SIAM REVIEW
Vol. 20, No. 3, July 1978

0036-1445/78/2003-0457 $01.50/0

COMPLEXITY OF COMBINATORIAL ALGORITHMS*

ROBERT ENDRE TARJAN†

Abstract. This paper examines recent work on the complexity of combinatorial algorithms, highlighting the aims of the work, the mathematical tools used, and the important results. Included are sections discussing ways to measure the complexity of an algorithm, methods for proving that certain problems are very hard to solve, tools useful in the design of good algorithms, and recent improvements in algorithms for solving ten representative problems. The final sect on suggests some directions for future research.

1. Introduction. In recent years there has been an explosive growth in research dealing with the development and complexity analysis of combinatorial algorithms. While much of this research is theoretical in nature, many of the newly discovered algorithms are very practical. These algorithms and the data manipulation techniques they use are valuable in both combinatorial and numeric computing. Some problems which at first glance are entirely numeric in character require for their efficient solution not only the proper numeric techniques but also the proper choice of data structures and of data manipulation methods. An example of such a problem is the solution of a system of linear equations when the coefficient matrix contains mostly zeros (Tewarson (1973)).

In this paper I shall survey some of the recent results on complexity of combinatorial algorithms, examine some of the ideas behind them, and suggest possible directions for future research. Section 2 of the paper discusses ways to measure the complexity of algorithms. Though several different measures are useful in different circumstances, I shall concentrate upon one measure, the worst-case running time of the algorithm as a function of the input size. Section 3 discusses techniques for proving that certain combinatorial problems are very hard to solve. The results in this area are a natural extension, perhaps more relevant for real-world computing, of the incompleteness and undecidability results of Gödel, Turing and others. Section 4 presents a small collection of general techniques which are useful in the construction of efficient combinatorial algorithms. Section 5 discusses efficient algorithms for solving ten representative problems. These problems illustrate the importance of the methods in § 4, and they include some, but certainly not all, of the combinatorial problems for which good algorithms are known. Section 6 suggests some unsolved problems and directions for future research. The Appendix contains a list of terminology for those unfamiliar with graph theory.

2. Machine models and complexity measures. In the early years of computing (before computer science was recognizable as an academic discipline), an individual confronted with a computational problem was likely to proceed in the following way. He or she[1] would ponder the problem for a while, formulate an algorithm for its solution, and write a computer program which would hopefully implement his algorithm. To test the algorithm's correctness, he would run the program on several

* Received by the editors April 19, 1977. Presented by invitation at the Special Symposium Honoring. The 30th Anniversary of the Founding of the Office of Naval Research held at the 1976 Fall Meeting of the Society for Industrial and Applied Mathematics, Georgia, Institute of Technology, Atlanta, Georgia, October 18–20, 1976.

† Computer Science Department, Stanford University, Stanford, California 94305. This work was supported in part by the National Science Foundation under Grant MCS75-2287 and in part by the Office of Naval Research under Contract N00014-76-C-0688.

[1] Henceforth I shall use "he" to denote any individual, male or female.

sets of data, "debugging" the program until it produced correct output for each set of sample input. To test the algorithm's efficiency, he would measure the time and storage space needed by his program to process the sample data, fit these measurements to curves (by eye, by least-squares fit, or by some other method), and claim that these curves measure the efficiency of the algorithm.

The drawbacks of this empirical approach are obvious. The development of very large programs, such as compilers and operating systems, requires a much more systematic method of checking correctness. This need has led computer scientists to devise methods for *proving* the correctness (and other properties) of programs (Floyd (1967), Manna (1969), Hoare (1969)). These methods use mathematical induction to establish that certain invariant relations hold whenever certain points in the program are reached. Computer scientists have also developed methods (such as "structured programming") for constructing easy-to-understand and easy-to-verify programs (Dahl, Dijkstra, and Hoare (1972)), and have formulated new programming languages to make these methods easy to apply (Wirth (1971)). The thrust of this research is to demonstrate that devising an algorithm and devising a proof of its correctness are inseparable parts of the same process. Perhaps the foremost advocate of this point of view is Dijkstra (Dahl, Dijkstra and Hoare (1972); Dijkstra (1976)).

Measuring efficiency by means of empirical tests has the same deficiency as checking correctness empirically; there is no guarantee that the result is reproducible on new sets of data. If an informed choice is to be made between two algorithms for solving the same problem, some more systematic information about the algorithms' complexity is needed. To be most useful, this information should be machine-independent; good algorithms tend to remain good even if they are expressed in different programming languages or run on different machines. Furthermore the measure should be both realistic and susceptible to theoretical study.

Complexity measures are of two kinds: those which are *static* (independent of the size and characteristics of the input data) and those which are *dynamic* (dependent upon the input data). A typical static measure is *program length*. Program length in some sense measures the *simplicity* and *elegance* of an algorithm (an algorithm with a short program and short correctness proof is simple; an algorithm with a short program and long correctness proof is elegant). This measure is most appropriate if programming time is important or if the program is to be run infrequently.

Dynamic complexity measures provide information about the resource requirements of the algorithm as a function of the characteristics of the input data. Typical dynamic measures are *running time* and *storage space*. These measures are appropriate if the program is to be run often. Running time is usually the most important factor restricting the size of problems which can be solved by computer; most of the problems to be examined in § 5 require only linear space for their solution. However, for problems with linear-time algorithms, storage space may be the limiting factor. Storage space has been used as a measure in proofs of the computational intractability of certain problems (see § 2), but most efficiency studies emphasize running time.

Dynamic measures require that we specify the input data. One possibility is to assume that the data for a given problem size is the worst possible. A *worst-case* measure of running time or storage space as a function of problem size provides a performance guarantee; the program will always require no more time or space than that specified by the bound. A worst-case measure is in this sense not unlike a proof of program correctness.

For some algorithms a worst case bound may be overly pessimistic; for instance, the simplex method of linear programming (Dantzig (1963)), which has an exponential worst-case time bound (Klee and Minty (1972)), seems to run much faster than

exponential on real-world problems (Dantzig (1963)). In such cases an "average" case or "representative" case may give a more realistic bound. For certain problem domains, such as sorting and searching (Knuth (1973)), average case analysis is almost always more realistic than worst-case analysis, and in these areas much average-case analysis has been done. However, average-case analysis has its drawbacks. It may be very hard to choose a good probability measure. For instance, assuming that different parts of the input data are independently distributed may make the analysis easier but may be an unrealistic assumption; furthermore even a relatively simple algorithm may rapidly destroy the independence. With average-case analysis one additionally runs the risk of being surprised by a very rare but very bad set of input data.

Any concrete complexity measure must be based on a computer model. One possible choice is the *random access machine* (Cook and Reckhow (1973)), which is an abstraction of a general-purpose digital computer. The *memory* of such a machine consists of an array of storage cells, each able to hold an integer. The storage cells are numbered consecutively from one; the number of a storage cell is its *address*. The machine also has a fixed finite set of *registers*, each able to hold an integer. (For problems involving real numbers, we allow storage cells and registers to hold real numbers.) In one step, the machine can transfer the contents of a register to a storage cell whose address is in a register, or transfer to a register the contents of a storage cell whose address is in a register, or perform an arithmetic operation on the contents of two registers, or compare the contents of two registers. A *program* of fixed finite length specifies the sequence of operations to be carried out. The initial configuration of memory represents the input data, and the final configuration of memory represents the output. The details of this machine model are unimportant in that reasonable variations do not affect running time or storage space by more than a constant factor.

A random access machine is sequential; it carries out one step at a time. Much work has been done on the computational complexity of parallel algorithms, but I shall not discuss this work here.

The random-access machine model provides a useful tool for realistically measuring the efficiency of particular algorithms, but it has serious drawbacks for lower bound studies. Since a single storage cell can hold an arbitrarily large integer, it is possible on a random-access machine to carry out computations in parallel by encoding several small numbers into one large one. One can avoid this problem by assuming that the time required for an integer operation is proportional to the length of its binary representation (Aho, Hopcroft and Ullman (1974)), or by requiring that all integers be bounded in absolute value by some constant times the size of the input data.

Random-access machines are extremely powerful; in particular, they can perform arithmetic on addresses. This ability is useful for representing multidimensional arrays (Knuth (1968)), performing radix sorts (Knuth (1973)), storing hash tables (Knuth (1973)), and the like. However, determining the theoretical limits of this capability seems to be a hard problem. Kolmogorov (1953), Kolmogorov and Uspenskii (1963), Knuth (1968), Schönhage (1973), and Tarjan (1977) have proposed machine models in which access to memory is by explicit reference only, and no address arithmetic is possible. I shall call such a machine a *linked memory machine*. These machines accurately model the capabilities of list-processing languages such as LISP and the list-processing features of general-purpose languages such as ALGOL-W and PL/1, and they appear to be more amenable to analysis than random-access machines.

Another very simple machine model, the *Turing machine* (Turing (1936–7)), has been used in many theoretical studies. A Turing machine has a memory consisting of a tape. The tape is divided into cells, each capable of holding one of a finite number of

symbols. The machine possesses a finite internal memory and a read/write head which can scan one tape cell at a time. In one step, the machine can read a tape cell, write a new symbol in the cell (erasing what was there previously), move the read/write head one cell forward or backward on the tape, and change the internal memory state. The decision as to what to do at each step depends only on the current internal memory state and the contents of the tape cell being read; this decision is encoded for each internal state and each tape symbol in a *decision table* which forms the program of the machine.

Turing proposed his machine model in 1936, before electronic digital computers existed: he was attempting to model computation processes in the abstract, without reference to any real computer. Though Turing's model is inadequate for a large part of concrete complexity research, its simplicity and the fact that any random access machine can be simulated on a Turing machine with only a polynomial blow-up in running time makes the Turing machine extremely useful for studying very difficult computational tasks. It is also valuable for studying problems where tapes are the storage device, as for instance in tape sorting (Knuth (1973)).

In lower bound studies the focus is often on some critical operation; one counts in the running time occurrences only of that critical operation. For instance, in sorting and selection problems it is useful to count only comparisons (or general binary decisions), measuring the complexity of a problem by the depth of a *decision tree* for it (Aho, Hopcroft and Ullman (1974)). In arithmetic and algebraic problems, it is useful to count only arithmetic operations and to assume that no decisions are made; i.e., that the computations performed are independent of the input data (for a particular problem size). In this case one measures the complexity of a problem by the length of a *straight-line program* (Aho, Hopcroft and Ullman (1974)). In other situations memory accesses may be the critical operations.

In this paper I shall use worst-case running time on a random-access machine as a measure of algorithmic complexity. This measure is useful and realistic for a wide range of combinatorial problems. I shall ignore constant factors in running time, since such constant factors depend upon the exact model of computation, they are often hard to compute, and they tend, at least for large-sized problems, to be washed out by asymptotic growth rates. To indicate functional relationships, I shall use the following notation. If f and g are functions of n, "$f(n)$ is $O(g(n))$" means $f(n) \leqq cg(n)$ for all n, where c is a suitable positive constant, and "$f(n)$ is $\Omega(g(n))$" means $f(n) \geqq cg(n)$ for all n, where c is a suitable positive constant.

3. Complexity of intractable problems. Inspired by Hilbert (1926) and other formalists, mathematicians of the early twentieth century hoped to find a formal system which would be adequate for expressing and verifying all mathematical truths. These hopes were dashed by Gödel (1931), who in his famous incompleteness theorem demonstrated that no method of proof could be both subject to mechanical verification and powerful enough to prove all theorems of elementary arithmetic. Their interest in the foundations of mathematics prompted logicians to confront the question, "What is mechanical verification?" or equivalently, "What is an algorithm?". Church (1936), Kleene (1936), Post (1936), Turing (1936–7), and others provided formal definitions of an algorithm. These definitions are superficially different but provably equivalent, in the sense that if a problem is solvable according to one definition of an algorithm, then it is solvable according to all the other definitions. This robustness of the notion of an algorithm is usually stated as *Church's thesis*: any algorithm (in the informal sense) can be expressed as a Turing machine, and any Turing machine expresses an algorithm.

Once a formal definition of an algorithm existed, it was possible for mathematicians to study the power of computation. Turing proved that *no* algorithm existed for determining whether a given Turing machine with a given input will ever halt. Other researchers discovered a number of such *undecidable problems* (Jones (1974)), which correspond in computer science to the incompleteness results of Gödel and others in logic. Perhaps the capstone to this research on computability is Matijasevic's 1970 proof, building on earlier work by Martin Davis and Julia Robinson, that Hilbert's tenth problem is undecidable (Davis, Matijasevic, and Robinson (1976)). Hilbert's tenth problem is to determine whether a given polynomial equation has a solution in integers.

Two proof techniques, *diagonalization* and *simulation*, pervade computability theory. Diagonalization is based on ancient self-reference paradoxes; Cantor (1874) used it to prove that there are more real numbers than integers and Gödel used it to prove his incompleteness result. One can use it in the following way to devise an undecidable problem. Suppose we are interested in yes-no questions about the integers, such as "Is n even?" or "Is n prime?" Suppose we have a listing $A_1, A_2, A_3, \cdots$ of all algorithms for answering such questions (for any of the standard definitions of an algorithm it is easy to produce such a listing). Consider the set S of integers such that n is an element of S if and only if algorithm A_n answers "no" (or does not answer at all) on input n. Then the question "Is n an element of S?" is undecidable, since each algorithm in the list $A_1, A_2, A_3, \cdots$ produces a wrong answer on at least one input (A_n is wrong on input n) and by Church's thesis this list contains all possible algorithms. Turing used the same idea to show the undecidability of the halting problem for Turing machines.

Simulation is a method for turning one problem or problem-solving method into another. Once we have one undecidable problem P_1, we can prove another problem P_2 undecidable by showing that if P_2 has an algorithm then this algorithm can be used to solve P_1. To accomplish this we provide an algorithm which converts an instance of problem P_1 into one or more instances of problem P_2, thus *reducing* P_1 to P_2 (or *transforming* P_1 into P_2). Similarly, to show that two definitions of an algorithm are equivalent, we show how to *simulate* an algorithm according to one definition by an algorithm according to the other definition.

The development of general-purpose digital computers made possible the implementation and execution of complicated algorithms, and the theory of computability became a matter of more than mathematical interest. However, this theory ignores questions of resource use, which limits its power to identify what is possible in practice. Many problems which obviously have algorithms seem to have no *good* algorithms. For instance, consider the *maximum stable set problem*: given a graph, find in it a maximum number of vertices, no two adjacent. Since a graph with n vertices has only 2^n subsets of vertices, an exponential-time algorithm for this problem exists. However no one has yet discovered a substantially faster algorithm for this problem.

Tables 3.1 and 3.2 illustrate the importance of this phenomenon. Table 3.1 estimates running times of algorithms with various time bounds. The table shows that constant factors become less and less important as problem size increases; on large problems the asymptotic growth rate of the time bound dominates the constant factor. The table also shows that running time grows explosively if the time bound is exponential. Table 3.2 estimates the maximum size of problems solvable in a given amount of time. Increasing the amount of time (or the speed of the machine) by a large factor does *not* substantially increase the size of problems solvable unless the time bound grows more slowly than exponential.

TABLE 3.1
Running time estimates. (One step = one microsecond; logarithms are base two.)

Complexity \ Size	20	50	100	200	500	1000
$1000n$	.02 sec	.05 sec	.1 sec	.2 sec	.5 sec	1 sec
$1000n \log n$	.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
$n^{\log n}$	.4 sec	1.1 hr	220 days	125 cent	5.10^8 cent	
$2^{n/3}$	.0001 sec	.1 sec	2.7 hr	3.10^4 cent		
2^n	1 sec	35 yr	3.10^4 cent			
3^n	58 min	2.10^9 cent				

TABLE 3.2
Maximum size of a solvable problem. (A factor of ten increase in machine speed corresponds to a factor of ten increase in time.)

Complexity \ Time	1 sec	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}
$1000n \log n$	1.4×10^2	7.7×10^3	5.2×10^5	3.9×10^7	3.1×10^9	2.6×10^{11}
$100n^2$	10^2	10^3	10^4	10^5	10^6	10^7
$10n^3$	46	2.1×10^2	10^3	4.6×10^3	2.1×10^4	10^5
$n^{\log n}$	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

Tables 3.1 and 3.2 suggest a natural division between *good* algorithms (those with worst-case time bounds polynomial in the size of the input) and *bad* algorithms. Edmonds (1965) was apparently the first to stress this distinction. I shall call a decidable problem *tractable* if it has a polynomial-time algorithm and *intractable* otherwise. The distinction between tractable and intractable problems is independent of the machine model, since any of the commonly used machine models can be simulated by any other with only a polynomial loss in running time. As Tables 3.1 and 3.2 show, it is not feasible to execute exponential-time algorithms on large problems. Many combinatorial problems are easily solvable in exponential time by exhaustively checking cases, but solving such problems in polynomial time seems to require much greater insight. Most known good algorithms have time bounds which are polynomials of small degree ($O(n^3)$ or better). It is a major task of complexity theory to identify which natural problems are tractable and which are intractable.

Hartmanis, Lewis and Stearns took the first steps toward exhibiting natural intractable problems (Hartmanis, Lewis and Stearns (1965); Hartmanis and Stearns (1965)). By diagonalizing over all algorithms with a given space bound $S_1(n)$, they were able to obtain a problem solvable in space $S_2(n)$ but not in space $S_1(n)$, for any space bounds $S_1(n)$ and $S_2(n)$ satisfying $\liminf_{n\to\infty} S_1(n)/S_2(n) = 0$ and a few other technical constraints. They proved a similar but somewhat weaker result for time complexity. These results imply in particular that there are problems solvable in exponential space but not in polynomial space, and problems solvable in exponential time but not in polynomial time.

Unfortunately, the intractable problems produced by diagonalization are not natural ones. Meyer and Stockmeyer (1972) proved the intractability of a natural problem. They showed that the problem of determining whether two regular expressions with squaring denote the same set requires exponential space (and hence exponential time) for its solution. A regular expression is a formula constructed from the symbols Λ, 0, 1, $\cup$, $\cdot$, *, (,) according to the following rules. Each such formula denotes a set of strings of zeros and ones.

3.1. 0 is a regular expression denoting the set $\{0\}$;
1 is a regular expression denoting the set $\{1\}$;
Λ is a regular expression denoting the set whose single element is the empty string.

3.2. If A and B are regular expressions denoting sets $L(A)$ and $L(B)$, respectively, then
$(A \cup B)$ is a regular expression denoting the set $L(A) \cup L(B)$;
$(A \cdot B)$ is a regular expression denoting the set

$$\{xy \mid x \in L(A) \text{ and } y \in L(B)\}.$$

A^* is a regular expression denoting the set consisting of the empty string and all strings formed by concatenating one or more strings in $L(A)$.

Meyer and Stockmeyer added an additional rule:

3.3. If A is a regular expression, then A^2 is a regular expression denoting the same set as $(A \cdot A)$.

To prove that the equivalence problem for two such expressions is intractable, Meyer and Stockmeyer used simulation. They devised a polynomial-time algorithm which, given a Turing machine, an input, and an exponential space bound, would construct a regular expression representing the computation of the Turing machine on the given input. The expression is such that it denotes the same set as $(0 \cup 1)^*$ if and only if the Turing machine does *not* accept the input within the given space bound. It follows that the equivalence problem for regular expressions with squaring is as hard (to within a polynomial time blow-up) as *any* yes-no question answerable in exponential space by a Turing machine. Since the Hartmanis, Lewis, Stearns result implies that *some* problem exists which can be solved in space 2^n but not space $2^n/n$, the equivalence problem for regular expressions must require exponential space.

In the last five years, several more such results have been discovered. Hunt (1973) showed that if set intersection is substituted for squaring the equivalence problem for regular expressions still requires exponential space. Stockmeyer and Meyer (1973) showed that if set subtraction is substituted for squaring the equivalence problem for regular expressions has a nonelementary space bound. Fischer and Rabin (1974) proved that testing the validity of a formula in Presburger arithmetic (the theory of natural numbers with + as the only operation) requires $2^{2^{cn}}$ space, for some positive constant c. Cardoza, Lipton and Meyer (1976) showed that the word problem

for Abelian groups requires exponential space. Jazayeri, Ogden and Rounds (1975) showed that testing the circularity of attribute grammars (a problem arising in programming language semantics) requires exponential time.

The idea in all these proofs is the same; one shows how to efficiently convert any computation with a particular space or time complexity into an instance of the given problem, and one appeals to the Hartmanis, Lewis, Stearns results to assert the existence of an intractable problem with the particular space or time complexity.

Significantly, a number of apparently intractable problems, such as the maximum stable set problem, are not included in the list of known intractable problems. These problems have the following property. If such a problem is phrased as a yes-no question, and the answer is "yes", then there is a polynomial-length proof of the answer. For instance, suppose we rephrase the maximum stable set problem as follows: "Does a given graph G contain a stable set of k vertices?" If the answer is yes, one can prove it by exhibiting the stable set and showing that its vertices are pairwise nonadjacent.

To formalize this notion of polynomial-length proof, we introduce *nondeterministic* machines. A nondeterministic machine may, at various times during its computation, make a *guess* as to what to do next. The machine accepts a given input if there exists some sequence of guesses which causes the machine to eventually answer "yes". We define the time (or space) required by the machine to accept an input as the *minimum* amount of time (or space) used by an accepting computation. The following nondeterministic algorithm solves the maximum stable set problem in polynomial time: First, guess a subset of k vertices. Next, check all pairs of these vertices for adjacency. Accept if no two of the vertices are adjacent. Let $\mathcal{P}$ denote the class of yes-no problems solvable deterministically in polynomial time and let $\mathcal{NP}$ denote the class of yes-no problems solvable nondeterministically in polynomial time. The question we wish to answer is, "Are there natural problems which are in $\mathcal{NP}$ but not in $\mathcal{P}$?"

Cook (1971) showed that $\mathcal{NP}$ contains certain "hardest" problems, called $\mathcal{NP}$-complete problems. A problem P is $\mathcal{NP}$-complete if it satisfies two properties:

3.4. P is in $\mathcal{NP}$.

3.5. If Q is in $\mathcal{NP}$ then Q is reducible to P in polynomial time.

To say that Q is reducible to P in polynomial time means that there is a (deterministic) polynomial-time algorithm which, given an instance of problem Q, will convert it into an instance of problem P, such that the answer to the instance of Q is "yes" if and only if the answer to the instance of P is "yes". If Q is reducible to P in polynomial time and P has a polynomial-time algorithm, then so does Q. Thus if any $\mathcal{NP}$-complete problem has a polynomial-time algorithm, $\mathcal{P} = \mathcal{NP}$.

Cook's main result was to show that the satisfiability problem of propositional calculus is $\mathcal{NP}$-complete. The satisfiability problem is to determine whether a given logical formula is true for at least one assignment of the values "true" and "false" to the variables. It is easy to show that this problem satisfies 3.4. Cook proved 3.5 by giving a polynomial-time algorithm for constructing, from a given nondeterministic Turing machine, a given input, and a given polynomial time bound, a logical formula such that the formula is satisfiable if and only if the Turing machine accepts the input within the time bound.

If one knows a single problem P to be $\mathcal{NP}$-complete, one can prove another problem Q $\mathcal{NP}$-complete by showing that Q is in $\mathcal{NP}$ and that P is reducible in polynomial time to Q; property 3.5 then follows from the transitivity of polynomial-time reducibility. Karp (1972) used this idea to exhibit a number of natural

$\mathcal{NP}$-complete problems. Others continued this work, and the number of known $\mathcal{NP}$-complete problems is now in the hundreds (see for instance Even, Itai and Shamir (1976); Garey, Johnson and Stockmeyer (1976); Garey, Johnson and Tarjan (1976); Karp (1975); Sahni (1974); Sethi (1975); and Ullman (1973b)). In addition to the satisfiability problem and the maximum stable set problem, the following problems are $\mathcal{NP}$-complete.

Subgraph isomorphism (Cook (1971)). Given two graphs G_1 and G_2, is G_1 isomorphic to a subgraph of G_2?

Graph coloring (Karp (1972)). Given a graph G, can its vertices be colored with k colors so that no two adjacent vertices have the same color? This problem is $\mathcal{NP}$-complete even if $k = 3$ and G is planar (Garey, Johnson and Stockmeyer (1976)), whereas it follows from Appel and Haken's proof of the four color conjecture (Appel and Haken (1977)) that there is a polynomial-time algorithm to color any planar graph with four colors.

Hamilton cycle (Karp (1972)). Given a graph G, does it contain a cycle which passes through every vertex exactly once? This problem is a special case of the traveling salesman problem (see § 4). It is $\mathcal{NP}$-complete even if G is planar (Garey, Johnson and Tarjan (1976)).

Subset sum (Karp (1972)). Given a set of numbers $n_1, n_2, \cdots, n_k$ and a sum s, does some subset of the numbers sum to exactly s?

Maximum planar subgraph (Liu and Geldmacher (1976)). Given a graph G, does it contain a planar subgraph with at least k edges?

A major open problem of complexity theory is to determine whether $\mathcal{P} = \mathcal{NP}$. A natural approach to this problem would be to try using diagonalization to exhibit a problem in $\mathcal{NP}$ but not in $\mathcal{P}$. However, recent work by Baker, Gill and Solovay (1975) suggests that diagonalization is impotent for resolving the $\mathcal{P} = \mathcal{NP}$? question. Even without a proof that $\mathcal{P} = \mathcal{NP}$, it is still fruitful to add new natural problems to the list of $\mathcal{NP}$-complete ones; the large amount of time spent by bright people fruitlessly searching for polynomial-time algorithms for $\mathcal{NP}$-complete problems is strong evidence that the $\mathcal{NP}$-complete problems are in fact intractable.

4. Techniques for good algorithms. Although many important combinatorial problems seem to be intractable, many others have good algorithms. A small number of data manipulation techniques form the basis for these algorithms. This section examines these techniques, which are outlined in Table 4.1.

Data structures. Any algorithm (good or bad) requires one or more *data structures* to represent the elements of the problem to be solved and the information computed during the solution process. A data structure is a composite object composed of elements related in specified ways. Associated with the data structure is a set of operations for manipulating its elements. Once a good implementation of a given data structure and its operations is known, one can regard the operations as primitives when implementing any algorithm which uses the data structure. The efficiency of the algorithm will depend to a large extent upon the implementation of the underlying data structure.

There are two data structures upon which all others are based: *arrays* and *linked structures*. An array is a collection of storage cells numbered consecutively. Two operations are associated with an array: given the number of a storage cell, one can either store a value in the storage cell (destroying the current value) or retrieve the current value from the storage cell. The memory of a random access machine and of

TABLE 4.1
Techniques for good algorithms.

1. Data structures (built from arrays and linked structures).
   a. Lists.
   b. Unordered sets.
   c. Ordered sets.
   d. Graphs.
   e. Trees.
2. Recursion.
   a. Dynamic programming.
3. Graph searching.
   a. Depth-first.
   b. Breadth-first.
      i. Shortest-first.
      ii. Lexicographic.
4. Optimization methods.
   a. Greed.
   b. Augmentation.
5. Data updating methods.
   a. Path compression.
   b. Partition refinement.
   c. Linear arrangement.
6. Graph mapping.
   a. Decomposition (by subgraphs).
   b. Shrinking (by graph homomorphism).

most digital computers is an array. One can use arrays to represent vectors, matrices, tensors, and multidimensional arrays (Knuth (1968)).

A linked structure consists of a collection of *records*. Each record is divided into a number of *items*, each with an identifying *name*. The structure of all records is identical. Items are of two kinds, *data items* and *reference items*. Data items contain data. Reference items contain pointers to records. Two operations are possible on a linked structure; given a pointer to a record, one can either store a value into an item in the record or retrieve the current value from an item in the record. Figure 4.1 illustrates a linked structure. Whereas array addresses are integers capable of being manipulated by arithmetic operations, no operations are allowed on linked structure pointers except storage, retrieval, and testing for equality. The memory of a linked memory machine is a linked structure, and most list-processing languages can be regarded as operating on linked structures.

It is easy to implement arrays and linked structures so that storage and retrieval require constant time. Linked structures can be implemented as collections of arrays (see Fig. 4.1); this makes list-processing easy in languages such as FORTRAN which do not possess an explicit list-processing facility. It seems to be impossible to implement an array as a linked structure in such a way that storage and retrieval take constant time, though I know of no proof of this fact.

Using arrays and linked structures, one can implement many different data structures. I shall consider here five classes of data structures: lists, unordered sets, ordered sets, graphs, and trees.

A *list* is a sequence of elements. The first element of a list is its *head*; the last element is its *tail*. Simple operations on a list include *scanning* the list to retrieve its elements in order, *adding* an element as the new head of the list (making the old head

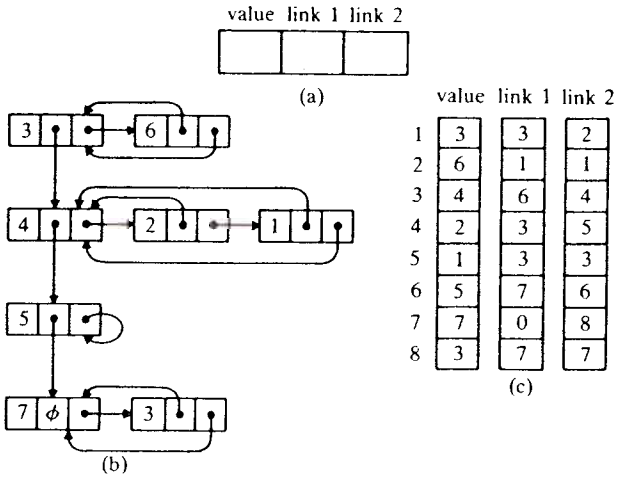


FIG. 4.1. *A linked structure and its representation by arrays.*
(a) *Record format.*
(b) *Linked structure.*
(c) *Representation by three arrays.*

the second element); adding an element as the new tail, *deleting* and retrieving the head of a list, and deleting and retrieving the tail of a list. Lists on which only a few of these operations are possible have special names. A *stack* is a list with addition and deletion allowed only at the head. A *queue* is a list with addition allowed only at the tail and deletion allowed only at the head. A *deque* (double-ended queue) is a list on which addition or deletion is possible at either end. One can implement a deque either as a circular array (addresses are computed modulo the size of the array) or as a singly linked structure (if deletion from the tail is not necessary). See Fig. 4.2. In either case,

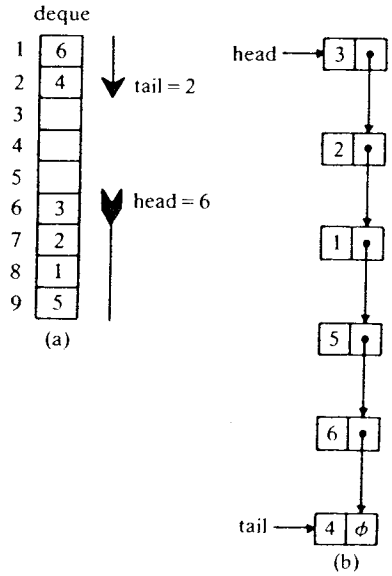


FIG. 4.2. *Representation of a deque containing* 3, 2, 1, 5, 6, 4.
(a) *Array representation.*
(b) *Linked representation.*

all operations except scanning require constant time. The array representation uses no space for storing pointers but requires that an amount of storage equal to the maximum size of the list be permanently allocated to the list.

Other important list operations include *concatenating* two lists (making the head of the second list the element following the tail of the first), *inserting* an element before or after an element whose location in the list is known, and *deleting* an element whose location in the list is known. These operations require a linked structure for their efficient implementation. A singly linked structure is sufficient for concatenation and for insertion after another element. Insertion before another element and deletion require a doubly linked structure. See Fig. 4.3. An alternate way to handle deletion is to provide each element with a flag which is set to "true" if the element is to be deleted. The element is not explicitly deleted until the next scan through the list.

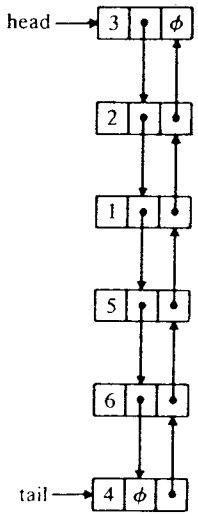


FIG. 4.3. *Representation of list* 3, 2, 1, 5, 6, 4 *by doubly linked structure.*

The list operations hardest to implement are *inserting* an element at the kth position in a list, *retrieving* the element at the kth position in a list, or *deleting* the element at the kth position in a list. It is possible to implement these operations to run in $O(\log n)$ time, where n is the size of the list, by using AVL trees (Knuth (1973)) or 2-3 trees (Aho, Hopcroft and Ullman (1974)), which are rather complicated linked structures. Recently Guibas, McCreight, Plass and Roberts (1977) have found a way to carry out these operations in $O(\log k)$ time.

An *unordered set* is a collection of distinct elements with no imposed relationship. Basic set operations are *adding* an element to a set, deleting an element from a set, and *testing* whether an element is in a set. One way to represent a set is by a singly linked list. Addition requires constant time but testing and deletion require $O(n)$ time, where n is the size of the set. Alternatively, if the elements of the set are values which can be compared and sorted, one can represent the set by an AVL tree or a 2-3 tree in such a way that all three operations require $O(\log n)$ time (Knuth (1973); Aho, Hopcroft and Ullman (1974)).

Another way to represent a set is by a *bit vector* (Aho, Hopcroft and Ullman (1974)), which is an array with one storage cell for each possible element. A storage

cell has two possible values: true, indicating that the set contains the element, and false, indicating that it does not. All three operations require constant time using this representation. Bit vector representation is only feasible if the number of possible elements is small.

If the number of possible elements is large, one can mimic the behavior of a bit vector by using a *hash table* (Knuth (1973)). A hash table consists of a moderately sized array and a *hashing function* which maps each possible element into an array address. If an element is present, the element (or a pointer to it) is stored at (or near) the address specified by the hashing function. Since two or more elements may hash to the same address, some mechanism must be provided for resolving such collisions. Hash tables are used extensively in compilers, and many papers have been written about them (see Knuth (1973), Morris (1968)). With a hash table, addition, deletion, and testing require $O(n)$ time in the worst case but only constant time on the average.

Additional set operations are useful if two or more sets exist. These include the ability to form a set which is the *union*, *intersection*, or *difference* of two sets. For most representations union, intersection, and difference require time proportional to the sum of the sizes of the sets. However, if the universe of elements is small enough so that a bit vector can fit into a few computer words and the computer possesses bit vector operations, then union, intersection, and difference require constant time.

An *ordered set* is a collection of elements, each with an associated numeric value. Two important operations on ordered sets are *sorting* the elements in increasing order and *selecting* the element with kth largest value. A variety of ways exist to sort n elements in $O(n \log n)$ time (Knuth (1973)); if binary comparisons are the only operations used to manipulate the values then $\Omega(n \log n)$ time is required in both the average and the worst case to sort (Knuth (1973)). Selecting the kth largest element requires $O(n)$ time (Blum, Floyd, Pratt, Rivest and Tarjan (1973); Schönhage, Paterson and Pippenger (1976)).

A *priority queue* is an ordered set on which the following operations are allowed: *adding* an element to the queue, *retrieving* the minimum-value element in the queue, and *deleting* an element whose location is known from the queue. By using binomial trees (Vuillemin (1977), Brown (1977)), leftist trees (Knuth (1973)), or 2-3 trees (Aho, Hopcroft and Ullman (1974)) one can implement priority queue operations so that they require $O(\log n)$ time, where n is the size of the queue. These implementations also allow one to combine two queues into a larger queue (destroying the smaller queues) in $O(\log n)$ time.

If the values of the elements in an ordered set are integers of moderate size, then the ordered set operations can be speeded up. Using a k-pass radix sort, one can sort n integers in the range 1 to m^k in $O(km+n)$ time (Knuth (1973)). Peter van Emde Boas has devised a method for implementing priority queues with integer values in the range 1 to n so that the queue operations require $O(\log \log n)$ time (van Emde Boas, Kaas and Zijkstra (1975)).

A *graph* is a set of *vertices* and a set of *edges*, each edge a pair of vertices. One way to represent a graph is by a two-dimensional array A, called an *adjacency matrix*. The value of $A(i, j)$ is one if (i, j) is an edge of the graph; otherwise the value of $A(i, j)$ is zero. An alternate way to represent a graph is by an *adjacency structure*, which is an array of lists, one for each vertex. The list for vertex i contains vertex j if and only if (i, j) is an edge of the graph. See Fig. 4.4.

The adjacency matrix representation saves space if the graph is dense (i.e., most possible edges are present); it also allows one to test the presence of a given edge in constant time. However, Anderaa and Rosenburg conjectured (Rosenburg (1973))

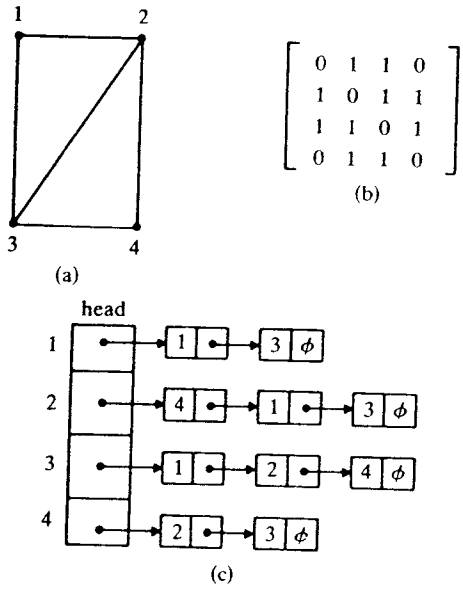


FIG. 4.4. *Representation of a graph.*
(a) *Graph.*
(b) *Adjacency matrix.*
(c) *Adjacency structure.*

and Rivest and Vuillemin (1975) proved that testing any nontrivial monotone[2] graph property requires $\Omega(n^2)$ probes of the adjacency matrix in the worst case, where n is the number of vertices in the graph. By using an adjacency structure, one can search a graph in $O(n+m)$ time, where m is the number of edges in the graph; thus representation by an adjacency structure is preferable for sparse graphs.

A *tree* is a graph without cycles. Since a tree is a graph it can be represented by an adjacency structure. A more compact way to represent a tree is to choose a root for the tree, compute the parent of each vertex with respect to this root, and store this information in an array (Fig. 4.5). This representation is usable as long as the tree is to be explored from leaves to root, which is often the case in problems involving trees.

Recursion. An important and very general algorithmic technique is *recursion*. Recursion is a method of solving a problem by reducing it to one or more subproblems. The subproblems are reduced in the same way. Eventually the subproblems become small enough that they can be solved directly. The solutions to the smaller subproblems are then combined to give solutions to the bigger subproblems, until the solution to the original problem is computed. As a simple example of a recursive algorithm, consider the following definition of the nth Fibonacci number:

$$F(n) := \textbf{if } (n = 1) \textbf{ or } (n = 2) \textbf{ then } 1 \textbf{ else } F(n-1) + F(n-2). \tag{4.1}$$

Using recursion, one can often state algorithms much more simply than would be possible without recursion. Many programming languages, including ALGOL, PL/1, and LISP, allow recursive procedures (procedures which call themselves). In a language without this facility, such as FORTRAN, one can implement a recursive

[2] A graph property is *nontrivial* if for any n the property is true for some graph of n vertices and false for some other graph of n vertices. A graph property is *monotone* if adding edges to a graph does not change the property from true to false.

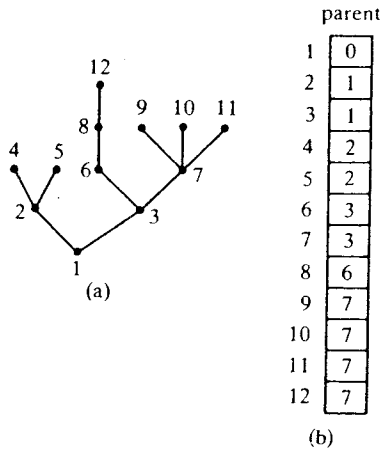


FIG. 4.5. *Representation of a tree.*
(a) *Tree.*
(b) *Parent array for root* 1.

algorithm by using a stack to store the generated subproblems (Aho, Hopcroft and Ullman (1974)).

Dynamic programming. Dynamic programming (Bellman (1957)) can be viewed as a special kind of recursion in which one keeps track of the generated subproblems and never solves the same problem twice. As an example of the work which can be saved in this way, consider the computation of the nth Fibonacci number. A recursive procedure based on (4.1) requires time proportional to the size of $F(n)$ to compute $F(n)$; such a procedure performs $F(n+1-i)$ computations of $F(i)$ for each i in the range from 1 to n. A better way to compute $F(n)$ is to compute each $F(i)$ just once for each value of i. The most efficient way to implement a dynamic programming algorithm is to set up a table of solutions to all subproblems, and to fill in the table from smallest to largest subproblem. Sometimes one can discard the solutions for small subproblems as the computation proceeds and re-use the space for larger subproblems. One can evaluate $F(n)$ in $O(n)$ time with two storage locations by using this idea. (Of course, using a closed-form expression for $F(n)$ results in an even faster computation.)

Dynamic programming has been used with great success on a number of combinatorial problems, including shortest path problems (Floyd (1962)), context-free language parsing (Younger (1967), Earley (1970)), error correction in context-free languages (Aho and Peterson (1972)), and construction of optimum binary search trees (Knuth (1971), Itai (1976)).

Graph searching. Most graph problems require for their solution a systematic method of exploring a graph. A *search* is an examination of the edges of a graph using the following procedure.

Step 1 (initialization): Mark all edges and vertices of the graph *new* (unexplored).

Step 2 (choose a new starting vertex): If no new vertex exists, halt. (The entire graph has been explored.) Otherwise, choose a new vertex and mark it *old* (explored).

Step 3 (explore an edge): If no new edges lead away from old vertices, go to Step 2. (All of the graph reachable from the current start vertex has been explored.) Otherwise, choose a new edge leading away from an old vertex. Mark the edge old. If the other endpoint of the edge is new, mark it old. Repeat Step 3.

Assume for simplicity that all vertices in the graph to be searched are reachable from the first start vertex selected in Step 2. Then the search generates a *spanning tree*. The root of the spanning tree is the start vertex. The edges of the spanning tree are the edges which lead to new vertices when explored in Step 3. The properties of the spanning tree depend upon the criteria used to select the starting vertex in Step 2 and the edges to explore in Step 3. For some simple graph problems, such as finding connected components (Hopcroft and Tarjan (1973c)), any order of exploration is satisfactory. However, for harder graph problems the exploration order is crucial.

In a *depth-first search*, the edge selected in Step 3 is an edge out of the *last* explored vertex with candidate edges. If a depth-first search is performed on an undirected graph, the generated spanning tree has the property that all nontree edges connect vertices related in the tree (Tarjan (1972)). See Fig. 4.6. If such a search is performed on a directed graph and the vertices are numbered from 1 to n as they are marked old, then no nontree edge leads from a vertex to a vertex which is both higher numbered and unrelated in the spanning tree (Tarjan (1972)). See Fig. 4.7. A depth-first search can be implemented as a recursive procedure or with an explicit stack to store the old vertices.

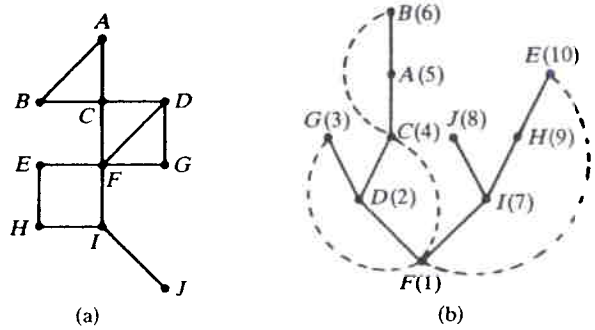


FIG. 4.6. *Depth-first search of an undirected graph.*
(a) *Graph.*
(b) *Spanning tree generated by search. Vertices numbered as explored.*

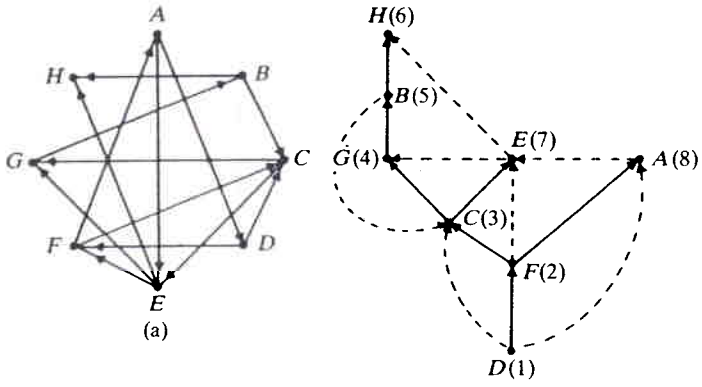


FIG. 4.7. *Depth-first search of a directed graph.*
(a) *Graph.*
(b) *Spanning tree generated by search. Vertices numbered as explored.*

In a *breadth-first search*, the edge selected in Step 3 is an edge out of the *first* explored vertex with candidate edges. Such a search partitions the vertices into *levels* depending upon their distance from the start vertex. In an undirected graph each edge connects vertices in the same level or in two adjacent levels; in a directed graph, no edge leads from a level to a level higher than the next level. See Fig. 4.8. A breadth-first search can be implemented using a queue to store the old vertices.

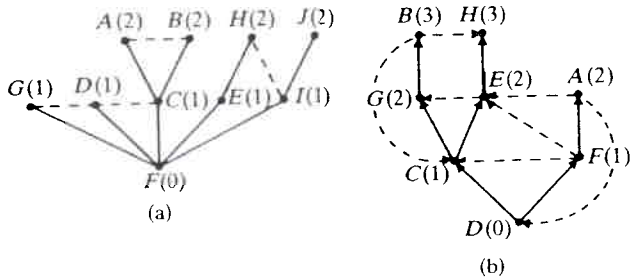


FIG. 4.8. *Breadth-first search. Level indicated in parentheses.*
(a) *Search of graph in Fig.* 4.6.
(b) *Search of graph in Fig.* 4.7.

Both depth-first and breadth-first search, if properly implemented using an adjacency structure to store the graph, require $O(n+m)$ time to explore an n-vertex, m-edge graph. Although these are the most important search methods, several others, including *topological search* (Knuth (1968)), *lexicographic search* (Sethi (1975); Rose, Tarjan, and Lueker (1976)), and *shortest-first search* (Dijkstra (1959), Johnson (1977)), are occasionally useful.

Optimization methods. A large class of problems requires the maximization of a function defined on a graph with weighted edges. It is usually possible to phrase these problems as linear or integer programming problems (Dantzig (1963), Nemhauser and Garfinkel (1972)), but better algorithms than general-purpose linear or integer programming methods are available for their solution. These algorithms use two techniques, *greed* and *augmentation*. The most general setting for these techniques is in matroid theory (Lawler (1976)), but one can understand and apply the techniques to graph problems without knowing about matroids.

Consider the problem of finding, in a set with weighted elements, a maximum-weight subset satisfying certain additional constraints. The following *greedy method* might be useful in solving this problem. Sort the elements by weight. Examine the elements in order, heaviest to lightest, building up a subset element-by-element. When examining an element, add it to the subset if some extension of the subset satisfies the constraint. Otherwise throw the element away. The resultant subset certainly satisfies the constraint. Under appropriate conditions, the subset will be of maximum possible weight. One problem to which this method is applicable is the minimum spanning tree problem (Kruskal (1956), Prim (1957), Dijkstra (1959), Yao (1975), Cheriton and Tarjan (1976)). Even if the greedy method does not produce optimal solutions, it may produce solutions which are close to optimal (Garey and Johnson (1976)), and it is usually easy to implement and fast.

In situations where the greedy method doesn't work, a method of iterative improvement sometimes does. The idea is to start with any solution to the constraints

and look for a way to *augment* the weight of the solution by making local changes. The new solution is then improved in the same way, and the process is continued until no improvement is possible. Under appropriate conditions such a locally maximal solution is also globally maximum. Even if the solution is not guaranteed to be maximum, the augmentation method may be a good heuristic; for instance, Lin (1965) has applied it with good results to the traveling salesman problem. The traveling salesman problem is to find a shortest cycle through all vertices of a graph with distances on the edges. The Hamilton cycle problem, a special case of the traveling salesman problem, is $\mathcal{NP}$-complete.

Data updating methods. Some problems require more sophisticated data manipulation than is possible with the simple data structures discussed early in this section. Three advanced techniques have been devised for dealing with three diverse problems which require dynamic updating of data. These techniques are *path compression*, *partition refinement*, and *linear arrangement*.

Path compression is a method of solving the following problem. Consider a universe of elements, partitioned initially into singleton sets. Associated with each element is a value. We wish to be able to carry out the following operations on the sets.

Union: Combine two sets into a single set, destroying the old sets.

Update: Modify the values of all elements in a given set in a consistent way.

Evaluate: Retrieve the value associated with a given element.

A situation of this kind occurs in the compilation of FORTRAN COMMON and EQUIVALENCE statements (Galler and Fischer (1964)) and in several other combinatorial problems (Tarjan (1975b)). The set union problem to be discussed in § 5 is the simplest such problem. Galler and Fischer (1964) proposed an algorithm for this problem using trees as a data structure. McIlroy and Morris confronted the set union problem when trying to compute minimum spanning trees and proposed an improved method using path compression on trees (Aho, Hopcroft and Ullman (1974)). Their method, which is very simple to program but very hard to analyze, generalizes to a number of other problems (Tarjan (1975b)). I shall discuss this method and its remarkable running time in § 5.

Another problem involving disjoint sets is the following. Suppose the vertices of a graph are initially partitioned into several subsets. We wish to find the coarsest partition which is a refinement of the given partition and which is preserved under adjacency, in the sense that if two vertices v and w are contained in the same subset of the partition, then the sets $A(v)=\{x \mid (v, x)$ is an edge$\}$ and $A(w)=\{x \mid (w, x)$ is an edge$\}$ intersect exactly the same number of times with each subset of the partition. This adjacency-preserving partition is easily computable in $O(nm)$ time. Hopcroft (1971) devised a more sophisticated algorithm which runs in $O(m \log n)$ time. Gries (1973) gives a nice description of this algorithm. Partition refinement is useful in solving the state minimization problem for finite automata (Harrison (1965)) and in testing graphs for isomorphism (Corneil and Gottlieb (1970)).

A third problem requiring a good data updating method is the *linear arrangement problem*: Given a set of n elements and a collection of subsets of the elements, can the elements be arranged in a line so that each subset occurs contiguously? This problem arises in biochemistry (Benzer (1959)) and in archaeology (Kendall (1969)). Booth and Lueker (1976) have devised a method of solving this problem in $O(n+m)$ time, where m is the total size of the subsets, using a data structure they call a P-Q tree.

Graph mapping. There are two methods of solving graph problems, *decomposition* and *shrinking*, which are related to the algebraic concepts of *subalgebra* and

homomorphism. One way to solve certain graph problems is to decompose the graph into several subgraphs, solve the problem on the subgraphs, and combine the solutions to give the solution for the entire graph. In most instances where this technique is useful, the subgraphs are components (maximal subgraphs) satisfying some connectivity relation. In order to apply the technique, one must know an efficient way to determine the components. Good algorithms exist for a variety of connectivity problems (Tarjan (1972), Hopcroft and Tarjan (1973a), Hopcroft and Tarjan (1973c), Pacault (1974), Tarjan (1974a), Tarjan (1975c)).

Another way to solve some graph problems is to shrink part of the graph to a single vertex, solve the problem on the shrunken graph by applying the idea recursively, and from this solution compute the solution on the original graph. The shrinking operation corresponds to taking a homomorphic image of the graph. Generally the part of the graph to be shrunk is a cycle or a union of cycles.

5. Ten tractable problems. There are hundreds of combinatorial problems for which good algorithms are known. This section examines ten such problems. I have selected the problems on the basis of their importance, the range of techniques they require, and my familiarity with them. The list is not meant to be exhaustive but to be representative of problems with good algorithms. Table 5.1 lists the problems and the techniques used in the best algorithms for them. Figure 5.1 shows improvements in solution time achieved recently for these problems.

TABLE 5.1
Ten tractable problems and methods for solving them.

1. Discrete Fourier transform (DFT):
   recursion.
2. Matrix multiplication (MM):
   recursion.
3. Linear equations on a planar graph (LEG):
   recursion, decomposition by connectivity, breadth-first search.
4. Global flow analysis (GFA):
   decomposition by connectivity, path compression, depth-first search.
5. Pattern matching on strings (PM):
   data structures.
6. Strong components (SC):
   depth-first search.
7. Planarity testing (PT):
   depth-first search.
8. Maximum network flow (MNF):
   augmentation, breadth-first search.
9. Graph matching (GM):
   augmentation, breadth-first search, cycle shrinking.
10. Set union (SU):
    path compression.

Discrete Fourier transform. Given an n-dimensional vector $(a_0, a_1, \cdots, a_{n-1})$, the discrete Fourier transform problem is to compute the vector $(b_0, b_1, \cdots, b_{n-1})$ given by $b_k = \sum_{i=0}^{n-1} a_i \omega^{ik}$, where $\omega^0, \omega^1, \cdots, \omega^{n-1}$ are the (complex) nth roots of one. This problem arises in signal processing. An algorithm for the discrete Fourier transform is useful as a subroutine in various arithmetic and algebraic problems, including polynomial evaluation and interpolation and integer and polynomial multiplication (Knuth (1969), Aho, Hopcroft and Ullman (1974), Borodin and Munro (1975)).

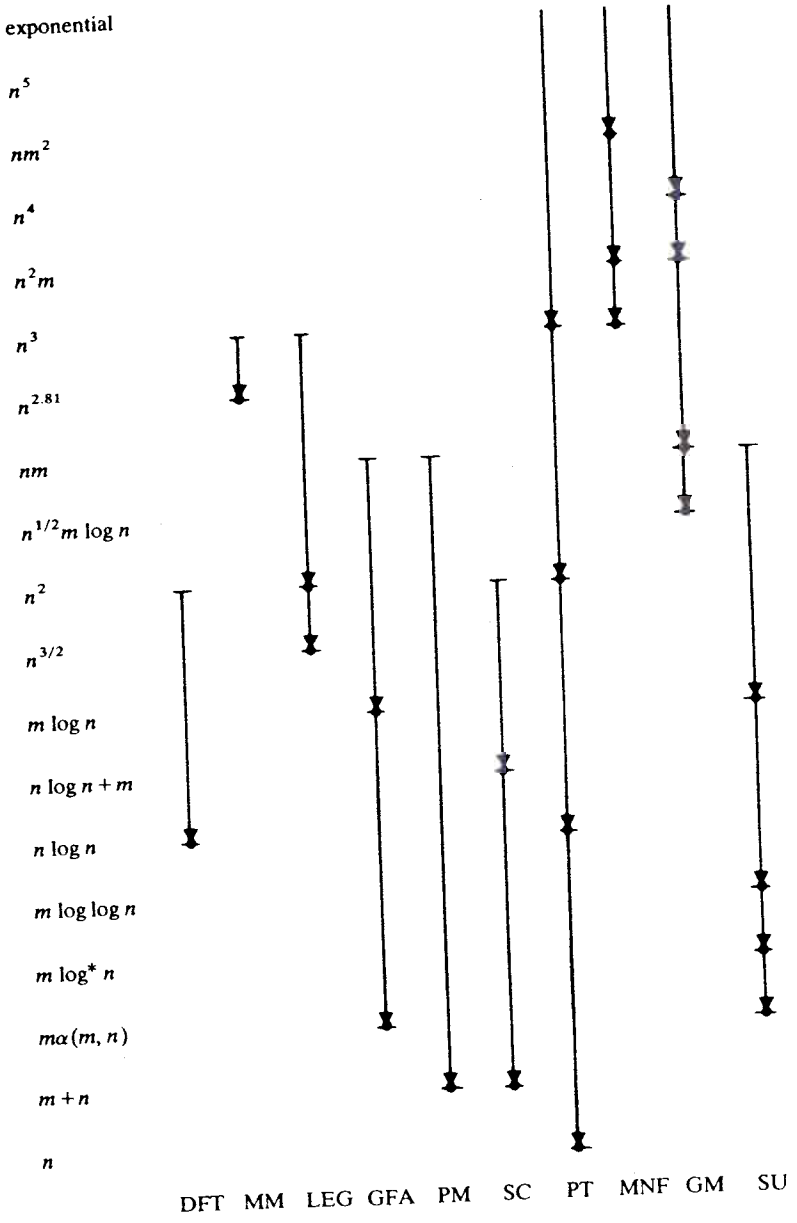


FIG. 5.1. *Recent complexity improvements.*
n = size (number of vertices in graph problems).
m = second parameter (number of edges in graph problems).

It is straightforward to compute the discrete Fourier transform in $O(n^2)$ time. Cooley and Tukey (1965) popularized an $O(n \log n)$-time method, called the *fast Fourier transform*. They were not the first to use the method, which originated at least as early as Runge and König (1924). The fast Fourier transform uses recursion to cut down the amount of computation. Recently Winograd (1975), (1976) proposed a method for computing the discrete Fourier transform using only $O(n)$ multiplications.

This method may be superior to the fast Fourier transform in practice, although Winograd has not analyzed the overall running time of his algorithm.

Matrix multiplication. Given two $n \times n$ matrices, the matrix multiplication problem is to determine their matrix product. The standard high school method of matrix multiplication requires $O(n^3)$ time. Strassen (1969) devised a way to multiply two 2×2 matrices with only seven multiplications, and used this in a recursive matrix multiplication algorithm requiring only $O(n^{\log_2 7})$ time. This surprising result has acted as a stimulus for much research in the complexity of algebraic problems. No one knows whether Strassen's algorithm is improvable. Strassen's algorithm has been used to compute transitive closures of graphs (Munro (1971), Fischer and Meyer (1971)) and to do context-free language parsing (Valiant (1975a)) in $O(n^{2.81})$ time.

A problem related to matrix multiplication is the *shortest path problem*. Given a directed graph with positive edge distances, the *single source shortest path problem* is to find the minimum distance from a given vertex to every other vertex. The *all pairs shortest path problem* is to find the minimum distance between all pairs of vertices. Dijkstra (1959) devised an algorithm for the single source problem which requires either $O(n^2)$ time or $O(m \log n)$ time depending upon the implementation, where n is the number of vertices and m the number of edges in the graph (Johnson (1977)). Floyd (1962) gave a way of solving the all pairs problem in $O(n^3)$ time. Fredman (1976) showed that the all pairs problem can be solved using $O(n^{2.5})$ comparisons and only $O(n^3(\log \log n/\log n)^{1/3})$ time total.

Linear equations on a planar graph. Suppose A is an $n \times n$ matrix, b is an $n \times 1$ vector of constants, x is an $n \times 1$ vector of variables, and we wish to solve the system of equations $Ax = b$. A standard method for doing this is Gaussian elimination (Forsythe and Moler (1967), Tewarson (1973)). First, the matrix A is decomposed into a product of two matrices, $A = LU$, such that L is *lower triangular* (i.e., L has no nonzero entries above the diagonal) and U is *upper triangular* (i.e., U has no nonzero entries below the diagonal). Then $Ax = b$ is solved in two steps, by solving $Ly = b$, called *frontsolving*, and solving $Ux = y$, called *backsolving*. Because L and U have special forms, frontsolving and backsolving are very efficient; the slowest part of Gaussian elimination is the first step, decomposing A into LU.

The decomposition of A proceeds by means of *row operations*. A row operation consists of adding a multiple of one row of A to another row of A. If the multiple is chosen correctly, the modified row will have a zero in a previously nonzero position. By systematically applying such row operations, one can transform the original matrix A into an upper triangular matrix U; the row operations performed define a lower triangular matrix L such that $LU = A$.

If A is originally a dense (mostly nonzero) matrix, then LU decomposition requires $O(n^2)$ space and $O(n^3)$ time, and frontsolving and backsolving require $O(n^2)$ time. In many large systems of equations, however, the matrix A is sparse. For a sparse matrix, the time and storage space required by Gaussian elimination depend in a complicated way upon the zero-nonzero structure of the matrix. In particular, a row operation may introduce new nonzeros (called fill-in) into positions originally zero. It is desirable to rearrange the matrix A by means of row and column permutations so that the fill-in and running time of Gaussian elimination are reduced.

For this purpose it is useful to represent the zero-nonzero structure of A by a graph G. The graph contains one vertex for each row and column of A and one edge

(i, j) for each nonzero entry (i, j) in A. If A is symmetric, G is undirected; if A is unsymmetric, G is directed. The graph G represents A and all matrices formed by simultaneously permuting rows and columns of A. By studying the properties of G, it may be possible to find a reordered version of A such that Gaussian elimination is efficient. (It is necessary to know that the permutations do not destroy the numeric stability of the elimination process. I shall ignore this issue here; see Forsythe and Moler (1967), Tewarson (1973).)

Parter (1961) was one of the first to suggest the usefulness of this approach. The idea has been extensively developed. For general results concerning the relationship between Gaussian elimination and graph theory, see Rose (1970); Harary (1971); Rose (1973); Rose, Tarjan and Lueker (1976); Duff (1976); and Rose and Tarjan (1977).

As an example of the improvement possible by taking advantage of sparsity, consider the graph in Fig. 5.2. Such a $k \times k$ *grid graph* arises in the numeric solution of differential equations. Ordinary dense Gaussian elimination requires $O(n^2)$ space and $O(n^3)$ time on such a matrix, if $n = k^2$. The *bandwidth* scheme of sparse elimination reduces the space to $O(n^{3/2})$ and the time to $O(n^2)$ (Cuthill and McKee (1969), Tewarson (1973)). George (1973) discovered an even better method, called *nested dissection*, which requires $O(n \log n)$ space and $O(n^{3/2})$ time. Hoffman, Martin and Rose (1973) showed that, to within a constant factor, nested dissection requires the least fill-in and computing time of any ordering scheme for Gaussian elimination on $k \times k$ grid graphs.

Nested dissection is a recursive method which uses the fact that a $(2k+1) \times (2k+1)$ grid graph consists of four $k \times k$ grid graphs and the $(4k+1)$-vertex boundary between them (Fig. 5.2). Many sparse matrices which arise in practice do not have such a nice structure, and one might ask whether nested dissection has any natural generalization. Recently Lipton, Rose and Tarjan (Tarjan (1976b)) discovered a way to extend nested dissection to arbitrary planar graphs so that the storage space is still $O(n \log n)$ and the running time still $O(n^{3/2})$. Such graphs arise in two-dimensional finite element problems (Martin and Carey (1973)).

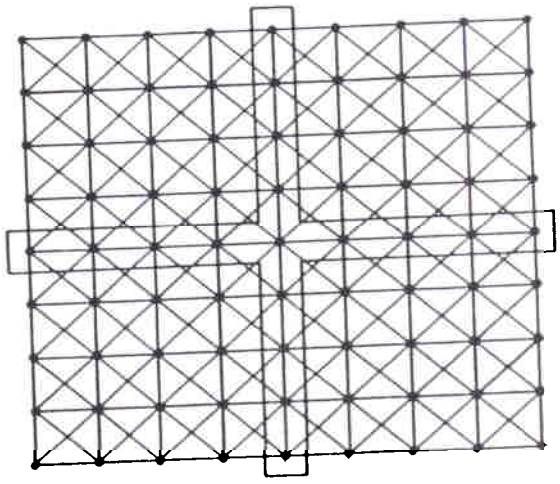


FIG. 5.2. *Nine × nine grid graph. Vertices in cross separate grid into four* 4×4 *grids.*

Global flow analysis. Systems of linear equations arise in contexts other than linear algebra. For instance, the shortest path problem can be formulated as a system of equations, with minimization replacing addition and addition replacing multiplication (Backhouse and Carré (1975)). Another situation where systems of linear equations occur is in the global flow analysis of computer programs. Suppose, for instance, that we wish to modify a computer program so that it does not recompute an expression unless the value of one of the variables in the expression has changed.

The first step in the analysis is to represent the program by a *flow graph*. Each vertex in the flow graph represents a *basic block* of the program (a set of program statements having a single entry point and a single exit point). Each edge in the flow graph represents a transfer of control from one basic block to another. The problem of determining, for each basic block, the set of *available expressions* (those which do not need to be recomputed) can then be formulated as a system of linear equations with one variable for each basic block. The variable is a *bit vector*, with one bit corresponding to each program expression, and appropriate bit vector operations replace addition and multiplication in the system of equations. The sparsity structure of the matrix corresponds to the flow graph representing the program. For further details of this correspondence, see Kildall (1973), Schaefer (1973), and Allen and Cocke (1976).

One can use standard Gaussian elimination techniques to compute available expressions, but it is often useful to take advantage of sparsity. The flow graphs of many computer programs have a special property called *reducibility*, which means in essence that every cycle has a single entry point from the starting block of the program. Allen (1970) and Cocke (1970) first formulated this notion of reducibility, presented an $O(nm)$-time algorithm to test for reducibility, and used this test in an $O(nm)$-time algorithm for global flow analysis of reducible graphs. Hopcroft and Ullman (1972) discovered an $O(m \log n)$-time test for reducibility, which Ullman (1973) combined with clever use of 2-3 trees to give an $O(m \log n)$-time method for global flow analysis. Kennedy (1975) discovered a rather different method for global flow analysis, which is $O(m \log n)$-time by a result of Aho and Ullman (1975). Graham and Wegman (1976) discovered how to use path compression to get yet another $O(m \log n)$-time algorithm. Tarjan (1974c) gave an $O(m\alpha(m, n))$-time algorithm for testing reducibility, and later improved the Graham–Wegman algorithm to run in $O(m\alpha(m, n))$ time (Tarjan (1975c)). Here $\alpha(m, n)$ is a functional inverse of Ackermann's function to be defined below.

Pattern matching on strings. Suppose x and y are two strings composed of characters selected from a finite alphabet, and we wish to determine where x occurs as a contiguous substring of y. If m is the length of x and n is the length of y, then a straightforward algorithm solves this problem in $O(nm)$ time. Knuth, Morris and Pratt (1977) devised an $O(n+m)$-time algorithm for pattern matching. Their algorithm processes the pattern x, creating a data structure representing a program to recognize the pattern. The algorithm then scans the string y character-by-character according to the steps of the program. Boyer and Moore (1975) proposed an even better algorithm, which, although it has an $O(n+m)$ running time in the worst case (Knuth, Morris and Pratt (1977)), requires only $O(n(\log_q m)/m)$ time on the average, where q is the alphabet size.

A generalization of the pattern matching problem is to find the longest common contiguous substring of two strings x and y. The pattern matching algorithms mentioned above do not seem to apply to this problem. Karp, Miller and Rosenburg (1972) described an $O(m+n)\log(m-n))$-time algorithm for longest common

substrings. Weiner (1973) discovered an algorithm using trees in a new way which solves the problem in $O(n+m)$ time. McCreight (1976) has provided a simplification and clean description of this algorithm.

Strong components. The strong components problem is to determine the strongly connected components of a given directed graph with n vertices and m edges. This problem occurs in finding the irreducible blocks of a nonsymmetric matrix (Forsythe and Moler (1967)), in finding ergodic subchains and transient states of a Markov chain (Fox and Landy (1968)), and in finding the transitivity sets of a set of permutations (McKay and Regener (1974)). Sargent and Westerberg (1964) gave an $O(n^2)$-time algorithm. Munro (1971) described an improved algorithm with a running time of $O(n \log n + m)$. Tarjan (1972) presented an $O(n+m)$-time algorithm which uses depth-first search and a few simple data structures to solve this problem.

Planarity testing. Let G be a graph. The planarity testing problem is to determine whether G can be drawn in a plane so that no two edges cross. Kuratowski (1930) provided an elegant mathematical characterization of planar graphs, showing that a graph G is planar if and only if it does not contain one of the two graphs in Fig. 5.3 as a generalized subgraph. Unfortunately, Kuratowski's criterion seems to be useless as a

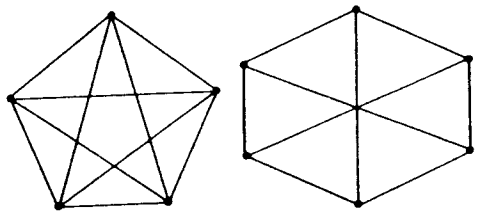


FIG. 5.3. *Kuratowski subgraphs.*

(a) K_5.
(b) $K_{3,3}$.

practical test for planarity. Auslander and Parter (1961) proposed an algorithm which tests planarity by trying to construct a planar representation for the graph. They gave no time bound for the algorithm, and their presentation contains an error: the proposed algorithm may run forever. Goldstein (1963) correctly formulated this algorithm, and Shirey (1969) gave an $O(n^3)$-time implementation of it. Hopcroft and Tarjan (1972) combined depth-first search and appropriate data structures in an $O(n \log n)$-time implementation, which was later simplified and improved to $O(n)$ (Hopcroft and Tarjan (1974)).

Lempel, Even and Cederbaum (1967) presented another good algorithm, without giving an explicit time bound. Their algorithm can easily be implemented to run in $O(n^2)$ time. Booth and Lueker (1976) showed how to use their P-Q tree data structure in an $O(n)$-time implementation of the algorithm.

Maximum network flow. Let G be a directed graph with two distinguished vertices, a *source* s and a *sink* t. For each edge e in G, let $c(e)$ be a nonnegative real-valued *capacity*. A *flow* f on G is a nonnegative value $f(e)$ on each edge such that, for all vertices v except s and t, the total flow on edges entering v is equal to the total flow leaving v. The *value* of the flow is the total flow leaving s (which is equal to the total flow entering t). The maximum network flow problem is to determine a flow $f(e)$ of maximum value satisfying $f(e) \leqq c(e)$ for all edges e.

Classic work by Ford and Fulkerson (1962) produced an elegant algorithm which augments flow along paths. Unfortunately, in the worst case their algorithm does not terminate, though it works exceedingly well in practice. Edmonds and Karp (1972), by using breadth-first search to guide the selection of augmenting paths, produced an $O(nm^2)$-time variation of the Ford–Fulkerson algorithm. Independently, Dinic (1970) used breadth-first search plus improved updating methods to achieve an $O(n^2m)$ time bound. The best algorithm so far found for this problem is due to Karzanov (1974), who improved Dinic's algorithm to obtain an $O(n^3)$ time bound.

Graph matching. If G is an undirected graph, the graph matching problem is to find a maximum number of edges in G, no two having a common endpoint. Such a set of edges is a *maximum matching*. An important special case of this problem is its restriction to *bipartite* graphs. A graph is bipartite if its vertices can be partitioned into two sets so that no edge connects two vertices in the same set.

The bipartite graph matching problem can be transformed via a linear-time algorithm into a network flow problem in which all edge capacities are one (Ford and Fulkerson (1962)); for such a problem, the Ford–Fulkerson algorithm has an $O(nm)$ time bound. Kuhn (1955) used results of Egerváry to obtain essentially the same algorithm, called the Hungarian method. Hopcroft and Karp (1973) used breadth-first search and improved updating methods to achieve an $O(n^{1/2}m)$ time bound. Their algorithm is essentially the same as Dinic's (Even and Tarjan (1975)).

Berge (1957) and Norman and Rabin (1959) proved that an augmenting path method can solve the maximum matching problem on nonbipartite graphs, though a good algorithm does not follow from their results. Edmonds (1965) used cycle shrinking plus the augmenting paths idea to give a polynomial-time algorithm. He claimed an $O(n^4)$ time bound, though it is not hard to implement Edmonds' algorithm to run in $O(n^2m)$ time. Lawler (1976) and Gabow (1976) independently gave $O(nm)$-time algorithms. Even and Kariv (1975) ingeniously combined the ideas of Hopcroft and Karp and the data structures of Gabow to obtain an $O(n^{1/2}m \log n)$-time algorithm.

Set union. Let $S_1, S_2, \ldots, S_n$ be n disjoint sets, each containing a single element. The disjoint set union problem is to carry out a sequence of operations of the following two types on the sets.

find (x): determine the name of the set containing element x.
union (A, B): add all elements of set B to set A (destroying set B).

The operations are to be carried out *on-line*; that is, each instruction must be completed before the next one is known. Assume for convenience that the sequence of operations contains exactly $n-1$ union operations (so that after the last union all elements are in one set) and $m \geqq n$ intermixed find operations (if $m < n$, some elements are never found).

Galler and Fischer (1964) proposed an algorithm for this problem in which each set is represented by a tree. Each vertex of the tree represents one element. The root of the tree contains the name of the set, and each tree vertex has a pointer to its parent in the tree. See Fig. 5.4. A find on element x is performed by starting at the vertex representing x and following parent pointers until reaching the root of the tree; this root contains the set name. A union of sets A and B is performed by making the root of the A tree the parent of the root of the B tree.

This algorithm requires $O(nm)$ time in the worst case, since an unfortunate sequence of unions can build up a tree consisting of a single long path. Galler and Fischer modified the union procedure in the following way: if B contains more

elements than A, then the root of the B tree is made the parent of the root of the A tree, and the name A is moved to the old root of B. See Fig. 5.5. This *weighted union* heuristic improves the algorithm considerably; Galler and Fischer proved an $O(m \log n)$ time bound.

McIlroy and Morris (Aho, Hopcroft and Ullman (1974)) modified the find procedure by adding a heuristic called *path compression*: after a find on element x, all vertices on the path from x to the root are made children of the root. See Fig. 5.6. This increases the time of a find by a constant factor but may save time on later finds.

The set union algorithm with path compression is very easy to program but very hard to analyze. Fischer (1972) proved an $O(mn^{1/2})$ upper bound and an $\Omega(m \log n)$ lower bound on the worst-case running time of the algorithm with path compression

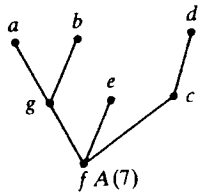
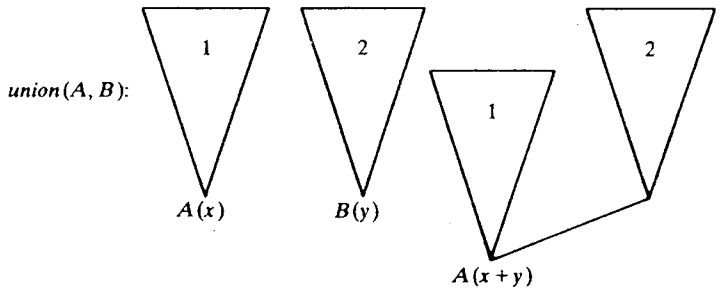
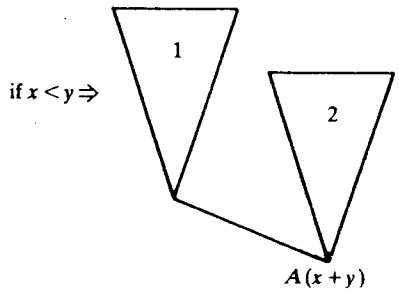


FIG. 5.4. *Representation of set* $A = \{a, b, c, d, e, f, g\}$ *by a tree. Size of set in parentheses.*



(a)



(b)

FIG. 5.5. *Implementation of union.*
(a) *Basic method.*
(b) *Weighting heuristic.*

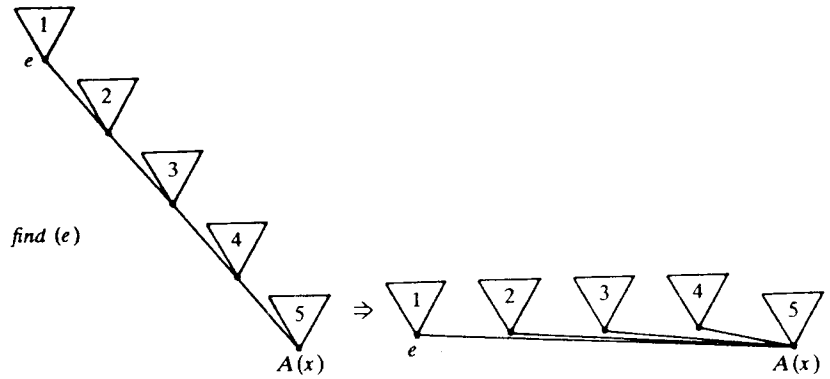


FIG. 5.6. *Implementation of find with path compression.*

but without weighted union. Paterson (1972) improved the upper bound to $O(m \log n)$ and thus determined the running time to within a constant factor for the case when m is $O(n)$. With both path compression and weighted union, the algorithm is even harder to analyze. Fischer (1972) proved an $O(m \log \log n)$ upper bound on the running time. Hopcroft and Ullman (1973) improved the upper bound to $O(m \log^* n)$, where

$$\log^* n = \min \{i \mid \overbrace{\log \log \cdots \log}^{i \text{ times}} n \leqq 1\}.$$

Tarjan (1975a) improved the upper bound to $O(m\alpha(m, n))$, where $\alpha(m, n)$ is a functional inverse of Ackermann's function (Ackermann (1928)) defined as follows.

For $i, j \geqq 0$ let the function $A(i, j)$ be defined by

$$\begin{aligned} &A(i, 0) = 0; \\ &A(0, j) = 2^j \quad \text{for } j \geqq i; \\ &A(i, 1) = A(i-1, 2) \quad \text{for } i \geqq 1; \\ &A(i, j) = A(i-1, A(i, j-1)) \quad \text{for } i \geqq 1, j \geqq 2. \end{aligned} \tag{5.1}$$

Let

$$\alpha(m, n) = \min \{i \geqq 1 \mid A(i, \lfloor 2m/n \rfloor) > \log_2 n\}.^3 \tag{5.2}$$

The bound $O(m\alpha(m, n))$ is a rather complicated one for such a simple algorithm. One may naturally ask whether it is improvable. Tarjan (1975a) showed that there are worst-case instances of the set union problem which require $\Omega(m\alpha(m, n))$ time when solved by the path compression algorithm. In fact *any* linked memory machine requires $\Omega(m\alpha(m, n))$ time in the worst case to solve the set union problem (Tarjan (1977)). Thus Ackermann's function is inherent in the problem.

6. Future directions. The field of combinatorial algorithms is too vast to cover in a single paper or even in a single book. I have tried to point out some of the major results and underlying ideas in this field, but there are certainly many important results I have had to omit. Though much work on combinatorial algorithms has been done, much remains to be done. In this concluding section I would like to suggest five areas for future research, areas in which relatively little work has been done but in which the rewards are potentially great.

[3] For any real number x, $\lfloor x \rfloor$ denotes the greatest integer not larger than x.

Is $\mathcal{P} = \mathcal{NP}$? Answering this question would be a major breakthrough in complexity theory. Although many people have attempted to solve this problem, very little progress has been made. It seems that some major new idea is needed; the evidence of Baker, Gill and Solovay (1975) suggests that diagonalization, the standard technique for proving problems hard, may not be powerful enough to show $\mathcal{P} \neq \mathcal{NP}$. It is even conceivable that the $\mathcal{P} = \mathcal{NP}$? problem cannot be solved within the framework of formal set theory (Hartmanis and Hopcroft (1976)).

More generally, we now know almost nothing about the relative power of deterministic and nondeterministic algorithms, and about the relationship between time and space as measures of complexity. Any results in this area would be important. Recently, Hopcroft, Paul and Valiant (1975) were able to show that any computation requiring $t(n)$ time on a multitape Turing machine can be carried out in $t(n)/\log t(n)$ space. Thus, at least for multitape Turing machines, space is a more valuable resource than time. This is the only such result known.

One approach to the $\mathcal{P} = \mathcal{NP}$? question is to consider for a particular $\mathcal{NP}$-complete problem a restricted class of algorithms and to show that every algorithm in this limited class requires more than polynomial time. Results of this kind have been obtained for the satisfiability problem (Galil (1975)), the maximum stable set problem (Chvatal (1976)), and the graph coloring problem (McDiarmid (1976)).

Another approach is to consider models of computation other than Turing machines. One possibility is to study the size of Boolean circuits for computing Boolean functions. For results in this area, see Savage (1976). A related model is the *pebble game* used by Hopcroft, Paul and Valiant to obtain their time-space trade-off result. A study of Boolean circuits and the pebble game leads rapidly to unanswered combinatorial questions (Valiant (1975b), (1976)).

Average-case analysis. Although most of the work on combinatorial algorithms outside the areas of sorting and searching has been worst-case analysis, average-case analysis is potentially important and useful. The results of Erdös and Renyi (1960) and others on random graphs form a starting place for average-case analysis of graph algorithms. Spira (1973) has devised an $O(n^2(\log n)^2)$ average time algorithm for the all-pairs shortest path problem, which Bloniarz, Fischer and Meyer (1976) modified to compute transitive closures in $O(n^2 \log n)$ average time. Schnorr (1977) has devised an $O(n \log n + m)$ average time transitive closure algorithm. Yao (1976), Doyle and Rivest (1976), and Knuth and Schönhage (1977) have analyzed the behavior of set union algorithms for several probability distributions. Much more work in this area is needed.

Gill (1974) and Rabin (1976) have proposed another kind of average-case model of complexity, in which the algorithm makes use of random choices. For such an algorithm, one may be able to say that the algorithm runs fast on the average independent of the input distribution, because the average is taken not over the input but over the possible computations of the algorithm on a given input. Algorithms exist for testing primality (Strassen and Solovay (1977), Rabin (1976)), finding closest points (Rabin (1976)), and hashing (Carter and Wegman (1977)) which are good in this sense.

Constant factors and algorithm trade-offs. The choice of an algorithm in practice may depend upon more than asymptotic tunning time. A simple algorithm may be better on intermediate-sized problems than a more complicated algorithm with a faster asymptotic running time but a larger constant factor. If two algorithms have the same asymptotic running time, then the constant factors may govern the choice

between them. More careful analysis to determine constant factors and trade-offs between algorithms would be a useful contribution to practical computing. Such analysis for a number of problems appears in Knuth's books (Knuth (1968), (1969), (1973)). A recent example of this research is work by Brown (1977), which compares implementations of priority queues and suggests that the binomial tree representation is best in most circumstances.

Low-order lower bounds. Just as little is known about the boundary between tractable and intractable problems, little is known about whether or not the existing good algorithms are improvable. Figure 5.1 suggests several tantalizing questions. Can two matrices be multiplied in less than $O(n^{2.81})$ time? Can the discrete Fourier transform be computed in less than $O(n \log n)$ time? Can maximum network flows be found in less than $O(n^3)$ time? Nonlinear lower bounds exist for only a few problems, such as sorting (Knuth (1973)), disjoint set union (Tarjan (1977)), and evaluation of symmetric functions (Strassen (1973)).

General properties of data structures and basic methods. The range of techniques and algorithms outlined in §§ 4 and 5 suggests a basic question: Confronted with a problem, how does one construct a good algorithm for it? Is there a "calculus of data structures" by which one can choose the appropriate data representation and techniques for a given problem? What makes one data structure better than another for a certain application? The known results cry out for an underlying theory to explain them. This is perhaps one of the most challenging problems facing researchers in algorithmic complexity.

Appendix: Terminology. (See also Berge (1962), Busacker and Saaty (1965), and Harary (1969)). A *graph* $G = (V, E)$ is an ordered pair consisting of a set V of *vertices* and a set E of *edges*. Either the edges are ordered pairs (v, w) of distinct vertices (the graph is *directed*), or the edges are unordered pairs of distinct vertices, also represented as (v, w) (the graph is *undirected*). If (v, w) is an edge, v and w are its *endpoints* and are *adjacent*. The edge (v, w) *leads from* v to w. (If undirected, the edge also leads from w to v.) A graph $G' = (V', E')$ is a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$. G' is *spanning* if $V' = V$. A graph $G' = (V', E')$ is a *homomorphic image* of G if there is a mapping from V onto V' such that $(x, y) \in E'$ if and only if $x = f(v)$ and $y = f(w)$ for some $(v, w) \in E$. G and G' are *isomorphic* if the mapping is one-to-one. A graph $G' = (V', E')$ is a *generalized subgraph* of G if G' is a subgraph of a homomorphic image of G.

A *path* from v_1 to v_n in G is a sequence of edges $(v_1, v_2), \cdots, (v_{n-1}, v_n)$. This path is said to *contain* edges $(v_1, v_2), \cdots, (v_{n-1}, v_n)$ and vertices $v_1, \cdots, v_n$, and to *avoid* all other edges and vertices. The path is *simple* if $v_1, \cdots, v_n$ are distinct except possibly v_1 and v_n; the path is a *cycle* if $v_1 = v_n$. The *transitive closure* of $G = (V, E)$ is the graph $G^+ = (V, E^+)$ such that $(v, w) \in E^+$ if and only if $v \neq w$ and there is a path from v to w in G.

An undirected graph is *connected* if there is a path from any vertex to any other vertex. A directed graph is *strongly connected* if there is a path from any vertex to any other vertex. The maximum connected (strongly connected) subgraphs of a graph are called its *connected components* (strongly connected components). A graph is *planar* if it can be drawn in the plane (with vertices as points and edges as simple curves) so that no two edges intersect except at a common endpoint.

A *tree* T is a connected, undirected graph which contains no cycles. In a tree there is a unique simple path between any pair of distinct vertices. A *rooted, undirected*

tree (T, r) is a tree with a distinguished vertex r, called the root. A *rooted, directed tree* is a directed graph T with a unique vertex r such that

(i) there is a path from r to any other vertex;
(ii) each vertex except r has exactly one edge leading to it;
(iii) r has no edges leading to it.

Any rooted, undirected tree (T, r) can be converted into a rooted, directed tree by directing each edge (v, w) so that v is contained in the path from r to w.

In a rooted, directed tree, a vertex w is a *descendant* of a vertex v (v is an *ancestor* of w) if there is a path from v to w. A vertex v is a *child* of v (v is the *parent* of w) if (v, w) is an edge in the tree. These definitions extend to rooted, undirected trees by directing the edges of the tree as above. If G is a graph, a *spanning tree* of G is a rooted tree which is a spanning subgraph of G.

A *partition* $\mathcal{S}$ of a set S is a collection of subsets $S_1, S_2, \cdots, S_k$ of S such that $\bigcup_{j=1}^{k} S_i = S$ and $S_i \cap S_j = \emptyset$ if $i \neq j$. If $\mathcal{S}, \mathcal{S}'$ are partitions of S, $\mathcal{S}$ is a *refinement* of $\mathcal{S}'$ (and $\mathcal{S}'$ is a *coarsening* of $\mathcal{S}$) if, for all $S_i \in \mathcal{S}$, there is some $\mathcal{S}_j' \in \mathcal{S}'$ such that $\mathcal{S}_i \subseteq \mathcal{S}_j'$.

REFERENCES

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

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

A. V. AHO AND T. G. PETERSON (1972), *A minimum distance error-correcting parser for context-free languages*, SIAM J. Comput., 1, pp. 305–312.

A. V. AHO AND J. D. ULLMAN (1975), *Node listings for reducible flow graphs*, Proc. Seventh Annual ACM Symp. on Theory of Computing, Association for Computing Machinery, New York, pp. 177–185.

F. E. ALLEN (1970), *Control flow analysis*, SIGPLAN Notices, 5, pp. 1–19.

F. E. ALLEN AND J. COCKE (1976), *A program data flow analysis procedure*, Comm. ACM, 19, pp. 137–146.

K. APPEL AND W. HAKEN (1977), *The solution of the four-color-map problem*, Sci. Amer., 237, pp. 108–121.

L. AUSLANDER AND S. V. PARTER (1961), *On imbedding graphs in the plane*, J. Math. and Mech., 10, pp. 517–523.

R. C. BACKHOUSE AND B. A. CARRÉ (1975), *Regular algebra applied to path-finding problems*, J. Inst. Math. Appl., 15, pp. 161–186.

T. BAKER, J. GILL AND R. SOLOVAY (1975), *Relativizations of the P = ?NP question*, SIAM J. Comput., 4, pp. 431–442.

R. E. BELLMAN (1957), *Dynamic Programming*, Princeton University Press, Princeton, NJ.

S. BENZER (1959), *On the topology of the genetic fine structure*, Proc. Nat. Acad. Sci. U.S.A., 45, pp. 1607–1620.

C. BERGE (1957), *Two theorems in graph theory*, Ibid., 43, pp. 842–844.

——— (1962), *The Theory of Graphs and its Applications*, A. Doig, transl., John Wiley, New York.

P. A. BLONIARZ, M. J. FISCHER AND A. R. MEYER (1976), *A note on the average time to compute transitive closures*, Tech. Memorandum 76, Laboratory for Computer Science, Mass. Inst. of Tech., Cambridge.

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

K. S. BOOTH AND G. S. LUEKER (1976), *Testing for the consecutive ones property, interval graphs, and graph planarity using P-Q tree algorithms*, Ibid., 13, pp. 335–379.

A. BORODIN AND I. MUNRO (1975), *The Computational Complexity of Algebraic and Numeric Problems*, American Elsevier, New York.

R. S. BOYER AND J. S. MOORE (1975), *A fast string searching algorithm*, Stanford Research Institute and Xerox Palo Alto Research Center, Palo Alto, CA, unpublished manuscript.

M. R. BROWN (1977), *The analysis of a practical and nearly optimal priority queue*, Ph.D. thesis, STAN-CS-77-600, Computer Science Dept., Stanford Univ., Stanford, CA.

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

G. CANTOR (1874), *Über eine Eigenshaft des Inbegriffes aller reellen algebraischen Zahlen*, J. Reine Angew. Math., 77, pp. 258–262.

E. CARDOZA, R. LIPTON AND A. R. MEYER (1976), *Exponential space complete problems for Petri nets and commutative semigroups: preliminary report*, Proc. Eight Annual ACM Symp. on Theory of Computing, Association for Computing Machinery, New York, pp. 50–54.

J. L. CARTER AND M. N. WEGMAN (1977), *Universal classes of hash functions*, Proc. Ninth ACM Symp. on Theory of Computing, Association for Computing Machinery, New York, pp. 106–112.

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

A. CHURCH (1936), *An unsolvable problem of elementary number theory*, Amer. J. Math., 58, pp. 345–363.

V. CHVATAL (1976), *Determining the stability number of a graph*, Tech. Rep. STAN-CS-76-583, Computer Science Dept., Stanford Univ., Stanford, CA.

J. COCKE (1970), *Global common subexpression elimination*, SIGPLAN Notices, 5, pp. 20–24.

S. A. COOK (1971), *The complexity of theorem proving procedures*, Proc. Third Annual ACM Symp. on Theory of Computing, Association for Computing Machinery, New York, pp. 151–158.

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

J. M. COOLEY AND J. W. TUKEY (1965), *An algorithm for the machine calculation of complex Fourier series*, Math. Comput., 19, pp. 297–301.

D. G. CORNEIL AND C. C. GOTLIEB (1970), *An efficient algorithm for graph isomorphism*, J. Assoc. Comput. Mach., 17, pp. 51–64.

E. CUTHILL AND J. MCKEE (1969), *Reducing the bandwidth of sparse symmetric matrices*, Proc. 24th National Conf. ACM, Association for Computing Machinery, New York, pp. 157–172.

O. J. DAHL, E. W. DIJKSTRA AND C. A. R. HOARE (1972), *Structured Programming*, Academic Press, New York.

G. B. DANTZIG (1963), *Linear Programming and Extensions*, Princeton University Press, Princeton, NJ.

M. DAVIS, Y. MATIJASEVIC AND J. ROBINSON (1976), *Hilbert's tenth problem. Diophantine equations: positive aspects of a negative solution*, Mathematical Developments Arising from Hilbert Problems, American Mathematical Society, Providence, RI, pp. 323–378.

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

——— (1976), *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ.

E. A. DINIC (1970), *Algorithm for solution of a problem of maximum flow in a network with power estimation*, Soviet Math. Dokl., 11, pp. 1277–1280.

J. DOYLE AND R. L. RIVEST (1976), *Linear expected time of a simple union-find algorithm*, Information Processing Lett., 5, pp. 146–148.

I. S. DUFF (1976), *A survey of sparse matrix research*, Tech. Rep. CSS 28, Computer Science and Systems Division, AERE Harwell, Oxfordshire, England.

J. EARLEY (1970), *An efficient context-free parsing algorithm*, Comm. ACM, 13, pp. 94–102.

J. EDMONDS (1965), *Paths, trees, and flowers*, Canad. J. Math., 17, pp. 449–467.

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

P. ERDÖS AND A. RENYI (1960), *On the evolution of random graphs*, Magyar Tud. Akad. Mat. Kut. Int. Közl., 5, pp. 17–61.

S. EVEN, A. ITAI AND A. SHAMIR (1976), *On the complexity of timetable and multicommodity flow problems*, SIAM J. Comput., 5, pp. 691–703.

S. EVEN AND O. KARIV (1975), *An $O(n^{2.5})$ algorithm for maximum matching in general graphs*, Conf. Record, IEEE 16th Annual Symp. on Foundations of Computer Science, Institute of Electrical and Electronics Engineers, New York, pp. 100–112.

S. EVEN AND R. E. TARJAN (1975), *Network flow and testing graph connectivity*, SIAM J. Comput., 4, pp. 507–518.

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

M. J. FISCHER AND A. R. MEYER (1971), *Boolean matrix multiplication and transitive closure*, Conf. Record, IEEE 12th Annual Symp. on Switching and Automata Theory, Institute of Electrical and Electronics Engineers, New York, pp. 129–131.

M. J. FISCHER AND M. O. RABIN (1974), *Super-exponential complexity of Presburger arithmetic*, Project MAC Tech. Memorandum 43, Mass. Inst. of Tech., Cambridge.

R. W. FLOYD (1962), *Algorithm 97: shortest path*, Comm. ACM, 5, p. 345.

——— (1967), *Assigning meaning to programs*, Mathematical Aspects of Computer Science, J. T. Schwartz, ed., American Mathematical Society, Providence, RI., pp. 19–32.

L. R. FORD AND D. R. FULKERSON (1962), *Flows in Networks*, Princeton University Press, Princeton, NJ.

G. FORSYTHE AND C. B. MOLER (1967), *Computer Solution of Linear Algebraic Systems*, Prentice-Hall, Englewood Cliffs, NJ.

B. L. FOX AND D. M. LANDY (1968), *An algorithm for identifying the ergodic subchains and transient states of a stochastic matrix*, Comm. ACM, 11, pp. 619–621.

M. L. FREDMAN (1976), *New bounds on the complexity of the shortest path problem*, SIAM J. Comput., 5, pp. 87–89.

H. N. GABOW (1976), *An efficient implementation of Edmonds' algorithm for maximum matching on graphs*, J. Assoc. Comput. Mach., 23, pp. 221–234.

Z. GALIL (1975), *On the validity and complexity of bounded resolution*, Proc. Seventh Annual ACM Symp. on Theory for Computing, Association for Computing Machinery, New York, pp. 72–82.

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

M. R. GAREY AND D. S. JOHNSON (1976), *Approximation algorithms for combinatorial problems: an annotated bibliography*, Algorithms and Complexity: New Directions and Recent Results, J. F. Traub, ed., Academic Press, New York, pp. 41–52.

M. R. GAREY, D. S. JOHNSON AND L. J. STOCKMEYER (1976), *Some simplified NP-complete graph problems*, Theoret. Comput. Sci., 1, pp. 237–267.

M. R. GAREY, D. S. JOHNSON AND R. E. TARJAN (1976), *The planar Hamiltonian circuit problem is NP-complete*, SIAM J. Comput., 5, pp. 704–714.

J. A. GEORGE (1973), *Nested dissection of a regular finite element mesh*, SIAM J. Numer. Anal., 10, pp. 345–363.

J. T. GILL III (1974), *Computational complexity of probabilistic Turing machines*, Proc. Sixth Annual ACM Symp. on theory of Computing, Association for Computing Machinery, New York, pp. 91–95.

K. GÖDEL (1931), *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I*, Monatsh. Math. und Phys., 38, pp. 173–198.

A. J. GOLDSTEIN (1963), *An efficient and constructive algorithm for testing whether a graph can be embedded in a plane*, Graph and Combinatorics Conf., Contract No. NONR 1858-(21), Office of Naval Research Logistics Proj., Dept. of Math., Princeton Univ., Princeton, NJ.

S. L. GRAHAM AND M. WEGMAN (1976), *A fast and usually linear algorithm for global flow analysis*, J. Assoc. Comput. Mach., 23, pp. 172–202.

D. GRIES (1973), *Describing an algorithm by Hopcroft*, Acta Informat., 2, pp. 97–109.

L. GUIBAS, E. MCCREIGHT, M. PLASS AND J. ROBERTS (1977), *A new representation for linear lists*, Proc. Ninth Annual ACM Symp. on Theory of Computing, Association for Computing Machinery, New York, pp. 49–60.

F. HARARY (1969), *Graph Theory*, Addison-Wesley, Reading, MA.

——— (1971), *Sparse matrices and graph theory*, Large Sparse Sets of Linear Equations, J. K. Reid, ed., Academic Press, London, pp. 139–150.

M. A. HARRISON (1965), *Introduction to Switching and Automata Theory*, McGraw-Hill, New York.

J. HARTMANIS AND J. E. HOPCROFT (1976), *Independence results in computer science*, SIGACT News, 8, pp. 13–24.

J. HARTMANIS, P. M. LEWIS II AND R. E. STEARNS (1965), *Classification of computations by time and memory requirements*, Proc. IFIP Congress 65, International Federation for Information Processing, Spartan Books, Washington, DC, pp. 31–35.

J. HARTMANIS AND R. E. STEARNS (1965), *On the computational complexity of algorithms*, Trans. Amer. Math. Soc., 117, pp. 285–306.

D. HILBERT (1926), *Über das Unendliche*, Math. Ann., 95, pp. 134–151.

C. A. HOARE (1969), *An axiomatic basis of computer programming*, Comm. ACM, 12, pp. 576–580, 583.

A. J. HOFFMAN, M. S. MARTIN AND D. J. ROSE (1973), *Complexity bounds for regular finite difference and finite element grids*, SIAM J. Numer. Anal., 10, pp. 364–369.

J. E. HOPCROFT (1971), *An n log n algorithm for minimizing states in a finite automaton*, Theory of Machines and Computations, Z. Kohavi and A. Paz, eds., Academic Press, New York, pp. 189–196.

J. E. HOPCROFT AND R. M. KARP (1973), *An $n^{5/2}$ algorithm for maximum matching in bipartite graphs*, SIAM J. Comput., 2, pp. 225–231.

J. HOPCROFT, W. PAUL AND L. VALIANT (1975), *On time versus space and related problems*, Conf. Record, IEEE 16th Annual Symp. on Foundations of Computer Science, Institute of Electrical and Electronics Engineers, New York, pp. 57–64.

J. HOPCROFT AND R. TARJAN (1972), *Planarity testing in V log V steps: extended abstract*, Information Processing 71, Vol. 1—Foundations and Systems, North-Holland, Amsterdam, pp. 85–90.

——— (1973a), *Dividing a graph into triconnected components*, SIAM J. Comput., 2, pp. 135–158.

——— (1973b), *A V log V algorithm for isomorphism of triconnected planar graphs*, J. Comput. System Sci., 7, pp. 323–331.

——— (1973c), *Algorithm 447: efficient algorithms for graph manipulation*, Comm. ACM, 16, pp. 372–378.

——— (1974), *Efficient planarity testing*, Assoc. Comput. Mach., 21, pp. 549–568.

J. E. HOPCROFT AND J. D. ULLMAN (1972), *An n log n algorithm for detecting reducible graphs*, Proc. 6th Annual Princeton Conf. on Info. Sciences and Systems, Dept. of Electrical Engineering, Princeton University, Princeton, NJ, pp. 119–122.

——— (1973), *Set merging algorithms*, SIAM J. Comput., 2, pp. 294–303.

H. B. HUNT III (1973), *The equivalence problem for regular expressions with intersection is not polynomial in tape*, Tech. Rep. 73–156, Computer Science Dept., Cornell Univ., Ithaca, NY.

A. ITAI (1976), *Optimal alphabetic trees*, SIAM J. Comput., 5, pp. 9–18.

M. JAZAYERI, W. F. OGDEN AND W. C. ROUNDS (1975), *The intrinsically exponential complexity of the circularity problem for attribute grammars*, Comm. ACM, 18, pp. 697–706.

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

J. P. JONES (1974), *Recursive undecidability—an exposition*, Amer. Math. Monthly, 81, pp. 724–738.

R. M. KARP (1972), *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, pp. 85–104.

——— (1975), *On the computational complexity of combinatorial problems*, Networks, 5, pp. 45–68.

R. M. KARP, R. E. MILLER AND A. L. ROSENBERG (1972), *Rapid identification of repeated patterns in strings, trees, and arrays*, Proc. Fourth ACM Symp. on Theory of Computing, Association for Computing Machinery, New York, pp. 125–136.

A. V. KARZANOV (1974), *Determining the maximal flow in a network by the method of preflows*, Soviet Math. Dokl., 15, pp. 434–437.

D. G. KENDALL (1969), *Incidence matrices, interval graphs, and seriation in archaeology*, Pacific J. Math., 28, pp. 565–570.

K. W. KENNEDY (1975), *Node listings applied to data flow analysis*, Conf. Record, Second ACM Symp. on Principles of Programming Languages, Association for Computing Machinery, New York, pp. 10–21.

G. A. KILDALL (1973), *A unified approach to global program optimization*, Conf. Record, ACM Symp. on Principles of Programming Languages, Association for Computing Machinery, New York, pp. 194–206.

V. KLEE AND G. J. MINTY (1972), *How good is the simplex algorithm?*, Inequalities-III, O. Shisha, ed., Academic Press, New York, pp. 159–175.

S. C. KLEENE (1936), *General recursive functions of natural numbers*, Math. Ann., 112, pp. 727–742.

D. E. KNUTH (1968), *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison-Wesley, Reading, MA.

——— (1969), *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, Addison-Wesley, Reading, MA.

——— (1971), *Optimum binary search trees*, Acta Informat., 1, pp. 14–25.

——— (1973), *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley, Reading, MA.

D. E. KNUTH, J. H. MORRIS, JR. AND V. R. PRATT (1977), *Fast pattern matching in strings*, SIAM J. Comput., 6, pp. 323–350.

D. E. KNUTH AND A. SCHÖNHAGE (1977), *The expected linearity of a simple equivalence algorithm*, Tech. Rep. STAN-CS-77-599, Computer Science Dept., Stanford Univ., Stanford, CA.

A. N. KOLMOGOROV (1953), *On the notion of an algorithm*, Uspehi Mat. Nauk., 8, pp. 175–176.

A. N. KOLMOGOROV AND V. A. USPENSKII (1963), *On the definition of an algorithm*, Amer. Math. Soc. Transl. II, 29, pp. 217–245.

J. B. KRUSKAL, JR. (1956), *On the shortest spanning subtree of a graph and the travelling salesman problem*, Proc. Amer. Math. Soc., 7, pp. 48–50.

H. W. KUHN (1955), *The Hungarian method for the assignment problem*, Naval Res. Logist. Quart., 2, pp. 83–97.

C. KURATOWSKI (1930), *Sur le probleme des corbes gauches en topologic*, Fund. Math., 15, pp. 271–283.

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

A. LEMPEL, S. EVEN AND I. CEDERBAUM (1967), *An algorithm for planarity testing of graphs*, Theory of Graphs: International Symposium: Rome, July, 1966, P. Rosenstiehl, ed., Gordon and Breach, New York, pp. 215–232.

S. LIN (1965), *Computer solution of the travelling salesman problem*, Bell System Tech. J., 44, pp. 2245–2269.

P. C. LIU AND R. C. GELDMACHER (1976), *On the deletion of nonplanar edges of a graph*, Dept. of Electrical Engineering, Stevens Inst. of Tech., Hoboken, NJ.

Z. MANNA (1969), *The correctness of programs*, J. Comput. System Sci., 3, pp. 119–127.

H. C. MARTIN AND G. F. CAREY (1973), *Introduction to Finite Element Analysis*, McGraw-Hill, New York.

E. M. MCCREIGHT (1976), *A space-economical suffix tree construction algorithm*, J. Assoc. Comput. Mach., 23, pp. 262–272.

C. MCDIARMID (1976), *Determining the chromatic number of a graph*, Tech. Rep. STAN-CS-76-576, Computer Science Dept., Stanford Univ., Stanford, CA.

J. MCKAY AND E. REGENER (1974), *Algorithm 482: transitivity sets*, Comm. ACM, 17, p. 470.

A. R. MEYER AND L. STOCKMEYER (1972), *The equivalence problem for regular expressions with squaring requires exponential space*, Conf. Record, IEEE 13th Annual Symp. on Switching and automata Theory, Institute of Electrical and Electronics Engineers, New York, pp. 125–129.

R. MORRIS (1968), *Scatter storage techniques*, Comm. ACM, 11, pp. 38–44.

I. MUNRO (1971), *Efficient determination of the transitive closure of a directed graph*, Information Processing Lett., 1, pp. 56–58.

G. NEMHAUSER AND R. GARFINKEL (1972), *Integer Programming*, John Wiley, New York.

R. Z. NORMAN AND M. O. RABIN (1959), *An algorithm for a minimum cover of a graph*, Proc. Amer. Math. Soc., 10. pp. 315–319.

J. F. PACAULT (1974), *Computing the weak components of a directed graph*, SIAM J. Comput., 3, pp. 56–61.

S. V. PARTER (1961), *The use of linear graphs in Gaussian elimination*, this Review, 3, pp. 119–130.

M. PATERSON (1972), Private communication.

E. L. POST (1936), *Finite combinatory processes—formalism I*, J. Symbolic Logic, 1, pp. 103–105.

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

M. O. RABIN (1976), *Probabilistic algorithms, Algorithms and Complexity: New Directions and Recent Results*, J. F. Traub, ed., Academic Press, New York, pp. 21–40.

R. RIVEST AND J. VUILLEMIN (1975), *A generalization and proof of the Anderaa–Rosenburg conjecture*, Proc. Seventh Annual ACM Symp. on Theory of Computing, Association for Computing Machinery, New York, pp. 6–11.

D. J. ROSE (1970), *Triangulated graphs and the elimination process*, J. Math. Anal. Appl., 32, pp. 597–609.

——— (1973), *A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations*, Graph Theory and Computing, R. Read, ed., Academic Press, New York, pp. 183–217.

D. J. ROSE AND R. E. TARJAN (1977), *Algorithmic aspects of vertex elimination on directed graphs*, SIAM J. Appl. Math., to appear.

D. J. ROSE, R. E. TARJAN AND G. S. LUEKER (1976), *Algorithmic aspects of vertex elimination on graphs*, SIAM J. Comput., 5, pp. 266–283.

A. L. ROSENBURG (1973), *On the time required to recognize properties of graphs: a problem*, SIGACT News, 5, pp. 15–16.

C. RUNGE AND H. KÖNIG (1924), *Die Grundlehren der mathematischen Wissenschaften*, vol. 11, Springer, Berlin.

S. SAHNI (1974), *Computationally related problems*, SIAM J. Comput., 3, pp. 262–279.

R. W. H. SARGENT AND A. W. WESTERBERG (1964), *'Speed up' in chemical engineering design*, Trans. Inst. Chem. Engrs., 42, pp. 190–197.

J. E. SAVAGE (1976), *The Complexity of Computing*, John Wiley, New York.

M. SCHAEFER (1973), *A Mathematical Theory of Global Program Optimization*, Prentice-Hall, Englewood Cliffs, NJ.

C. P. SCHNORR (1977), *An algorithm for transitive closure with linear expected time*, unpublished manuscript, Fachbereich Mathematik, Universität Frankfurt, Frankfurt, Germany.

A. SCHÖNHAGE (1973), *Real-time simulation of multidimensional Turing machines by storage modification machines*, Project MAC Tech. Memorandum 37, Mass. Inst. of Tech., Cambridge.

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

R. SETHI (1975), *Complete register allocation problems*, SIAM J. Comput., 4, pp. 226–248.

——— (1976), *Scheduling graphs on two processors*, Ibid., 5, pp. 73–82.

R. W. SHIREY (1969), *Implementation and analysis of efficient graph planarity testing algorithms*, Ph.D. thesis, Univ. of Wisconsin, Madison.

P. M. SPIRA (1973), *A new algorithm for finding shortest paths in a graph of positive arcs in average time* $O(n^2 \log^2 n)$, SIAM J. Comput., 2, pp. 28–32.

L. J. STOCKMEYER AND A. R. MEYER (1973), *Word problems requiring exponential time: preliminary report*, Proc. Fifth Annual ACM Symp. on Theory of Computing, Association for Computing Machinery, New York, pp. 1–9.

V. STRASSEN (1969), *Gaussian elimination is not optimal*, Numer. Math., 13, pp. 354–356.
——— (1973), *Die Berechnungskomplexetät von elementarysymmetrischen Funktionen und von Interpolations koeffizienten*, Ibid., 20, pp. 238–251.
V. STRASSEN AND R. SOLOVAY (1977), *A fast Monte-Carlo test for primality*, SIAM J. Comput., 1, pp. 84–85.
R. E. TARJAN (1972), *Depth-first search and linear graph algorithms*, SIAM J. Comput., 1, pp. 146–160.
——— (1974a), *A new algorithm for finding weak components*, Information Processing Lett., 3, pp. 13–15.
——— (1974b), *Finding dominators in directed graphs*, SIAM J. Comput., 3, pp. 62–89.
——— (1974c), *Testing flow graph reducibility*, J. Comput. System Sci., 9, pp. 355–365.
——— (1975a), *Efficiency of a good but not linear disjoint set union algorithm*, J. Assoc. Comput. Mach., 22, pp. 215–225.
——— (1975b), *Applications of path compression on balanced trees*, Tech. Rep. STAN-CS-75-512, Computer Science Dept., Stanford Univ., Stanford, CA.
——— (1975c), *Solving path problems on directed graphs*, Tech. Rep. STAN-CS-75-528, Computer Science Dept., Stanford Univ., Stanford, CA.
——— (1976a), *Graph theory and Gaussian elimination*, Sparse Matrix Computations, J. R. Bunch and D. J. Rose, eds., Academic Press, New York, pp. 3–22.
——— (1976b), *Generalized nested dissection*, unpublished notes.
——— (1977), *Reference machines require non-linear time to maintain disjoint sets*, Proc. Ninth Annual ACM Symp. on Theory of Computing, Association for Computing Machinery, New York, pp. 18–29.
R. P. TEWARSON (1973), *Sparse Matrices*, Academic Press, New York.
A. M. TURING (1936–7), *On computable numbers, with an application to the Entscheidungs problem*, Proc. London Math. Society, ser. 2, 42, pp. 230–265; corrections, 43 (1937), pp. 544–546.
J. D. ULLMAN (1973a), *A fast algorithm for the elimination of common subexpressions*, Acta Informat., 2, pp. 191–213.
——— (1973b), *Polynomial complete scheduling problems*, Proc. Fourth Symp. on Operating System Principles, ACM Operating Systems Review, 7, Association for Computing Machinery, New York, pp. 96–101.
L. G. VALIANT (1975a), *General context-free recognition in less than cubic time*, J. Comput. System Sci., 10, pp. 308–315.
——— (1975b), *On non-linear lower bounds in computational complexity*, Proc. Seventh Annual ACM Symp. on Theory of Computing, Association for Computing Machinery, New York, pp. 45–53.
——— (1976), *Some conjectures relating to super linear complexity bounds*, unpublished manuscript, Centre for Computer Studies, University of Leeds, England.
P. VAN EMDE BOAS, R. KAAS AND E. ZIJLSTRA (1975), *Design and implementation of an efficient priority queue*, Mathematisch Centrum, Amsterdam.
J. VUILLEMIN (1977), *A data structure for manipulating priority queues*, Comm. ACM, to appear.
P. WEINER (1973), *Linear pattern matching algorithms*, IEEE 14th Annual Symp. on Switching and Automata Theory, Institute for Electrical and Electronics Engineers, New York, pp. 1–11.
S. WINOGRAD (1975), *The effect of the field of constants on the number of multiplications*, Proc. Sixteenth Annual Symp. on Foundations of Computer Science, Institute for Electrical and Electronics Engineers, New York, pp. 1–2.
——— (1976), *On computing the discrete Fourier transform*, Proc. Nat. Acad. Sci. U.S.A., 73, pp. 1005–1006.
N. WIRTH (1971), *The programming language Pascal*, Acta Informat., 1, pp. 35–63.
A. C. YAO (1975), *An $O(|E| \log \log |V|)$ algorithm for finding minimum spanning trees*, Information Processing Lett., 4, pp. 21–23.
——— (1976), *On the average behavior of set merging algorithms*, Proc. Eighth Annual ACM Symp. on Theory of Computing, Association for Computing Machinery, New York, pp. 192–195.
D. H. YOUNGER (1967), *Recognition and parsing of context-free languages*, Information and Control, 10, pp. 189–208.