OPERATIONS RESEARCH
Vol. 26, No. 5, September–October 1978

0030-364X/78/2605-0718 $01.25

Combinatorial Problems: Reducibility and Approximation

SARTAJ SAHNI

University of Minnesota, Minneapolis, Minnesota

ELLIS HOROWITZ

University of Southern California, Los Angeles, California

(Received October 1976; accepted January 1978)

Recent research in the theory of algorithms has determined that many classical operations research problems are computationally related; i.e., an efficient algorithm for one implies the existence of efficient algorithms for everyone or a proof that one is inherently difficult implies they are all so. This paper presents a tutorial of this concept. In contrast to other surveys it does so by selecting a few problems (knapsack, traveling salesperson, multiprocessor scheduling, and flow shop) and carefully shows how they are related. References to most other problems of interest to operations researchers are given. The second part of this paper is a survey on the relatively recent progress in the study of approximation algorithms. These algorithms hold great promise for they work fast and in many cases are guaranteed to work well. Again the techniques for devising and analyzing these algorithms are explained through the continued use of these examples.

THIS ISSUE, devoted to the interactions between computer science and operations research, provides an ideal opportunity for us. For as computer scientists, we find more and more that our own work is making incursions into what was heretofore considered strictly operations research territory. This has made us very aware of the different styles of research that these groups follow. This article presents, in tutorial fashion, our view of some of the major contributions that computer science has made to algorithm research. As such it stresses the kinds of questions and answers that computer scientists have devised about algorithms. Our treatment of topics is selective, but we intend to choose problems of long-time interest to operations researchers to show how recent results from computer science have complemented existing knowledge about these traditional problems.

Scholars of operations research have been studying algorithms for many years and so have computer scientists. Historically, the latter group has concentrated on sorting and searching for data processing applications, and numerical analysis for mathematical computation. As the discipline has matured, a greater research emphasis has been placed on studying the algorithm concept in general. With that growth has come renewed interest in classical operations research problems since algorithms for their solution are of great interest and variety. In this paper we intend to select some problems that are very familiar to readers of this journal (the knapsack problem, a scheduling problem, traveling salesperson problem, and flow-shop problem) and review the latest results about them from computer science.

What are the contributions of computer science to the study of algorithms? We list four major points:

(i) a great impetus to the idea that algorithms should be formally analyzed in addition to being empirically tested;
(ii) the study of how to express an algorithm stressing clarity and verification;
(iii) the development of the notion of reducibility, i.e. problems related in terms of their execution-time complexity;
(iv) the study of general techniques for devising and analyzing approximation algorithms.

In Section 1 we briefly discuss the major points that have been made regarding the exposition of algorithms. At the same time we introduce the basic ideas underlying the complexity analysis of an algorithm. In Section 2 the concept of a nondeterministic algorithm is introduced, followed by the notions of *NP*-hard and *NP*-complete. These concepts are the cornerstone of reducibility theory. We first indicate the importance of knowing that a problem is *NP*-complete. Then we use four well-known operations research problems to show how to resolve *NP*-completness. References to the literature that contains other relevant *NP*-complete problems are given.

Recognizing that the best-known algorithms for *NP*-complete and *NP*-hard problems are intractable, in the sense of worst-case performance, in Section 3 we survey some research that holds a great deal of promise for adequately solving these problems. This is the growing study of approximation algorithms. There are many different kinds of approximation algorithms. Some generate solutions that are guaranteed to be within some fixed fraction of the optimal solution. Others are parameterized and are capable of generating solutions with value as close to the optimal as desired. Finally, Section 4 surveys a relatively new approach to tackling *NP*-hard problems. This approach consists of designing fast algorithms that generate either optimal or near optimal solutions to almost all instances of a problem.

1. ALGORITHMS AND THEIR ANALYSIS

Writing Algorithms

Given n integer weights w_i, n profits p_i, and a positive integer M that is the maximum allowable weight, the knapsack optimization problem is

$$\max \textstyle\sum_{i=1}^{n} p_i x_i$$
$$\textstyle\sum_{i=1}^{n} w_i x_i \leqq M$$
$$x_i \in \{0, 1\},\ 1 \leqq i \leqq n.$$

Algorithm 1 implements a backtracking approach to the knapsack problem. The quantities W^*, P^* and $(x_1, \cdots, x_n)$ are the weight, profit and assignments of the best solution found so far. This algorithm was taken verbatim from a recent issue of an operations research publication. In many ways it is fairly typical of the manner in which algorithms are being presented today. Judged in the computer science context, it would be condemned as highly "unstructured."

Algorithm 1

```
 1. Initialize $W \leftarrow P \leftarrow k \leftarrow 0$ and $P^* \leftarrow -1$.
 2. Set $k \leftarrow k+1$ and $U \leftarrow W+w_k$. If $U>M$, then go to 4. Otherwise, set
    $W \leftarrow U$, $P \leftarrow P+P_k$, and $y_k \leftarrow 1$.
 3. If $k<N$, go to 2. Otherwise, set $W^* \leftarrow W$, $P^* \leftarrow P$ and $x_i \leftarrow y_i$ for $i=1, 2,$
    $\cdots, N$ and go to 6.
 4. Set $y_k \leftarrow 0$ and calculate a bound $B$ as described below in 8, 9, 10. If
    $B>P^*$, go to 3.
 5. Set $k \leftarrow k-1$. Exit if $k=0$.
 6. If $y_k=0$ go to 5.
 7. Set $W \leftarrow W-w_k$, $P \leftarrow P-p_k$ and go to 4.
    Calculation of B:
 8. Initialize $B \leftarrow P$; $C \leftarrow W$, and $i \leftarrow k$.
 9. Set $i \leftarrow i+1$. If $i>N$ return $B$; otherwise, set $C \leftarrow C+w_i$.
10. If $C<M$, set $B \leftarrow B+p_i$ and go to 9. Otherwise, return $B \leftarrow B+$
    $[1-(C-M)/w_i]p_i$.
```

In Algorithm 2 we see the same algorithm after some transformations have been applied to Algorithm 1. These alterations do not modify the computation sequence; they are purely a matter of form. The purpose of these transformations is to present this algorithm in as clear a manner as possible. The precise syntax we have chosen does not conform exactly to any existing programming language, but the style of the program is typical of a modern presentation.

Algorithm 2

```
    procedure KNAPSACK($M, N, w_1, \cdots, w_N, p_1, \cdots, p_N, W^*, V^*, x_1,$
    $\cdots, x_N$)

1       //input: $M$, the size of the knapsack
2                $N$, the number of weights and profits
3                $w_k$, the integer weights
```

```
                $p_k$, the corresponding profits
                (the $w_k$ and $p_k$ arranged so that $p_k/w_k \geqq p_{k+1}/w_{k+1}$, $1 \leqq k < N$)
    output:     $W^*$, the final weight of the knapsack
                $P^*$, the final maximum profit
                $x_k$, either zero or one indicating whether $w_k$ is not in (zero)
                or is in (one) the knapsack//
    $W \leftarrow P \leftarrow 0$; $k \leftarrow 1$; $P^* \leftarrow -1$
    loop
      while $W + w_k \leqq M$ and $k \leqq N$ do   //add in object $k$//
                $W \leftarrow W + w_k$; $P \leftarrow P + p_k$; $y_k \leftarrow 1$; $k \leftarrow k+1$
      repeat
      if $k > N$ then $P^* \leftarrow P$; $(x_1, \cdots, x_N) \leftarrow (y_1, \cdots, y_N)$; $k \leftarrow N$//update
                                                                   solution//
                else $y_k \leftarrow 0$   //object $k$ does not fit//
      endif
      while BOUND($P, W, k, M$) $\leqq P^*$ do   //backtracking step//
                while $y_k \neq 1$ and $k \neq 0$ do   //find last item in//
                  $k \leftarrow k-1$
                repeat
                if $k=0$ then stop endif   //search completed//
                $y_k \leftarrow 0$; $W \leftarrow W - w_k$; $P \leftarrow P - p_k$   //remove $k$th item//
      repeat
      $k \leftarrow k+1$
    repeat
  end KNAPSACK

  procedure BOUND($P, W, k, M$)
  //input: $P$, current profit total $= \sum_{i=1}^{k} p_i y_i$
                $W$, current weight total $= \sum_{i=1}^{k} w_i y_i$
                $k$, index of last removed item
                $M$, knapsack size
    output; $B$, a new profit//
    $B \leftarrow P$; $C \leftarrow W$
    for $i \leftarrow k+1$ to $N$ do
      $C \leftarrow C + w_i$
      if $C < M$ then $B \leftarrow B + p_i$
                else return($B + (1 - (C-M)/w_i) * p_i$)
      endif
    repeat
    return ($B$)
  end BOUND
```

What is better (or different) about this second version? The procedure KNAPSACK has been clearly named and its input and output variables are clearly listed. In lines 1–8 these variables are defined using the convention that anything between a pair of double slashes (//) is a comment. The executable statements on lines 9–25 are indented in order to reveal the computational nesting of the algorithm. In-line comments describe the purpose of key statements. In general, a program should be written so that it reads down the page, as with any piece of prose. The various statements used here make it easier to construct the algorithm in this way. All of these stylistic conventions and more can be found in Kernighan and Plauger [50].

An important consequence of these stylistic improvements is that it is now much easier to see what is happening in the algorithm. This in turn means it will be easier to prove and debug. To substantiate this claim, we now proceed to describe and prove the algorithm.

The algorithm implements a backtracking approach that assigns variables in the order they are given. The quantities W^*, P^*, and $(x_1, \cdots, x_n)$ are the weight, profit, and assignments of an optimal solution. The subroutine BOUND determines the maximum profit that can be obtained by completing the given partial filling $(y_1, \cdots, y_k)$. It is quite clear that once $y_1, \cdots, y_k$ have been fixed, we cannot do any better than fill in the objects $k+1$, $k+2$, $\cdots, j$ until we reach object $j+1$, which does not fit in. Now a fraction j that fills the knapsack is introduced. This observation is true because the objects are ordered such that $p_i/w_i \geqq p_{i+1}/w_{i+1}$, $1 \leqq i < N$. The loop lines 10–25 tries out all possible fillings $(y_1, \cdots, y_N)$, eliminating those prefixes $(y_1, \cdots, y_{k-1})$ that cannot possibly lead to a filling with value more than P^* (i.e., the highest solution value found so far). Such prefixes are easily identified since for them BOUND$(P, W, k-1, M) \leqq P^*$. Thus at the start of each iteration of the main loop we have BOUND$(P, W, k-1, M) > P^*$. This is certainly true for the first iteration, as $P^*=1$ and $p_i>0$, $w_i \leqq M$, $1 \leqq i \leqq n$. This condition is ensured for all successive iterations by the *while* loop of lines 17–23. The *while* loop of lines 11–13 places consecutive objects into the knapsack until either all objects are exhausted $(k=N)$ or we reach an object that does not fit in $(W+w_k>M)$. In case all objects have been exhausted, then $(y_1, \cdots, y_N)$ corresponds to a knapsack filling with profit $>P^*$. This follows from the observation that BOUND$(P, W, k-1, M) > P^*$ and objects k, $k+1$, $\cdots$, N have been placed into the knapsack. In case $W+w_k>M$, then y_k is set to 0. Now if BOUND$(P, W, k, M) \leqq P^*$, then the loop of lines 17–23 finds the next (i.e., lexically next) prefix for which BOUND$(P, W, k, M) > P^*$. In case there is no next prefix with this property, then the optimal solution has been found and the algorithm terminates in line 21.

Interesting reading on the subject of structured programming can be found in [12, 36, 54].

Analyzing Algorithms

Algorithm analysis can be conveniently thought of as consisting of two phases: a priori analysis and a posteriori testing. The latter occurs once the program has been completely written and shown to be free of errors. Results of such testing are generally given in the form of actual times (e.g., milliseconds) produced when the program was run on specific data sets. Results of a posteriori testing are tremendously useful, but they have limitations on the extent to which they may be interpreted. In particular, they are data specific, which means that the algorithm may behave very differently on other data sets. This is very common for many operations research problems such as the one we have just seen. Also, these empirical results are machine specific and even compiler specific as compilers vary in their ability to generate good code.

A priori analysis has none of these latter deficiencies. Its basic strategy is to take a completely specified algorithm and to study the frequency of execution of each step. The time for one execution of each step is assumed to be a constant, as long as those steps are sufficiently basic in nature —see [1] for more details. The initial strategy is to derive an upper bound on this frequency of execution, giving a formula that most accurately bounds the maximum (worst-case) number of executions of any step. These bounds are generally given using the "big-oh" notation.

DEFINITION. *$O(f(n))$ denotes the set of all $g(n)$ such that there exist positive constants c, n_0 such that $|g(n)| \leqq cf(n)$ for all $n \geqq n_0$.*

Most of the sorting and searching algorithms that computer scientists study have worst-case computing times that are bounded by a polynomial in n, the number of inputs. A typical collection of such times is $O(n)$, $O(n \log n)$ and $O(n^2)$. This is in marked contrast to most operations research problems, which typically take exponential time, e.g., $O(2^n)$ or $O(n!)$. The distinction between algorithms with polynomial versus non-polynomial time is important because, despite the tremendous increases in the speed of modern-day computers, algorithms whose time is exponential invariably take too long to execute, even on inputs of moderate size. However, as these times are upper bounds, for many problems they do not accurately describe the algorithm's performance on most data sets. The most striking example of this situation is the simplex method, which has been shown to be inefficient in the worst case [52], but which is known to work quite well on many problems. Since analyses of average computing time are far harder to obtain than worst-case bounds, often one relies on empirical testing to study the average computing time. See [34] for an empirical study of the knapsack problem.

Suppose we perform an a priori analysis of procedure BOUND used in KNAPSACK. The time for line 6 is taken to be a constant, or $O(1)$. Similarly, the time for one execution of every step is $O(1)$. Lines 7–14 represent a loop that can terminate in two ways. Ignoring the exit on line 10, the loop would be entered for $i=k+1$ to N or $N-k$ times. Since k can be as low as one, $O(N-1)$ bounds the total time. Similarly, for the second exit, if $C=0$, $k=1$, $j=N$ and $\sum w_i > M$, the time is $O(N-1)$.

The analysis of KNAPSACK is more complex since the number of iterations does not obviously depend on N. What is required is a clear understanding of the way elements are being selected for the knapsack. This is provided by the proof, which indicates that all possible subsets of the N elements may form an initial sequence. Elementary combinatorics tells us that this number is 2^N. Since no more than N elements can be added to the knapsack at each iteration, the computing time bound is $O(N2^N)$. This upper bound does not reflect the fact that the algorithm is really fast for many inputs [34]. A method for estimating the average efficiency of backtracking algorithms is given in [55]. A more complete discussion of how to analyze an algorithm and useful mathematical notation pertinent to such analyses can be found in [37].

2. REDUCIBLE PROBLEMS AND THEIR COMPLEXITY

In this section we introduce the notion of a nondeterministic algorithm and its complexity. From these definitions there follows the concept of the P and NP classes of problems and the notion of NP-complete and NP-hard. Some well-known combinatorial problems from operations research are shown to be NP-complete or NP-hard and hence computationally related. References are given to other papers that deal with other NP-hard or NP-complete problems of interest to operations researchers.

Nondeterministic Algorithms

Up to now the notion of algorithm that we have been using has the property that the result of every operation is uniquely defined. Algorithms with this property are termed *deterministic algorithms.* Such algorithms agree with the way programs are executed on a computer. From a theoretical framework we can remove this restriction on the outcome of every operation. We can allow algorithms to contain operations whose outcome is not uniquely defined but is limited to a specified set of possibilities. The machine executing such operations is allowed to choose any one of these outcomes. This leads to the concept of a *nondeterministic algorithm.* To specify such algorithms we introduce three new functions:

(i) *choice*(S) . . . arbitrarily chooses one of the elements of set S;
(ii) *failure* . . . signals an unsuccessful completion;
(iii) *success* . . . signals a successful completion.

Thus, the assignment statement $X \leftarrow$ *choice* $(1{:}n)$ could result in X being assigned any one of the integers in the range $[1, n]$. There is no rule specifying how this choice is to be made. The *failure* and *success* signals are used to define a computation of the algorithm. One way to view this computation is to say that whenever there is a set of choices that leads to a successful computation, then one such set of choices is made and the algorithm terminates successfully. A nondeterministic algorithm terminates unsuccesfully if and only if there exists no set of choices leading to a success signal. A machine capable of executing a nondeterministic algorithm in this way is called a *nondeterministic machine.*

Example 1. Consider the problem of searching for an element x in a given set of elements $A(1)$ to $A(n)$, $n \geqq 1$. We are required to determine an index j such that $A(j)=x$ or $j=0$ if x is not in A. A nondeterministic algorithm for this is

```
j←choice (1:n)
if A(j)=x then print (j); success endif
print ('0'); failure.
```

From the way a nondeterministic computation is defined, it follows that the number '0' can be output if and only if there is no j such that $A(j)=x$. The computing times for *choice, success,* and *failure* are taken to be $0(1)$. Thus the above algorithm is of nondeterministic complexity $0(1)$. Note that since A is not ordered, every deterministic search algorithm is of complexity at least $0(n)$.

Example 2 [Sorting]. Let $A(i)$, $1 \leqq i \leqq n$ be an unsorted set of positive integers. The nondeterministic algorithm NSORT(A, n), Algorithm 3, sorts the numbers into nondecreasing order and then outputs them in this order. An auxiliary array $B(1{:}n)$ is used for convenience. Line 1 initializes B to zero, although any value different from all $A(i)$ will do. In the loop of lines 2–6 each $A(i)$ is assigned to a position in B. Line 3 nondeterministically determines this position. Line 4 ascertains that $B(j)$ has not already been used. Thus, the order of the numbers in B is some permutation of the initial order in A. Lines 7–9 verify that B is sorted in nondecreasing order. A successful completion is achieved if and only if the numbers are output in nondecreasing order. Since there is always a set of choices at line 3 for such an output order, algorithm NSORT is a sorting algorithm. Its complexity is $0(n)$. Note that all deterministic sorting algorithms must have a complexity at least $0(n \log n)$.

Algorithm 3

```
    procedure NSORT(A, n)
      //sort n positive integers//
      integer A(n), B(n)
1     B←0 //initialize B to zero//
2     for i←1 to n do
3         j←choice(1:n)
4         if B(j)≠0 then failure endif
```

```
        B(j)←A(i)
    repeat
    for i←1 to n−1 do //verify order//
        if B(i)>B(i+1) then failure endif
    repeat
    print (B)
    success
  end NSORT
```

A deterministic interpretation of a nondeterministic algorithm can be made by allowing unbounded parallelism in computation. Each time a choice is to be made, the algorithm makes several copies of itself. One copy is made for each of the possible choices. Thus, many copies are executing at the same time. The first copy to reach a successful completion terminates all other computations. If a copy reaches a failure completion, then only that copy of the algorithm terminates. Note that the *success* and *failure* signals are equivalent to *stop* statements in deterministic algorithms. They may not be used in place of *return* statements. While this interpretation may enable one to better understand nondeterministic algorithms, it is important to remember that a nondeterministic machine does not make any copies of an algorithm every time a choice is to be made. Instead, it has the ability to select a "correct" element from the *choice* function (if such an element exists) every time a choice is to made. A "correct" element is defined relative to a shortest sequence of choices that leads to a successful termination. In case there is no sequence of choices leading to a successful termination, we shall assume that the algorithm terminates in one unit of time with output "unsuccessful computation." If there are many shortest sequences of choices leading to a succesful termination, a "correct" element is defined relative to any one of these sequences. Whenever successful termination is possible, a nondeterministic machine makes a sequence of choices that is a shortest sequence leading to a successful termination. Since the machine we are defining is fictitious, it is not necessary for use to concern ourselves with "how" the machine can make a "correct" choice at each step.

It is possible to construct nondeterministic algorithms for which many different choice sequences lead to a successful completion. Procedure NSORT of Example 2 is one such algorithm. If the numbers $A(i)$ are not distinct, then many different permutations will result in a sorted sequence. If NSORT were written to output the permutation used rather than the $A(i)$'s in sorted order, then its output would not be uniquely defined. We shall concern ourselves only with those nondeterministic algorithms that generate a unique output. In particular, we shall consider only *nondeterministic decision algorithms.* Such algorithms generate only a zero or one as their output; i.e., a binary decision is made. A successful completion is made if and only if the output is '1'. A '0' is output if and only if there is no sequence of choices leading to a successful completion. The output statement is implicit in the signals *success* and *failure.* No explicit output statements are permitted in a decision algorithm. Clearly, our earlier definition of a nondeterministic computation implies that the output from a decision algorithm is uniquely defined by the input parameters and the algorithm specification.

While the idea of a decision algorithm may appear very restrictive at this time, many optimization problems can be recast into decision problems with the property that the decision problem can be solved in polynomial time if and only if the corresponding optimization problem can. In other cases, we can at least make the statement that if the decision problem cannot be solved in polynomial time, then the optimization problem cannot either.

Example 3 [Max Clique]. A maximax complete subgraph of a graph $G=(V, E)$ is a *clique.* V is the set of vertices and E the set of edges of G. The size of the clique is the number of vertices in it. The max clique problem is to determine the size of a largest clique in G. The corresponding decision problem is to determine if G has a clique of size at least k for some given k. Let DCLIQUE(G, k) be a deterministic decision algorithm for the clique decision problem. If the number of vertices in G is n, the size of a max clique in G can be found by making several applications of DCLIQUE. DCLIQUE is used once for each k, $k=n, n-1, n-2, \cdots$ until the output from DCLIQUE is 1. If the time complexity of DCLIQUE is $f(n)$, then the size of a max clique can be found in time $n*f(n)$. Also, if the size of a max clique can be determined in time $g(n)$, then the decision problem may be solved in time $g(n)$. Hence, the max clique problem can be solved in polynomial time if and only if the clique decision problem can be solved in polynomial time.

Example 4 [0/1—Knapsack]. The knapsack decision problem is to determine if there is a 0/1 assignment of values to x_i, $1\leqq i\leqq n$ such that $\sum_{i=1}^{n} p_i x_i \geqq R$ and $\sum_{i=1}^{n} w_i x_i \leqq M$. R is a given integer. The p_i's and w_i's are non-negative integers. Clearly, if the knapsack decision problem cannot be solved in deterministic polynomial time, then the optimization problem cannot either.

Before we proceed further, it is necessary to arrive at a uniform parameter, n, to measure complexity. We shall assume that n is the length of the input to the algorithm. We will assume all inputs are integer. Rational inputs can be provided by specifying pairs of integers. Generally, the length of an input is measured assuming a binary representation; i.e. if the number 10 is to be input, then in binary it is represented as 1010. Its length is 4 (4 bits). In general, a positive integer k has a length of $\lfloor \log_2 k \rfloor + 1$ bits when represented in binary. The length of the binary representation of 0 is 1. The size or length, n, of the input to an algorithm is the sum of the lengths of the individual numbers being input. In case the input is given using a different representation (say radix r) then the length of a positive number k is $\lfloor \log_r k \rfloor+1$. Thus, in decimal notation $r=10$ and the number 100 has a length $\log_{10} 100+1=3$ digits. Since $\log_r k = \log_2 k/\log_2 r$, the length of any input using radix $r(r>1)$ representation is $c(r)*n$, where n is the length using a binary representation and $c(r)$ is a number that is fixed for a given r. When inputs are given using a radix $r=1$, we shall say the input is in unary form. In unary form the number 5 is input as 11111. Thus, the length of a positive integer k is k. It is important to observe that the length of a unary input is exponentially related to the length of the corresponding r-ary input for radix r, $r>1$.

Example 5 [Max Clique]. The input to the max clique problem may be provided as a sequence of edges and an integer k. Each edge in $E(G)$ is a pair of numbers (i, j). The size of the input for each edge (i, j) is $\lfloor \log_2 i \rfloor + \lfloor \log_2 j \rfloor + 2$ if a binary representation is assumed. The input size is $n = \sum_{(i,j)\in E(G), i<j} (\lfloor \log_2 i \rfloor + \lfloor \log_2 j \rfloor + 2) + \lfloor \log_2 k \rfloor + 1$. Note that if G has only one connected component, then $n \geqq |V|$. Thus, if this decision problem cannot be solved by an algorithm of complexity $p(n)$ for some polynomial $p(\ )$, then it cannot be solved by an algorithm of complexity $p(|V|)$.

Example 6 [0/1 Knapsack]. Assuming p_i, w_i, M and R are all integers, the input size for the knapsack decision problem is $m=\sum_{i=1}^{n} (\lfloor \log_2 p_i \rfloor + \lfloor \log_2 w_i \rfloor) + \lfloor \log_2 M \rfloor + \lfloor \log_2 R \rfloor + 2n + 2$. Note that $m \geqq n$. If the input is given in unary notation, then the input size s is $\sum p_i + \sum w_i + M + R$. Note that the knapsack decision and optimization problems can be solved in time $p(s)$ for some polynomial $p(\ )$. (See for example, the dynamic programming algorithm of [34].)

DEFINITION. *The time required by a nondeterministic algorithm performing on any given input is the minimum number of steps needed to reach a successful completion if there exists a sequence of choices leading to such a completion. A nondeterministic algorithm is of complexity $O(f(n))$ if for all inputs of size n, $n \geqq n_0$, that result in a sucessful completion the time required is at most $c \cdot f(n)$ for some constants c and n_0.*

In the above definition we assume that each computation step is of a fixed cost. In word-oriented computers this is guaranteed by the finiteness of each word. When this is not the case, it is necessary to consider the cost of individual instructions. Thus, the addition of two m bit numbers takes $0(m)$ time, their multiplication takes $0(m^2)$ time using classical multiplication, etc. To see the necessity of this consider procedure SUM (Algorithm 4). This is a deterministic algorithm for the sum of subsets decision problem. It uses an $M + 1$ bit word S. The ith bit in S is zero if and only if no subset of the integers $A(j)$, $1 \leqq j \leqq n$ sums to i. The bits are numbered 0 to M left to right. The function SHIFT shifts the bits in S, left by $A(i)$ bits. The total number of steps for this algorithm is only $0(n)$. However, each step moves M bits of data and would really take $0(M)$ time. Thus, the true complexity is $0(nM)$ and not $0(n)$.

Algorithm 4

```
procedure SUM(A, n, M)
    integer A(n)
    S←1 //S is an M bit word bit 0 is set to 1//
    for i←1 to n do
        S←S or SHIFT(S, A(i))
    repeat
    if Mth bit in S=0 then print ('no subset sums to M')
                      else print ('a subset sums to M')
    endif
end SUM
```

The virtue of the idea of nondeterministic algorithms is that often what would be very complex to write down deterministically is very easy to write nondeterministically. In fact, it is very easy to obtain polynomial-time nondeterministic algorithms for many problems that can be deterministically solved by a systematic search in a solution space of polynomial depth and exponential size.

Example 7 [Knapsack Decision Problem]. Procedure DKP (Algorithm 5) is a nondeterministic polynomial-time algorithm for the knapsack decision problem. Lines 1 to 3 assign 0/1 values to $X(i)$, $1 \leqq i \leqq n$. Line 4 checks to see if this assignment is feasible and if the resulting profit is at least R. A successful termination is possible if and only if the answer to the decision problem is yes. The time complexity is $0(n)$. If m is the input length using a binary representation, the time is $0(m)$.

Algorithm 5

```
  procedure DKP(P, W, n, M, R, X)
     integer P(n), W(n), R, X(n)
1    for i←1 to n do
2        X(i)←CHOICE (0, 1)
3    repeat
4    if $\sum_{1 \leq i \leq n} W(i)*X(i) > M$ or $\sum_{1 \leq i \leq n} P(i)*X(i) < R$ then failure
                                                           else success
5    endif
  end DKP
```

Example 8 [Satisfiability]. Let x_1, x_2, $\cdots$, denote boolean variables (their value is either true or false). Let x_i denote the negation of x_i. A *literal* is either a variable or its negation. A formula in the propositional calculus is an expression that can be constructed using literals and the operations *and* and *or*. Examples of such formulas are $(x_1 \wedge x_2) \vee (x_3 \wedge \bar{x}_4)$; $(x_3 \vee \bar{x}_4) \wedge (x_1 \vee \bar{x}_2)$. $\vee$ denotes *or* and $\wedge$ denotes *and*. A formula is in *conjunctive normal form* (CNF) if it is represented as $\wedge_{i=1}^{k} c_i$, where the c_i are clauses each represented as $\vee l_{i_j}$. The l_{i_j} are literals. It is in *disjunctive normal form* (DNF) if and only if it is represented as $\vee_{i=1}^{k} c_i$ and each clause c_i is represented as $\wedge l_{i_j}$. Thus $(x_1 \wedge x_2) \vee (x_3 \wedge \bar{x}_4)$ is in DNF, while $(x_3 \vee \bar{x}_4) \wedge (x_1 \vee \bar{x}_2)$ is in CNF. The *satisfiability* problem is to determine if a formula is true for any assignment of truth values to the variables. *CNF satisfiability* is the satisfiability problem for CNF formulas.

It is easy to obtain a polynomial-time nondeterministic algorithm that terminates successfully if and only if a given propositional formula $E(x_1, \cdots, x_n)$ is satisfiable. Such an algorithm could proceed by simply choosing one of the 2^n possible assignments of truth values to $(x_1, \cdots, x_n)$ and verifying that $E(x_1, \cdots, x_n)$ is true for that assignment.

Procedure EVAL (Algorithm 6) does this. The nondeterministic time required by the algorithm is $0(n)$ to choose the value of $(x_1, \cdots, x_n)$ plus whatever time is needed to deterministically evaluate E for that assignment.

```
Algorithm 6
  procedure EVAL (E, n)
    //Determine if the propositional formula E is satisfiable. The vari-
    ables are $x_i$, $1 \leqq i \leqq n$//
    boolean x(n)
    for i←1 to n do //choose a truth value assignment//
      $x_i$←choice(true, false)
    repeat
    if E($x_1$, ···, $x_n$) is true then success //satisfiable//
                                  else failure
    endif
  end EVAL
```

The Classes *NP*-hard and *NP*-complete

The idea that one problem could be reduced to another is not new. As far back as 1960, Dantzig [13] showed how certain problems could be transformed into zero-one integer programming problems. Bradley has shown how to transform zero-one integer problems into knapsack problems [3]. More recently Cook [10] and Karp [44] have formalized and extended the notion of problem reducibility to the point where we now know a host of computationally related problems, including the knapsack, traveling salesperson, multiprocessor scheduling, integer programming, flow-shop, and graph problems such as finding maximum cliques and colorability. We now proceed to define Cook's class of reducible problems.

An algorithm is of *polynomial complexity* if its computing time is $0(p(n))$ for every input of size n and some fixed polynomial $p(\ )$.

DEFINITION. *P is the set of all decision problems solvable by a deterministic algorithm in polynomial time. NP is the set of all decision problems solvable by a nondeterministic algorithm in polynomial time.*

Since deterministic algorithms are just a special case of nondeterministic ones, we can conclude that $P \subseteq NP$. What we do not know, and what has become perhaps the most famous unsolved problem in computer science, is whether $P=NP$ or $P \neq NP$.

Is it possible that for all of the problems in NP there exist polynomial-time deterministic algorithms that have remained undiscovered? This seems unlikely, at least because of the tremendous effort that has already been expended by so many people on these problems. Nevertheless, a proof that $P \neq NP$ is just as elusive and seems to require as yet undiscovered techniques. But as with many famous unsolved problems, they serve to generate other useful results, and the $P \overset{?}{=} P$ question is no exception.

In considering this problem Cook formulated the following question: Is there any single problem in NP such that if we showed it to be in P, then that would imply that $P=NP$? Cook answered his own question in the affirmative with the following theorem.

THEOREM 1 (Cook). *Satisfiability is in P if and only if $P=NP$.*

Proof. See [10] or [37].

We are now ready to define the NP-hard and NP-complete classes of problems. First we define the notion of reducibility.

DEFINITION. *If L_1 and L_2 are problems, L_1 reduces to L_2 (also written $L_1 \propto L_2$) if and only if there is a way to solve L_1 by a deterministic polynomial time algorithm using a deterministic algorithm that solves L_2 in polynomial time.*

This definition implies that if we have a polynomial time algorithm for L_2 then we can solve L_1 in polynomial time. One may readily verify that $\propto$ is a transitive relation (i.e., if $L_1 \alpha L_2$ and $L_2 \alpha L_3$ then $L_1 \alpha L_3$).

DEFINITION. *A problem L is NP-hard if and only if satisfiability reduces to L (satisfiability$\propto L$). A problem L is NP-complete if and only if it is NP-hard and $L \in NP$.*

It is easy to see that there are NP-hard problems that are not NP-complete. Only a decision problem can be NP-complete. However, an optimization problem may be NP-hard. Furthermore, if L_1 is a decision problem and L_2 an optimization problem, it is quite possible that $L_1 \propto L_2$. One may trivially show that the knapsack decision problem reduces to the knapsack optimization problem. For the clique problem one may easily show that the clique decision problem reduces to the clique optimization problem. In fact, we can also show that these optimization problems reduce to their corresponding decision problems. Yet optimization problems cannot be *NP*-complete while decision problems can. One can also show the existence of *NP*-hard decision problems that are not *NP*-complete.

DEFINITION. *Two problems L_1 and L_2 are said to be polynomially equivalent if and only if $L_1 \propto L_2$ and $L_2 \propto L_1$.*

In order to show a problem, L_2, *NP*-hard, it is adequate to show $L_1 \propto L_2$ where L_1 is some problem already known to be *NP*-hard. Since $\propto$ is a transitive relation, it follows that if satisfiability$\propto L_1$ and $L_1 \propto L_2$, then satisfiability$\propto L_2$. To show an *NP*-hard decision problem *NP*-complete, we have just to exhibit a polynomial-time nondeterministic algorithm for it. Later sections will show many problems to be *NP*-hard. While we shall restrict ourselves to decision problems, it should be clear that the corresponding optimization problems are also *NP*-hard.

Some *NP*-Complete Operations Research Problems

In this section we take four well-known problems and give very simple proofs that they are *NP*-complete. Part of the simplicity of the proofs derives from our assuming other problems as being *NP*-complete. Then in the last section we point to other papers and surveys of interesting *NP*-complete problems. The proofs for the first three problems will make use of the following known *NP*-complete problem [44].

Partition decision problem. Given n weights s_i determine if there exists a solution to $\sum_{i=1}^{n} s_i x_i = \frac{1}{2}\sum_{i=1}^{n} s_i$ where x_i equals zero or one.

Knapsack decision problem. Here we show that the knapsack decision problem is *NP*-complete. In Section 2 we exhibited a nondeterministic algorithm which establishes that knapsack is in *NP*. The partition problem can be shown reducible to the knapsack problem by the following argument: For any instance $s_1, \cdots, s_n$ of the partition problem, let $w_i = p_i = s_i$, $1 \leqq i \leqq n$ and $R = M = \frac{1}{2}\sum_{i=1}^{n} s_i$. If the answer to the knapsack decision problem is yes, then the answer is also yes for the partition problem and vice versa.

Scheduling decision problem. Given two identical processors and n jobs with times t_i, $1 \leqq i \leqq n$, we wish to find a schedule whose finish time is bounded by T.

We will show that the partition problem is reducible to scheduling, or partitions $\propto$ scheduling. Clearly, the scheduling problem is in *NP* because we can write a nondeterministic algorithm that guesses a schedule and then computes the finish time. Thus we only have to show how the partitions problem can be reduced to the scheduling problem. Let $t_i = s_i$ and $T = \frac{1}{2}\sum_{i=1}^{n} s_i$. If we let $x_i = 1$ if job i is scheduled on processor 1 and $x_i = 0$ otherwise, then the schedule's finish time is given by $\max\{\sum_{i=1}^{n} t_i x_i, 2T - \sum_{i=1}^{n} t_i x_i\}$. Clearly, there exists a schedule with finish time $\leqq T$ if and only if there is a partition of $s_1, \cdots, s_n$.

Flow-shop decision problem. Given $m > 2$ machines and n jobs each with m tasks, task i of job j must be performed on machine i and task i must be completed before task $i+1$ can start. The time required by task i of job j, is t_{ij}. Does there exist a schedule with finish $\leqq T$?

Again, we show this problem is *NP*-complete by reducing it to the partition problem. The problem is clearly in *NP*. The nondeterministic algorithm guesses the permutations of the tasks on each machine and

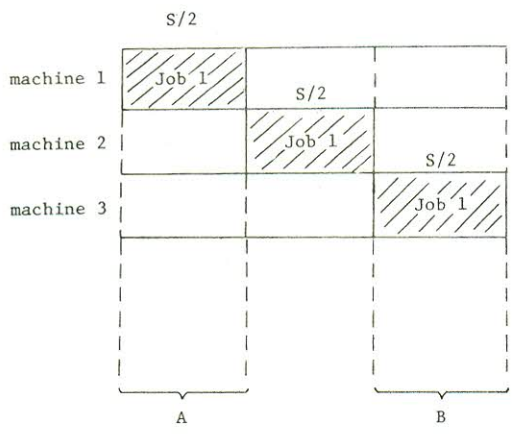

Figure 1

then checks to see if the schedule is feasible and meets the finish time criteria. Now given a partition problem we construct a flow-shop problem.

Let $m = 3$ and let $S = \sum_{i=1}^{n} s_i$. There are $n+1$ jobs, each with three tasks described by the three-tuples $(S/2, S/2, S/2)$, $(0, s_i, 0)$ $1 \leqq i \leqq n$ and $T = 3S/2$. Then any schedule with finish time $\leqq 3S/2$ must process job 1 continuously from start to finish (see Figure 1). The remaining jobs can be finished in time T if and only if there exists a partition of the s_i into two equal sets in the time periods A and B as indicated.

Hamiltonian cycle problem. Given a graph G, does it contain a cycle that includes every vertex? (A cycle is a path with no repeated vertices except the first and the last.)

This problem was shown to be *NP*-complete by Karp [44].

Traveling-salesperson decision problem. Given a complete graph on n vertices with costs attached to the edges, does there exist a Hamiltonian cycle with cost $\leqq M$?

Algorithm 7 is a nondeterministic algorithm for the traveling salesperson problem. It is clearly of nondeterministic polynomial complexity. Hence, the traveling-salesperson decision problem is in *NP*. To show this problem *NP*-complete, do the following.

Let $G=(V, E)$ be a graph. Define another graph $G'=(V', E')$ such that $V'= V$ and E' contains all possible edges (u', v'), where $u', v' \in V'$, $u' \neq v'$. Define the cost $c(u', v')$ of edge (u', v') to be 1 if $(u, v) \in E$ and 2 if $(u, v) \notin E$. Clearly, if $|V|=n$, then G' has a minimum cost tour of length n if and only if G has a Hamiltonian cycle. Hence Hamiltonian cycle α traveling-salesperson decision problem.

Algorithm 7

```
procedure TSP(G, n, M)
   //G is a complete graph with n vertices. TSP determines if a
     Hamiltonian cycle of cost $\leq M$ exists.//
   $N \leftarrow \{1, \cdots, n\}$
   $v_1 \leftarrow choice(N)$
   $N \leftarrow N - \{v_1\}$
   $T \leftarrow 0$
   for $k \leftarrow 2$ to n do
       $v_k \leftarrow choice(N)$                 //choose a new vertex//
       $N \leftarrow N - \{v_k\}$                   //remove it from set N//
       $T \leftarrow T + \mathrm{cost}((v_{k-1}, v_k))$   //add the new cost//
   repeat
   $T \leftarrow T + \mathrm{cost}((v_n, v_1))$
   if $T \leq M$  then success
               else failure
   endif
end TSP
```

Other reducible problems. Many combinatorial problems of interest to the operations research community have been shown to be *NP*-hard. *NP*-hard integer problems related to integer network flows and multicommodity flows may be found in [14, 64]. The timetable problem is shown *NP*-hard in [14]. Problems related to deterministic scheduling may be found in [4–6, 8, 21, 25, 28, 29, 44, 57, 63, 68, 72, 73]. *NP*-hard graph problems appear in [10, 19, 26, 33, 44, 45, 61]. Some problems concerned with n-person and 2-person games appear in [64, 71]. Integer programming, quadratic programming, and quadratic assignment type problems appear in [32, 45, 64, 70].

A more complete discussion of *NP*-hard and *NP*-complete concepts and a proof of Theorem 1 can be found in [37]. See [17] for more on nondeterministic algorithms; [53] contains a discussion of the the terminology used.

3. APPROXIMATION ALGORITHMS

Definitions and Motivation

In the preceding section we saw strong empirical evidence to support the claim that no *NP*-hard problem can be solved in polynomial time. Yet many *NP*-hard optimization problems have great practical significance and it is desirable to solve large instances of these problems in a "reasonable" amount of time. The best-known algorithms for *NP*-hard problems have a worst-case complexity that is exponential in the number of inputs. While the results of Section 2 may favor abandoning the quest for polynomial time algorithms, there is still plenty of room for improvement in an exponential algorithm. For example, we may look for algorithms with subexponential complexity, say $2^{\sqrt{n}}$ or $n^{\log n}$. The discovery of a subexponential algorithm for an *NP*-hard problem would increase the maximum problem size that could actually be solved. However, for large problem instances, even an $0(n^4)$ algorithm may require too much computational effort. Clearly, what is needed is an algorithm of low polynomial complexity (say, $0(n)$ *or* $0(n^2)$).

The use of heuristics in an existing algorithm may enable it to quickly solve a large instance of a problem, provided the heuristic "works" on that instance. A heuristic, however, does not "work" equally effectively on all problem instances. Exponential algorithms, even coupled with heuristics, will still show exponential behavior on some set of inputs. If we are to produce an algorithm of low polynomial complexity to "solve" an *NP*-hard optimization problem, then it will be necessary to relax the meaning of "solve." Instead of demanding an exact solution to the optimization problem, we require only an approximate solution. Recall that an exact solution is a feasible solution with optimal value. We shall define an approximate solution to be a feasible solution with value close to that of an exact solution. While at first one may discount the virtue of an approximate solution, one should bear in mind that often the data for the problem instance being solved are known only approximately. Hence, an approximate solution (provided its value is "sufficiently" close to that of an exact solution) may be no less meaningful than an exact solution. In the case of *NP*-hard problems it is very unlikely that we can find exact solutions using a feasible amount of computing time. An approximate solution may be all one can get in a reasonable amount of time.

Example 9. Consider the knapsack instance $n=3$, $M=100$, $\{p_1, p_2, p_3\} = \{20, 10, 19\}$ and $\{w_1, w_2, w_3\} = \{65, 20, 35\}$. $(x_1, x_2, x_3)=(1, 1, 1)$ is not a feasible solution as $\sum w_i x_i > M$. The solution $(x_1, x_2, x_3)=(1, 0, 1)$ is an optimal solution. Its value $\sum p_i x_i$ is 39. Hence $F^*(I)=39$ for this instance. The solution $(x_1, x_2, x_3)=(1, 1, 0)$ is suboptimal. Its value is $\sum p_i x_i=30$. This is a candidate for a possible output from an approximation algorithm. In fact, every feasible solution (in this case all three element 0/1 vectors other than (1, 1, 1) are feasible) is a candidate for output by an approximation algorithm. If the solution (1, 1, 0) is generated by an approximation algorithm in this instance, then $\hat{F}(I)=30$. $|F|^*(I)-\hat{F}(I)|=9$ and $|F^*(I)-\hat{F}(I)|/F^*(I)=0.3$.

Let L be a problem such as the knapsack problem or the traveling salesperson problem. Let I be an instance of the problem L and let $F^*(I)$ be the value of an optimal solution to I. An approximation algorithm will in general produce a feasible solution to I whose value $\hat{F}(I)$ is less than (greater than) $F^*(I)$ in case L is a maximization (minimization) problem. We define several categories of approximation algorithms.

Let $\mathscr{A}$ be an algorithm that generates a feasible solution to every instance I of a problem L. Let $F^*(I)$ be the value of an optimal solution to I and let $\hat{F}(I)$ be the value of the solution generated by $\mathscr{A}$.

DEFINITION. *$\mathscr{A}$ is an absolute approximation algorithm for problem L if and only if for every instance I of L, $|F^*(I)-\hat{F}(I)|\leqq k$ for some constant k.*

DEFINITION. *$\mathscr{A}$ is an f(n)-approximate algorithm if and only if for every instance I of size n, $|F^*(I)-\hat{F}(I)|/F^*(I)\leqq f(n)$. (Here it is assumed that $F^*(I)>0$. Also note that in case L is a maximization problem, then any algorithm capable of generating a feasible solution to every instance of P is a 1-approximate algorithm.)*

DEFINITION. *An ϵ-approximate algorithm is an f(n)-approximate algorithm for which $f(n)\leqq\epsilon$ for some constant ϵ.*

The next three difinitions are due to Garey and Johnson [22]. These definitions are in terms of an algorithm $\mathscr{A}$ that has ϵ as a parameter.

DEFINITION. *$\mathscr{A}(\epsilon)$ is an approximation scheme if for every given $\epsilon>0$ and problem instance I, $\mathscr{A}(\epsilon)$ generates a solution such that $|F^*(I)-\hat{F}(I)|/F^*(I)\leqq\epsilon$.*

DEFINITION. *An approximation scheme is a polynomial-time approximation scheme if for every fixed $\epsilon>0$ it has a computing time that is polynomial in the problem size.*

DEFINITION. *An approximation scheme whose computing time is a polynomial both in the problem size and in $1/\epsilon$ is a fully polynomial time approximation scheme.*

Clearly, the most desirable kind of approximation algorithm is an absolute approximation algorithm (provided it is sufficiently fast). Unfortunately, for most *NP*-hard problems it can be shown that fast algorithms of this type exist only if *P*=*NP*. Surprisingly, this statement is true even for the existence of *f*(*n*)-approximate algorithms for certain *NP*-hard problems.

Example 10. Consider the following approximation algorithm for the 0/1 knapsack problem: consider the objects in nonincreasing order of p_i/w_i. If object i fits, then set $x_i=1$; otherwise, set $x_i=0$. When this algorithm is used on the instance of Example 9, the objects are considered in the order 1, 3, 2. The result is $(x_1, x_2, x_3)=(1, 0, 1)$. The optimal solution is obtained. Now, consider the following instance: $n=2$, $(p_1, p_2)=(2, r)$, $(w_1, w_2)=(1, r)$ and $M=r$. When $r>1$, the optimal solution is $(x_1, x_2)=(0, 1)$. Its value, $F^*(I)$, is r. The solution generated by the approximation algorithm is $(x_1, x_2)=(1, 0)$. Its value, $\hat{F}(I)$, is 2. Hence $|F^*(I)-\hat{F}(I)|=r-2$. Our approximation algorithm is not an absolute approximation algorithm as there exists no constant k such that $|F^*(I)-\hat{F}(I)|\leqq k$ for all instances I. Furthermore, note that $|F^*(I)-\hat{F}(I)|/F^*(I)=1-2/r$. This approaches 1 as r becomes large. $|F^*(I)-\hat{F}(I)|/F^*(I)\leqq 1$ for every feasible solution to every knapsack instance. Since the above algorithm always generates a feasible solution, it is a 1-approximate algorithm. It is, however, not an ϵ-approximate algorithm for any ϵ, $\epsilon<1$.

Corresponding to the notions of absolute approximation algorithm and $f(n)$-approximate algorithms, we may define approximation problems in the obvious way. Thus, the 0.5-approximate knapsack problem is to find any 0/1 feasible solution with $|F^*(I)-\hat{F}(I)|/F^*(I)\leqq 0.5$.

As we shall see, approximation algorithms are usually just heuristics or rules that on the surface look like they might solve the optimization problem exactly. However, they do not. Instead, they only guarantee to generate solutions with value within some constant or some factor of the optimal value. Being heuristic in nature, these algorithms are very much dependent on the individual problem being solved.

Absolute Approximations

Planar graph coloring. There are very few *NP*-hard optimization problems for which polynomial-time absolute approximation algorithms are known. One problem is that of determining the minimum number of colors needed to color a planar graph $G=(V, E)$. It is known that every planar graph is four-colorable. One may easily determine if a graph is 0, 1 or 2 colorable. It is zero colorable if and only if $V=\phi$. It is 1-colorable if and only if $E=\phi$. G is two colorable if and only if it is bipartite. Determining if a planar graph is three colorable is *NP*-hard. However, all planar graphs are four colorable. An absolute approximation algorithm with $|F^*(I)-\hat{F}(I)|\leqq 1$ is easy to obtain. Algorithm 8 is such an algorithm. It finds an exact answer when the graph can be colored using at most two colors. Since we can determine whether or not a graph is bipartite in time $0(|V|+|E|)$, the complexity of the algorithm is $0(|V|+|E|)$.

Algorithm 8

```
colors ACOLOR(V, E)
```

```
  //determine the minimum number of color needed to color the
    planar graph $G=(V, E)$//
  case
    :$V=\phi$: return (0)
    :$E=\phi$: return (1)
    :$G$ is bipartite: return (2)
    :else: return (4)
  endcase
end ACOLOR
```

Maximum programs stored. Assume that we have n programs and two tapes each of length L. Let l_i be the amount of tape needed to store the ith program. Determining the maximum number of these n programs that can be stored on the two tapes (without splitting a program over the tapes) is NP-hard.

THEOREM 2. *Partition α maximum programs stored.*

Proof. Let $\{s_1, \cdots, s_n\}$ define an instance of the partition problem. We may assume $\sum s_i=2T$. Define an instance of the maximum program problem as follows: $L=T$ and $l_i=s_i$, $1 \leqq i \leqq n$. Clearly, $\{s_1, \cdots, s_n\}$ has a partition if and only if all n programs can be stored on two tapes.

By considering programs in order of nondecreasing tape requirement l_i, we can obtain a polynomial-time absolute approximation algorithm. Procedure PSTORE assumes $l_1 \leqq \cdots \leqq l_n$ and assigns programs to tape 1 so long as enough space remains on this tape. Then it begins assigning programs to tape 2. In addition to the time needed to initially sort the programs into nondecreasing order of l_i, $0(n)$ time is needed to obtain the storage assignment.

Algorithm 9. Approximation algorithm to store programs

```
procedure PSTORE ($l$, $n$, $L$)
  //assume $l_i \leqq l_{i+1}$, $1 \leqq i < n$//
    $i \leftarrow 1$
    for $j \leftarrow 1$ to 2 do
      sum$\leftarrow$0 //amount of tape $j$ already assigned//
      while sum$+l_i \leqq L$ do
        print ('store program', $i$, 'on tape', $j$)
        sum$\leftarrow$sum$+l_i$
        $i \leftarrow i+1$; if $i>n$ then return endif
      repeat
    repeat
end PSTORE
```

THEOREM 3. *Let I be any instance of the program storage problem. Let $F^*(I)$ be the maximum number of programs that can be stored on two* *tapes of length L each. Let $\hat{F}(I)$ be the number of programs stored using procedure PSTORE. Then $|F^*(I) - \hat{F}(I)| \leqq 1$.*

Proof. Assume that k programs are stored when Algorithm 9 is used. Then $\hat{F}(I) = k$. Consider the program storage problem when only one tape is available. In this case, considering programs in order of nondecreasing tape requirement maximizes the number of programs stored. Assume that p programs get stored when this strategy is used on a single tape of length $2L$. If $p > k+1$, then $\sum_1^{k+2} l_i \leqq 2L$. Let q be the number of programs assigned to tape 1 by Algorithm 9. Since $\sum_1^{q+1} l_i > L$, it follows that $\sum_1^q l_i + l_{k+1} > L$. Also, since $\sum_{q+1}^{k+1} l_i > L$, it follows that $\sum_{q+1}^{k} l_i + l_{k+2} > L$. Hence $\sum_1^{k+2} l_i > 2L$. This contradicts the earlier relation $\sum_1^{k+2} l_i \leqq 2L$. Hence $p \leqq k+1$. It is easy to verify that $k \leqq F^*(I) \leqq p$. Hence $|F^*(I) - \hat{F}(I)| \leqq 1$.

NP-Hard Absolute Approximations

The absolute approximation algorithms for the planar graph coloring and the maximum program storage problems are very simple and straightforward. Thus, one may expect that polynomial-time absolute approximation algorithms exist for most other *NP*-hard problems. Unfortunately, for the majority of *NP*-hard problems one can provide very simple proofs to show that a polynomial-time absolute approximation algorithm exists if and only if a polynomial-time exact algorithm does. Let us look at some sample proofs.

THEOREM 4. *Knapsack α absolute approximation knapsack.*

Proof. Assume we have a polynomial-time absolute approximation algorithm, $\mathscr{A}$, for the knapsack problem and that this algorithm guarantees solutions such that $|F^*(I) - \hat{F}(I)| \leqq k$. Any instance I of the knapsack problem may be stated as

$$\max \sum_{i=1}^n p_i x_i,\ \sum_{i=1}^n w_i x_i \leqq M \text{ and } x_i = 0 \text{ or } 1,\ 1 \leqq i \leqq n.$$

From this instance I, we can construct another instance I' as below

$$\max \sum_{i=1}^n (k+1) p_i x_i,\ \sum_{i=1}^n w_i x_i \leqq M,\ x_i = 0 \text{ or } 1,\ 1 \leqq i \leqq n.$$

It should be clear that I and I' have the same feasible and optimal solutions. Furthermore, any solution to I' that is generated by algorithm $\mathscr{A}$ is such that $F^*(I') - \hat{F}(I') = 0$ and so is an optimal solution to both I and I'. Moreover, the length of I' is at most $(\log k)$(length of I). Hence, from a polynomial-time absolute approximation algorithm for the knapsack problem we can obtain a polynomial-time exact algorithm.

Now consider the problem of obtaining a maximum clique of an undirected graph. The following theorem shows that obtaining a polynomial-time absolute approximation algorithm for this problem is as hard as obtaining a polynomial-time algorithm for the exact problem.

THEOREM 5. *Max clique $\propto$ absolute approximation max clique.*

Proof. Assume that the algorithm for the absolute approximation problem gets solutions such that $|F^*(I)-\hat{F}(I)|\leq k$. From any given graph $G=(V, E)$, we construct another graph $G'=(V', E')$ such that G' consists of $k+1$ copies of G connected together such that there is an edge between every two vertices in distinct copies of G; i.e., if $V=\{v_1, \cdots, v_n\}$, then $V'=\cup_{i=1}^{k+1}\{v_1^{i}, \cdots, v_n^{i}\}$ and $E'=(\cup_{i=1}^{k+1}\{(v_p^{i}, v_r^{i})\,|\,(v_p, v_r)\in E\})\cup\{(v_p^{i}, v_r^{j})\,|\,i\neq j\}$. Clearly, the maximum clique size in G is q if and only if the maximum clique size in G' is $(k+1)q$. Further, any clique in G' that is within k of the optimal clique size in G' must contain a sub-clique of size

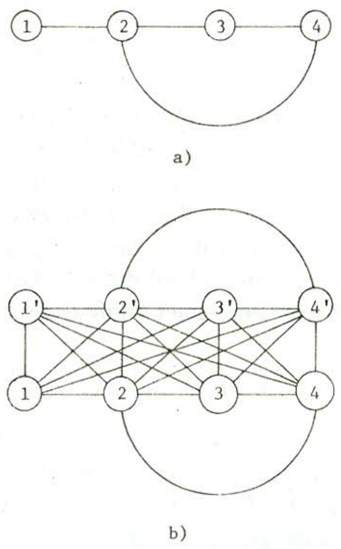

Figure 2. Graphs for Example 11.

q that is a clique of size q in G. Hence, we can obtain a maximum clique for G from a k-absolute approximate maximum clique for G'.

Example 11. Figure 2(b) shows the graph G' that results when the construction of Theorem 5 is applied to the graph of Figure 2(a). We have assumed $k=1$. The graph of Figure 2(a) has two cliques. One consists of the vertex set $\{1, 2\}$ and the other $\{2, 3, 4\}$. Thus, an absolute approximation algorithm for $k=1$ could output either of the two as solution cliques. In the graph of Figure 2(b), however, the two cliques are $\{1, 2, 1', 2'\}$ and $\{2, 3, 4, 2', 3', 4'\}$. Only the latter may be output. Hence an absolute approximation algorithm for $k=1$ will output the maximum clique.

ϵ-Approximations

Scheduling independent tasks. Obtaining minimum finish-time schedules on m, $m>2$ identical processors is *NP*-hard. There exists a very simple scheduling rule that generates schedules with a finish time very close to that of an optimal schedule. An instance I of the scheduling problem is defined by a set of n task times, t_i, $1 \leqq i \leqq n$, and m, the number of processors. The scheduling rule we are about to describe is known as the LPT (longest processing time) rule. An LPT schedule is a schedule that results from this rule.

DEFINITION. *An LPT schedule is one that is the result of an algorithm that whenever a processor becomes free, assigns that task whose task time is the largest of those tasks not yet assigned. Ties are broken in an arbitrary manner.*

It is possible to implement the LPT rule so that at most $0(n \log n)$ time is needed to generate an LPT schedule for n tasks on m processors. The preceding example shows that the LPT rule does not necessarily generate an optimal schedule. How bad can LPT schedules be relative to optimal schedules? This question is answered by the following theorem, which is due to Graham [30].

THEOREM 6. *Let $F^*(I)$ be the finish time of an optimal m processor schedule for instance I of the task scheduling problem. Let $\hat{F}(I)$ be the finish time of an LPT schedule for the same instance. Then*

$$|F^*(I)-\hat{F}(I)|/F^*(I) \leqq 1/3-1/(3m).$$

Proof. See [30].

Efficient ϵ-approximate algorithms exist for many scheduling problems. Relevent references are [9, 18, 20, 27, 30, 31, 38, 40, 41].

Bin packing. In this problem we are given n objects that have to be placed in bins of equal capacity L. Object i requires l_i units of bin capacity. The objective is to determine the minimum number of bins needed to accommodate all n objects. No object may be placed partly in one bin and partly in another.

The bin-packing problem may be regarded as a variation of the scheduling problem considered earlier. The bins represent processors and L is the time by which all tasks must be completed. l_i is the processing requirement of task i. The problem is to determine the minimum number of processors needed to accomplish this. An alternative interpretation is to regard the bins as tapes. L is the length of a tape and l_i the tape length needed to store program i. The problem is to determine the minimum number of tapes needed to store all n programs. Clearly, many interpretations exist for this problem.

THEOREM 7. *The bin-packing problem is NP-hard.*

Proof. To see this consider the partition problem. Let $\{s_1, \cdots, s_n\}$ be an instance of the partition problem. Define an instance of the bin-packing problem as follows: $l_i = s_i$, $1 \leqq i \leqq n$ and $L = \sum s_i/2$. Clearly, the minimum number of bins needed is 2 if and only if there is a partition for $\{s_1, \cdots, s_n\}$.

One can devise many simple heuristics for the bin-packing problem. These will not, in general, obtain optimal packings. They will, however, obtain packings that use only a "small" fraction of bins more than an optimal packing. Four simple heuristics are:

I. *First Fit* (FF). Index the bins 1, 2, $\cdots$. All bins are initially filled to level zero. Objects are considered for packing in the order 1, $\cdots$, n. To pack object i, find the least index j such that bin j is filled to a level r, $r \leqq L - l_i$. Pack i into bin j. Bin j is now filled to level $r + l_i$.

II. *Best Fit* (BF). The initial conditions on the bins and objects are the same as for FF. When object i is being considered, find the least j such that bin j is filled to a level r, $r \leqq L - l_i$ and r is as large as possible. Pack i into bin j. Bin j is now filled to level $r + l_i$.

III. *First Fit Decreasing* (FFD). Reorder the objects so that $l_i \geqq l_{i+1}$, $1 \leqq i < n$. Now use First Fit to pack the objects.

IV. *Best Fit Decreasing* (BFD). Reorder the objects so that $l_i \geqq l_{i+1}$, $1 \leqq i < n$. Now use Best Fit to pack the objects.

THEOREM 8. *Let I be an instance of the bin-packing problem and let $F^*(I)$ be the minimum number of bins needed for this instance. The packing generated by either FF or BF uses no more than* $17/10F^*(I)+2$ *bins. The packing generated by either FFD or BFD uses no more than $11/9F^*(I)+4$ bins. These bounds are the best possible bounds for the respective algorithms.*

Proof. See [43].

Efficient ϵ-approximation algorithms are known for many other problems. Some references are [7, 11, 15, 16, 42, 46, 48, 49].

NP-hard ϵ-Approximation Problems

As in the case of absolute approximations, there exist many *NP*-hard optimization problems for which the corresponding ϵ-approximation problems are also *NP*-hard. Let us look at some of these. To begin, consider the traveling salesperson problem. We shall make use of the Hamiltonian cycle problem, which was shown *NP*-complete in [44].

THEOREM 9. *Hamiltonian cycle $\propto$ ϵ-approximate traveling salesperson.*

Proof. Let $G = (N, A)$ be any graph. Construct the complete graph $G_1(V,$

E) such that $V=N$ and $E=\{(u, v)|u, v\in V \text{ and } u\neq v\}$. Define the weighting function $w{:}E\rightarrow Z$ to be

$$w(u, v) = \begin{cases} 1 \text{ if } (u, v)\in A \\ k \text{ otherwise.} \end{cases}$$

Let $n=|N|$. For $k>1$, the traveling salesperson problem on G_1 has a solution of length n if and only if G has a Hamiltonian cycle. Otherwise, all solutions to G_1 have length $\geqq k+n-1$. If we choose $k\geqq(1+\epsilon)n$, then the only solutions approximating a solution with value n (if there was a Hamiltonian cycle in G_1) also have length n. Consequently, if the ϵ-approximate solution has length $\leqq(1+\epsilon)n$, then it must be of length n. If it has length $>(1+\epsilon)n$, then G has no Hamiltonian cycle.

Another NP-hard ϵ-approximation problem is the 0/1 integer programming problem. In the optimization version of this problem we are given a linear optimization function $f(x)=\sum_{i=1}^{n}p_ix_i+p_0$. We are required to find a 0/1 vector $(x_1, \cdots, x_n)$ such that $f(x)$ is optimized (either maximized or minimized) subject to the constraints that $\sum_{j=1}^{n}a_{ij}x_j\leqq b_i$, $1\leqq i\leqq k$. k is the number of constraints. Note that the 0/1-knapsack problem is a special case of the 0/1 integer programming problem just described. Hence, this problem is also NP-hard. We now show that the corresponding ϵ-approximation problem is NP-hard for all ϵ, $\epsilon>0$. This is true even when there is only one constraint (i.e., $k=1$).

THEOREM 10. *Partition α ϵ-approximate integer programming.*

Proof. Let $(a_1, \cdots, a_n)$ be an instance of the partition problem. Construct the following 0/1 integer program:

$$\begin{aligned} &\min 1+k(m-\textstyle\sum_{r=1}^{n}a_ix_i),\\ &\textstyle\sum_{r=1}^{n}a_ix_i\leqq m,\\ &x_i=0 \text{ or } 1,\ 1\leqq i\leqq n \end{aligned}$$

where $m=\sum a_i/2$. The value of an optimal solution is 1 if and only if the a_i's have a partition. If they do not, then every optimal solution has a value at least $1+k$. Suppose there is a polynomial-time ϵ-approximate algorithm for the 0/1 integer programming problem for some ϵ, $\epsilon>0$. Then, by choosing $k>\epsilon$ and using the above construction, this approximation algorithm can be used to solve, in polynomial time, the partition problem. The given partition instance has a partition if and only if the ϵ-approximate algorithm generates a solution with value 1. All other solutions have value $\hat{F}(I)$ such that $|F^*(I)-\hat{F}(I)|/F^*(I)\geqq k>\epsilon$.

As a final example of an ϵ-approximation problem that is NP-hard for all ϵ, $\epsilon>0$, consider the quadratic assignment problem. In one interpretation this problem is concerned with optimally locating m plants. There are n possible sites for these plants, $n\geqq m$. At most one plant may be located in any of these n sites. We shall use x_{ik}, $1\leqq i\leqq n$, $1\leqq k\leqq m$ as mn 0/1 variables. $x_{ik}=1$ if and only if plant k is to be located at site i. The location of the plants is to be chosen so as to minimize the total cost of transporting goods between plants. Let d_{kl} be the amount of goods to be transported from plant k to plant l. $d_{kk}=0$, $1 \leqq k \leqq m$. Let c_{ij} be the cost of transporting one unit of the goods from site i to site j. $c_{ii}=0$, $1 \leqq i \leqq n$. The quadratic assignment problem has the following mathematical formulation:

$$\max f(x)=\sum_{i,j=1}^{n} \sum_{k,l=1}^{m} c_{ij} d_{kl} x_{ik} x_{jl}$$

$$\text{(a) } \sum_{k=1}^{m} x_{ik} \leqq 1, \quad 1 \leqq i \leqq n$$

$$\text{(b) } \sum_{i=1}^{n} x_{ik}=1, \quad 1 \leqq k \leqq m$$

$$\text{(c) } x_{ik}=0, 1 \text{ for all } i, k$$

$$c_{ij}, d_{kl} \geqq 0, \quad 1 \leqq i, j \leqq n, \quad 1 \leqq k, l \leqq m$$

Condition (a) ensures that at most one plant is located at any site. Condition (b) ensures that every plant is located at exactly one site. $f(x)$ is the total transportation cost.

THEOREM 11. *Hamiltonian cycle* $\propto$ *ϵ-approximate quadratic assignment.*

Proof. Let $G=(N, A)$ be an undirected graph with $m=|N|$. The following quadratic assignment instance is constructed from G: $n=m$

$$c_{ij}=\begin{cases} 1 & i=(j \bmod m)+1, \ 1 \leqq k, j \leqq m \\ 0 & \text{otherwise.} \end{cases}$$

$$d_{kl}=\begin{cases} 1 & \text{if } (k, l) \in A, \ 1 \leqq k, l \leqq m \\ \omega & \text{otherwise.} \end{cases}$$

The total cost, $f(\gamma)$, of an assignment, γ, of plants to locations is $\sum_{i=1}^{n} c_{ij} d_{\gamma(i)\gamma(j)}$ where $j=(i \bmod m)+1$ and $\gamma(i)$ is the index of the plant assigned to location i. If G has a Hamiltonian cycle $i_1, \cdots, i_n, i_1$, then the assignment $\gamma(j)=i_j$ has a cost $f(\gamma)=m$. In case G has no Hamiltonian cycle, then at least one of the values $d_{\gamma(i),\gamma(i \bmod m+1)}$ must be ω and so the cost becomes $\geqq m+\omega-1$. Choosing $\omega>(1+\epsilon)m$ results in optimal solutions with a value of m if G has a Hamiltonian cycle and value $>(1+\epsilon)m$ if G has no Hamiltonian cycle. Thus, from an ϵ-approximate solution it can be determined whether or not G has a Hamiltonian cycle.

Many other ϵ-approximation problems are known to be *NP*-hard. While the three problems just discussed were *NP*-hard for all ϵ, $\epsilon>0$, it is quite possible that an ϵ-approximation problem be *NP*-hard only for ϵ in some range, say, $0<\epsilon<r$. For $\epsilon>r$ there may exist simple polynomial-time approximation algorithms.

In [69] and [70] many other ϵ-approximation problems are shown to be *NP*-hard for all ϵ, $\epsilon>0$. In [23] the graph coloring problem is shown to be *NP*-hard for all ϵ, $0<\epsilon \leqq 1$.

Polynomial-Time Approximation Schemes

Scheduling independent tasks. We have seen that the LPT rule leads to a $(1/3-1/(3m))$-approximate algorithm for the problem of obtaining an m processor schedule for n tasks. A polynomial-time approximation scheme is also known for this problem. This scheme relies on the following scheduling rule: (i) Let k be some specified and fixed interger. (ii) Obtain an optimal schedule for the k longest tasks. (iii) Schedule the remaining $n-k$ tasks using the LPT rule.

Example 12. Let $m=2$; $n=6$ $(t_1, t_2, t_3, t_4, t_5, t_6)=(8, 6, 5, 4, 4, 1)$ and $k=4$. The four longest tasks have task times 8, 6, 5 and 4, respectively. An optimal schedule for these has finish time 12 (Figure 3(a)). When the remaining two tasks are scheduled using the LPT rule, the schedule of Figure 3(b) results. This has finish time 15. Figure 3(c) shows an optimal schedule. This has finish time 14.

THEOREM 12. (Graham). *Let I be an m processor instance of the scheduling problem. Let $F^*(I)$ be the finish time of an optimal schedule for I and let $\hat{F}(I)$ be the length of the schedule generated by the above scheduling rule. Then*

$$|F^*(I)-\hat{F}(I)|/F^*(I)\leqq(1-1/m)/(1+\lfloor k/m\rfloor).$$

Proof. See [29].

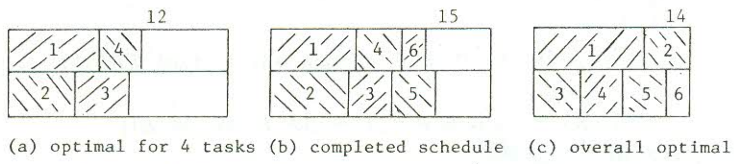

Figure 3. Using the approximation scheme with $k=4$.

Using the result of Theorem 12, we can construct a polynomial-time ϵ-approximation scheme for the scheduling problem. This scheme has ϵ as an input variable. For any input ϵ it computes an integer k such that $\epsilon<(1-1/m)/(1+\lfloor k/m\rfloor)$. This defines the k to be used in the scheduling rule described above. Solving for k, we obtain that any integer k, $k>(m-1)/\epsilon-m$ will guarantee ϵ-approximate schedules. The time required to obtain such schedules, however, depends mainly on the time needed to obtain an optimal schedule for k tasks on m machines. With a branch-and-bound algorithm, this time is $0(m^k)$. The time needed to arrange the tasks such that $t_i\geqq t_{i+1}$ and also to obtain the LPT schedule for the remaining $n-k$ tasks is $0(n \log n)$. Hence the total time needed by the ϵ-approximate scheme is $0(n \log n+m^k)=0(n \log n+m^{((m-1)/\epsilon-m))})$. Since this time is not polynomial in $1/\epsilon$ (it is exponential in $1/\epsilon$), this approximation scheme is not a fully polynomial-time approximation scheme. It is a polynomial-time approximation scheme (for any fixed m) as the computing time is polynomial in the number of tasks n.

0/1—Knapsack

The 0/1-knapsack heuristic proposed in Example 10 does not result in an ϵ-approximate algorithm for any ϵ, $0<\epsilon<1$. Consider the heuristic described by procedure ϵ-APPROX (Algorithm 10). In this procedure P and W are the sets of profits and weights, respectively. It is assumed that $p_i/w_i \geqq p_{i+1}/w_{i+1}$, $1 \leqq i < n$. M is the knapsack capacity and k a non-negative integer. In the loop of lines 2–5, all $\sum_{i=1}^{k}\binom{n}{i}$ different subsets, I, consisting of at most k of the n objects are generated. If the currently generated subset I is such that $\sum_{i\in I} w_i > M$, it is discarded (as it is infeasible). Otherwise, the space remaining in the knapsack (i.e., $M - \sum_{i\in I} w_i$) is filled using the heuristic described in Example 10. This heuristic is stated more formally as function L (Algorithm 11).

Algorithm 10. Heuristic algorithm for knapsack problem.

```
   procedure ε-APPROX(P, W, M, n, k)
      //(i) the size of a combination is the number of objects in it;
        (ii) the weight of a combination is the sum of the weights of the
             objects in that combination;
       (iii) k is a non-negative integer that defines the order of the
             algorithm//
1            PMAX←0;
2            for all combinations I of size ≤k and weight ≤M do
3               $P_I \leftarrow \sum_{i\in I} p_i$
4               PMAX←max {PMAX, $P_I + L(I, P, W, M, n)$}
5            repeat
6  end ε-APPROX
```

Algorithm 11. Subalgorithm for procedure ϵ-APPROX.

```
  function L(I, P, W, M, n)
     S←0; i←1; T←M− $\sum_{i\in I} w_i$ //initialize//
     for i←1 to n do
        if i∉I and $w_i$≦T then S←S+$p_i$
                                 T←T−$w_i$
        endif
     repeat
     return (S)
  end L
```

Example 13. Consider the knapsack problem instance with n=8 objects, size of knapsack=M=110, P={11, 21, 31, 33, 43, 53, 55, 65} and W={1, 11, 21, 23, 33, 43, 45, 55}.

The optimal solution is obtained by putting objects 1, 2, 3, 5, and 6 into the knapsack. This results in an optimal profit, P^*, of 159 and a weight of 109.

We obtain the following approximations for different k:

(a) $k=0$, PMAX is just the lower bound solution $L(\emptyset, P, W, M, n)$; PMAX=139; $x=(1, 1, 1, 1, 1, 0, 0, 0)$; $W=89$; $(P^*-\text{PMAX})/P^*=20/159=0.126$.
(b) $k=1$, PMAX=151; $x=(1, 1, 1, 1, 0, 0, 1, 0)$; $W=101$; $(P^*-\text{PMAX})/P^*=8/159=0.05$.
(c) $k=2$, PMAX$=P^*=159$, $x=(1, 1, 1, 0, 1, 1, 0, 0)$; $W=109$.

Table I gives the details for $k=1$. It is interesting to note that the combinations $I=\{1\}, \{2\}, \{3\}, \{4\}, \{5\}$ need not be tried since for $I=\{\phi\}$ x_6 is the first x_i that is 0 and so these combinations will yield the same

TABLE I
EXPANSION OF EXAMPLE 13, FOR $k=1$

I	PMAX	P_I	R_I	L	PMAX= max {PMAX, P_I + L}	x_{optimal}
ϕ	0	11	1	128	139	(1,1,1,1,1,0,0,0)
6	139	53	43	96	149	(1,1,1,1,0,1,0,0)
7	149	55	45	9	151	(1,1,1,1,0,0,1,0)
8	151	65	55	63	151	(1,1,1,1,0,0,1,0)

* Note that rather than update x_{optimal} it is easier to update the optimal I and recompute x_{optimal} at the end.

PMAX as $I=\{\phi\}$. This will be true for all combinations I that include only objects for which x_i was 1 in the solution for $I=\{\phi\}$.

THEOREM 13. *Let J be an instance of the knapsack problem. Let n, M, P and W be as defined for procedure ϵ-APPROX. Let P^* be the value of an optimal solution for J. Let PMAX be as defined by procedure ϵ-APPROX on termination. Then $|P^*-PMAX|/P^*<1/(k+1)$.*

Proof. See [65].

The time required by algorithm 10 is $0(n^{k+1})$. To see this, note that the total number of subsets tried is $\sum_{i=0}^{k}\binom{n}{i}$ and $\sum_{i=0}^{k}\binom{n}{i}=0(n^k)$. The subalgorithm L has complexity $0(n)$. So the total time is $0(n^{k+1})$.

Algorithm ϵ-APPROX may be used as a polynomial time approximation scheme. For any given ϵ, $0<\epsilon<1$ we may choose k to be the least integer greater than or equal to $1/\epsilon-1$. This will guarantee a fractional error in the solution value of at most ϵ. The computing time is $0(n^{1/\epsilon})$.

In order to get a feel for how the approximation scheme might perform in practice, a simulation was conducted. A sample of 600 knapsack instances was used. This sample included problems with $n=15, 20, 25, 30, \cdots, 60$. For each problem size, 60 instances were generated. These 60 instances included five from each of the following six distributions:

I. random weights w_i and random profits p_i, $1 \leqq w_i, p_i \leqq 100$.
II. random weights w_i and random profits p_i, $1 \leqq w_i, p_i \leqq 1000$.
III. random weights w_i, $1 \leqq w_i \leqq 100$, $p_i = w_i + 10$.
IV. random weights w_i, $1 \leqq w_i \leqq 1000$, $p_i = w_i + 100$.
V. random profits p_i, $1 \leqq p_i \leqq 100$, $w_i = p_i + 10$.
VI. random profits p_i, $1 \leqq p_i \leqq 1000$, $w_i = p_i + 100$.

Random profits and weights were chosen from a uniform distribution over the given range. For each set of p's and w's, two M's were used: $M = 2 * \max\{w_i\}$ and $M = \sum w_i/2$. This makes for a total of 600 problem instances. Table II summarizes the results and gives the number of

TABLE II
RESULTS OF SIMULATION FOR SET OF 600 PROBLEMS

	|P* - PMAX /P*|									
Method	0 (Optimal value)	.001	.005	.01	.02	.03	.04	.05	.1	.25
L(Ø,P,S,M,n)	239	267	341	390	443	484	511	528	583	600
.5-APPROX	360	404	477	527	567	585	593	598	600	
.33-APPROX	483	527	564	581	596	600				

Figures give number of solutions that were within r percent of the true optimal solution value; r is the figure in the column head.

problems for which $(P^* - \text{PMAX})/P^*$ was in a particular range. 0.5-APPROX is ϵ-APPROX with $k=1$ and 0.33-APPROX is ϵ-APPROX with $k=2$. As is evident, the observed $|P^* - \text{PMAX}|/P^*$ values are much less than indicated by the worst-case bound of Theorem 13. Table III gives

TABLE III
COMPUTING TIMES USING THE 0.5-APPROXIMATE ALGORITHM

Problem size n	100	200	500	1000	2000	3000	4000	5000
Computing Time	.25	.9	3.5	14.6	60.4	98.3	180.	350.
Estimated % difference min{p/PMAX,.5}*100	2.5%	1.3%	.5%	.25%	.12%	.08%	.06%	.04%

$M = \sum w_{i/2}$; $w_i, p_i \epsilon [1, 1000]$; times in seconds.

the result of a simulation for large n. Computing times are for a FORTRAN program run on an IBM 360/65 computer.

Nemhauser and Wolsey [60] present a polynomial-time approximation scheme to find the maximum of a submodular set function.

Fully Polynomial-Time Approximation Schemes

The approximation algorithms and schemes we have seen so far are particular to the problem considered. There is no set of well-defined techniques that one may use to obtain such algorithms. The heuristics used depended very much on the particular problem being solved. For the case of fully polynomial-time approximation schemes, we can identify three underlying techniques. These techniques apply to a variety of optimization problems. We shall discuss these three techniques in terms of maximization problems.

We shall assume the maximization problem to be of the form:

$$\begin{array}{lll} \max \sum_{i=1}^{n} p_i x_i & & \\ \sum_{i=1}^{n} a_{ij} x_i \leqq b_j, & 1 \leqq j \leqq m & \\ x_i = 0 \text{ or } 1 & 1 \leqq i \leqq n & \end{array} \qquad (1)$$

where p_i, $a_{ij} \geqq 0$. Without loss of generality, we will assume that $a_{ij} \leqq b_j$, $1 \leqq i \leqq n$ and $1 \leqq j \leqq m$.

If $1 \leqq k \leqq n$, then the assignment $x_i = y_i$, $1 \leqq i \leqq k$ will be said to be a *feasible assignment* if and only if there exists at least one feasible solution to (1) with $x_i = y_i$, $1 \leqq i \leqq k$. A *completion* of a feasible assignment $x_i = y_i$, $1 \leqq i \leqq k$ is any feasible solution to (1) with $x_i = y_i$, $1 \leqq i \leqq k$. Let $x_i = y_i$, $1 \leqq i \leqq k$ and $x_i = z_i$, $1 \leqq i \leqq k$ be two feasible assignments such that for at least one j, $1 \leqq j \leqq k$, $y_j \neq z_j$. Let $\sum p_i y_i = \sum p_i z_i$. We shall say that $y_1, \cdots, y_k$ *dominates* $z_1, \cdots, z_k$ if and only if there exists a completion $y_1, \cdots, y_k, y_{k+1}, \cdots, y_n$ such that $\sum_{i=1}^{n} p_i y_i$ is greater than or equal to $\sum_{i=1}^{n} p_i z_i$ for all completions $z_1, \cdots, z_n$ of $z_1, \cdots, z_k$. The approximation techniques to be discussed will apply to those problems that can be formulated as (1) and for which simple rules can be found to determine when one feasible assignment dominates another. Such rules exist for example for problems solvable by the dynamic programming technique. Some such problems are 0/1-knapsack, job sequencing with deadlines, job sequencing to minimize finish time, and job sequencing to minimize weighted mean finish time.

One way to solve problems stated as above is to systematically generate all feasible assignments starting from the null assignment. Let $S^{(i)}$ represent the set of all feasible assignments for $x_1, \cdots, x_i$. Then $S^{(0)}$ represents the null assignment and $S^{(n)}$ the set of all completions. The answer to our problem is an assignment in $S^{(n)}$ that maximizes the objective function. The solution approach is then to generate $S^{(i+1)}$ from $S^{(i)}$, $1 \leqq i < n$. If an $S^{(i)}$ contains two feasible assignments $y_1, \cdots, y_i$ and $z_1, \cdots, z_i$ such that $\sum p_j y_j = \sum p_j z_j$, then use of the dominance rules enables us to discard or kill that assignment which is dominated. (In some cases the dominance rules may permit the discarding or killing of a feasible assignment even when $\sum p_j y_j \neq \sum p_j z_j$. This happens, for instance, in the knapsack problem (see [34, 59]). Following the use of the dominance rules, it is the case that for each feasible assignment in $S^{(i)}$ $\sum_{j=1}^{i} p_j x_j$ is distinct. However, despite this, it is possible for each $S^{(i)}$ to contain twice as many feasible assignments as in $S^{(i-1)}$. This results in a worst-case computing time that is exponential in n.

The approximation methods we are about to discuss are called rounding, interval partitioning, and separation. These methods will restrict the number of distinct $\sum_{j=1}^{i} p_j x_j$ to be only a polynomial function of n. The error introduced will be within some prespecified bound. Since these methods are discussed in detail in [67], we shall only have a brief discussion here.

Rounding. The aim of rounding is to start from a problem instance, I, formulated as in (1) and to transform it to another problem instance I' that is easier to solve. This transformation is carried out in such a way that the optimal solution value of I' is "close" to the optimal solution value of I. In particular, if we are provided with a bound, ϵ, on the fractional difference between the exact and approximate solution values, then we require that $|(F^*(I)-F^*(I'))/F^*(I)| \leqq \epsilon$, where $F^*(I)$ and $F^*(I')$ represent the optimal solution values of I and I', respectively.

I' is obtained from I by changing the objective function to $\max \sum q_i x_i$. Since I and I' have the same constraints, they have the same feasible solutions. Hence, if the p_i's and q_i's differ by only a "small" amount, the value of an optimal solution to I' will be close to the value of an optimal solution to I.

Given the p_i's and an ϵ, what should the q_i's be so that $(F^*(I')-F^*(I))/F^*(I) \leqq \epsilon$ and $\sum_{i=0}^{n} |S^{(i)}| \leqq u(n, 1/\epsilon)$ where u is a polynomial in n and $1/\epsilon$? Once we can figure this out, we will have a fully polynomial approximation scheme for our problem since it is possible to go from $S^{(i-1)}$ to $S^{(i)}$ in time proportional to $0(S^{(i-1)})$.

Let LB be an estimate for $F^*(I)$ such that $F^*(I) \geqq LB$. Clearly, we may assume $LB \geqq \max_i \{p_i\}$. If $\sum_{i=1}^{n} |p_i - q_i| \leqq \epsilon F^*(I)$, then $(F^*(I)-F^*(I'))/F^*(I) \leqq \epsilon$. If we define $q_i = p_i - \text{rem}(p_i, (LB \cdot \epsilon)/n)$, where $\text{rem}(a, b)$ is the remainder of a/b, i.e., $\text{rem}(a, b) = a - \lfloor a/b \rfloor b$. (E.g., $\text{rem}(7, 6) = 1$ and $\text{rem}(2.2, 1.3) = 0.9$). Since $\text{rem}(p_i, LB \cdot \epsilon/n) < LB \cdot \epsilon/n$, it follows that $\sum |p_i - q_i| < LB \cdot \epsilon \leqq F^* \cdot \epsilon$. Hence, if an optimal solution to I' is used as an optimal solution for I, then the fractional error is less than ϵ.

In order to determine the time required to solve I' exactly, it is useful to introduce another problem I'' with s_i, $1 \leqq i \leqq n$ as its objective function coefficients. Define $s_i = \lfloor (p_i \cdot n)/(LB \cdot \epsilon) \rfloor$, $1 \leqq i \leqq n$. It is easy to see that $s_i = (q_i \cdot n)/(LB \cdot \epsilon)$. Clearly, the $S^{(i)}$'s corresponding to the solution of I' and I'' will have the same number of tuples. (r_1, t_1) is a tuple in an $S^{(i)}$ for I' if and only if $((r_1 \cdot n)/(LB \cdot \epsilon), t_1)$ is a tuple in the $S^{(i)}$ for I''. Hence, the time needed to solve I' is the same as that needed to solve I''. Since $p_i \leqq LB$, it follows that $s_i \leqq \lfloor n/\epsilon \rfloor$. Hence $|S^{(i)}| \leqq 1 + \sum_{j=1}^{i} s_j \leqq 1 + i \lfloor n/\epsilon \rfloor$ and so $\sum_{i=0}^{n-1} |S^{(i)}| \leqq n + \sum_{i=0}^{n-1} i \lfloor n/\epsilon \rfloor = 0(n^3/\epsilon)$. Thus, if we can go from $S^{(i-1)}$ to $S^{(i)}$ in $0(|S^{(i-1)}|)$ time, then I'' and hence I' can be solved in $0(n^3/\epsilon)$ time.

Moreover, the solution for I' would be an ϵ-approximate solution for I, and we would thus have a fully polynomial-time approximation scheme. When using rounding, we will actually solve I'' and use the resulting optimal solution as the solution to I.

Rounding as described in its full generality results in $0(n^3/\epsilon)$ time approximation schemes. It is possible to specialize this technique to the specific problem being solved. In particular, we can obtain specialized and asymptotically faster polynomial-time approximation schemes for the knapsack problem as well as for the problem of scheduling tasks on two processors to minimize finish time. Ibarra and Kim [39] use rounding to obtain an $0(n(\log n+1/\epsilon^2))$ scheme for the 0/1 knapsack problem. Lawler [56] has obtained an $0(n \log(1/\epsilon)+1/\epsilon^4)$ approximation scheme for the same problem.

Interval partitioning. Unlike rounding, interval partitioning does not transform the original problem instance into one that is easier to solve. Instead, an attempt is made to solve the problem instance I by generating a restricted class of the feasible assignments for $S^{(0)}, S^{(1)}, \cdots, S^{(n)}$. Let P_i be the maximum $\sum_{j=1}^{i} p_j x_j$ among all feasible assignments generated for $S^{(i)}$. Then the profit interval $[0, P_i]$ is divided into subintervals each of size $P_i\epsilon/(n-1)$ (except possibly the last interval that may be a little smaller). All feasible assignments in $S^{(i)}$ with $\sum_{j=1}^{i} p_j x_j$ in the same subinterval are regarded as having the same $\sum_{j=1}^{i} p_j x_j$ and the dominance rules are used to discard all but one of them. The $S^{(i)}$ resulting from this elimination is used in the generation of $S^{(i+1)}$. Since the number of subintervals for each $S^{(i)}$ is at most $\lceil n/\epsilon \rceil+1$, $|S^{(i)}| \leqq \lceil n/\epsilon \rceil+1$. Hence, $\sum_{i=1}^{n} |S^{(i)}|=0(n^2/\epsilon)$. The error introduced in each feasible assignment due to this elimination in $S^{(i)}$ is less than the subinterval length. This error may, however, propagate from $S^{(1)}$ up through $S^{(n)}$, and the error is additive. If $\hat{F}(I)$ is the value of the optimal generated using interval partitioning and $F^*(I)$ the value of a true optimal, it follows that $F^*(I)-\hat{F}(I) \leqq (\epsilon \sum_{i=1}^{n-1} P_i)/(n-1)$. Since $P_i \leqq F^*(I)$, it follows that $(F^*(I)-\hat{F}(I))/F^*(I) \leqq \epsilon$, as desired.

In many cases the algorithm may be speeded by starting with a good estimate, LB for $F^*(I)$ such that $F^*(I) \geqq LB$. The subinterval size is then $LB \cdot \epsilon/(n-1)$ rather than $P_i\epsilon/(n-1)$. When a feasible assignment with value greater than LB is discovered, the subinterval size can be chosen as described above.

Separation. Assume that in solving a problem instance I, we have obtained an $S^{(i)}$ with feasible solutions having the following $\sum_{1 \leqq j \leqq i} p_j x_j$ values: 0, 3.9, 4.1, 7.8, 8.2, 11.9, 12.1. Further assume that the interval size $P_i\epsilon/(n-1)$ is 2. Then the subintervals are $[0, 2)$, $[2, 4)$, $[4, 6)$, $[6, 8)$, $[8, 10)$, $[10, 12)$ and $[12, 14)$. Each value above falls in a different subinterval, and hence no feasible assignments are eliminated. However, there are three pairs of assignments with values within $P_i\epsilon/(n-1)$. If the dominance rules are used for each pair, only four assignments will remain. The error introduced is at most $P_i\epsilon/(n-1)$. More formally, let a_0, a_1, $\cdots$, a_r be the distinct values of $\sum_{j=1}^{i} p_j x_j$ in $S^{(i)}$. Let us assume $a_0 < a_1 < \cdots < a_r$. We will construct a new set J from $S^{(i)}$ by making a left-to-right scan and retaining a tuple only if its value exceeds the value of the last tuple in J by more than $P_i\epsilon/(n-1)$.

The analysis for this strategy is the same as that for interval partitioning. The same comments regarding the use of a good estimate for $F^*(I)$ hold here, too.

Intuitively, one may expect separation to always work better than interval partitioning. In [67] it is shown that this need not be the case. However, empirical studies with one problem indicate interval partitioning to be inferior in practice.

In [35, 39, 66] these techniques are applied to the following problems to obtain fully polynomial-time approximation schemes: 0/1 knapsack problem, integer knapsack problem, job sequencing with deadlines, minimizing weighted mean finish time, finding an optimal SPT schedule, and finding minimum finish time schedules.

At this point, one might well ask the question: What kind of *NP*-hard problems can have fully polynomial-time approximation schemes? Clearly, no *NP*-hard ϵ-approximation problem can have such a scheme unless $P=NP$. A stonger result [24] may be proven. No *NP*-hard problem that is *NP*-hard in the strong sense may have a fully polynomial-time approximation scheme unless $P=NP$.

DEFINITION. *Let L be some problem. Let I be an instance of L and let LENGTH(I) be the number of bits in the representation of I. Let MAX(I) be the magnitude of the largest number in I. For some fixed polynomial p let L_p be problem L restricted to those instances I in which MAX(I)≦p(LENGTH(I)). Problem L is NP-hard in the strong sense if and only if there exists a polynomial p such that L_p is NP-hard.*

Examples of problems that are *NP*-hard in the strong sense are: Hamiltonian cycle, node cover, feedback arc set, traveling salesperson, max clique. The 0/1-knapsack problem is probably not *NP*-hard in the strong sense because when MAX(I)≦p(LENGTH(I)) then I can be solved in time 0(LENGTH(I)^2*p(LENGTH(I)) using the dynamic programming algorithm of [34] or [59].

THEOREM 14. *Let L be an optimization problem such that all feasible solutions to all possible instances have a value that is a positive integer. Further, assume that for all instances I of L, the optimal value F*(I) is bounded by a polynomial function p in the variables LENGTH(I) and MAX(I), i.e., 0<F*(I)<p(LENGTH(I), MAX(I)) and F*(I) integer. If L has a fully polynomial-time approximation scheme, then L has an exact*

algorithm whose complexity is a polynomial in LENGTH(I) and MAX(I).

Proof. See [24].

When Theorem 14 is applied to integer-valued problems that are *NP*-hard in the strong sense, we see that no such problem can have a fully polynomial-time approximation scheme unless $P=NP$.

4. PROBABILISTICALLY GOOD ALGORITHMS

The approximation algorithms of Section 3 had the nice property that their worst-case performance could be bounded by some constants (k in the case of an absolute approximation and ϵ in the case of an ϵ-approximation). The requirement of bounded performance tends to categorize as bad other algorithms that usually work well. Some algorithms with unbounded performance may in fact almost always either solve the problem exactly or generate a solution that is exceedingly close in value to an optimal solution. Such algorithms are good in a probabilistic sense. If we pick a problem instance I at random, then there is a very high probability that the algorithm will generate a very good approximate solution. In this section we shall consider two algorithms with this property. Both algorithms are for *NP*-hard problems.

First, since we shall be carrying out a probabilistic analysis of the algorithms, we need to define a sample space of inputs. The sample space is set up by first defining a sample space S_n for each problem size n. Problem instances of size n are drawn from S_n. Then the overall sample space is the infinite Cartesian product $S_1\times\cdots\times S_n\cdots$. An element of the sample space is a sequence $X=x_1,\cdots,x_n,\cdots$ such that x_i is drawn from S_i.

DEFINITION (Karp [47]). *An algorithm $\mathscr{A}$ solves a problem P almost everywhere (abbreviated a. e.) if, when $X=x_1,\cdots,x_n,\cdots$ is drawn from the sample space $S_1\times\cdots\times S_n,\cdots$, the number of x_i on which the algorithm fails to solve P is finite with probability 1.*

Since both the algorithms we shall be discussing are for *NP*-hard graph problems, we shall first describe the sample space for which the probabilistic analysis will be carried out. Let $p(n)$ be a function such that $0\leqq p(n)\leqq 1$ for all $n\geqq 0$. A random n-vertex graph is constructed by including edge (i,j), $i\neq j$, with probability $p(n)$.

The first algorithm we shall consider is due to Posa [62]. Posa's algorithm is to find a Hamiltonian cycle in an undirected graph. Informally, Posa's algorithm proceeds as follows. First, an arbitrary vertex (say vertex 1) is chosen as the start vertex. The algorithm maintains a simple path P starting from vertex 1 and ending at vertex k. Initially P is a trivial path with $k=1$, i.e, there are no edges in P. At each iteration of the algorithm an attempt is made to increase the length of P. This is done by considering an edge (k,j) incident to the end point k of P. When edge (k,j) is being considered, one of three possibilities exists:

(i) [$j=1$ and path P includes all the vertices of the graph.]
In this case a Hamiltonian cycle has been found and the algorithm terminates.

(ii) [j is not on the path P.]
In this case the length of path P is increased by adding (k,j) to it. j becomes the new end point of P.

(iii) [j is already on path P.]
Now there is a unique edge $e=(j,m)$ in P such that the deletion of e and the inclusion of (k,j) to P result in a simple path. e is deleted and (k,j) added to P. P is now a simple path with end point m.

The algorithm is constrained so that case (iii) does not generate two paths of the same length having the same end point. With a proper choice of data representations, this algorithm can be implemented to run in time

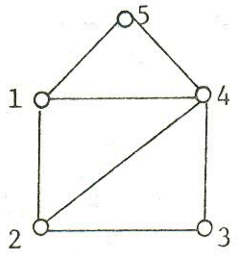

Figure 4

$0(n^2)$, where n is the number of vertices in the graph G. It is easy to see that this algorithm does not always find a Hamiltonian cycle in a graph that contains such a cycle. However, Posa has shown the following:

THEOREM 15. *If $p(n) \sim (\alpha \ln n)/n$, $\alpha>1$, then the preceding algorithm finds a Hamiltonian cycle (a. e.).*

Proof. See [62].

Example 14. Let us try out the above algorithm on the five-vertex graph of Figure 4. The path P initially consists of vertex 1 only. Assume edge (1,4) is chosen. This represents case (ii) and P is expanded to $\{1,4\}$. Assume edge (4,5) is chosen next. Path P now becomes $\{1,4,5\}$. Edge (1,5) is the only possibility for the next edge. This results in case (iii) and P becomes $\{1,5,4\}$. Now assume edges (4,3) and (3,2) are considered. P becomes $\{1,5,4,3,2\}$. If edge (1,2) is considered next, a Hamiltonian cycle is found and the algorithm terminates.

Angulin and Valiant [2] present a faster algorithm to detect the presence of Hamiltonian cycles in both directed and undirected graphs.

Their algorithm also has a high probability of detecting a Hamiltonian cycle when such a cycle exists.

We now consider a probabilistically good algorithm that is for the problem of finding a maximum independent set of a graph G. (A set S of vertices is said to be independent in G if no two vertices in S are adjacent in G). This problem is just the complement of the max clique problem. For any given graph G and set of vertices S, define the function $m(S)$ such that $m(S)=0$ if every vertex in G is either in S or is adjacent to some vertex in S; otherwise, $m(S)$=index of a vertex not in S and not adjacent to any vertex in S. The following algorithm uses this function to construct an independent set of G.

```
procedure INDEP($G$)
    $S \leftarrow \emptyset$
    while $m(S) \neq 0$ do
        $S \leftarrow S \cup m(S)$
    end
end INDEP
```

One can easily construct examples of n vertex graphs for which INDEP generates independent sets of size 1 when in fact a maximum independent set contains $n-1$ vertices. However, for certain probability distributions it can be shown that INDEP generates good approximations almost everywhere. If $F^*(I)$ and $\hat{F}(I)$ represent the size of a maximum independent set and one generated by algorithm INDEP, respectively, then the following theorem is obtained:

THEOREM 16. *If $p(n)=c$, for some constant c, then for every $\epsilon>0$ we have $(F^*(I)-\hat{F}(I))/F^*(I) \leqq 0.5+\epsilon$ (a.e.).*

Proof. See [47].

Algorithm INDEP can easily be implemented to have polynomial complexity. Some other *NP*-hard problems for which probabilistically good algorithms are known are: Euclidean traveling salesperson, minimal colorings of graphs, and set covering (see [47]). Kim [51] has carried out a probabilistic analysis of the performance of several fast heuristics for the problem of scheduling n independent tasks on 2 identical machines to minimize finish time. Lin and Kernighan [58] present an algorithm for the traveling salesperson problem. Empirical results indicate it to be good; however probabilistic analysis has not yet been carried out.

ACKNOWLEDGMENTS

This research was supported, in part, by the National Science Foundation under grants numbered MCS 76-06089, DCR76-10081 and MCS 74-21024.

REFERENCES

1. A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison Wesley, Reading, Mass., 1974.
2. D. Angulin and L. Valiant, "Fast Probabilistic Algorithms for Hamiltonian Circuits and Matchings," *Proceedings of the 9th Annual ACM Symposium on Theory of Computing*, pp. 30–41, 1977.
3. G. H. Bradley, "Equivalent Integer Programs and Canonical Problems," *Management Sci.* **17,** 354–366 (1961).
4. P. Brucker, M. Garey, and D. Johnson, "Scheduling Equal Length Tasks under Tree Like Precedence Constraints to Minimize Maximum Lateness" (to appear in *Math. Opns. Res.*).
5. J. Bruno, E. G. Coffman Jr., and R. Sethi, "Algorithms for Minimizing Mean Flow Time," *Proc. IFIP Congr.* **74,** 504–510 (1974).
6. J. Bruno, E. G. Coffman Jr., and R. Sethi, "Scheduling Independent Tasks to Reduce Mean Finishing-Time," *Comm. ACM* **17,** 382–387 (1974).
7. N. Christofides, "Worst Case Analysis of a New Heuristic for the Traveling Salesman Problem," Management Sci. Res. Report #388, Carnegie Mellon University, 1976.
8. E. G. Coffman Jr., *Computer and Job-Shop Scheduling Theory*, John Wiley & Sons, New York, 1976.
9. E. Coffman, M. Garey, and D. Johnson, "An Application of Bin-Packing to Multiprocessor Scheduling" *SIAM J. Comput.* **7,** 1–17 (1978).
10. S. A. Cook, "The Complexity of Theorem-Proving Procedures," *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pp. 151–158, 1971.
11. G. Cornuejols, M. Fisher, and G. Nemhauser, "Location of Bank Accounts to Optimize Float: An Analytic Study of Exact and Approximate Algorithms," *Management Sci.* **23,** 789–810 (1977).
12. O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*, Academic Press, New York, 1972.
13. G. G. Dantzig, "On the Significance of Solving Linear Programming Problems with Some Integer Variables," *Econometrica* **28,** 30–44 (1960).
14. S. Even, A. Itai, and A. Shamir, "On the Complexity of a Timetable and Multicommodity Flow Problems," *SIAM J. Comput.* **5,** 691–703 (1976).
15. M. Fisher, G. Nemhauser, and L. Wolsey, "An Analysis of Approximations for Maximizing Submodular Set Functions—II," CORE discussion paper #7629, Catholic University of Louvain, Belgium, 1976 (to appear in *Math. Prog. Studies*).
16. M. Fisher, G. Nemhauser, and L. Wolsey, "An Analysis of Approximations for Finding a Maximum Weight Hamiltonian Circuit" (to appear in *Opns. Res.*).
17. R. W. Floyd, "Nondeterministic Algorithms," *J. Assoc. Comput. Mach.* **14,** 636–644 (1967).
18. G. Fredrickson, M. Hecht, and C. Kim, "Approximation Algorithms for Some Routing Problems," pp. 216–227, *Proceedings of 17th Annual Symposium on Foundation of Computer Science*, Houston, Texas, 1976.
19. M. Garey, R. Graham, and D. Johnson, "Some NP-Complete Geometric

Problems," *Proceedings of 8th Annual ACM Symposium on Theory of Computing*, pp. 10–22, May 1976.

20. M. Garey, R. Graham, and D. Johnson, "Performance Guarantees for Scheduling Algorithms," *Opns Res.* **26,** 3–21 (1978).
21. M. Garey and D. Johnson, "Complexity Results for Multiprocessor Scheduling under Resource Constraints," *Siam J. Comput.* **4,** 397–411 (1975).
22. M. Garey and D. Johnson, "Approximation Algorithms for Combinatorial Problems: An Annotated Bibliography," in *Algorithms and Complexity: Recent Results and New Directions*, pp. 41–52, J. Traub (ed.), Academic Press, 1976.
23. M. Garey and D. Johnson, "The Complexity of Near Optimal Graph Coloring," *J. Assoc. Comput. Mach.* **23,** 43–49 (1976).
24. M. Garey and D. Johnson, "'Strong' NP-Completeness Results: Motivation Examples and Implications," Bell Laboratories Report, Murray Hill, N. J. 1976 (to appear in *J. Assoc. Comput. Mach.*).
25. M. Garey, D. Johnson, and R. Sethi, "The Complexity of Flow Shop Scheduling," *Math. Opns. Res.* **1,** 117–129, (1976).
26. M. Garey, D. Johnson, and L. Stockmeyer, "Some Simplified NP-Complete Problems," *J. Theory Comput. Sci.* **1,** 237–267 (1976).
27. T. Gonzalez, O. Ibarra, and S. Sahni, "Bounds for LPT Schedules on Uniform Processors," *SIAM J. Comput.* **6,** 155–166 (1977).
28. T. Gonzalez and S. Sahni, "Open Shop Scheduling to Minimize Finish Time," *J. Assoc. Comput. Mach.* **23,** 665–679 (1976).
29. T. Gonzalez and S. Sahni, "Flowshop and Jobshop Schedules: Complexity and Approximation," *Opns. Res.* **26,** 36–52 (1978).
30. R. Graham, "Bounds for Certain Multiprocessing Anomalies," *Bell System Tech. J.* **45,** 1563–1581 (1966).
31. R. Graham, "Bounds on Multiprocessing Timing Anomalies," *SIAM J. Appl. Math.* **17,** 416–429 (1969).
32. E. Gurari and O. Ibarra, "An *NP*-Complete Number Theoretic Problem," University of Minnesota, Computer Science Technical Report, TR-77-12, 1977.
33. P. P. Hermann, "On Reducibility among Combinatorial Problems," MIT MAC Report TR-113, December 1973.
34. E. Horowitz and S. Sahni, "Computing Partitions with Application to the Knapsack Problem," *J. Assoc. Comput. Mach.* **21,** 277–292 (1974).
35. E. Horowitz and S. Sahni, "Exact and Approximate Algorithms for Scheduling Nonidentical Processors," *J. Assoc. Comput. Mach.* **23,** 317–327 (1976).
36. E. Horowitz and S. Sahni, *Fundamentals of Data Structures*, Computer Science Press, Potomac, Md., 1976.
37. E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, Potomac, Md., 1978.
38. E. Horvath, S. Lam, and R. Sethi, "A Level Algorithm for Preemptive Scheduling," *J. Assoc. Comput. Mach.* **24,** 32–43 (1977).
39. O. Ibarra and C. Kim, "Fast Approximation Algorithms for the Knapsack and Sum of Subsets Problems," *J. Assoc. Comput. Mach.* **22,** 463–468 (1975).

40. O. IBARRA AND C. KIM, "Heuristic Algorithms for Scheduling Independent Tasks on Nonidentical Processors," *J. Assoc. Comput. Mach.* **24,** 280–289 (1977).
41. O. IBARRA AND C. KIM, "Scheduling for Maximum Profits/Minimum Time," (to appear in *Math. Opns. Res.*).
42. D. JOHNSON, "Approximation Algorithms for Combinatorial Problems," *J. Comput. Syst. Sci.* **9,** 256–278 (1974).
43. D. JOHNSON, A. DEMERS, J. ULLMAN, M. GAREY, AND R. GRAHAM, "Performance Bounds for Simple One-Dimensional Bin Packing Algorithms," *SIAM J. Comput.* **3,** 299–325 (1974).
44. R. KARP, "Reducibility among Combinatorial Problems," *Complexity of Computer Computations,* pp. 85–104, R. E. Miller and J. W. Thatcher (eds.), Plenum Press, New York, 1972.
45. R. KARP, "On the Computational Complexity of Combinatorial Problems," *Networks* **5,** 45–68 (1975).
46. R. KARP, "The Fast Approximate Solution of Hard Combinatorial Problems," *Proceedings Sixth Southeastern Conference on Combinatorics, Graph Theory and Computing,* pp. 15–34, Winnipeg, 1975.
47. R. KARP, "The Probabilistic Analysis of Some Combinatorial Search Algorithms," in *Algorithms and Complexity: New Directions and Recent Results,* pp. 1–20, J. Traub (ed.), Academic Press, New York, 1976.
48. R. KARP, A. C. MCKELLAR, AND C. K. WONG, "Near-Optimal Solutions to a 2-Dimensional Placement Problem," *SIAM J. Comput.* **4,** 271–286 (1975).
49. M. KAUFMAN, "An Almost-Optimal Algorithm for the Assembly Line Scheduling Problem," *IEEE Trans. Comput.* C-23, 1169–1174 (1974).
50. B. W. KERNIGHAN AND P. J. PLAUGER, *The Elements Of Programming Style,* McGraw-Hill, New York, 1974.
51. C. KIM, "Analysis of the Expected Performance of Algorithms for the Partition Problem," University of Maryland, Computer Science Technical Report, 1976.
52. V. KLEE AND G. J. MINTY, "How Good Is the Simplex Algorithm?" in Inequalities III, pp. 159–175, O. Shisha (ed.), Academic Press, New York, 1972.
53. D. KNUTH, "Postscript about NP-Hard Problems," *ACM SIGACT News* **6,** 15–16 (1974).
54. D. KNUTH, "Structured Programming with Go To's," *ACM Surv.* **6,** 261–302 (1974).
55. D. KNUTH, "Estimating the Efficiency of Backtrack Programs," *Math. Comp.* **29,** 121–136 (1975).
56. E. LAWLER, "Fast Approximation Algorithms for Knapsack Problems," *Proceedings, 18th Annual Symposium on Foundations of Computer Science,* Providence, Rhode Island, pp. 206–213, 1977.
57. J. K. LENSTRA, "Sequencing by Enumerative Methods," Ph.D. thesis, Mathematisch Centrum, Amsterdam, 1976.
58. S. LIN AND P. KERNIGHAN, "An Effective Heuristic Algorithm for the Traveling Salesman Problem," *Opns. Res.* **21,** 498–516 (1973).
59. G. NEMHAUSER AND Z. ULLMAN, "Discrete Dynamic Programming and Capital Allocation," *Management Sci.* **15,** 494–505 (1969).

60. G. NEMHAUSER AND L. WOLSEY, "Best Algorithms for Approximating the Maximum of a Submodular Set Function," CORE discussion paper #7636, Catholic University of Louvain, Belgium, 1976 (to appear in *Math. Opns. Res.*).
61. C. H. PAPADIMITRIOU AND K. STEIGLITZ, "Some Complexity Results for the Traveling Salesman Problem," *Proceedings Eighth Annual ACM Symposium on Theory of Computing*, pp. 1–9, May 1976.
62. L. POSA, "Hamiltonian Circuits in Random Graphs," *Discrete Math.* **14,** 359–369 (1976).
63. A. H. G. RINNOOY KAN, "Machine Scheduling Problems," Ph.D. thesis, Mathematisch Centrum, Amsterdam, 1976.
64. S. SAHNI, "Computationally Related Problems," *SIAM J. Comput.* **3,** 262–279 (1974).
65. S. SAHNI, "Approximate Algorithms for the 0/1 Knapsack Problem," *J. Assoc. Comput. Mach.* **22,** 115–124 (1975).
66. S. SAHNI, "Algorithms for Scheduling Independent Tasks," *J. Assoc. Comput. Mach.* **23,** 114–127 (1976).
67. S. SAHNI, "General Techniques for Combinatorial Approximation," *Opns. Res.* **25,** 920–936 (1977).
68. S. SAHNI AND Y. CHO, "Complexity of Scheduling Shops with No Wait in Process," University of Minnesota Technical Report, TR-77-20, Dec. 1977.
69. S. SAHNI AND T. GONZALEZ, "P-Complete Problems and Approximation Solutions," *Proc. 15th Annual IEEE Symp. on Switching and Automata Theory*, pp. 28–32, 1974.
70. S. SAHNI AND T. GONZALEZ "P-Complete Approximation Problems," *J. Assoc. Comput. Mach.* **23,** 555–565 (1976).
71. T. J. SCHAEFER, "Complexity of Decision Problems Based on Finite Two-Person Perfect-Information Games," *Proceedings Eighth Annual ACM Symposium on Theory of Computing*, pp. 41–49, May 1976.
72. R. SETHI, "On the Complexity of Mean Flow Time Scheduling," *Math. Opns. Res.* **2,** 320–330 (1977).
73. J. D. ULLMAN, "*NP*-Complete Scheduling Problems," *J. Comput. Syst. Sci.* **10,** 384–393 (1975).