OPERATIONS RESEARCH, Vol. 25, No. 6, November–December 1977

General Techniques for Combinatorial Approximation

SARTAJ SAHNI

University of Minnesota, Minneapolis, Minnesota

(Received June 1976; accepted April 1977)

This is a tutorial on general techniques for combinatorial approximation. In addition to covering known techniques, a new one is presented. These techniques generate fully polynomial time approximation schemes for a large number of NP-complete problems. Some of the problems they apply to are: 0-1 knapsack, integer knapsack, job sequencing with deadlines, minimizing weighted mean flow times, and optimal SPT schedules. We also present experimental results for the job sequencing with deadlines problem.

MANY COMBINATORIAL optimization problems are known to be NP-complete [10, 15, 16, 19]. An NP-complete problem can be solved in polynomial time iff all other NP-complete problems can also be solved in polynomial time. The class of NP-complete problems includes such difficult problems as the traveling salesman, multicommodity network flows, integer programming with bounded variables, set cover, and node cover problems. There is no known polynomial time algorithm for any of these problems. Moreover, mounting empirical evidence (i.e., the identification of more and more NP-complete problems) suggests that it is very likely that no polynomial time algorithms exist for any of these problems. This realization has led many researchers to develop polynomial time approximation algorithms for some NP-complete optimization problems. An approximation algorithm for an optimization problem is generally a heuristic that attempts to obtain a solution whose value is "close" to the optimal value. For many problems the data themselves are only an estimate. The exact values may be slightly different from these estimates. In such cases it is probably just as meaningful to find a solution whose value is close to the optimal value as it is to find an optimal solution (as the optimal solution may not remain so once the exact data values are known). For the case of NP-complete problems, the study of approximation algorithms derives an even stronger motivation from the fact that all known optimization algorithms for these problems require an exponential amount of time (measured as a function of problem size) and the expectation that these problems will never be solvable by polynomial time algorithms.

Even on the fastest computers, exponential time algorithms are feasible only for relatively small problem sizes. It is better to be able to obtain an approximately optimal solution than no solution at all.

Many researchers have presented heuristics that obtain reasonably good solutions. Until recently the performance of these heuristics was measured only through computational tests. The heuristic was programmed and the solutions compared with known optimal solutions. Thus, Lin and Kernighan [13] obtain computational results showing that for the test set used, their local interchange heuristic for the traveling salesman problem obtained solutions with value almost equal to the exact solution value. On the basis of this test can we make any predictions about how the heuristic will perform on other problem instances? The answer is no. We can only hope that it will do this well on other data sets, but we cannot be sure (especially since the optimal solution values may not be known). A major thrust of recent research on approximation algorithms has been the insistence on performance guarantees for heuristics. This involves the establishment of bounds on maximum difference between the value of an optimal solution and the value of the solution generated by the heuristic. While performance bounds were obtained as early as 1966 by Graham [4] for certain scheduling heuristics, the development of the class of NP-complete problems has led to a renewed interest in approximation methods that are guaranteed to get solutions with value within some specified fraction of the optimal solution value.

DEFINITION 1. *An algorithm will be said to be an ϵ-approximate algorithm for a problem P iff it is the case that for every instance I of P, $|F^*(I)-\hat{F}(I)|/F^*(I) \leqq \epsilon$. $F^*(I)>0$ is the value of an optimal solution to I, $\hat{F}(I)$ is the value of the solution generated by the algorithm, and ϵ is some constant. For a maximization problem we require $0 \leqq \epsilon < 1$ and for a minimization problem $\epsilon \geqq 0$.*

Many known polynomial time approximation algorithms are also ϵ-approximation algorithms. Thus while it seems necessary to spend, in the worst case, a nonpolynomial amount of time to obtain optimal solutions to NP-complete problems, we can get to within an ϵ factor of the optimal for some such problems in polynomial time. For yet others [19] it is known that the problem of obtaining ϵ-approximate solutions is also NP-complete.

In some cases it is possible to obtain an approximation algorithm that for every $\epsilon>0$ generates solutions that are ϵ-approximate [7–9, 17, 18]. In the terminology of Garey and Johnson [2, 3] such an algorithm is called an approximation scheme. Definitions 2–4 are due to Garey and Johnson.

DEFINITION 2. *An approximation scheme for a problem P is an algorithm that, given an instance I and a desired degree of accuracy $\epsilon>0$, constructs a problem solution with value $\hat{F}(I)$, such that, if $F^*(I)>0$ is the value of an optimal solution to I, then $|F^*(I)-\hat{F}(I)|/F^*(I) \leqq \epsilon$.*

DEFINITION 3. *An approximation scheme is a polynomial time approximation scheme if for every fixed $\epsilon>0$ it has a polynomial computing time. (References 7–9, 17, 18 present such schemes for some NP-complete problems.)*

While the existence of polynomial time approximation schemes for NP-complete problems may appear surprising, for some problems one can in fact obtain approximation schemes with a computing time polynomial in both the input size and $1/\epsilon$. Such schemes are called *fully polynomial time approximation schemes.*

DEFINITION 4. *A fully polynomial time approximation scheme is a polynomial time approximation scheme whose computing time is a polynomial in both the input size and $1/\epsilon$.*

Garey and Johnson [2] present an annotated bibliography of research on combinatorial approximation. Ibarra and Kim [8] present an $0(n/\epsilon^2+n\log n)$ algorithm for the 0-1 knapsack problem. This algorithm is a fully polynomial time approximation scheme. Sahni [18] and Horowitz and Sahni [7] present $0(n^2/\epsilon)$ fully polynomial time approximation schemes for several machine-scheduling problems. In Section 1 we present a tutorial on two of the techniques used in [7, 8, 18] to obtain these schemes. These techniques are called *rounding* [8] and *interval partitioning* [18]. Both techniques are very general and applicable to a wide variety of optimization problems. Next we present a third technique, called *separation*, which is a modification of interval partitioning. This technique is as general as the others. While it results in approximation schemes with the same worst-case complexity as those obtained when interval partitioning is used, intuition backed by experimental results indicates that it performs better than interval partitioning. These three general methods for combinatorial optimization have the added advantage that heuristics that could be applied to the exact algorithm can also be used with the approximation scheme. We shall see an example of this in Section 2. For many of the other approximation algorithms that have been obtained for NP-complete problems this is not true.

In Section 2 we present implementation details and experimental results comparing interval partitioning and separation as applied to the job sequencing with deadlines problem. The approximation algorithms are also compared with an exact algorithm for this problem. It will be seen that, at least on the test data, the heuristics obtained solutions with values much closer to the optimal than suggested by their worst-case analysis.

1. GENERAL APPROXIMATION TECHNIQUES

We state the three approximation techniques in terms of maximization problems. The extension to minimization problems is immediate. We shall assume the maximization problem to be of the form

$$\begin{aligned} &\max \textstyle\sum_{i=1}^{n} c_i x_i \\ &\textstyle\sum_{i=1}^{n} a_{ij} x_i \leqq b_j, \quad 1 \leqq j \leqq m \\ &x_i = 0 \text{ or } 1, \quad 1 \leqq i \leqq n, \end{aligned} \tag{1}$$

where c_i, $a_{ij} \geqq 0$ for all i and j. 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* iff 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 $\sum c_i y_i = \sum c_i z_i$. We shall say that $y_1, \cdots, y_k$ *dominates* $z_1, \cdots, z_k$ iff there exists a completion $y_1, \cdots, y_k, y_{k+1}, \cdots y_n$ such that $\sum_{i=1}^{n} c_i y_i$ is greater than or equal to $\sum_{i=1}^{n} c_i z_i$ 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 many problems solvable by dynamic programming [1, 14, 15].

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 $\sum c_j y_j = \sum c_j z_j$, then use of the dominance rules enables us to discard that assignment which is dominated. (In some cases the dominance rules may permit the discarding of a feasible assignment even when $\sum c_j y_j \neq \sum c_j z_j$. This happens, for instance, in the knapsack problem [5, 15].) Following the use of the dominance rules, it is the case that for each feasible assignment in $S^{(i)}$ $\sum_{j=1}^{i} c_j x_j$ is distinct. However, despite this, it is possible for each $S^{(i)}$ to contain twice as many feasible assignments as $S^{(i-1)}$. This results in a worst-case computing time that is exponential in n. The approximation methods we are about to discuss will restrict the number of distinct $\sum_{j=1}^{n} c_j x_j$ to be only a polynomial function of n. The error introduced will be within some prespecified bound. The methods are computationally efficient only when there exist efficient dominance rules to eliminate all but one of a set of assignments that yield the same profit.

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. The transformation from I to I' is carried out in the following way: First we observe that the feasible solutions to I are independent of the c_i, $1 \leqq i \leqq n$. Thus if I' has the same constraints as I and the c_i' differ from the c_i by "small amounts" the optimal solution to I will be a feasible solution to I'. In addition, the solution value in I' will be close to that in I. For example, if the c_i in I have the values $(c_1, c_2, c_3, c_4) = (1.1, 2.1, 1001.6, 1002.3)$, then if we construct I' with $(c_1', c_2', c_3', c_4') = (0, 0, 1000, 1000)$ it is easy to see that the value of any solution in I is at most 7.1 more than the value of the same solution in I'. This worst-case difference is achieved only when $x_i = 1$, $1 \leqq i \leqq 4$ is a feasible solution for I (and hence also for I'). Since $a_{ij} \leqq b_j$, $1 \leqq i \leqq n$ and $1 \leqq j \leqq m$, it follows that $F^*(I) \geqq 1002.3$ (as one feasible solution is $x_1 = x_2 = x_3 = 0$ and $x_4 = 1$). But $F^*(I) - F^*(I') \leqq 7.1$ and so $(F^*(I) - F^*(I'))/F^*(I) \approx 0.007$.

Solving I by using the procedure outlined above, the feasible assignments in $S^{(i)}$ could have the following distinct profit values: $S^{(0)} = \{0\}$, $S^{(1)} = \{0, 1.1\}$, $S^{(2)} = \{0, 1.1, 2.1, 3.2\}$, $S^{(3)} = \{0, 1.1, 2.1, 3.2, 1001.6, 1002.7, 1003.7, 1004.8\}$, $S^{(4)} = \{0, 1.1, 2.1, 3.2, 1001.6, 1002.3, 1002.7, 1003.4, 1003.7, 1004.4, 1004.8, 1005.5, 2003.9, 2005, 2006, 2007.1\}$. Thus, barring any elimination of feasible assignments resulting from the dominance rules or from any heuristic, the solution of I using the procedure outlined above would require the computation of $\sum_{i=0}^{n} |S^{(i)}| = 31$ feasible assignments. The feasible assignments for I' have the following values: $S^{(0)} = \{0\}$, $S^{(1)} = \{0\}$, $S^{(2)} = \{0\}$, $S^{(3)} = \{0, 1000\}$, $S^{(4)} = \{0, 1000, 2000\}$. Note that $\sum_{i=0}^{n} |S^{(i)}|$ is only 8. Hence I' can be solved in about one-fourth the time needed for I. An inaccuracy of at most 0.7 % is introduced.

Given the c_i's and an ϵ, what should the c_i''s be so that $(F^*(I) - F^*(I'))/F^*(I) \leqq \epsilon$ and $\sum_{i=0}^{n} |S^{(i)}| \leqq p(n)$ for some 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 $0(|S^{(i-1)}|)$. (We shall see this in greater detail in the examples.)

Let LB be an estimate for $F^*(I)$ such that $F^*(I) \geqq \text{LB}$. Clearly, we may assume $\text{LB} \geqq \max_i \{c_i\}$. If $\sum_{i=1}^{n} |c_i - c_i'| \leqq \epsilon F^*(I)$, then it is clear that $(F^*(I) - F^*(I'))/F^*(I) \leqq \epsilon$. Define $c_i' = c_i - \text{rem}(c_i, (\text{LB}\epsilon)/n)$, where

$\text{rem}(a, b)$ is the remainder of a/b, i.e., $a - \lfloor a/b \rfloor b$ (e.g., $\text{rem}(7, 6) = 1/6$ and $\text{rem}(2.2, 1.3) = 0.9$).[1] Since $\text{rem}(c_i, \text{LB}\epsilon/n) < \text{LB}\epsilon/n$, it follows that $\sum |c_i - c_i'| < \text{LB}\epsilon \leqq F^*\epsilon$. Hence, if the optimal solution to I' is used as an optimal solution for I, 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 c_i'', $1 \leqq i \leqq n$ as its objective function coefficients. Define $c_i'' = \lfloor (c_i n)/(\text{LB}\epsilon) \rfloor$, $1 \leqq i \leqq n$. It is easy to see that $c_i'' = (c_i' n)/(\text{LB}\epsilon)$. Clearly, the $S^{(i)}$'s corresponding to the solutions of I' and I'' will have the same number of tuples. (p_1, t_1) is a tuple in an $S^{(i)}$ for I' iff $((p_1 n)/(\text{LB}\epsilon), t_1)$, 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 $c_i \leqq \text{LB}$, it follows that $c_i'' \leqq \lfloor n/\epsilon \rfloor$. Hence $|S^{(i)}| \leqq 1 + \sum_{j=1}^{i} c_j'' \leqq 1 + i\lfloor n/\epsilon \rfloor$ 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.

Example 1. The 0-1 knapsack problem is formulated as

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

While solving this problem by successively generating $S^{(0)}, S^{(1)}, \cdots, S^{(n)}$, the feasible assignments for $S^{(i)}$ may be represented by tuples of the form (p, t), where $p = \sum_{j=1}^{i} c_j x_j$ and $t = \sum_{j=1}^{i} w_j x_j$. One may easily verify the validity of the following dominance rule [6, 15]: The assignment corresponding to (p_1, t_1) dominates that corresponding to (p_2, t_2) iff $t_1 \leqq t_2$ and $p_1 \geqq p_2$.

Let us solve the following instance of the 0-1 knapsack problem: $n = 5$, $M = 1112$ and $(c_1, c_2, c_3, c_4, c_5) = (w_1, w_2, w_3, w_4, w_5) = \{1, 2, 10, 100, 1000\}$. Since $c_i = w_i$, $1 \leqq i \leqq 5$, the tuples (p, t) in $S^{(i)}$, $0 \leqq i \leqq 5$ will have $p = t$. Consequently, it is necessary to retain only one of the two coordinates p, t. The $S^{(i)}$ obtained for this instance are: $S^{(0)} = \{0\}$; $S^{(1)} = \{0, 1\}$; $S^{(2)} = \{0, 1, 2, 3\}$; $S^{(3)} = \{0, 1, 2, 3, 10, 11, 12, 13\}$; $S^{(4)} = \{0, 1, 2, 3, 10, 11, 12, 13, 100, 101, 102, 103, 110, 111, 112, 113\}$; $S^{(5)} = \{0, 1, 2, 3, 10, 11, 12, 13, 100, 101, 102, 103, 110, 111, 112, 113, 1000, 1001, 1002, 1003, 1010, 1011, 1012, 1013, 1100, 1101, 1102, 1103, 1110, 1111, 1112\}$. The optimal solution has value $\sum c_i x_i = 1112$.

Now let us use rounding on the above problem instance to find an approximate solution with value at most 10 % less than the optimal value. We thus have $\epsilon = 1/10$. Also, we know that $F^*(I) \geqq \text{LB} \geqq \max\{c_i\} = 1000$. The

[1] $\lfloor x \rfloor$ is the largest integer not greater than x.

problem I' to be solved is: $n=5$, $M=1112$ $(c_1'', c_2'', c_3'', c_4'', c_5'')=(0, 0, 0, 5, 50)$ and $(w_1, w_2, w_3, w_4, w_5)=(1, 2, 10, 100, 1000)$. Hence $S^{(0)}=S^{(1)}=S^{(2)}=S^{(3)}=\{(0, 0)\}$; $S^{(4)}=\{(0, 0), (5, 100)\}$; $S^{(5)}=\{(0, 0), (5, 100), (50, 1000), (55, 1100)\}$.

The optimal solution is $(x_1, x_2, x_3, x_4, x_5)=(0, 0, 0, 1, 1)$. Its value in I'' is 55 and in the original problem 1100. The error $(F^*(I)-\hat{F}(I))/F^*(I)$ is therefore $12/1112<0.011<\epsilon$. At this time we see that the solution may be improved by setting either $x_1=1$ or $x_2=1$ or $x_3=1$.

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. Thus Ibarra and Kim [8] obtain an $0(n/\epsilon^2+n \log n)$ ϵ-approximate algorithm for the 0-1 knapsack problem and an $0(n/\epsilon^2)$ ϵ-approximate algorithm for the unrestricted nonnegative integer knapsack problem. Both their algorithms use rounding.

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 $\sum_{j=1}^{i} c_j x_j$ amongst all feasible assignments generated for $S^{(i)}$. Then the profit interval $[0, P_i]$ is divided into subintervals each of size $P_i\epsilon/n$ (except possibly the last interval, which may be a little smaller). All feasible assignments in $S^{(i)}$ with $\sum_{j=1}^{i} c_j x_j$ in the same subinterval are regarded as having the same $\sum_{j=1}^{i} c_j x_j$ 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 $S^{(n)}$. However, the error is additive. If $\hat{F}(I)$ is the value of the optimal generated by interval partitioning and $F^*(I)$ is the value of a true optimal, it follows that $F^*(I)-\hat{F}(I) \leqq (\epsilon \sum_{i=1}^{n} P_i)/n$. 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 up by starting with a good estimate, LB for $F^*(I)$ such that $F^*(I) \geqq \text{LB}$. The subinterval size is then $\text{LB}\epsilon/n$ rather than $P_i\epsilon/n$. When a feasible assignment with value greater than LB is discovered, the subinterval size can be chosen as described above.

Example 2. Consider the same instance of the 0-1 knapsack problem as in Example 1. $\epsilon=1/10$ and $F^* \geqq \text{LB} \geqq 1000$. We can start with a subinterval size of $\text{LB}\epsilon/n=1000/50=20$. Since all tuples (p, t) in $S^{(i)}$ have $p=t$, only p will be explicitly retained. The intervals are $[0, 20)$, $[40, 60)$, $\cdots$ etc.

Using interval partitioning we obtain: $S^{(0)}=S^{(1)}=S^{(2)}=S^{(3)}=\{0\}$; $S^{(4)}=\{0, 100\}$; $S^{(5)}=\{0, 100, 1000, 1100\}$.

The optimal generated by interval partitioning is $(x_1, x_2, x_3, x_4, x_5)=(0, 0, 0, 1, 1)$ and the value $\hat{F}(I)=1100$. $(F^*(I)-\hat{F}(I))/F^*(I)=12/1112<0.011<\epsilon$. Again, the solution value may be improved by using a heuristic to change some of the x_i's from 0 to 1.

Separation

Assume that in solving a problem instance I, we have obtained an $S^{(i)}$ with feasible solutions having the following values of $\sum_{j=1}^{i} c_j x_j$: 0, 3.9, 4.1, 7.8, 8.2, 11.9, 12.1. Further assume that the interval size $P_i\epsilon/n$ 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 so no feasible assignments are eliminated. However, there are three pairs of assignments with values within $P_i\epsilon/n$. If the dominance rules are used for each pair, only 4 assignments will remain. The error introduced is at most $P_i\epsilon/n$. More formally, let $a_0, a_1, a_2, \cdots, a_r$ be the distinct values of $\sum_{j=1}^{i} c_j x_j$ in $S^{(i)}$. Let us assume $a_0<a_1<a_2 \cdots <a_r$. 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 $P_i\epsilon/n$. This is described by the following algorithm:

```
J←assignment corresponding to a_0; XP←a_0
for j←1 to r do
  if a_j>XP+P_iε/n then
  [put assignment corresponding to a_j into J, XP←a_j]
end.
```

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 always to work better than interval partitioning. The following example illustrates that this need not be the case. However, empirical studies with one problem (see Section 2) indicate that interval partitioning is inferior in practice.

Example 3. Using separation on the data of Example 1 yields the same $S^{(i)}$ as obtained using interval partitioning. We have already seen an instance where separation performs better than interval partitioning. Now we shall see an example where interval partitioning does better than separation. Assume that the subinterval size $\text{LB}\epsilon/n$ is 2. Then the intervals are $[0, 2)$, $[2, 4)$, $[4, 6) \cdots$ etc. Assume further that $(c_1, c_2, c_3, c_4, c_5)=(3, 1, 5.1, 5.1, 5.1)$. Then, following the use of interval partitioning, we have: $S^{(0)}=\{0\}$; $S^{(1)}=\{0, 3\}$; $S^{(2)}=\{0, 3, 4\}$; $S^{(3)}=\{0, 3, 4, 8.1\}$; $S^{(4)}=\{0, 3, 4, 8.1, 13.2\}$; $S^{(5)}=\{0, 3, 4, 8.1, 13.2, 18.3\}$.

Using separation with $LB\epsilon/n=2$ we have: $S^{(0)}=\{0\}$; $S^{(1)}=\{0, 3\}$; $S^{(2)}=\{0, 3\}$; $S^{(3)}=\{0, 3, 5.1, 8.1\}$; $S^{(4)}=\{0, 3, 5.1, 8.1, 10.2, 13.2\}$; $S^{(5)}=\{0, 3, 5.1, 8.1, 10.2, 13.2, 15.3, 18.3\}$.

The three methods for obtaining fully polynomial time approximation schemes can be applied to a wide variety of problems. Some of these problems are 0-1 knapsack problem [8]; integer knapsack problem [8]; job sequencing with deadlines [18]; minimizing weighted mean finish time [7, 18]; finding an optimal SPT schedule [18]; and finding minimum finish time schedules on identical, uniform and nonidentical machines [7, 18].

2. JOB SEQUENCING WITH DEADLINES

The problem here is to sequence a set of n jobs onto one machine. Each job i has associated with it a processing time requirement t_i, a deadline d_i, and a profit p_i which is earned only if the processing of job i is completed by d_i. If a job is not processed by its deadline, no profit is earned. The objective is to find a schedule (i.e., permutation of the n jobs) that maximizes the profit.

Any feasible solution may be represented as a tuple (p, σ), where σ is a permutation specifying the schedule and p the profit. It is clear that all jobs that are not completed by their deadlines might as well be processed after all jobs that are. Furthermore, those jobs not completed by their deadlines can be processed in any order. Hence, it is sufficient if σ specifies only those jobs that are to be completed by their deadlines and the required order for them. It is well-known [12] that if J represents a subset of jobs that can be completed by their deadlines, one possible permutation is to process jobs in order of nondecreasing deadlines. Hence, if we assume $d_{i+1} \geqq d_i$, $1 \leqq i < n$, the job sequencing with deadlines problem may be formulated as

$$\begin{aligned} \max \ & \textstyle\sum_{i=1}^{n} p_i x_i \\ & \textstyle\sum_{j=1}^{i} t_i x_i \leqq d_j, \ 1 \leqq j \leqq n \\ & x_i = 0 \text{ or } 1, \ 1 \leqq i \leqq n. \end{aligned}$$

In solving this problem, a feasible assignment $x_i=y_i, 1 \leqq i \leqq k$ will be represented as a tuple (p, t), where $p=\sum_{i=1}^{k} p_i y_i$ and $t=\sum_{i=1}^{k} t_i y_i$. The values y_i can be reconstructed from the sets $S^{(i)}$, $0 \leqq i \leqq n$.

The dominance rule is fairly straightforward: (p_1, t_1) dominates (p_2, t_2) iff $t_1 \leqq t_2$ and $p_1 \geqq p_2$.

Our first pass at obtaining an exact algorithm would result in the following:

1. $S^{(0)} \leftarrow \{(0, 0)\}$ //null assignment//
2. **for** $i \leftarrow 1$ **to** n **do**
3. Generate $S^{(i)}$ from $S^{(i-1)}$ using the dominance rules to eliminate tuples
4. **end**
5. optimal solution value corresponds to tuple in $S^{(n)}$ with maximum p
6. an optimal schedule can be found using a backward search starting from $S^{(n)}$.

In the appendix we will fill in the details of the algorithm and show that when the p_i's, t_i's and d_i's are integer, its worst-case computing time is $0(\min\{2^n, n\sum_{j=1}^{n} p_j, n\sum_{j=1}^{n} t_j, \sum_{j=1}^{n} d_j)$. The complete algorithm of the appendix will be referred to as JSD.

The first thing we can do to improve the performance of JSD is introduce heuristics that will tend to reduce $|S^{(i)}|$. A possible heuristic is:

Heuristic. Let LB be a lower bound on the value of an optimal solution. Let $PLEFT=\sum_{j=i+1}^{n} p_j$. Then all tuples in $S^{(i)}$ with profit value less than LB-PLEFT can be discarded, as these tuples cannot lead to a tuple in $S^{(n)}$ with profit value greater than or equal to LB.

The power of this heuristic clearly lies in our ability to get a good estimate for LB. One way to estimate LB dynamically in JSD is to consider the tuple (p, t) with the largest p generated so far. We may use $LB=p$. A better estimate may be obtained by considering (p, t) as before and then adding to this some of the tasks not yet considered. The appendix describes how this heuristic may be introduced in JSD without introducing excessive overhead. The resulting algorithm is called E. Slight modifications to algorithm E result in its transformation into a fully polynomial time approximation scheme. The appendix describes the necessary modifications for both interval partitioning and separation. The resulting algorithms are called I and S, respectively, and both have a computing time of $0(n^2/\epsilon)$.

The algorithms E, I, and S were programmed in FORTRAN and run on a CDC CYBER 74. The programs were compared over a variety of data. Algorithms I and S were run with $\epsilon=0.1$. Three data sets were used:

Data Set A: random profits $p_i \epsilon [1, 100]$, $t_i=p_i$ and $d_i=\sum_{i=1}^{n} t_i/2$.

Data Set B: random $p_i \epsilon [1, 100]$; $t_i=p_i$ and random $d_i \epsilon [t_i, t_i+25n]$.

Data Set C: random $p_i \epsilon [1, 100]$; random $t_i \epsilon [1, 100]$ and random $d_i \epsilon [t_i, t_i+25n]$.

The program had a capacity to solve all problems generating no more than 9000 tuples (i.e., $\sum_{i=0}^{n} |S^{(i)}| \leqq 9000$). For each data set an attempt was made to run 10 problems each of size 5, 15, 25, 35, 45, $\cdots$. Tables I–III give the average times and space requirements for the three algorithms. The standard deviation in the observed space and time requirements is also given. The first thing to note is that the time and space requirements for algorithm E do not grow as rapidly as indicated by the worst-case analysis. (This of course does not mean that they will not grow in such a manner for some other data set.) Secondly, the standard deviation in time and space requirements even within a given data set is so large that it does not make sense to talk about a "typical" computing

time for the three algorithms. For a fixed n the time and space required by each of the algorithms is highly dependent on the specific values of the p_i, t_i and d_i. For all three data sets the approximation algorithms outperformed the exact one on both space and time for $n \geqq 15$. In the case of data set A the difference is most marked. On all the problems generated, algorithm S generated fewer tuples than did algorithm I (thus lending support to our intuitive evaluation of the separation technique). Algorithm S also required less time than algorithm I except on problems from data set C with $n \leqq 65$.

TABLE I
DATA SET A

N	Time (msec) per algorithm:						Space (no. of tuples) per algorithm:					
	E		I		S		E		I		S	
	Mean	STD	Mean	STD	Mean	STD	Mean	STD	Mean	STD	Mean	STD
5	1.6	0.66	2.6	0.49	2.2	0.98	22.1	7.2	20.9	6.07	19.9	5.8
15	51.3	7.8	33.8	4.31	25.6	3.72	1350.1	208.49	635.1	80.81	557.2	84.31
25	200.6	13.3	109.2	6.68	85.5	4.92	5746.1	423.06	2122.5	129.65	1912.5	115.49
35	**	**	231.6	30.73	186.7	26.56	**	**	4652.9	678.65	4217.3	640.59
45	**	**	**	**	331.00	34.71	**	**	**	**	7560.9	863.13

(**) Problem generated more than 9000 tuples.

TABLE II
DATA SET B

N	Time (msec) per algorithm:						Space (no. of tuples) per algorithm:					
	E		I		S		E		I		S	
	Mean	STD	Mean	STD	Mean	STD	Mean	STD	Mean	STD	Mean	STD
5	2.3	0.90	2.4	0.49	2.6	0.49	23.0	9.27	21.8	7.68	21.0	7.39
15	34.0	9.98	23.2	5.11	17.4	4.15	819.9	267.95	391.3	104.18	332.7	90.82
25	107.5	23.75	59.8	12.46	48.7	11.07	2776.1	638.57	1073.8	242.96	955.9	245.6
35	**	**	149.4	22.48	123.3	21.34	**	**	2806.6	489.31	2634.8	505.74
45	**	**	244.1	33.84	200.7	28.24	**	**	4522.5	734.99	4024.1	701.74
55	**	**	306.4	67.29	255.0	55.91	**	**	5678.4	1372.94	5345.7	1282.97

(**) Generated more than 9000 tuples.

Even though algorithms I and S were run with $\epsilon = 0.1$, the observed differences were much less than this. Of the 140 problems for which the optimal solution value was obtained by algorithm E, I generated 90 optimal solutions while S generated 85 optimal solutions. The breakdown by data sets is summarized in Table IV.

Finally, it should be mentioned that Kohler [11] has also applied interval partitioning to the job sequencing with deadlines problems. He uses heuristics similar to the ones discussed here and also provides a comparison on the performance of the exact algorithm E with and without the heuristics. Using different test data he concludes that algorithm I does not represent a significant improvement over algorithm E. A close look at his results indicates that the number of tuples generated by both E and I were almost the same and so, for the test data used, algorithm E actually had only a $0(n^2)$ time.

TABLE III
DATA SET C

N	Time (msec per) algorithm:						Space (no. of tuples) per algorithm:					
	E		I		S		E		I		S	
	Mean	STD	Mean	STD	Mean	STD	Mean	STD	Mean	STD	Mean	STD
5	2.1	0.94	2.00	0.77	2.1	1.3	11.4	5.04	10.9	5.07	10.7	4.92
15	9.2	2.75	7.7	1.35	9.1	2.62	84.2	30.5	76.0	22.28	71.9	18.61
25	23.7	5.1	21.1	4.06	24.8	5.53	293.9	83.21	268.8	63.79	260.5	61.45
35	57.6	14.47	45.3	10.82	55.1	11.5	784.5	246.01	622.9	194.36	570.9	181.04
45	88.8	23.06	69.4	20.9	71.4	19.44	1177.6	509.71	964.4	392.2	916.80	387.91
55	151.5	35.34	121.1	29.53	137.0	37.17	2326.1	694.43	1836.0	558.83	1734.7	548.01
65	238.2	50.72	172.6	28.78	192.4	31.64	3805.6	1083.81	2720.0	572.4	2574.7	536.85
75	305.1	41.93	257.2	37.84	222.0	30.69	6011.6	1006.02	4089.2	733.72	3885.2	697.67
85	**	**	308.7	55.28	266.1	44.99	**	**	4832.5	1003.65	4557.1	1005.68

(**) Generated more than 9000 tuples.

TABLE IV
RELATIVE PERFORMANCE OF I AND S

Performance	Data set		
	A	B	C
Total number of problems solved by algorithm E	80	30	30
Optimal solutions by I	54	20	16
Optimal solutions by S	53	18	14
Average fractional error in nonoptimal solutions by I	0.0025	0.0047	0.0040
Average fractional error in nonoptimal solutions by S	0.0024	0.0047	0.0040
No. of I solutions better than S	3	7	9
No. of S solutions better than I	1	7	6

On the basis of our results, Kohler's results, and the theoretical analysis,

we can conclude that for some problem instances I and S will have a run time almost the same as E and for other instances I and S will outperform E. In general, we expect S to perform better than I.

APPENDIX

We describe the details of the implementations of algorithms JSD, E, I, and S discussed in Section 2. The algorithms are described in the algorithmic language SPARKS [6] rather than the language in which they were programmed, FORTRAN, since it is much easier to read and understand a SPARKS algorithm than a FORTRAN program. Coding a SPARKS algorithm as a FORTRAN program is a trivial task.

Algorithm JSD is arrived at by building upon the first pass of the exact algorithm discussed in Section 2. Let us concentrate on lines 2–5. Line 6 of the first pass algorithm is fairly straightforward and simply involves the use of binary search. This will become apparent once we discuss the implementation of lines 2–5. The sets $S^{(i)}$ are of varying size. Each element is a pair of numbers (p, t). To use space efficiently, we represent all the $S^{(i)}$ in two arrays P and T with F_i pointing to the start of $S^{(i)}$. To be able to implement the domination rule efficiently, each $S^{(i)}$ is generated in such a way that if (p_1, t_1) and (p_2, t_2) are two tuples in $S^{(i)}$ and $t_1 < t_2$ then (p_1, t_1) precedes (p_2, t_2). (Note that because of the domination rules it must be that $p_1 < p_2$, as otherwise (p_2, t_2) gets eliminated.) In generating $S^{(i)}$ from $S^{(i-1)}$ we note that when $x_i = 0$, all the tuples in $S^{(i-1)}$ represent feasible assignments. When $x_i = 1$ the feasible assignments are obtained from the tuples of $S^{(i-1)}$ as follows: If (p, t) is a tuple in $S^{(i-1)}$ then $(p+p_i, t+t_i)$ represents a feasible assignment iff $t+t_i \leqq d_i$. Thus one way to generate $S^{(i)}$ from $S^{(i-1)}$ would be to first generate the set $J = \{(p+p_i, t+t_i) | (p, t) \epsilon S^{(i-1)}$ and $t+t_i \leqq d_i\}$ and then to combine the tuples in $S^{(i-1)}$ and J together, obtaining $S^{(i)}$. During this step the dominance rule can be used to ensure that the tuples in $S^{(i)}$ have distinct p's and t's. Our implementation of this process generates the elements of J in parallel with the generation of $S^{(i)}$. As each new element of J is generated, the domination rules is used to merge this term together with all remaining terms in $S^{(i-1)}$ with a smaller or equal time coordinate into $S^{(i)}$. Algorithm JSD uses a subalgorithm PARTS, which carries out the function of line 6 of our first pass algorithm. In addition, the dominance rule is used in parallel with the generation of $S^{(i)}$. The inputs to the algorithm are: $n \cdots$number of jobs; $p \cdots p_i$ is profit for job i; $t \cdots t_i$ is time needed by job i; $d \cdots d_i$ is deadline for job i: $d_i \geqq d_{i-1}$, $2 \leqq i \leqq n$. The other variables used in JSD have the following meaning: $(P_j, T_j) \cdots$a tuple in one of the $S^{(i)}$'s; $F_i \cdots$index of first tuple in $S^{(i)}$, (P_j, T_j), $F_i \leqq j < F_{i+1}$ are the tuples in $S^{(i)}$; $XP \cdots$profit value of last tuple put in $S^{(i)}$; $k \cdots$index of next tuple from $S^{(i-1)}$ to be merged into $S^{(i)}$; next $\cdots$index of next tuple to be generated for $S^{(i)}$; $l$point $\cdots$index of first tuple in $S^{(i-1)}$; $h$point $\cdots$index of last tuple in $S^{(i-1)}$.


```
Line    procedure JSD(n, p, t, d)
 1        F0←1; P1←0; T1←0 //initialize S(0)//
 2        lpoint←hpoint←1 //start and end of S(0)//
 3        next←2; F1←2
 4        for i←1 to n do //compute S(i)//
 5          k←lpoint; XP←−1
 6          u←largest m, lpoint≦m≦hpoint, such that Tm+ti≦di
 7          for j←lpoint to u do //generate J and merge//
 8            (p', t')←(Pj+pi, Tj+ti)
               //merge in from S(i−1) using dominance rules//
 9            while k≦hpoint and Tk<t' do
10              if Pk>XP then [Pnext←XP←Pk
11                             Tnext←Tk
12                             next←next+1]
13              k←k+1
14            end
15            if k≦hpoint and Tk=t' then [p'←max {p', Pk}
16                                        k←k+1]
17            if p'>XP then [Pnext←XP←p'
18                           Tnext←t'
19                           next←next+1]
20          end
            //merge in remaining terms from S(i−1)//
21          j←least m, k≦m≦hpoint, such that Pm>XP
22          for l←j to hpoint do
23            Pnext←Pl; Tnext←Tl; next←next+1
24          end
            //initialize for S(i+1)
25          lpoint←hpoint+1; hpoint←next−1; Fi+1←next
26        end
27        call PARTS     //Phpoint is optimal solution value//
28      end JSD.
```

Analysis of JSD

Lines 1–3 take $0(1)$ time. For each iteration of the loop of lines 4–26, line 5 takes $0(1)$ and line 6 $0(\log |S^{(i-1)}|)$ if binary search is used. The time for lines 7–24 is $0(|S^{(i-1)}|)$. This follows from the observation that in each iteration of the loops of lines 9–14 and 22–24, k increases by 1. The total increment in k is $|S^{(i-1)}|$. The number of iterations of the loop of lines 7–20 is $(u - l\text{point} + 1)$, which is no more than $|S^{(i-1)}|$. Line 21 takes no more than $\log(|S^{(i-1)}|)$ time. Line 25 takes $0(1)$ time. Hence each iteration of the loop of lines 4–26 requires $0(|S^{(i-1)}|)$ time. The total time for this loop is therefore $0(\sum |S^{(i-1)}|)$. The procedure PARTS can use binary search on $S^{(n)}, S^{(n-1)}, \cdots, S^{(1)}$ to find the optimal solution. Hence the time for this is $0(\sum |\log S^{(i-1)}|)$. The total time for JSD is therefore $0(\sum |S^{(i-1)}|)$. In the worst case $|S^{(i)}| = 2|S^{(i-1)}|$. With $|S^{(0)}| = 1$, this gives

$\sum_{i=1}^{n} |S^{(i-1)}| \leq \sum_{i=0}^{n-1} 2^i = 2^n - 1$. The worst-case time for JSD is therefore $0(2^n)$. In case all the p_i, t_i and d_i are integer then, as a consequence of the dominance rules, it follows that $|S^{(i)}| \leq \min\{\sum_{j=1}^{i} p_j, \sum_{j=1}^{i} t_j, d_i\} + 1$. In this case we obtain $0(\min\{2^n, n\sum_{j=1}^{n} p_j, n\sum_{j=1}^{n} t_j, \sum_{i=1}^{n} d_i\})$ as the time bound for JSD. The space bound is the same as the time bound. Algorithm JSD can be speeded up slightly by carrying out the iteration of line 4 only for $1 \leq i < n$. Then the index from line 6 will give an optimal solution.

The heuristic discussed in Section 2 may be introduced into algorithm JSD by introducing the following code after line 4:

4.1 $(p, t) \leftarrow (p_{h\text{point}}, t_{h\text{point}})$
4.2 **for** $j \leftarrow i$ **to** n **do**
4.3 **if** $t + t_j \leq d_j$ **then** $[t \leftarrow t + t_j;\ p \leftarrow p + p_j]$
4.4 **end.**

For some data sets it is possible that p as generated above is less than the previous estimate of the lower bound. Hence, if LB is the previous lower bound used, then the new lower bound to use becomes:

4.5 LB$\leftarrow$max {LB, p}.

Even though this heuristic may seem simple, experimental results will show it to be very effective. The remaining changes needed in JSD to implement the heuristic are:

(i) Add line 3.1 PLEFT$\leftarrow \sum_{i=1}^{n} p_i$;
(ii) Add line 4.6 PLEFT$\leftarrow$PLEFT$-p_i$;
(iii) Replace line 5 with

5 $k \leftarrow$least m, $l$point$\leq m \leq h$point such that PLEFT$+P_m \geq$ LB;
5.1 $XP \leftarrow P_k - 1$.

It is easy to see that the inclusion of the heuristic to JSD does not alter the worst-case computing time (except by a constant factor). We call the resulting exact algorithm, algorithm E. In order to introduce the interval partitioning scheme into algorithm E, we note that the computation for LB will have to be altered slightly. As a result of discarding some tuples in going from $S^{(i-1)}$ to $S^{(i)}$, it is likely that the LB as computed for $S^{(i-1)}$ may be more than the best obtainable from $S^{(i)}$. If DEL is the interval size used, then the optimal obtainable from $S^{(i)}$ is at least LB-DEL. Hence line 4.5 should be changed to

4.5 LB$\leftarrow$max {LB-DEL, p}.

The interval size is readily computed as LBϵ/n. By making appropriate changes to lines 10, 17–19 and 23, the interval partitioning scheme can be carried out in parallel with the generation of $S^{(i)}$ from $S^{(i-1)}$. The resulting computing time is $0(n^2/\epsilon)$. Call this algorithm, algorithm I.

The approximation algorithm incorporating the separation scheme is obtained by computing DEL as above and by changing line 4.5 of algorithm E as described above.

In addition, the following changes are made:

(i) Line 10—replace conditional by $P_k > XP + \text{DEL}$;
(ii) Lines 17–19 are executed only if $p' > XP + \text{DEL}$;
(iii) Line 21—replace XP by $XP + \text{DEL}$;
(iv) Replace line 23 by

if $P_l > XP + \text{DEL}$ **then** $[P_{\text{next}} \leftarrow XP \leftarrow P_l$
$T_{\text{next}} \leftarrow T_l$
next$\leftarrow$next$+1]$.

The computing time for the resulting algorithm is $0(n^2/\epsilon)$. Call this algorithm, algorithm S.

It should be clear from our discussion on the implementation of the two approximation methods that the overhead involved is small. This is borne out by the experimental results presented in Section 2.

ACKNOWLEDGMENT

This research was supported in part by NSF grants DCR 74-10081 and MCS 76-21024.

REFERENCES

1. R. Bellman and S. Dreyfus, *Applied Dynamic Programming*, Princeton University Press, Princeton, N.J., 1962.
2. M. Garey and D. Johnson, "Approximation Algorithms for Combinatorial Problems: An Annotated Bibliography," in *Algorithms and Complexity*, J. Traub (Ed.), pp. 41–52, Academic Press, New York, 1976.
3. M. Garey and D. Johnson, "Approximation Algorithms for Combinatorial Problems: Prospects and Limitations," Lecture by D. Johnson presented at the Symposium on Algorithms and Complexity, Carnegie Mellon Institute, Pittsburgh, 1976.
4. R. Graham, "Bounds for Certain Multiprocessing Anomalies," *Bell Systems Tech. J.* **4**, 1563–1581 (1966).
5. E. Horowitz and S. Sahni, "Computing Partitions with Applications to the Knapsack Problem," *J. Assoc. Comput. Machinery* **21**, 277–292 (1974).
6. E. Horowitz and S. Sahni, *Fundamentals of Data Structures*, Computer Science Press, Woodland Hills, Calif., 1976.
7. E. Horowitz and S. Sahni, "Exact and Approximate Algorithms for Scheduling Nonidentical Processors," *J. Assoc. Comput. Machinery* **23**, 317–327 (1976).
8. O. Ibarra and C. Kim, "Fast Approximation Algorithms for the Knapsack and Sum of Subsets Problems," *J. Assoc. Comput. Machinery* **22**, 463–468 (1975).
9. D. Johnson, "Approximation Algorithms for Combinatorial Problems," *J. Comput. Syst. Sci.* **9**, 256–278 (1974).

10. M. KARP, "Reducibility among Combinatorial Problems," in *Complexity of Computer Computations*, pp. 85–103, R. E. Miller and J. W. Thatcher (Eds.), Plenum Press, New York, 1972.
11. W. KOHLER, "Computational Experience with Efficient Exact and Approximate Algorithms for an NP-Complete Scheduling Problem," Technical Report ECE-CS-75-13, University of Massachusetts, Amherst, December 1975.
12. E. LAWLER AND J. MOORE, "A Functional Equation and Its Application to Resource Allocation and Sequencing Problems," *Management Sci.* **16,** 85–103 (1969).
13. S. LIN AND B. KERNIGHAN, "An Effective Heuristic Algorithm for the Travelling Salesman Problem," *Opns. Res.* **21,** 498–516 (1973).
14. G. NEMHAUSER, *Introduction to Dynamic Programming*, John Wiley & Sons, New York, 1966.
15. G. NEMHAUSER AND Z. ULLMAN, "Discrete Dynamic Programming and Capital Allocation," *Management Sci.* **15,** 494–505 (1969).
16. S. SAHNI, "Computationally Related Problems," *SIAM J. Computing* **3,** 277–292 (1974).
17. S. SAHNI, "Approximate Algorithms for the 0/1 Knapsack Problem," *J. Assoc. Comput. Machinery* **22,** 115–124 (1975).
18. S. SAHNI, "Algorithms for Scheduling Independent Tasks," *J. Assoc. Comput. Machinery* **23,** 114–127 (1976).
19. S. SAHNI AND T. GONZALEZ, "*P*-Complete Approximation Problems," *J. Assoc. Comput. Machinery* **23,** 555–565 (1976).