An edge in $M$ is a *matching edge*; every edge not in $M$ is *free*. A vertex is *matched* if it is incident to a matching edge and *free* otherwise. An *alternating path* or *cycle* is a simple path or cycle whose edges are alternately matching and free. The *length* of an alternating path or cycle is the number of edges it contains; its *weight* is the total weight of its free edges minus the weight of its matching edges. An alternating path is *augmenting* if both its ends are free vertices. If $M$ has an augmenting path then $M$ is not of maximum size, since we can increase its size by one by interchanging matching and free edges along the path. We call this an *augmentation.* The following theorem is analogous to Lemma 8.2:

THEOREM 9.1 [3, 13]. *Let $M$ be a matching, $\overline{M}$ a matching of maximum size, and $k = |\overline{M}| - |M|$. Then $M$ has a set of $k$ vertex-disjoint augmenting paths, at least one of length at most $n/k - 1$.*

*Proof.* Let $M \oplus \overline{M}$ be the *symmetric difference* of $M$ and $\overline{M}$, the set of edges in $M$ or in $\overline{M}$ but not in both. Every vertex is adjacent to at most two edges of $M \oplus \overline{M}$; thus the subgraph of $G$ induced by $M \oplus \overline{M}$ consists of a collection of paths and even-length cycles that are alternating with respect to $M$ (and to $\overline{M}$). $M \oplus \overline{M}$ contains exactly $k$ more edges in $\overline{M}$ than in $M$; thus it contains at least $k$ paths that begin and end with an edge of $\overline{M}$. These paths are vertex-disjoint and augmenting for $M$; at least one has length at most $n/k - 1$. □

Berge [2] and Norman and Rabin [20] proved a weaker form of Theorem 9.1: A matching is of maximum size if and only if it has no augmenting path. We can construct a maximum size matching by beginning with the empty matching and repeatedly performing augmentations until there are no augmenting paths; this takes at most $n/2$ augmentations. We call this the *augmenting path method* for *maximum matching*. Before discussing how to find augmenting paths, let us obtain a result for weighted matchings analogous to Theorem 8.12.

THEOREM 9.2. *Let $M$ be a matching of maximum weight among matchings of size $|M|$, let $p$ be an augmenting path for $M$ of maximum weight, and let $M'$ be the matching formed by augmenting $M$ using $p$. Then $M'$ is of maximum weight among matchings of size $|M| + 1$.*

*Proof.* Let $\overline{M}$ be a matching of maximum weight among matchings of size $|M| + 1$. Consider the symmetric difference $M \oplus \overline{M}$. Define the weight of a path or cycle in $M \oplus \overline{M}$ with respect to $M$. Any cycle or even-length path in $M \oplus \overline{M}$ must have weight zero; a cycle or path of positive or negative weight contradicts the choice of $M$ or $\overline{M}$, respectively. $M \oplus \overline{M}$ contains exactly one more edge in $\overline{M}$ than in $M$; thus we can pair all but one of the odd-length paths so that each pair has an equal number of edges in $M$ and in $\overline{M}$. Each pair of paths must have total weight zero; a positive or negative weight pair contradicts the choice of $M$ or $\overline{M}$. Augmenting $M$ using the remaining path gives a matching of size $|M| + 1$ and of the same weight as $\overline{M}$. The theorem follows. □

Theorem 9.2 implies that the augmenting path method will compute maximum weight matchings of all possible sizes if we always augment using a maximum weight augmenting path. The analogue of Lemma 8.4 holds for matchings; namely, this method will augment along paths of successively decreasing weight. Thus if we want a maximum weight matching, we can stop after the last augmentation along a path of positive weight.

**9.3. Blossoms.** There remains the problem of finding augmenting paths, maximum weight or otherwise. The natural way to find an augmenting path is to search from the free vertices, advancing only along alternating paths. If a search from one free vertex reaches another, we have found an alternating path. This method works fine for bipartite graphs, but on nonbipartite graphs there is a subtle difficulty: a vertex can appear on an alternating path in either parity, where we call a vertex *even*

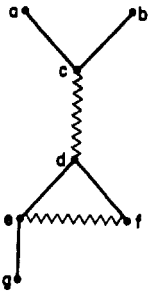



FIG. 9.2. *A graph in which it is hard to find an augmenting path. If we search from a and allow one visit per vertex, labeling c and e odd prevents discovery of the augmenting path [a, c, d, f, e, g]. Allowing two visits per vertex may produce the supposed augmenting path [a, c, d, e, f, d, c, b].*

if it is an even distance from the starting free vertex and *odd* otherwise. (Edmonds [3] called even vertices "outer" and odd vertices "inner".) If during the search we do not allow two visits to a vertex, one in each parity, we may miss an augmenting path; if we allow visits in both parities we may generate a supposedly augmenting path that is not simple. (See Fig. 9.2.)

Such an anomaly can only occur if $G$ contains the configuration shown in Fig. 9.3, consisting of an alternating path $p$ from a free vertex $s$ to an even vertex $v$ and an edge from $v$ to another even vertex $w$ on $p$. We call the odd-length cycle formed by $\{v, w\}$ and the part of $p$ from $w$ to $v$ a *blossom*; vertex $w$ is the *base* of the blossom and the part of $p$ from $s$ to $w$ is the *stem* of the blossom. Edmonds [3] discovered how to

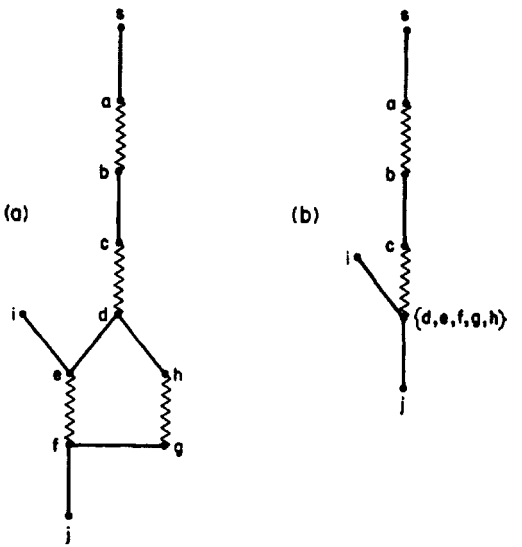



FIG. 9.3. *Shrinking a blossom.* (a) *Blossom defined by path from s to d and cycle [d, e, f, g, h, d]. Vertex d is the base.* (b) *Shrunken blossom. Augmenting path from s to i(j) corresponds to augmenting path in original graph going around blossom clockwise (counterclockwise).*

deal with this situation: We shrink the blossom to a single vertex, called a *shrunken blossom*, and look for an augmenting path in the shrunken graph $G$.

In our discussion we shall sometimes not distinguish between the expanded and shrunken forms of a blossom; the graph being considered will resolve this ambiguity. The following theorem justifies blossom-shrinking:

THEOREM 9.3. *If $G'$ is formed from $G$ by shrinking a blossom $b$, then $G'$ contains an augmenting path if and only if $G$ does.*

*Proof* (only if). Suppose $G'$ contains an augmenting path $p$. If $p$ avoids $b$, then $p$ is an augmenting path in $G$. If $p$ contains $b$, either $b$ is a free vertex or $p$ contains the matching edge incident to $b$. In either case expansion of the blossom either leaves $p$ an augmenting path or breaks $p$ into two parts, one of which contains the base of blossom, that can be reconnected to form an augmenting path by inserting a path going around the blossom in the appropriate direction from the base (see Fig. 9.3). Thus $G$ contains an augmenting path. □

The "if" direction of Theorem 9.3 is harder to prove; we shall obtain it by proving the correctness of an algorithm developed by Edmonds [3] that finds augmenting paths using blossom-shrinking. The algorithm consists of an exploration of the graph that shrinks blossoms as they are encountered. The algorithm builds a forest consisting of trees of alternating paths rooted at the free vertices. For purposes of the algorithm we replace every undirected edge $\{v, w\}$ by a pair of directed edges $[v, w]$ and $[w, v]$. Each vertex is in one of three states: *unreached*, *odd*, or *even*. For any matched vertex $v$, we denote by *mate* $(v)$ the vertex $w$ such that $\{v, w\}$ is a matching edge. For each vertex $v$ the algorithm computes $p(v)$, the parent of $v$ in the forest. Initially every matched vertex is unreached and every free vertex $v$ is even, with $p(v) =$ **null**. The algorithm consists of repeating the following step until an augmenting path is found or there is no unexamined edge $[v, w]$ with $v$ even (see Fig. 9.4):

EXAMINE EDGE (Edmonds). Choose an unexamined edge $[v, w]$ with $v$ even and examine it, applying the appropriate case below:

*Case* 1. $w$ is odd. Do nothing. This case occurs whenever $\{v, w\}$ is a matching edge and can also occur if $\{v, w\}$ is free.

*Case* 2. $w$ is unreached and matched. Make $w$ odd and *mate* $(w)$ even; define $p(w) = v$ and $p(\textit{mate}\ (w)) = w$.

*Case* 3. $w$ is even and $v$ and w are in different trees. Stop; there is an augmenting path from the root of the tree containing $v$ to the root of the tree containing $w$.

*Case* 4. $w$ is even and $v$ and $w$ are in the same tree. Edge $\{v, w\}$ forms a blossom. Let $u$ be the nearest common ancestor of $v$ and $w$. Condense every vertex that is a descendant of $u$ and an ancestor of $v$ or $w$ into a blossom $b$; define $p(b) = p(u)$ and $p(x) = b$ for each vertex $x$ such that $p(x)$ is condensed into $b$.

We call this the *blossom-shrinking algorithm*. A vertex (either an original or a blossom) is *shrunken* if it has been condensed into a blossom (and thus no longer appears in the graph); any odd, even, or shrunken vertex is *reached*. We regard a blossom $b$ as containing not only the vertices on the cycle forming $b$ but also all

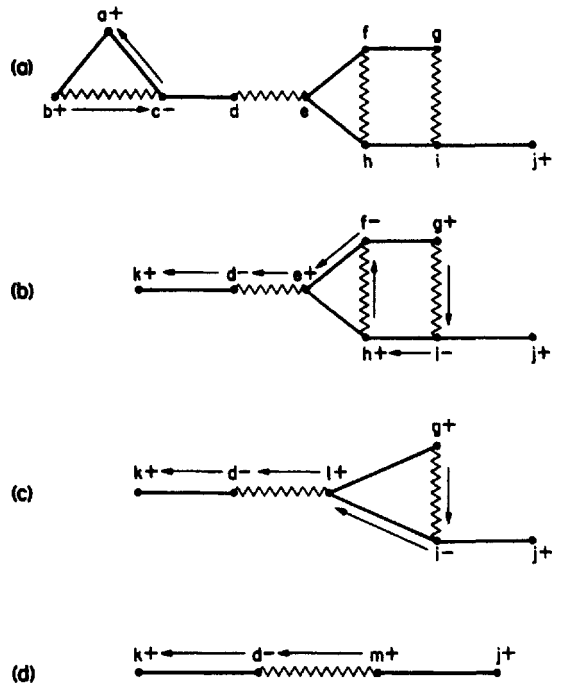



FIG. 9.4. *Execution of the blossom-shrinking algorithm. Plus denotes an even vertex, minus an odd vertex. Arrows denote parents.* (a) *After examining* $[a, c]$ (*Case* 2). (b) *After examining* $[b, a]$ (*Case* 4), $[c, d]$, $[e, f]$, $[h, i]$, *and* $[g, f]$ (*Case* 1). *Vertex* $k = \{a, b, c\}$. (c) *After examining* $[e, h]$. *Vertex* $l = \{e, f, h\}$. (d) *After examining* $[l, g]$. *Vertex* $m = \{g, i, l\}$. *On examining* $[j, m]$ (*Case* 3), *the algorithm halts with success.*

shrunken vertices combined through repeated shrinking to form the blossoms on the cycle; that is, we treat containment as transitive.

THEOREM 9.4. *The blossom-shrinking algorithm succeeds* (*stops in Case* 3) *if and only if there is an augmenting path in the original graph.*

*Proof.* If the algorithm succeeds, there is an augmenting path in the current shrunken graph. This path can be expanded to an augmenting path in the original graph by expanding blossoms in the reverse order of their shrinking, reconnecting the broken parts of the path each time a blossom on the path is expanded, as described in the proof of Theorem 9.3 (only if).

To prove the converse, we first note several properties of the algorithm. If $v$ is a reached, matched vertex, then *mate* $(v)$ is also reached. If $v$ is a shrunken, free vertex, then $v$ is contained in a free blossom. If the algorithm stops with failure, any two even or shrunken vertices that were adjacent at some time during the computation are condensed into a single blossom when the algorithm halts. To verify this third claim, suppose to the contrary that $\{v, w\}$ is an edge such that $v$ and $w$ are both even or shrunken. Without loss of generality suppose $v$ became even or shrunken after $w$. Eventually either $v$ and $w$ will be condensed into a common

blossom, or an edge corresponding to $[v, w]$ in the current shrunken graph will be examined; such an examination causes $v$ and $w$ to be condensed into a common blossom.

Suppose the algorithm fails but there is an augmenting path $p = [x_0, x_1, \cdots, x_{2l+1}]$. Consider the situation after the algorithm halts.

It suffices to show that $x_i$ is even (or shrunken) for all even $i$, $0 \leqq i \leqq 2l$. For then symmetry implies that $x_i$ is even (or shrunken) for all $i$, $0 \leqq i \leqq 2l + 1$, a contradiction. Thus let $i$ be the least even index such that $x_i$ is not even or shrunken. Observe that $x_{i-1}$ is not even: $i > 0$ and $x_{i-1}$ is the mate of $x_i$. Further $x_{i-1}$ is not odd. Since $x_{i-2}$ is even, $x_{i-1}$ is reached, which implies $x_{i-1}$ is even. Let $j$ be the smallest index less than $i-1$ such that $x_j, x_{j+1}, \cdots, x_{i-1}$ are even. All of these vertices are in the same blossom. But this blossom has two bases: $x_{i-1}$ is the base, since its mate $x_i$ is not in the blossom; $x_j$ is the base, since its mate is not in the blossom ($j$ is even, and either $j > 0$ and its mate is $x_{j-1}$, or $j = 0$ and it has no mate). This is impossible, which implies that the algorithm must halt with success if there is an augmenting path. □

Theorem 9.4 implies the "if" part of Theorem 9.3. Let $G'$ be formed from $G$ by shrinking a blossom $b$. Suppose we run the algorithm in parallel on $G$ and $G'$. On $G$, we begin by following the path to and around the blossom and shrinking it. On $G'$, we begin by following the path to $b$. Now the algorithm is in exactly the same state on $G$ and $G'$, and it will succeed on $G$ if and only if it succeeds on $G'$.

We conclude this section with two easy-to-prove observations about the blossom-shrinking algorithm and its use in the augmenting path method. After performing an augmentation, we need not immediately expand all blossoms; expansion is required only when a blossom is on an augmenting path or when it becomes an odd vertex. Suppose that while searching for an augmenting path we generate a tree such that every edge $[v, w]$ with $v$ in the tree has been examined and every edge $[w, v]$ with $v$ but not $w$ in the tree has $v$ odd. (Edmonds called such a tree *Hungarian*.) Then we can permanently delete from the graph all vertices in the tree and in its blossoms; none will ever again be on an augmenting path, no matter what augmentations occur.

**9.4. Algorithms for nonbipartite matching.** The augmenting path method, using blossom-shrinking to find augmenting paths, will find a maximum size matching in polynomial time. Edmonds claimed an $O(n^4)$ time bound, which is easy to obtain; see the book of Papadimitriou and Steiglitz [21]. Witzgall and Zahn [22] gave a related algorithm that instead of shrinking blossoms uses a vertex labeling scheme to keep track of the blossoms implicitly; they did not discuss running time. Balinski [1] gave a similar algorithm that runs in $O(n^3)$ time. Both Gabow [9], [10] and Lawler [18] discovered how to implement Edmonds's algorithm to run in $O(n^3)$ time. As Gabow noted, the running time can be reduced to $O(nm\alpha(m, n))$ using the disjoint set union algorithm discussed in Chapter 2. The linear-time set union algorithm of Gabow and Tarjan [11] further reduces the running time, to $O(nm)$. We shall describe how to implement blossom-shrinking to attain the best of these bounds.

The hard part is to keep track of blossoms. We do this by manipulating only the

vertices in the original graph. Each vertex $v$ in the current shrunken graph corresponds to the set of original vertices condensed to form it. At any time during the running of the algorithm these sets partition the original vertices; we maintain this partition using the operations makeset, link, and find defined in Chapter 2. We define the *origin* of a vertex $v$ in the shrunken graph inductively to be $v$ if $v$ is an original vertex or the origin of the base of $v$ if $v$ is a blossom. We label the canonical vertex of each set (see Chapter 2) with the origin of the current vertex corresponding to the set; that is, if $v$ is an original vertex, *origin* (find ($v$)) is the origin of the current vertex into which $v$ has been condensed.

We represent the vertices in the current shrunken graph by their origins. Instead of modifying the edges of the graph as blossoms are shrunk, we retain the original edges and use origin and find to convert them into edges in the shrunken graph. More specifically, let $v' =$ *origin* (find ($v$)) for any original vertex $v$. Then $[v', w']$ is the current edge corresponding to original edge $[v, w]$. Note that if $v$ is unreached or odd, $v' = v$.

As we explore the graph we generate a spanning forest, which we represent by defining *predecessors* of the odd vertices. When examination of an original edge $[v, w]$ causes an unreached vertex $w$ to become odd, we define *pred* ($w$) $= v$. From predecessors and mates we can compute parents in the forest as follows: if $v$ is an origin, its parent $p(v)$ is *mate* ($v$) if $v$ is even, *pred* ($v$)' if $v$ is odd; we assume that *mate* ($v$) = **null** if $v$ is a free vertex.

We also compute certain information necessary to construct an augmenting path. For each odd vertex $v$ condensed into a cycle we define a *bridge*. Suppose the examination of an original edge $[v, w]$ causes a blossom to form containing odd vertex $x$. We define *bridge* ($x$) to be $[v, w]$ if $x$ is an ancestor of $v'$ before condensing or to be $[w, v]$ if $x$ is an ancestor of $w'$.

Initialization for each vertex $v$ consists of defining *origin* ($v$) $= v$, executing makeset ($v$), and making $v$ even if it is free and unreached if it is matched. To execute the algorithm we repeat the following step until detecting an augmenting path or running out of unexamined edges $[v, w]$ such that $v'$ is even (see Fig. 9.5):

EXAMINE EDGE. Choose an unexamined edge $[v, w]$ such that $v'$ is even and examine it, applying the appropriate case below.

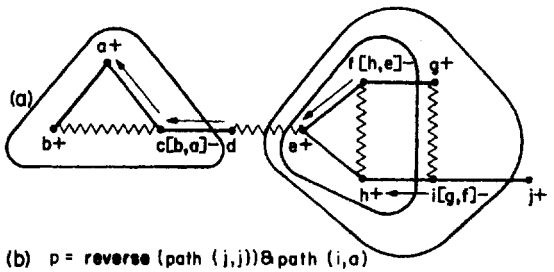
*Case* 1. $w'$ is odd. Do nothing.

*Case* 2. $w'$ is unreached. Make $w'$ odd and *mate* ($w'$) even; define *pred* ($w'$) $= v$.

*Case* 3. $w'$ is even and $v'$ and $w'$ are in different trees. Stop; there is an augmenting path.

*Case* 4. $w'$ is even, $v' \neq w'$, and $v'$ and $w'$ are in the same tree. A blossom has been formed. Let $u$ be the nearest common ancestor of $v'$ and $w'$. For every vertex $x$ that is a descendant of $u$ and an ancestor of $v'$, perform link (find ($u$), find ($x$)) and if $x$ is odd define *bridge* ($x$) $= [v, w]$. For every vertex $x$ that is a descendant of $u$ and an ancestor of $w'$, perform link (find ($u$), find ($x$)) and if $x$ is odd define *bridge* ($x$) $= [w, v]$. Define *origin* (find ($u$)) $= u$.

To complete the implementation we must fill in a few more details. We need a way to choose unexamined edges $[v, w]$ such that $v'$ is even. For this purpose we




(b) p = reverse (path (j,j)) & path (i,a)

= [j] & [i] & reverse (path (g,g)) & path (f,a)

= [j, i, g] & [f] & reverse (path (h,h)) & path (e,a)

= [j, i, g, f, h] & [e, d] & path (c,a)

= [j, i, g, f, h, e, d] & [c] & reverse (path (b,b)) & path (a,a)

= [j, i, g, f, h, e, d, c, b, a].

FIG. 9.5. *Efficient implementation of blossom-shrinking.* (a) *Labeling of graph in Fig. 9.4. Plus denotes an even, minus an odd vertex; arrows denote predecessors. Edges next to shrunken odd vertices are bridges. Blossoms are circled. The origins of k, l, m are a, e, e, respectively.* (b) *Construction of augmenting path.*

maintain the set of such edges, from which we delete one edge at a time. Initially the set contains all edges $[v, w]$ such that $v$ is free. When an unreached vertex $v$ becomes even or an odd vertex $v$ is condensed into a blossom, we add all edges $[v, w]$ to the set. By varying the examination order we can implement various search strategies.

We also need a way to distinguish between Case 3 and Case 4 and to determine the set of edges to be condensed into a blossom if Case 4 applies. When examining an edge $[v, w]$ such that $w'$ is even, we ascend through the forest simultaneously from $v'$ and from $w'$, computing $v_0 = v'$, $w_0 = w'$, $v_1$, $w_1$, $v_2$, $w_2$, $\cdots$, where $v_{i+1} = p(v_i)$ and $w_{i+1} = p(w_i)$. We stop when reaching different free vertices from $v$ and from $w$ (Case 3 applies), or when reaching from $w'$ a vertex $u$ previously reached from $v'$ or vice versa (Case 4 applies). In the latter case $u$ is the nearest common ancestor of $v'$ and $w'$, and the blossom consists of the vertices $v_0, v_1, \cdots, v_j = u$ and $w_0, w_1, \cdots, w_k = u$. The number of vertices visited by this process is $O(n)$ in Case 3, at most twice the number of vertices on the cycle defining the blossom in Case 4.

The total number of vertices on all blossom cycles is at most $2n - 2$, since there are at most $n - 1$ blossoms and shrinking a blossom of $k$ vertices reduces the number of vertices in the graph by $k - 1$. A simple analysis shows that the disjoint set operations, of which there are $n$ makeset, at most $n - 1$ link and $O(m)$ find operations, dominate the running time of the algorithm. If we use the data structure of Chapter 2 to implement makeset, link and find, the time to detect an augmenting

path is $O(m\alpha(m, n))$. The Gabow–Tarjan set union algorithm [11] is also usable and reduces the running time to $O(m)$.

There remains the problem of constructing an augmenting path once one is detected. Suppose the algorithm stops in Case 3, having found an edge $[v, w]$ such that $v'$ and $w'$ are even and in different trees. Let $x$ be the root of the tree containing $v'$ and $y$ the root of the tree containing $w'$; the algorithm determines $x$ and $y$ in the process of detecting an augmenting path. Then **reverse** (*path* $(v, x)$) & *path* $(w, y)$ is an augmenting path, where **reverse** reverses a list (see Chapter 1) and *path* is defined recursively as follows (see Fig. 9.5):

$$path\ (v, w) = \begin{cases} [v] \text{ if } v = w, \\ [v, mate\ (v)]\ \&\ path\ (pred\ (mate\ (v)), w) \text{ if} \\ \quad v \neq w \text{ and } v \text{ is even}, \\ [v]\ \&\ \textbf{reverse}\ (path\ (x, mate\ (v)))\ \&\ path\ (y, w) \text{ if} \\ \quad v \neq w \text{ and } v \text{ is odd, where } [x, y] = bridge\ (v). \end{cases}$$

The function *path* $(v, w)$ defines an even-length alternating path from $v$ to $w$ beginning with a matching edge, under the assumption that at some time during the running of the blossom-shrinking algorithm $v'$ is a descendant of $w$ in the forest. An induction on the number of blossoms shrunk verifies that *path* is correct. The time required to compute *path* $(v, w)$ is proportional to the length of the list returned, since with an appropriate implementation of lists we can perform concatenation and reversal in $O(1)$ time (see §1.3). With this method the time needed to construct an augmenting path is $O(n)$, and the time to find a maximum size matching is either $O(nm\alpha(m, n))$ or $O(nm)$ depending on the disjoint set implementation.

This algorithm is not the last word on unweighted nonbipartite matching. Even and Kariv [6], [14], in a remarkable tour-de-force, managed to generalize the Hopcroft–Karp bipartite matching algorithm by including blossom-shrinking. Their algorithm, though complicated, runs in $O(\min\{n^{2.5}, \sqrt{n}\, m \log \log n\})$ time. Micali and Vazirani [19], using the same ideas, obtained a simplified algorithm with a running time of $O(\sqrt{n}\, m)$. Thus the best time bounds for unweighted bipartite and nonbipartite matching are the same.

The situation is similar for weighted nonbipartite matching. Edmonds [4] obtained an $O(n^4)$-time algorithm that combines maximum weight augmentation (Theorem 9.2) with blossom-shrinking. With Edmonds's method it is necessary to perserve shrunken blossoms from augmentation to augmentation, only expanding or shrinking a blossom under certain conditions determined by the search for a maximum weight augmenting path. Gabow [9] and Lawler [18] independently discovered how to implement this algorithm to run in $O(n^3)$ time. Recently Galil, Micali, and Gabow [12] reduced the time bound to $O(n\, m \log n)$ by using an extension of meldable heaps (see Chapter 3) to speed up the search for an augmenting path. Thus the best known time bound for weighted nonbipartite matching is $O(\min\{n^3, n\, m \log n\})$. This is slightly larger than the best known bound for the bipartite case, $O(n\, m \log_{(2+m/n)} n)$. Curiously, the bound *is* the same as the

best known bound for maximum network flow, although the algorithms for the two problems use different data structures and techniques. We conjecture that there is an $O(n\,m \log (n^2/m))$-time algorithm for weighted bipartite matching.

## References


[1] M. L. BALINSKI, *Labelling to obtain a maximum matching*, in Proc. Combinatorial Mathematics and its Applications, North Carolina Press, Chapel Hill, NC, 1967, pp. 585–602.

[2] C. BERGE, *Two theorems in graph theory*, Proc. Natl. Acad. Sci., 43 (1957), pp. 842–844.

[3] J. EDMONDS, *Paths, trees, and flowers*, Canad. J. Math., 17 (1965), pp. 449–467.

[4] ———, *Matching and a polyhedron with* 0-1 *vertices*, J. Res. Nat. Bur. Standards Sect. B, 69 (1965), pp. 125–130.

[5] J. EGERVÁRY, *Matrixok kombinatórius tulajdonságairól*, Mat. Fiz. Lapok, 38 (1931), pp. 16–28. (In Hungarian.)

[6] S. EVEN AND O. KARIV, *An $O(n^{2.5})$ algorithm for maximum matching in general graphs*, in Proc. 16th Annual IEEE Symposium on Foundations of Computer Science, 1975, pp. 100–112.

[7] S. EVEN AND R. E. TARJAN, *Network flow and testing graph connectivity*, SIAM J. Comput., 4 (1975), pp. 507–518.

[8] L. R. FORD, JR. AND D. R. FULKERSON, *Flows in Networks*, Princeton Univ. Press, Princeton, NJ, 1962.

[9] H. N. GABOW, *Implementation of algorithms for maximum matching on nonbipartite graphs*, Ph.D. thesis, Dept. Computer Science, Stanford Univ., Stanford, CA, 1973.

[10] ———, *An efficient implementation of Edmonds' algorithm for maximum matching on graphs*, J. Assoc. Comput. Mach., 23 (1976), pp. 221–234.

[11] H. N. GABOW AND R. E. TARJAN, *A linear-time algorithm for a special case of disjoint set union*, Proc. Fifteenth Annual ACM Symposium on Theory of Computing, 1983, 246–251.

[12] Z. GALIL, S. MICALI AND H. GABOW, *Maximal weighted matching on general graphs*, in Proc. 23rd Annual IEEE Symposium on Foundations of Computer Science, 1982, pp. 255–261.

[13] J. E. HOPCROFT AND R. M. KARP, *An $n^{5/2}$ algorithm for maximum matching in bipartite graphs*, SIAM J. Comput., 2 (1973), pp. 225–231.

[14] O. KARIV, *An $O(n^{2.5})$ algorithm for maximum matching in general graphs*, Ph.D. thesis, Dept. Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel, 1976.

[15] D. KÖNIG, *Graphen und Matrizen*, Mat. Fiz. Lapok, 38 (1931), pp. 116–119.

[16] ———, *Theorie der endlichen und unendlichen Graphen*, Chelsea, New York, 1950.

[17] H. W. KUHN, *The Hungarian method for the assignment problem*, Naval Res. Logist. Quart., 2 (1955), pp. 83–98.

[18] E. L. LAWLER, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York, 1976.

[19] S. MICALI AND V. V. VAZIRANI, *An $O(\sqrt{|V|}\cdot|E|)$ algorithm for finding maximum matching in general graphs*, in Proc. 21st Annual IEEE Symposium on Foundations of Computer Science, 1980, pp. 17–27.

[20] R. Z. NORMAN AND M. O. RABIN, *An algorithm for a minimum cover of a graph*, Proc. Amer. Math. Soc., 10 (1959), pp. 315–319.

[21] C. H. PAPADIMITRIOU AND K. STEIGLITZ, *Combinatorial Optimization: Networks and Complexity*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[22] C. WITZGALL AND C. T. ZAHN, JR., *Modification of Edmonds' maximum matching algorithm*, J. Res. Nat. Bur. Standards Sect. B, 69 (1965), pp. 91–98.

*This page intentionally left blank*

# References


W. ACKERMANN, *Zum Hilbertschen Aufbau der reellen Zahlen*, Math. Ann., 99 (1928), pp. 118–133.

G. M. ADEL'SON-VEL'SKII AND E. M. LANDIS, *An algorithm for the organization of information*, Soviet Math. Dokl., 3 (1962), pp. 1259–1262.

B. ALLEN AND I. MUNRO, *Self-organizing search trees*, J. Assoc. Comput. Mach., 25 (1978), pp. 526–535.

A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

———, *On finding lowest common ancestors in trees*, SIAM J. Comput., 5 (1975), pp. 115–132.

B. ASPVALL AND R. E. STONE, *Khachiyan's linear programming algorithm*, J. Algorithms, 1 (1980), pp. 1–13.

A. O. L. ATKIN AND R. G. LARSON, *On a primality test of Solovay and Strassen*, SIAM J. Comput., 11 (1982), pp. 789–791.

M. L. BALINSKI, *Labelling to obtain a maximum matching*, Proc. Combinatorial Mathematics and its Applications, Univ. North Carolina Press, Chapel Hill, NC, 1967, pp. 585–602.

L. BANACHOWSKI, *A complement to Tarjan's result about the lower bound on the complexity of the set union problem*, Inform. Process. Lett., 11 (1980), pp. 59–65.

R. BAYER, *Symmetric binary B-trees: Data structure and maintenance algorithms*, Acta Inform., 1 (1972), pp. 290–306.

R. BAYER AND E. MCCREIGHT, *Organization of large ordered indexes*, Acta Inform., 1 (1972), pp. 173–189.

R. E. BELLMAN, *On a routing problem*, Quart. Appl. Math., 16 (1958), pp. 87–90.

S. W. BENT, D. D. SLEATOR AND R. E. TARJAN, *Biased search trees*, SIAM J. Compnt., submitted.

C. BERGE, *Two theorems in graph theory*, Proc. Natl. Acad. Sci., 43 (1957), pp. 842–844.

———, *Graphs and Hypergraphs*, North-Holland, Amsterdam, 1973.

C. BERGE AND A. GHOUILA-HOURI, *Programming, Games, and Transportation Networks*, John Wiley, New York, 1965.

J. R. BITNER, *Heuristics that dynamically organize data structures*, SIAM J. Comput., 8 (1979), pp. 82–110.

M. BLUM, R. W. FLOYD, V. R. PRATT, R. L. RIVEST AND R. E. TARJAN, *Time bounds for selection*, J. Comput. System Sci., 7 (1973), pp. 448–461.

F. BOCK, *An algorithm to construct a minimum directed spanning tree in a directed network*, in Developments in Operations Research, Gordon and Breach, New York, 1971, pp. 29–44.

J. A. BONDY AND U. S. R. MURTY, *Graph Theory with Applications*, North-Holland, New York, 1976.

O. BORŮVKA, *O jistém problému minimálním*, Práca Moravské Přírodovědecké Společnosti, 3 (1926), pp. 37–58. (In Czech.)

M. R. BROWN, *Implementation and analysis of binomial queue algorithms*, SIAM J. Comput., 7 (1978), pp. 298–319.

R. G. BUSACKER AND P. J. GOWEN, *A procedure for determining a family of minimal-cost network flow patterns*, Technical Paper 15, O.R.O., 1961.

R. G. BUSACKER AND T. L. SAATY, *Finite Graphs and Networks: An Introduction with Applications*, McGraw-Hill, New York, 1965.

P. M. CAMERINI, L. FRATTA AND F. MAFFIOLI, *A note on finding optimum branchings*, Networks, 9 (1979), pp. 309–312.

B. CARRÉ, *Graphs and Networks*, Clarendon Press, Oxford, 1979.

J. L. CARTER AND M. N. WEGMAN, *Universal classes of hash functions*, J. Comput. System. Sci., 18 (1979), pp. 143–154.

G. J. CHAITIN, *On the length of programs for computing finite binary sequences*, J. Assoc. Comput. Mach., 13 (1966), pp. 547–569.

R. V. CHERKASKY, *Algorithm of construction of maximal flow in networks with complexity of* $O(V^2\sqrt{E})$ *operations*, Mathematical Methods of Solution of Economical Problems, 7 (1977), pp. 112–125. (In Russian.)

D. CHERITON AND R. E. TARJAN, *Finding minimum spanning trees*, SIAM J. Comput., 5 (1976), pp. 724–742.

G. CHOQUET, *Etude de certains réseaux de routes*, C. R. Acad. Sci. Paris, 206 (1938), pp. 310–313.

N. CHRISTOFIDES, *Graph Theory: An Algorithmic Approach*, Academic Press, New York, 1975.

Y. J. CHU AND T. H. LIU, *On the shortest arborescence of a directed graph*, Sci. Sinica, 14 (1965), pp. 1396–1400.

A. COBHAM, *The intrinsic computational difficulty of functions*, in Proc. 1964 International Congress for Logic Methodology and Philosophy of Science, Y. Bar-Hillel, ed., North-Holland, Amsterdam, 1964, pp. 24–30.

S. A. COOK, *The complexity of theorem-proving procedures*, Proc. Third ACM Symposium on Theory of Computing, 1971, 151–158.

S. A. COOK AND R. A. RECKHOW, *Time-bounded random access machines*, J. Comput. System Sci., 7 (1973), pp. 354–375.

D. COPPERSMITH AND S. WINOGRAD, *On the asymptotic complexity of matrix multiplication*, SIAM J. Comput., 11 (1982), 472–492.

C. A. CRANE, *Linear lists and priority queues as balanced binary trees*, Tech. Rep. STAN-CS-72-259, Computer Science Dept., Stanford Univ., Stanford, CA, 1972.

G. B. DANTZIG, *Linear Programming and Extensions*, Princeton Univ. Press, Princeton, NJ, 1963.

———, *All shortest routes in a graph*, in Theory of Graphs, International Symposium, Gordon and Breach, New York, 1967, pp. 91–92.

———, *Comments on Khachian's algorithm for linear programming*, Technical Report SOR 79-22, Dept. Operations Research, Stanford Univ., Stanford, CA, 1979.

M. DAVIS, Y. MATIJASEVIC AND J. ROBINSON, *Hilbert's tenth problem. Diophantine equations: positive aspects of a negative solution*, in Mathematical Developments Arising from Hilbert Problems, American Mathematical Society, Providence, RI, 1976, pp. 323–378.

R. B. DIAL, *Algorithm* 360: *Shortest path forest with topological ordering*, Comm. ACM, 12 (1969), pp. 632–633.

R. DIAL, F. GLOVER, D. KARNEY AND D. KLINGMAN, *A computational analysis of alternative algorithms for finding shortest path trees*, Networks, 9 (1979), pp. 215–248.

E. W. DIJKSTRA, *A note on two problems in connexion with graphs*, Numer. Math., 1 (1959), pp. 269–271.

———, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

E. A. DINIC, *Algorithm for solution of a problem of maximum flow in a network with power estimation*, Soviet Math. Dokl., 11 (1970), pp. 1277–1280.

S. E. DREYFUS, *An appraisal of some shortest-path algorithms*, Oper. Res., 17 (1969), pp. 395–412.

J. EDMONDS, *Paths, trees, and flowers*, Canad. J. Math., 17 (1965), pp. 449–467.

———, *Matching and a polyhedron with* 0-1 *vertices*, J. Res. Nat. Bur. Standards Sect. B, 69 (1965), pp. 125–130.

———, *Optimum branchings*, J. Res. Nat. Bur. Standards Sect. B, 71 (1967), pp. 233–240.

J. EDMONDS AND D. R. FULKERSON, *Bottleneck extrema*, Memorandum RM-5375-PR, RAND Corp., Santa Monica, CA, 1968.

J. EDMONDS AND R. M. KARP, *Theoretical improvements in algorithmic efficiency for network flow problems*, J. Assoc. Comput. Mach., 19 (1972), pp. 248–264.

J. EGERVÁRY, *Matrixok kombinatórius tulajdonságairól*, Mat. Fiz. Lapok, 38 (1931), pp. 16–28. (In Hungarian.)

P. ELIAS, A. FEINSTEIN AND C. E. SHANNON, *Note on maximum flow through a network*, IRE Trans. Inform. Theory, IT-2 (1956), pp. 117–119.

S. EVEN, *Graph Algorithms*, Computer Science Press, Potomac, MD, 1979.

S. EVEN AND O. KARIV, *An $O(n^{2.5})$ algorithm for maximum matching in general graphs*, in Proc. 16th Annual IEEE Symposium on Foundations of Computer Science, 1975, pp. 100–112.

S. EVEN AND R. E. TARJAN, *Network flow and testing graph connectivity*, SIAM J. Comput., 4 (1975), pp. 507–518.

M. J. FISCHER, *Efficiency of equivalence algorithms*, in Complexity of Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 153–168.

M. J. FISCHER AND M. O. RABIN, *Super-exponential complexity of Presburger arithmetic*, in Complexity of Computation, R. M. Karp, ed., American Mathematical Society, Providence, RI, 1974, pp. 27–41.

R. W. FLOYD, *Algorithm* 96: *Ancestor*, Comm. ACM, 5 (1962), pp. 344–345.

———, *Algorithm* 97: *Shortest path*, Comm. ACM, 5 (1962), p. 345.

———, *Algorithm* 245: *Treesort* 3, Comm. ACM, 7 (1964), p. 701.

L. R. FORD, JR., *Network flow theory*, Paper P-923, RAND Corp., Santa Monica, CA, 1956.

L. R. FORD, JR. AND D. R. FULKERSON, *Maximal flow through a network*, Canad. J. Math., 8 (1956), pp. 399–404.

———, *Flows in Networks*, Princeton Univ. Press, Princeton, NJ, 1962.

G. E. FORSYTHE AND C. B. MOLER, *Computer Solution of Linear Algebraic Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1967.

M. L. FREDMAN, *New bounds on the complexity of the shortest path problem*, SIAM J. Comput., 5 (1976), pp. 83–89.

H. N. GABOW, *Implementation of algorithms for maximum matching on nonbipartite graphs*, Ph.D. thesis, Dept Electrical Engineering, Stanford Univ., Stanford, CA, 1973.

———, *An efficient implementation of Edmonds' algorithm for maximum matching on graphs*, J. Assoc. Comput. Mach., 23 (1976), pp. 221–234.

H. N. GABOW AND R. E. TARJAN, *A linear-time algorithm for a special case of disjoint set union*, Proc. Fifteenth Annual ACM Symposium on Theory of Computing, 1983, 246–251.

———, *Efficient algorithms for a family of matroid intersection problems*, J. Algorithms, to appear.

Z. GALIL, *An $O(V^{5/3}E^{2/3})$ algorithm for the maximal flow problem*, Acta Inform., 14 (1980), pp. 221–242.

Z. GALIL, S. MICALI AND H. GABOW, *Maximal weighted matching on general graphs*, in Proc. 23rd Annual IEEE Symposium on Foundations of Computer Science, 1982, pp. 255–261.

Z. GALIL AND A. NAAMAD, *An $O(EV\log^2 V)$ algorithm for the maximal flow problem*, J. Comput. System Sci., 21 (1980), pp. 203–217.

B. A. GALLER AND M. J. FISCHER, *An improved equivalence algorithm*, Comm. ACM, 7 (1964), pp. 301–303.

M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, 1979.

J. GILSINN AND C. WITZGALL, *A performance comparison of labeling algorithms for calculating shortest path trees*, National Bureau of Standards Technical Note 772, U.S. Dept. of Commerce, Washington, DC, 1973.

K. GÖDEL, *Über formal unentscheidbare Sätze der Principia Mathematica und verwandten Systeme* I, Monatsch. Math. und Phys., 38 (1931), pp. 173–198.

R. L. GRAHAM AND P. HELL, *On the history of the minimum spanning tree problem*, Ann. History of Computing, to appear.

L. J. GUIBAS AND R. SEDGEWICK, *A dichromatic framework for balanced trees*, in Proc. 19th Annual IEEE Symposium on Foundations of Computer Science, 1978, pp. 8–21.

F. HARARY, *Graph Theory*, Addison-Wesley, Reading, MA, 1969.

F. HARARY, R. Z. NORMAN AND D. CARTWRIGHT, *Structural Models: An Introduction to the Theory of Directed Graphs*, John Wiley, New York, 1965.

J. E. HOPCROFT AND R. M. KARP, *An $n^{5/2}$ algorithm for maximum matching in bipartite graphs*, SIAM J. Comput., 2 (1973), pp. 225–231.

J. E. HOPCROFT AND J. D. ULLMAN, *Set-merging algorithms*, SIAM J. Comput., 2 (1973), pp. 294–303.

T. C. HU, *The maximum capacity route problem*, Oper. Res., 9 (1961), pp. 898–900.

S. HUDDLESTON AND K. MEHLHORN, *A new data structure for representing sorted lists*, Acta Inform., 17 (1982), pp. 157–184.

———, *Robust balancing in B-trees*, in Lecture Notes in Computer Science, 104, Springer-Verlag, Berlin, (1981), pp. 234–244.

M. IRI, *A new method of solving transportation-network problems*, J. Oper. Res. Soc. Japan, 3 (1960), pp. 27–87.

V. JARNÍK, *O jistém problému minimálním*, Práca Moravské Přírodovědecké Společnosti, 6 (1930), pp. 57–63. (In Czech.)

M. JAZAYERI, W. F. OGDEN AND W. C. ROUNDS, *The intrinsically exponential complexity of the circularity problem for attribute grammars*, Comm. ACM, 18 (1975), pp. 697–706.

W. S. JEWELL, *Optimal flow through networks*, Interim Technical Report 8, Massachusetts Institute of Technology, Cambridge, MA, 1958.

D. B. JOHNSON, *Priority queues with update and finding minimum spanning trees*, Inform. Process. Lett., 4 (1975), pp. 53–57.

———, *Efficient algorithms for shortest paths in sparse networks*, J. Assoc. Comput. Mach., 24 (1977), pp. 1–13.

D. S. JOHNSON, *The* NP*-completeness column: An ongoing guide*, J. Algorithms, 4 (1981), pp. 393–405.

O. KARIV, *An* $O(n^{2.5})$ *algorithm for maximum matching in general graphs*, Ph.D. thesis, Dept. of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel, 1976.

R. M. KARP, *Reducibility among combinatorial problems*, in Complexity of Computer Computations, R. E. Miller, ed., Plenum Press, New York, 1972, pp. 85–103.

———, *On the complexity of combinatorial problems*, Networks, 5 (1975), pp. 45–68.

R. M. KARP AND M. O. RABIN, *Efficient randomized pattern-matching algorithms*, J. Assoc. Comput. Mach., to appear.

A. V. KARZANOV, *Determining the maximal flow in a network by the method of preflows*, Soviet Math. Dokl., 15 (1974), pp. 434–437.

K. KENNEDY AND J. SCHWARTZ, *An introduction to the set theoretical language* SETL, Comput. Math. Appl., 1 (1975), pp. 97–119.

A. KERSHENBAUM, *A note on finding shortest path trees*, Networks, 11 (1981), pp. 399–400.

L. G. KHACHIAN, *A polynomial algorithm for linear programming*, Soviet Math. Dokl., 20 (1979), pp. 191–194.

V. KLEE, *Combinatorial optimization: What is the state of the art?*, Math. Oper. Res., 5 (1980), pp. 1–26.

S. C. KLEENE, *Representation of events in nerve nets and finite automata*, in Automata Studies, C. E. Shannon and J. McCarthy, eds., Princeton Univ. Press, Princeton, NJ, 1956, pp. 3–40.

M. KLEIN, *A primal method for minimal cost flows*, Management Sci., 14 (1967), pp. 205–220.

———, *Theorie der endlichen und unendlichen Graphen*, Chelsea, New York, 1950.

D. E. KNUTH, *The Art of Computer Programming, Vol.* 1: *Fundamental Algorithms*, 2nd ed., Addison-Wesley, Reading, MA, 1973.

———, *The Art of Computer Programming, Vol.* 3: *Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

———, *A generalization of Dijkstra's algorithm*, Inform. Process. Lett., 6 (1977), pp. 1–5.

A. KOLMOGOROV, *Three approaches to the quantitative definition of information*, Problems Inform. Transmission, 1 (1965), pp. 1–7.

D. KÖNIG, *Graphen und Matrizen*, Mat. Fiz. Lapok, 38 (1931), pp. 116–119.

J. B. KRUSKAL, *On the shortest spanning subtree of a graph and the traveling salesman problem*, Proc. Amer. Math. Soc., 7 (1956), pp. 48–50.

H. W. KUHN, *The Hungarian method for the assignment problem*, Naval Res. Logist. Quart., 2 (1955), pp. 83–98.

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

J. LUKASZIEWICZ, K. FLOREK, J. PERKAL, H. STEINHAUS AND S. ZUBRZYCKI, *Sur la liaison et la division des points d'un ensemble fini*, Colloq. Math., 2 (1951), pp. 282–285.

F. MAFFIOLI, *Complexity of optimum undirected tree problems: A survey of recent results*, in Analysis and Design of Algorithms in Combinatorial Optimization, International Center for Mechanical Sciences, Courses and Lectures 266, Springer-Verlag, New York, 1981.

D. MAIER AND S. C. SALVETER, *Hysterical B-trees*, Inform. Process. Lett., 12 (1981), pp. 199–202.

V. M. MALHOTRA, M. P. KUMAR AND S. N. MAHESHWARI, *An $O(|V|^3)$ algorithm for finding maximum flows in networks*, Inform. Process. Lett., 7 (1978), pp. 277–278.

S. MICALI AND V. V. VAZIRANI, *An $O(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matching in general graphs*, in Proc. 21st Annual IEEE Symposium on Foundations of Computer Science, 1980, pp. 17–27.

E. MINIEKA, *Optimization Algorithms for Networks and Graphs*, Marcel Dekker, New York, 1978.

E. F. MOORE, *The shortest path through a maze*, in Proc. International Symposium on the Theory of Switching, Part II, April 2–5, 1957, The Annals of the Computation Laboratory of Harvard University, 30, Harvard Univ. Press, Cambridge, MA, 1959.

E. NAGEL AND J. R. NEWMAN, *Gödel's Proof*, New York Univ. Press, New York, 1958.

T. A. J. NICHOLSON, *Finding the shortest route between two points in a network*, Comput. J., 9 (1966), pp. 275–280.

J. NIEVERGELT AND E. M. REINGOLD, *Binary search trees of bounded balance*, SIAM J. Comput., 2 (1973), pp. 33–43.

R. Z. NORMAN AND M. O. RABIN, *An algorithm for a minimum cover of a graph*, Proc. Amer. Math. Soc., 10 (1959), pp. 315–319.

H. OLIVIÉ, *A new class of balanced search trees: Half-balanced binary search trees*, Tech. Rep. 80-02, Interstedelijke Industriele Hogeschool Antwerpen-Mechelen, Antwerp, 1980.

———, *On $\alpha$-balanced binary search trees*, in Lecture Notes in Computer Science 104, Springer-Verlag, Berlin, 1981, pp. 98–108.

C. H. PAPADIMITRIOU AND K. STEIGLITZ, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

U. PAPE, *Implementation and efficiency of Moore algorithms for the shortest route problem*, Math. Programming, 7 (1974), pp. 212–222.

W. J. PAUL, J. I. SEIFERAS AND J. SIMON, *An information-theoretic approach to time bounds for on-line computation*, J. Comput. System Sci., 23 (1981), pp. 108–126.

J.-C. PICARD AND M. QUEYRANNE, *Selected applications of maximum flows and minimum cuts in networks*, Rapport Technique EP-79-R-35, Département de Génie Industriel, Ecole Polytechnique de Montréal, Montreal, 1979.

I. POHL, *Bi-directional search*, in Machine Intelligence, Vol. 6, B. Meltzer and D. Michie, eds., Edinburgh Univ. Press, Edinburgh, 1971, pp. 124–140.

R. C. PRIM, *Shortest connection networks and some generalizations*, Bell System Tech. J., 36 (1957), pp. 1389–1401.

M. QUEYRANNE, *Theoretical efficiency of the algorithm 'capacity' for the maximum flow problem*, Math. Oper. Res., 5 (1980), pp. 258–266.

M. O. RABIN, *Probabilistic algorithms*, in Algorithms and Complexity: New Directions and Recent Results, J. F. Traub, ed., Academic Press, New York, 1976, pp. 21–39.

E. M. REINGOLD, J. NIEVERGELT AND N. DEO, *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, Englewood Cliffs, NJ, 1977.

R. L. RIVEST AND J. VUILLEMIN, *On recognizing graph properties from adjacency matrices*, Theoret. Comput. Sci., 3 (1976), pp. 371–384.

A. SCHÖNHAGE, *Storage modification machines*, SIAM J. Comput., 9 (1980), pp. 490–508.

A. SCHÖNHAGE, M. PATERSON AND N. PIPPENGER, *Finding the median*, J. Comput. System Sci., 13 (1975), pp. 184–199.

Y. SHILOACH, *An $O(n \cdot I \log^2 I)$ maximum-flow algorithm*, Tech. Rep. STAN-CS-78-802, Computer Science Dept., Stanford Univ., Stanford, CA, 1978.

Y. SHILOACH AND U. VISHKIN, *An $O(n^2 \log n)$ parallel max-flow algorithm*, J. Algorithms, 3 (1982), pp. 128–146.

D. D. SLEATOR, *An $O(nm \log n)$ algorithm for maximum network flow*, Tech. Rep. STAN-CS-80-831, Computer Science Dept., Stanford Univ., Stanford, CA, 1980.

D. D. SLEATOR AND R. E. TARJAN, *A data structure for dynamic trees*, J. Comput. System Sci., 24 (1983) to appear; also in Proc. 13th Annual ACM Symposium on Theory of Computing, 1981, pp. 114–122.

———, *Self-adjusting binary trees*, Proc. Fifteenth Annual AM Symposium on Foundations of Computing, 1983, pp. 235–245.

R. SOLOVAY AND V. STRASSEN, *A fast Monte-Carlo test for primality*, SIAM J. Comput., 6 (1977), pp. 84–85.

P. M. SPIRA AND A. PAN, *On finding and updating spanning trees and shortest paths*, SIAM J. Comput., 4 (1975), pp. 375–380.

T. A. STANDISH, *Data Structure Techniques*, Addison-Wesley, Reading, MA, 1980.

M. N. S. SWAMY AND K. THULASIRAMAN, *Graphs, Networks, and Algorithms*, John Wiley, New York, 1981.

R. E. TARJAN, *Depth-first search and linear graph algorithms*, SIAM J. Comput., 1 (1972), pp. 146–160.

———, *Finding dominators in directed graphs*, SIAM J. Comput., 3 (1974), pp. 62–89.

———, *Efficiency of a good but not linear set union algorithm*, J. Assoc. Comput. Mach., 22 (1975), pp. 215–225.

———, *Finding optimum branchings*, Networks, 7 (1977), pp. 25–35.

———, *Complexity of combinatorial algorithms*, SIAM Rev., 20 (1978), pp. 457–491.

———, *A class of algorithms which require nonlinear time to maintain disjoint sets*, J. Comput. System Sci., 18 (1979), pp. 110–127.

———, *Applications of path compression on balanced trees*, J. Assoc. Comput. Mach., 26 (1979), pp. 690–715.

———, *Recent developments in the complexity of combinatorial algorithms*, in Proc. 5th IBM Symposium on Mathematical Foundations of Computer Science, IBM Japan, Tokyo, 1980, pp. 1–28.

———, *A unified approach to path problems*, J. Assoc. Comput. Mach., 28 (1981), pp. 577–593.

———, *Minimum spanning trees*, Technical Memorandum, Bell Laboratories, Murray Hill, NJ, 1981.

———, *Shortest paths*, Technical Memorandum, Bell Laboratories, Murray Hill, NJ, 1981.

———, *Sensitivity analysis of minimum spanning trees and shortest path trees*, Inform. Process. Lett., 14 (1982), pp. 30–33.

———, *A simple version of Karzanov's blocking flow algorithm*, to appear.

———, *Updating a balanced search tree in $O(1)$ rotations*, Inform. Proc. Letters, to appear.

R. E. TARJAN AND J. VAN LEEUWEN, *Worst-case analysis of set union algorithms*, J. Assoc. Comput. Mach., to appear.

A. M. TURING, *On computable numbers, with an application to the Entscheidungs problem*, Proc. London Math. Soc., 2–42 (1936), pp. 230–265. Correction, ibid., 2–43, pp. 544–546.

J. VUILLEMIN, *A data structure for manipulating priority queues*, Comm. ACM, 21 (1978), pp. 309–314.

R. A. WAGNER, *A shortest path algorithm for edge-sparse graphs*, J. Assoc. Comput. Mach., 23 (1976), pp. 50–57.

S. WARSHALL, *A theorem on Boolean matrices*, J. Assoc. Comput. Mach., 9 (1962), pp. 11–12.

M. N. WEGMAN AND J. L. CARTER, *New hash functions and their use in authentication and set equality*, J. Comput. System Sci., 22 (1981), pp. 265–279.

J. W. J. WILLIAMS, *Algorithm* 232: *Heapsort*, Comm. ACM, 7 (1964), pp. 347–348.

N. WIRTH, *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

C. WITZGALL AND C. T. ZAHN, JR., *Modification of Edmonds' maximum matching algorithm*, J. Res. Nat. Bur. Standards Sect. B, 69 (1965), pp. 91–98.

A. YAO, *An $O(|E|\log\log|V|)$ algorithm for finding minimum spanning trees*, Inform. Process. Lett., 4 (1975), pp. 21–23.

N. ZADEH, *More pathological examples for network flow problems*, Math. Programming, 5 (1973), pp. 217–224.

———, *A bad network problem for the simplex method and other minimum cost flow algorithms*, Math. Programming, 5 (1973), pp. 255–266.

(continued from inside front cover)

JERROLD E. MARSDEN, *Lectures on Geometric Methods in Mathematical Physics*
BRADLEY EFRON, *The Jackknife, the Bootstrap, and Other Resampling Plans*
M. WOODROOFE, *Nonlinear Renewal Theory in Sequential Analysis*
D. H. SATTINGER, *Branching in the Presence of Symmetry*
R. TEMAM, *Navier-Stokes Equations and Nonlinear Functional Analysis*
MIKLÓS CSÖRGO, *Quantile Processes with Statistical Applications*
J. D. BUCKMASTER AND G. S. S. LUDFORD, *Lectures on Mathematical Combustion*
R. E. TARJAN, *Data Structures and Network Algorithms*
PAUL WALTMAN, *Competition Models in Population Biology*
S. R. S. VARADHAN, *Large Deviations and Applications*
KIYOSI ITÔ, *Foundations of Stochastic Differential Equations in Infinite Dimensional Spaces*
ALAN C. NEWELL, *Solitons in Mathematics and Physics*
PRANAB KUMAR SEN, *Theory and Applications of Sequential Nonparametrics*
LÁSZLÓ LOVÁSZ, *An Algorithmic Theory of Numbers, Graphs and Convexity*
E. W. CHENEY, *Multivariate Approximation Theory: Selected Topics*
JOEL SPENCER, *Ten Lectures on the Probabilistic Method*
PAUL C. FIFE, *Dynamics of Internal Layers and Diffusive Interfaces*
CHARLES K. CHUI, *Multivariate Splines*
HERBERT S. WILF, *Combinatorial Algorithms: An Update*
HENRY C. TUCKWELL, *Stochastic Processes in the Neurosciences*
FRANK H. CLARKE, *Methods of Dynamic and Nonsmooth Optimization*
ROBERT B. GARDNER, *The Method of Equivalence and Its Applications*
GRACE WAHBA, *Spline Models for Observational Data*
RICHARD S. VARGA, *Scientific Computation on Mathematical Problems and Conjectures*
INGRID DAUBECHIES, *Ten Lectures on Wavelets*
STEPHEN F. MCCORMICK, *Multilevel Projection Methods for Partial Differential Equations*
HARALD NIEDERREITER, *Random Number Generation and Quasi-Monte Carlo Methods*
JOEL SPENCER, *Ten Lectures on the Probabilistic Method, Second Edition*
CHARLES A. MICCHELLI, *Mathematical Aspects of Geometric Modeling*
ROGER TEMAM, *Navier–Stokes Equations and Nonlinear Functional Analysis, Second Edition*
GLENN SHAFER, *Probabilistic Expert Systems*
PETER J. HUBER, *Robust Statistical Procedures, Second Edition*
J. MICHAEL STEELE, *Probability Theory and Combinatorial Optimization*
WERNER C. RHEINBOLDT, *Methods for Solving Systems of Nonlinear Equations, Second Edition*
J. M. CUSHING, *An Introduction to Structured Population Dynamics*
TAI-PING LIU, *Hyperbolic and Viscous Conservation Laws*
MICHAEL RENARDY, *Mathematical Analysis of Viscoelastic Flows*
GÉRARD CORNUÉJOLS, *Combinatorial Optimization: Packing and Covering*
IRENA LASIECKA, *Mathematical Control Theory of Coupled PDEs*