OPERATIONS RESEARCH
Vol. 26, No. 1, January-February 1978

0030-364X/78/2601-0003 $01.25
© 1978 Operations Research Society of America

Performance Guarantees for Scheduling Algorithms

M. R. GAREY, R. L. GRAHAM, and D. S. JOHNSON
Bell Laboratories, Murray Hill, New Jersey

(Received June 1976; accepted September 1976)

One approach to coping with the apparent difficulty of many schedule-optimization problems, such as occur in machine shops and computer processing, is to devise efficient algorithms that find schedules guaranteed to be "near-optimal." This paper presents an introduction to this approach by describing its application to a well-known multiprocessor scheduling model and illustrating the variety of algorithms and results that are possible. The paper concludes with a brief survey of what has been accomplished to date in the area of scheduling using this approach.

SCHEDULING PROBLEMS arise in many practical circumstances and under a wide variety of guises. Many are basically optimization problems having the following form: Given a collection of tasks to be scheduled on a particular processing system, subject to various constraints, find a feasible schedule that minimizes (or in some cases maximizes) the value of a given objective function. Thus it is not surprising that much of the work on scheduling theory has been devoted to the design and analysis of *optimization algorithms*—algorithms that, when given a particular instance of a scheduling problem, construct an optimal feasible schedule for that instance.

Unfortunately, although it is not difficult to design optimization algorithms (e.g., exhaustive search is usually applicable), the goal of designing efficient optimization algorithms has proved much more difficult to attain. In fact, all but a few (e.g., [6, 11, 21, 27]) schedule-optimization problems are considered insoluble except for small or specially structured problem instances. For these scheduling problems no efficient optimization algorithm has yet been found and, indeed, none is expected. This pessimistic outlook has been bolstered by recent results showing that most scheduling problems belong to the infamous class of "NP-complete problems" [4, 30, 35, 36]).

For these reasons practitioners often are willing to settle merely for a feasible schedule, so long as they have some indication that it is a reasonably good one. Though there are problems for which finding even a feasible schedule seems computationally hopeless, for most schedule-optimization problems there do exist simple heuristic algorithms that find feasible schedules quickly. In this paper we shall discuss one particularly attractive approach to analyzing how good, relative to optimal schedules, are the schedules constructed by such heuristics. This approach deals with proving "performance guarantees."

At this point it is convenient to introduce a bit more formalism into our discussion. A scheduling *problem* will consist of two parts: a *model* and an *objective function.* The model describes the system, including the kinds of tasks and the constraints on their processing, the types of processors and their number, and all other properties necessary to specify feasible schedules. The objective function assigns a "value" to each feasible schedule. An *instance* of such a problem is merely a specification of a particular system, set of tasks, and set of constraints conforming to the model. Given an instance I, the model tells us what the feasible schedules for I look like and the objective function tells us what their values are. The *optimal schedule value* for I, which we denote by $\mathrm{OPT}(I)$, is the minimum (or in some cases the maximum) of the values for all feasible schedules. An *optimal schedule* is a feasible schedule for I having value $\mathrm{OPT}(I)$.

A *scheduling algorithm* A for a particular problem is a procedure that, given any instance I of that problem, produces a feasible schedule for I. We let $A(I)$ denote the value of the schedule found by A when applied to I. If $A(I)$ always equals $\mathrm{OPT}(I)$, then we call A an *optimization algorithm.* Otherwise, we shall call A an *approximation algorithm*, with the implied hope that A will find near-optimal schedules.

A traditional method for evaluating approximation algorithms has been to run them on selected sample problem instances. Indeed, for some algorithms this is still the only practical approach. However, this approach does suffer from some major drawbacks. First is the difficulty of choosing a convincing set of realistic sample problem instances. There is always the danger of omitting, through choice or accident, the instances for which our algorithm performs poorly. Second is the difficulty of using this approach to obtain an absolute performance measure on our algorithm. Since it is so difficult to find the optimal schedules to which we would like to compare our constructed schedules, this evaluation approach seems better adapted to comparing alternative heurstics than to determining how "near-optimal" our algorithm is.

An alternative theoretical approach is to evaluate approximation algorithms using probabilistic techniques. For example, one might derive the expected values of $A(I)$ and $\mathrm{OPT}(I)$ and compare them. Results of this type (though not exclusively for scheduling problems) have appeared recently [31]. Although this approach can yield useful and interesting information about algorithms, it too has its drawbacks. First of all, it is often very difficult to determine a probability distribution that can be dealt with mathematically and that also mirrors the problem instances that arise in practice. Secondly, given such a distribution, the current state of our abilities is such that many additional, and unjustified, assumptions must usually be made to complete a probabilistic analysis of even the most simple-minded algorithms.

Moreover, both these approaches suffer from another drawback. Although they may give some indication of average case performance, neither approach tells you anything about how close $A(I)$ will be to $\mathrm{OPT}(I)$ in a particular instance. It is here that the performance guarantee approach can give useful results.

Basically, a performance guarantee is a theorem that bounds the worst-case behavior of a particular approximation algorithm. For a minimization problem it might take the form: "For all instances I, $A(I) \leqq r \cdot \mathrm{OPT}(I) + d$," where $r \geqq 1$ and d are specified constants. In general, the additive constant d will be asymptotically negligible (and for many results of this type it is 0). The dominant factor will be the ratio r.

In analyzing an algorithm we will want to find the smallest r for which such a theorem can be proved, i.e., the best possible guarantee. We can show that no better bound can be proved by constructing problem instances on which the algorithm performs essentially as poorly as the bound allows. More precisely, if we can show that for some constant d' and all $N > 0$ there exist problem instances I with $\mathrm{OPT}(I) > N$ and $A(I) > r' \cdot \mathrm{OPT}(I) - d'$, then clearly no performance bound can be proved with $r < r'$. Just as our upper-bound theorems provide a guarantee that the algorithm will never do worse than is stated, these examples provide a warning as to how bad it may be. We say that we have determined the worst-case behavior of an algorithm if the ratio r of our guarantee is the same as the ratio r' of our warning. This common value then must be the smallest ratio that can be guaranteed by the algorithm, and we call it the *worst-case performance ratio* or simply the *performance ratio* for the algorithm.

By determining the performance ratio for an approximation algorithm, we obtain information that supplements that obtained by the other approaches and that provides us with a useful and rigorously defined quantity with which different approximation algorithms can be compared. There are, of course, drawbacks to worst-case analysis, too. In practice an algorithm may perform much better than it does in the worst case. The problem instances causing worst-case behavior may be contrived and unnatural. However, a good worst-case bound may be reassuring in a way that an average case result cannot be. It gives a bound that always holds, no matter what the problem instance. If nothing else, algorithms with performance guarantees can make ideal starting points for branch-and-bound techniques or more elaborate heuristics.

In Section 1 we illustrate the performance-guarantee approach by considering algorithms for one of the most basic problems of deterministic scheduling theory. These algorithms will provide examples of the variety of phenomena that can occur. We also give some indication of how one goes about proving performance guarantees and analyzing worst-case behavior. Section 2 presents a brief survey of the literature, indicating other scheduling problems for which such results have been proved and discussing how some of these results are interrelated. Section 3 concludes briefly with some general observations on the design of good approximation algorithms and the methodology of proving performance guarantees.

1. SCHEDULING INDEPENDENT TASKS

One model that fits many scheduling problems is the following. The system is a set of m identical processors $\{P_1, \cdots, P_m\}$ that operate in parallel. The set $\mathbf{T} = \{T_1, \cdots, T_n\}$ of tasks to be executed consists of *independent* tasks (no ordering constraints between them), each of which has an execution time $\tau(T_i)$ and requires only one processor.

The only processing constraints are that no processor can execute more than one task at a time and that, once a processor begins executing a task T_i, it continues executing T_i until its completion $\tau(T_i)$ time units later (non-preemptive scheduling).

A feasible schedule for $\mathbf{T}$ assigns to each $T_i \in \mathbf{T}$ a processor $p(T_i)$, $1 \leqq p(T_i) \leqq m$, and a starting time $s(T_i) \geqq 0$ such that, if $p(T_i) = p(T_j)$ and $i \neq j$, then the two execution intervals $(s(T_i), s(T_i) + \tau(T_i))$ and $(s(T_j), s(T_j) + \tau(T_j))$ are disjoint.

We may represent a schedule by a "Gantt chart" as shown in Figure 1. Each task is represented by a rectangle whose length corresponds to its execution time, and for each processor we have a row consisting of the tasks it executes. The cross-hatched regions represent times during which a processor is idle.

Although this description of the model has been in abstract mathematical terms, the model does correspond to a simplified version of many real processing systems. Examples are a computer center containing a number of computers or even a typing pool, where the typists are the processors and the letters and papers to be typed are the tasks.

In such a system one desirable goal might be to get all the work done as soon as possible. We shall be interested primarily in the objective function corresponding to this goal, called the *finishing time* or *makespan* of a schedule. The finishing time for a schedule is the earliest time at which all tasks have been completed and is equal to $\max\{s(T_i) + \tau(T_i) : T_i \in \mathbf{T}\}$. The finishing time for the schedule in Figure 1 is 6.

This problem of finding a feasible schedule with minimum finishing time, which we shall call the *independent task-scheduling problem*, is a well-known example of the type of intractable problem mentioned above. It is NP-complete for $m \geqq 2$, and no optimization algorithm is known that is practical for anything besides very small problem instances when $m \geqq 3$. (Certain optimization algorithms based on the knapsack problem seem to work well in practice for $m=2$, despite the fact that for some problem instances they can take exponential time.)

Therefore, we lower our sights somewhat and seek quick ways to generate feasible schedules. There are, of course, very simple ways to do this. For instance, one could assign all the tasks to the same processor and set $s(T_i) = \sum_{j=1}^{i-1} \tau(T_i)$ for $1 \leqq i \leqq n$. This clearly will not yield very good schedules. In fact, it is easy to see that there are problem instances I for which this algorithm A has $A(I) = m \cdot OPT(I)$. Thus in searching for approximation algorithms that generate good schedules, we should use at least a modicum of intelligence. Consider the following method, called the *list-scheduling algorithm*, for generating feasible schedules.

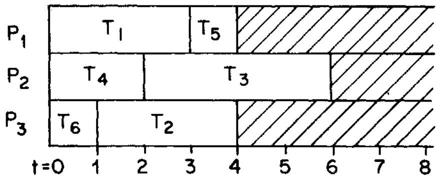


Figure 1. A Gantt chart.

Let F_j denote the current finishing time of processor P_j in the partial schedule constructed so far. Initially we set all $F_j = 0$. We then treat the tasks in the order given, assigning each task in turn to the processor with the current earliest finishing time (so as to minimize the increase in the latest finishing time). More formally, we start with $i=1$ and proceed as follows:

1) Choose the smallest $j \geqq 1$ such that $F_j \leqq F_{j'}$, $1 \leqq j' \leqq m$. (Find a processor with the earliest finishing time.)
2) Set $p(T_i) = j$, $s(T_i) = F_j$. (Assign T_i to that processor and start T_i as soon as all earlier tasks on that processor are completed.)
3) Update $F_j = F_j + \tau(T_i)$.
4) If $i=n$, the schedule is completed and has overall finishing time equal to $\max\{F_j : 1 \leqq j \leqq m\}$. Otherwise, set $i=i+1$ and go on to step 1 for the new T_i.

This is clearly a fast method for constructing feasible schedules. It can be implemented to run in time proportional to $n \log m$. Moreover, our introduction of a little common sense into the procedure has paid off, as the following performance-guarantee theorem shows.

THEOREM 1 [16]. *If A is the list-scheduling algorithm and I is an instance of the independent task-scheduling problem with m processors, then* $A(I) \leqq (2-1/m)\mathrm{OPT}(I)$.

How does one go about proving such a result when, as we have already seen, determining the precise value of $\mathrm{OPT}(I)$ is so hard? In this case we don't need to know the precise value of $\mathrm{OPT}(I)$; lower bounds on $\mathrm{OPT}(I)$ will do. Clearly, two lower bounds that must be satisfied are $\mathrm{OPT}(I) \geqq \max\{\tau(T_i): 1 \leqq i \leqq n\}$ and $\mathrm{OPT}(I) \geqq (1/m)\sum_{i=1}^{n} \tau(T_i)$.

Now consider the schedule generated by A and let T_k be a task finishing at time $A(I)$. When T_k was assigned its starting time, we must have had $F_j \geqq A(I)-\tau(T_k)$ for all j, $1 \leqq j \leqq m$, by the way the algorithm works. Therefore, $\sum_{i=1}^{n} \tau(T_i) \geqq \tau(T_k)+m(A(I)-\tau(T_k))$. Rearranging and dividing by m, we get

$$A(I) \leqq (1/m)\sum_{i=1}^{n} \tau(T_i)+((m-1)/m)\tau(T_k) \leqq \mathrm{OPT}(I) + ((m-1)/m)\mathrm{OPT}(I)$$

by our two lower bounds. The theorem follows.

That Theorem 1 gives the best performance guarantee possible for the list-scheduling algorithm follows from examples like those given in Figure 2, where we show both an optimal schedule and the list schedule for problem instances specified by: $\mathbf{T}=\{T_1, \cdots, T_{2m-1}\}$; $\tau(T_i)=m-1$, $1 \leqq i \leqq m-1$; $\tau(T_i)=1$, $m \leqq i \leqq 2m-2$; $\tau(T_{2m-1})=m$.

Thus we have determined the worst-case behavior of the list-scheduling algorithm. If we consider the general problem in which m is not fixed but is specified as part of the problem instance, the worst-case performance ratio is 2. For the subproblems corresponding to each fixed m (where m is part of the model rather than a variable associated with each problem instance), the worst-case performance ratio is $2-1/m$.

The question arises, can we find an algorithm that does still better? One way of trying to find a better algorithm is to examine the worst-case examples for the previous algorithm to see if there are any further common sense principles we could use to avoid them. Looking at Figure 2, we observe that the poor performance of the list-scheduling algorithm might be attributed to delaying the longest task until last. A natural way of avoiding this is to preorder the items so that they are in *decreasing order* of execution time, i.e., $\tau(T_1) \geqq \tau(T_2) \geqq \cdots \geqq \tau(T_n)$.

Thus one might propose the following algorithm, which often is called the LPT (largest processing time) algorithm [17]. First, reindex the tasks so that they are in decreasing order of execution time. Then apply the list-scheduling algorithm to the reordered task set. The LPT algorithm is again a quick and simple procedure. It can be implemented to run in time proportional to $n\log mn$.

THEOREM 2 [17]. *If A is the LPT algorithm and I is any instance of the independent task scheduling problem with m processors, then* $A(I) \leqq (4/3 - 1/3m)OPT(I)$.

We shall not prove this guarantee here. The details are in [17]. However, we do note that the proof is not as simple as that for Theorem 1. As algorithms become more sophisticated, the proofs unfortunately tend to

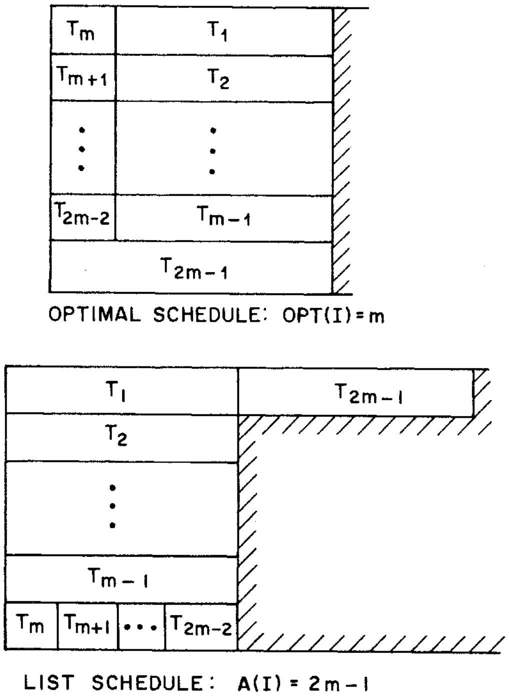


Figure 2. Problem instances yielding $A(I) = (2 - 1/m) \cdot \mathrm{OPT}(I)$ for the list-scheduling algorithm.

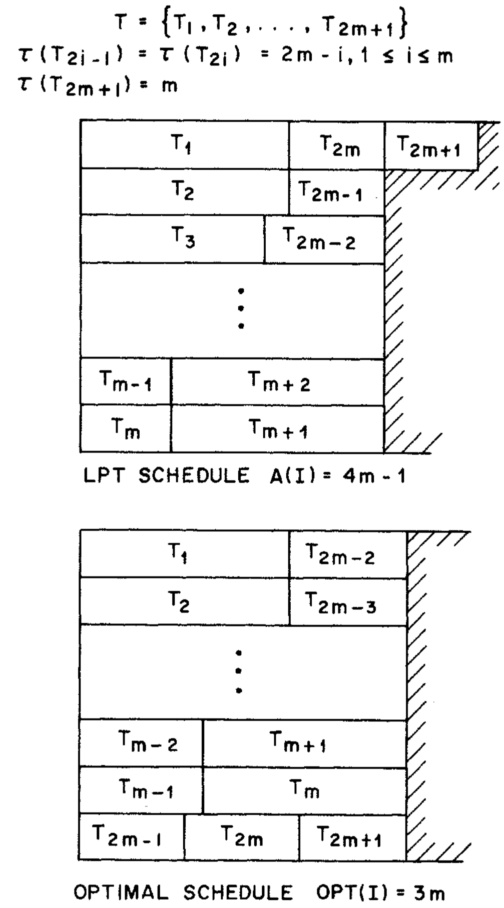
become more complex. Here one argues by contradiction, assuming a counter-example exists and deriving conflicting consequences as to what the optimum and the LPT schedules must look like.

Thus, this situation must be contrasted with that of Theorem 1, where the proof implied bounds on $A(I)$ both in terms of $\mathrm{OPT}(I)$ and in terms of the easily computed lower bound

$$\mathrm{LB}(I) = \max\{(1/m)\textstyle\sum_{i=1}^{n} \tau(T_i),\ \max\{\tau(T_i) : 1 \leqq i \leqq m\}\}.$$

If A is the LPT algorithm, the best bound on $A(I)$ in terms of $\mathrm{LB}(I)$ is given by $A(I) \leqq (2 - 2/(m+1))\mathrm{LB}(I)$, a considerably worse ratio than is needed for a bound in terms of $\mathrm{OPT}(I)$. The reason for the divergence between the two bounds is that $LB(I)$ is not a very good lower bound for

OPT(I). The problem instance given by $\mathbf{T}=\{T_1, \cdots, T_{m+1}\}$ and $\tau(T_i)=m$, $1 \leqq i \leqq m+1$, in fact yields OPT$(I)=(2-2/(m+1))$LB(I). Hence $(2-2/(m+1))$LB(I) is the best possible bound for an approximation algorithm in terms of LB(I). Thus, although it is nice to have easily com-



T = {T1, T2, . . . , T2m+1}
τ(T2i-1) = τ(T2i) = 2m - i, 1 ≤ i ≤ m
τ(T2m+1) = m
T1
T2m
T2m+1
T2
T2m-1
T3
T2m-2
Tm-1
Tm+2
Tm
Tm+1
LPT SCHEDULE A(I) = 4m - 1
T1
T2m-2
T2
T2m-3
Tm-2
Tm+1
Tm-1
Tm
T2m-1
T2m
T2m+1
OPTIMAL SCHEDULE OPT(I) = 3m

Figure 3. Problem instances yielding $A(I)=(4/3-1/3m)\cdot$OPT(I) for the LPT algorithm.

putable bounds on $A(I)$, we usually will get more information about how good an algorithm is by proving a guarantee in terms of OPT(I), rather than in terms of some straightforward lower bound on OPT(I).

Returning to the main line of our discussion, we note that the guarantee of Theorem 2 is the best possible for LPT, as follows from examples like those given in Figure 3. However, a study of these examples yields no obvious ideas about how to improve the algorithm further without attempting an entirely new approach. In the case of independent task scheduling, the best polynomial time-bounded approximation algorithm yet found—MULTIFIT [5]—does not use list scheduling at all.

The basic idea is as follows. Suppose there were some deadline D by which time all tasks must be completed. One might then try to schedule the tasks by what is called the *first fit decreasing* method. Essentially this method treats the tasks in order of decreasing execution times, assigning each task in turn to the lowest indexed processor which it will not cause to exceed the deadline. We can specify this procedure more formally as follows:

1. Index the tasks so that they are in decreasing order of execution time. Initially set $F_j=0$, $1\leqq j\leqq m$, and start with $i=1$.
2. Choose the smallest j, $1\leqq j\leqq m$, such that $F_j+\tau(T_i)\leqq D$. If no such j exists (i.e., $F_j+\tau(T_i)>D$, $1\leqq j\leqq m$), then halt. We have failed.
3. Set $p(T_i)=j$, $s(T_i)=F_j$.
4. Update $F_j=F_j+\tau(T_i)$.
5. If $i=n$, we have succeeded in constructing a schedule with $\max\{F_j\colon 1\leqq j\leqq m\}\leqq D$. Otherwise, set $i=i+1$ and go on to the next task.

For a particular value of D, this procedure will either succeed or fail. The idea of MULTIFIT is to find, using binary search techniques, a small value of D for which the procedure succeeds. For full details of the implementation, see [5]. We shall not describe them here, except to point out that the overall algorithm runs in time proportional to that for LPT and has a worst-case ratio for the general problem that is provably less than 1.22 and probably equals $20/17=1.176\cdots$ This is the best performance bound proven to date for any fast approximation algorithm for this scheduling problem.

Thus, as might be expected, our search for better and better worst-case performance ratios finally runs out of gas. At least for the time being there is an $r>1$ such that no known efficient approximation algorithm has a performance ratio better than r, even when we take "efficient" to mean "running time bounded by a polynomial function of the input size," as is often done in computer science and the theory of NP-complete problems [30].

Recall, however, that we have been dealing with the general independent task-scheduling problem, where the number m of processors is specified by the problem instance. A different situation arises if we fix m in advance as part of the model. We shall illustrate this by considering the two-processor independent task-scheduling problem, where the model specifies $m=2$.

The three algorithms mentioned so far all apply to this restricted problem and have worst-case performance ratios of $3/2$, $7/6$, and $8/7$, respectively. Now that m is fixed, however, other approaches become reasonable. Consider the sequence of algorithms $A_1, A_2, \cdots$, where A_k is defined as follows:

1. Let $\mathbf{T}' \subseteq \mathbf{T}$ contain exactly the $2k$ tasks with the longest execution times ($\mathbf{T}'=\mathbf{T}$ if $n \leqq 2k$).
2. Find an optimum schedule S' for $\mathbf{T}'$.
3. Extend S' to a feasible schedule for all of $\mathbf{T}$ using the list-scheduling algorithm applied to $\mathbf{T}-\mathbf{T}'$.

Step 1 can be performed in time $0(n\log n)$ by sorting, and step 3 can be done in time proportional to n. Thus, even if we find the optimum schedule $\mathbf{T}'$ by trying all possibilities, the time for the whole algorithm is at most proportional to $n \log n+2^k$, a polynomial in the input size since 2^k is a constant, independent of the number of tasks. Moreover, we have the following performance guarantee, proved in [17].

THEOREM 3. *If I is any instance of the two-processor independent task-scheduling problem, then $A_k(I) \leqq (1+1/2(k+1))OPT(I)$.*

Thus $\langle A_1, A_2, \cdots \rangle$ is a sequence of polynomial time algorithms whose worst-case performance ratios approach 1, the best possible ratio. Any desired level of accuracy can be guaranteed, provided we are willing to pay for it. Unfortunately, the price rises can be steep. It may not take much work to guarantee a better ratio than we could with any of our old algorithms—a ratio of $9/8$ can be guaranteed by A_3 at a cost proportional to $3n+8$. However, the exponential dependence of the running time of A_k on k means that the algorithms rapidly become impractical. For instance, to guarantee a ratio of 1.01 could take time proportional to $n \log n+2^{49}$ and, although this is bounded by a polynomial in n, the constant term 2^{49} would be sufficient to use up more than a few computer budgets.

Thus, although the first few algorithms A_k may be of practical import, the overall sequence is at best of theoretical interest. We call such a sequence a *polynomial time approximation scheme* [13], as all the algorithms in it have the same basic design and each does run in polynomial time. The difficulty with the polynomial time approximation scheme we have just described is that it is not polynomial in the amount of accuracy desired.

This should not be surprising. If exponential time appears to be required to find an optimal solution, we would expect the cost of approximate solutions to approach this limit as their guaranteed accuracy increased. What is perhaps surprising is that the limit can be approached much more slowly than we have seen so far. Recent work by Sahni [41] and Ibarra and Kim [22] has shown that we can design what might be called *fully polynomial time approximation schemes* [13]. In particular, for the two-processor scheduling problem we have been discussing, Sahni [41] presents general algorithms that, when given a problem instance I and a desired performance ratio $1+1/k$, find feasible schedules with $A(I) \leqq (1+1/k)\mathrm{OPT}(I)$ and that have running times proportional to kn^2 and k^2n, respectively. We say these are "fully polynomial" because the running time is no longer exponential in k. The time is still exponential in the number of digits of accuracy desired, but with k removed from the exponent, this type of approximation scheme may be able to guarantee very high accuracy using reasonable amounts of computer time.

The key ideas involved in these fully polynomial time approximation schemes center on the use of dynamic programming and rounding techniques. We illustrate these by presenting an $0(kn^2)$ scheme for the two-processor independent task scheduling problem.

As we shall describe the basic algorithm, it will be used only to generate the set $\mathbf{T}_1$ of tasks that are executed by the first processor P_1. Since we have only two processors in the system, a feasible schedule can be constructed easily from $\mathbf{T}_1$ merely by assigning all the remaining tasks to the other processor P_2. We generate $\mathbf{T}_1$ as follows.

Let $T = \frac{1}{2}\sum_{i=1}^{n} \tau(T_i)$, and $\delta = T/kn$. (Observe that we must have $\mathrm{OPT}(I) \geqq T$.) For each task $T_i \in \mathbf{T}$, let $v(T_i) = [\tau(T_i)/\delta]$, where $[x]$ denotes the greatest integer not exceeding x. (This is the "rounding" referred to above.)

Initially, set $U_j = \infty$, $1 \leqq j \leqq kn$, $U_0 = 0$. (This is the dynamic programming variable.) As the algorithm proceeds, $U_j < \infty$ will mean that we have found a subset $\mathbf{T}' \subseteq \mathbf{T}$ such that $\sum_{T_i \in \mathbf{T}'} v(T_i) = j$. The value of U_j will be the index of the highest indexed T_i in the first such subset found. The dynamic programming routine proceeds as follows, starting with $i=1$.

1. For each $U_j = \infty$, $v(T_i) \leqq j \leqq kn$, set $U_j = i$ whenever $U_{j-v(T_i)} < \infty$.
2. If $i = n$, halt. (The variables have received their final values.) Otherwise, set $i = i+1$ and go on to the next task.

At the conclusion of the above procedure, $U_j < \infty$ if and only if there is a subset $\mathbf{T}' \subseteq \mathbf{T}$ such that $\sum_{T_i \in \mathbf{T}'} v(T_i) = j$. Let $j^* = \max\{j : U_j < \infty\}$. Our output set $\mathbf{T}_1$ will be the first subset found such that $\sum_{T_i \in \mathbf{T}_1} v(T_i) = j^*$. The elements of $\mathbf{T}_1$ can be recovered easily using the values of U_j^* and the other U_j.

We claim that the schedule S obtained by assigning the tasks of $\mathbf{T}_1$ to P_1 and the tasks of $\mathbf{T}_2 = \mathbf{T} - \mathbf{T}_1$ to P_2 has a finishing time that is guaranteed to obey $A(I) \leqq (1+1/k)\mathrm{OPT}(I)$. First, suppose $j^* \geqq (k-1)n$. Then $[(k-1)/k]T \leqq \sum_{T_i \in \mathbf{T}'} \tau(T_i) \leqq [(k+1)/k]T$. Since $\sum_{T_i \in \mathbf{T}} \tau(T_i) = 2T$, the same inequality must hold with $\mathbf{T}_1$ replaced by $\mathbf{T}_2$. Thus the overall finishing time for S does not exceed $[(k+1)/k]T \leqq (1+1/k)\mathrm{OPT}(I)$.

On the other hand, suppose $j^* \leqq (k-1)n$. Let $\mathbf{T}_1{}^*$ be the set of tasks executed in an optimal schedule by the processor that becomes idle first (P_1 if both processors become idle simultaneously). Then $\mathrm{OPT}(I)=2T-\sum_{T_i\in \mathbf{T}_1^*}\tau(T_i)$. However, we also must have

$$T\geqq \sum_{T_i\in \mathbf{T}_1}\tau(T_i)\geqq \delta\sum_{T_i\in\mathbf{T}_1}v(T_i)=\delta j^*\geqq \delta\sum_{T_i\in\mathbf{T}_1^*}v(T_i)$$
$$\geqq \sum_{T_i\in\mathbf{T}_1^*}(\tau(T_i)-\delta)\geqq \sum_{T_i\in\mathbf{T}_1^*}\tau(T_i)-n\delta$$

and hence $\sum_{T_i\in\mathbf{T}_2}\tau(T_i)\leqq \mathrm{OPT}(I)+n\delta=\mathrm{OPT}(I)+T/k\leqq(1+1/k)\cdot\mathrm{OPT}(I)$. Thus in either case $A(I)\leqq(1+1/k)\mathrm{OPT}(I)$, as claimed.

A careful examination of the algorithm for finding $\mathbf{T}_1$ and the corresponding schedule shows that the running time is indeed proportional to kn^2, as claimed. Thus, to guarantee a ratio of 1.01, this approximation scheme requires time proportional to $100n^2$, as opposed to $n\log n+2^{49}$ for the previous scheme. For moderate values of n, this new scheme may well be practical, although it must be pointed out that for large values of k and n even fully polynomial time approximation schemes become of more theoretical than practical interest. (In addition, they also have the drawback of requiring more computer storage—the number of memory locations required here is proportional to kn.)

Before concluding this section, we remark that our performance-guarantee formulation allows for the possibility of something even better than an approximation scheme. Even though optimum schedules might be unobtainable, one could conceive of a fast algorithm that guaranteed $A(I)\leqq\mathrm{OPT}(I)+d$, for some $d>0$. Results of this form do in fact exist for some problems [32, 42]. However, they are not likely for problems like the independent task scheduling problem we have been discussing, as here the difference between an optimal schedule and a suboptimal one can be made arbitrarily large by scaling. That is, given a problem instance I, we can obtain a new instance "$k\cdot I$" merely by multiplying all execution times by k. There will be an exact correspondence between feasible schedules for I and $k\cdot I$, and we would expect $A(k\cdot I)=k\cdot A(I)$ and $\mathrm{OPT}(k\cdot I)=k\cdot\mathrm{OPT}(I)$. Thus $A(k\cdot I)-\mathrm{OPT}(k\cdot I)=k[A(I)-\mathrm{OPT}(I)]$, and this can be made arbitrarily large for any algorithm A that is not an optimization algorithm merely by taking a sufficiently large value of k.

2. OTHER RESULTS AND MODELS

The model discussed in the previous section is very simple and basic. However, similar results continue to hold as we add to it more complexity to reflect real-life situations. In this section we briefly survey some of the variations on the basic problem that have been studied from the performance-guarantee viewpoint. (Earlier surveys, less current but more detailed, can be found in [18, 19, 37].)

Resource Constraints

In the real world tasks often may require more resources than just a processor. In our example of a typing pool, tasks may require not only typists but also typewriters, copying machines, table space, etc. Thus we might extend our model so that each task has, in addition to an execution time $\tau(T_i)$, a number of *resource requirements* $R_1(T_i), \cdots, R_s(T_i)$ for the resources $R_1, \cdots, R_s$, with B_j denoting the total amount of R_j available at any one time. In any feasible schedule, if S_t is the set of tasks being executed at time t, we must have $\sum_{T\in S_t} R_j(T) \leqq B_j$ for all $t \geqq 0$ and all j, $1 \leqq j \leqq s$.

The extension of the basic model to include resources is treated in [9]. As might be expected, the worst-case performance ratios for simple algorithms increase as the number s of resources increases. Interestingly enough, however, the natural generalization of the list-scheduling algorithm still guarantees a ratio of 2 if there is only one resource and $m \geqq n$ (that is, there are enough processors so that all tasks could be executed simultaneously if it were not for the resource constraints).

Much more work has been devoted to the special case where all tasks have the same execution time. When $m \geqq n$ and $s=1$, this problem is equivalent to the well-studied "bin packing" problem of [24–26], and the best performance ratio guaranteed to date by a fast algorithm is $^{11}\!/_{9}$. (The algorithm is essentially the "first fit decreasing" procedure mentioned in Section 1.) The case where $s=1$ and $m \leqq n$ and hence where there are processors as well as resource constraints is studied in [33, 34]. This case for $s \geqq 1$ is examined in [10].

Precedence Constraints

The assumption that tasks can be executed in an arbitrary order is not always justified. Sometimes it may be the case that one task must be completed before another one can begin, for example, if some output of the first task is needed for execution of the second. We incorporate this type of constraint into the model by allowing a partial order "$<$" to be defined on the set of tasks. We use "$T_i < T_j$" to mean that T_i must be completed before T_j can begin; that is, in any feasible schedule we must have $s(T_j) \geqq s(T_i) + \tau(T_i)$.

Although there has been much research into scheduling models involving precedence constraints, little has been done from the performance-guarantees approach. It is interesting to note that, despite the increased complexity when precedence constraints are added, a natural generalization of the list-scheduling algorithm still generates exactly the same performance ratio, $A(I) \leqq (2-1/m)\mathrm{OPT}(I)$, when there are m processors. However, when precedence constraints are added to the resource-constrained version of the problem, algorithms seem to have significantly poorer performance ratios [9, 10].

Nonidentical Processors

So far we have been assuming that all our processors were identical. However, it is quite reasonable that one might encounter different speed computers, or different speed typists, or that task times might vary in a more complicated fashion as a function of processor. The basic model, extended to include different speed processors, is studied from a performance-guarantee viewpoint in [14, 20, 38]. The case where task times may vary arbitrarily from processor to processor is treated in [1, 23]. In another variation one might order the processors by "capability," with each task only executable by processors that are "big" enough for it. This extension is treated in [28, 29].

Restricted Models

Even when a real-word problem matches one of our models, it may be that the problem does not require the full generality of the model. There may be additional constraints on the kinds of problem instances that arise. Perhaps there are only two processors. Perhaps all the execution times are equal or belong to a limited set of possible values. Perhaps no resource requirement is ever larger than half the available amount. Perhaps the precedence constraints are never more complex than a tree. We have already seen examples of how such restrictions, if properly taken into account, may allow us to prove stronger guarantees, and many of the papers already cited include such special case results.

Other Models

The application of the performance-guarantee approach has not been restricted to our basic model and its variants. Stone and Fuller [42] consider a problem of scheduling access to a rotating computer memory device and actually prove a bound of the form $A(I) \leqq \mathrm{OPT}(I)+1$ for a very simple heuristic. (A similar "difference" result has been obtained in [32] for a very restricted version of our basic model.) One of the popular scheduling models that has not yet received much attention from the performance-guarantee point of view is that of job-shop and flow-shop scheduling. The only paper to date on this model is [15], which shows that some first attempts at possible approximation algorithms do not have very satisfactory worst-case performance ratios.

Other Objective Functions

There are, of course, other aspects of a feasible schedule one might want to optimize besides finishing time, although this is the measure that to date has received the most attention. One might be interested in minimizing the total finishing time $[\sum_{i=1}^{n}(s(T_i)+\tau(T_i))]$, which reflects the average length of time until a task is completed. Approximation algorithms for a problem involving this objective function are treated briefly in [1[, which also considers how algorithms designed to minimize finishing time perform under this new objective function.

An even more intriguing example of the behavior of one algorithm under two different objective functions occurs in [3]. Here the model is the same as our basic model, but the objective function is $\sum_{j=1}^{m} F_j^2$, where F_j is the finishing time of the last task on the jth processor. The list-scheduling algorithm, which had a worst-case performance ratio of $4/3$ for the objective function of overall finishing time, has a worst-case performance ratio for this alternative objective function lying between 1.03 and 1.04.

Other possible objective functions might measure the total weighted finishing time, the number of tardy tasks or the total weighted tardiness (in models where tasks have deadlines), or the number of processors required. Except for this latter objective function, which provides another way of embedding the bin-packing problem into scheduling theory, none of these measures has yet received much attention from the performance-guarantee point of view.

There also are a number of possible objective functions for which the optimization problem is a maximization problem rather than a minimization problem like all the ones we have treated so far. The knapsack problem may be viewed as a scheduling-maximization problem in which tasks have values as well as execution times, and the goal is to schedule on a single processor the highest valued collection of tasks possible such that all finish before a given deadline. Approximation algorithms and polynomial time approximation schemes have been presented for this and related problems in [2, 22, 40].

(For a maximization problem, a performance guarantee can be put in the format $\mathrm{OPT}(I) \leqq r \cdot A(I) + d$. $A(I)$ and $\mathrm{OPT}(I)$ have been interchanged so that the range of possible values for r remains $1 \leqq r \leqq \infty$. Thus worst-case performance ratios for both minimization and maximization problems can be viewed in the same framework.)

Further, one can have very problem-specific objective functions and still be able to prove performance guarantees. Nemhauser and Yu [39] consider a problem of train scheduling and construct an objective function representing expected profits. They then present what is in effect a fully polynomial time approximation scheme for the corresponding maximization problem.

Finally, we should mention that for some objective functions, a performance bound in terms of the "worst" possible solution may be of interest (see, for instance, [7]).

Other Combinatorial Problems

Before concluding this brief survey, we should point out that approximation algorithms and the performance-guarantee approach are not restricted to scheduling problems. Results have also been obtained in such diverse areas as graph coloring, set covering, the traveling-salesman problem, and various other placement, packing, and routing problems. See [12] for an annotated bibliography.

3. CONCLUDING REMARKS

We have attempted to introduce the reader to a relatively new approach to the analysis of heuristic scheduling algorithms. We conclude with some informal advice to the prospective user of this performance-guarantee approach. The analysis of the worst-case behavior of an algorithm is the real heart of the approach. Our treatment of the basic scheduling problem in Section 1 serves as one model of how to proceed.

The best technique seems to be to converge on the worst-case performance ratio from both sides. Prove the best guarantee you can, find examples on which the algorithm behaves poorly, and then try to narrow the gap by working alternatively on the upper and lower bounds. It is, of course, not necessary to determine the worst-case performance ratio exactly; the upper-bound guarantee is in the nature of things more important than the lower bound examples. However, information gained from failing to find bad examples may be of use in proving a better guarantee (and vice versa). Thus we recommend the use of this alternating technique for as much as one can get from it.

As to how one actually generates examples and proves bounds, both processes appear to be very problem-specific. An examination of the literature to see the methods used for other (related) problems can provide a valuable background and insight into some general techniques that may be useful. However, the predominant impression left by such a survey is that each specific problem seems to require special techniques. In addition, one cannot help noticing the great length and complexity required in the proofs for some very simple algorithms. Fortunately, difficult proofs are not the rule and indeed often are not needed unless one wants to determine the exact value of the worst-case performance ratio (which may be more of theoretical than practical significance). Many bounds *can* be proved quite simply, as we saw in Section 1. The numerous papers cited that prove performance guarantees are testimony to the viability of this approach.

REFERENCES

1. J. Bruno, E. G. Coffman, Jr., and R. Sethi, "Scheduling Independent Tasks to Reduce Mean Finishing Time," *Comm. ACM* **17**, 382–387 (1974).
2. A. K. Chandra, D. S. Hirschberg, and C. K. Wong, "Approximate Algo-

rithms for the Knapsack Problem and Its Generalizations," Report RC 5616, IBM Research Center, Yorktown Heights, N.Y., 1975.

3. A. K. Chandra and C. K. Wong, "Worst-case Analysis of a Placement Algorithm Related to Storage Allocation," *SIAM J. Comput.* **4,** 249–263 (1975).
4. E. G. Coffman, Jr. (Ed.), *Computer and Job/Shop Scheduling Theory,* John Wiley & Sons, New York, 1976.
5. E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson, "An Application of Bin Packing to Multiprocessor Scheduling," (to appear).
6. E. G. Coffman, Jr., and R. L. Graham, "Optimal Scheduling for Two Processor Systems," *Acta Informatica* **1,** 200–213 (1972).
7. G. Cornuejols, M. L. Fisher, and G. L. Nemhauser, "An analysis of heuristics and relaxations for the uncapacitated location problem," *Management Science,* in press.
8. G. B. Dantzig, "Discrete Variable Extremum Problems," *Opns. Res.* **5,** 266–277 (1957).
9. M. R. Garey and R. L. Graham, "Bounds for Multiprocessor Scheduling with Resource Constraints," *SIAM J. Comput.* **4,** 187–200 (1975).
10. M. R. Garey, R. L. Graham, D. S. Johnson, and A. C.-C. Yao, "Resource Constrained Scheduling as Generalized Bin Packing," *J. Combinational Theory (Series A),* **21,** 257–298 (1976).
11. M. R. Garey and D. S. Johnson, "Scheduling Tasks with Nonuniform Deadlines on Two Processors," *J. Assoc. Comput. Machinery* **23,** 461–467 (1976).
12. M. R. Garey and D. S. Johnson, "Approximation Algorithms for Combinatorial Problems: An Annotated Bibliography," in *Algorithms and Complexity: New Directions and Recent Results,* pp. 41–52, J. F. Traub (Ed.), Academic Press, New York, 1976.
13. M. R. Garey and D. S. Johnson, " 'Strong' NP-Completeness Results: Motivation, Examples, and Implications" (to appear).
14. T. Gonzalez, O. H. Ibarra, and S. Sahni, "Bounds for LPT Schedules on Uniform Processors," Computer Science Technical Report 75-1, University of Minnesota, Minneapolis, 1975.
15. T. Gonzalez and S. Sahni, "Flow Shop and Job Shop Schedules," Computer Science Technical Report 75-14, University of Minnesota, Minneapolis, 1975.
16. R. L. Graham, "Bounds for Certain Multiprocessing Anomalies," *Bell Sys. Tech. J.* **45,** 1563–1581 (1966).
17. R. L. Graham, "Bounds on Multiprocessing Timing Anomalies," *SIAM J. Appl. Math.* **17,** 416–429 (1969).
18. R. L. Graham, "Bounds on Multiprocessing Anomalies and Related Problems," in *AFIPS Conference Proceedings Vol. 40,* pp. 205–217, AFIPS Press, Montvale, N. J., 1972.
19. R. L. Graham, "Bounds on the Performance of Scheduling Algorithms," in *Computer and Job/Shop Scheduling Theory,* pp. 165–228, E. G. Coffman, Jr. (Ed.), John Wiley & Sons, New York, 1976.
20. E. Horowitz and S. Sahni, "Exact and Approximate Algorithms for Scheduling Nonidentical Processors," *J. Assoc. Comput. Machinery* **23,** 317–327 (1976).

21. T. C. Hu, "Parallel Sequencing and Assembly Line Problems," *Opns. Res.* **9,** 841–848 (1961).
22. O. H. Ibarra and C. E. Kim, "Fast Approximation Algorithms for the Knapsack and Sum of Subset Problems," *J. Assoc. Comput. Machinery* **22,** 463–468 (1975).
23. O. H. Ibarra and C. E. Kim, "Heuristic Algorithms for Scheduling Independent Tasks on Nonidentical Processors," Computer Science Technical Report 75-7, University of Minnesota, Minneapolis, 1975.
24. D. S. Johnson, "Near-Optimal Bin Packing Algorithms," doctoral thesis, Massachusetts Institute of Technology, 1973.
25. D. S. Johnson, "Fast Algorithms for Bin Packing," *J. Comput. Syst. Sci.* **8,** 272–314 (1974).
26. D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham, "Worst-Case Performance Bounds for Simple One-Dimensional Packing Algorithms," *SIAM J. Comput.* **3,** 299–325 (1974).
27. S. M. Johnson, "Optimal Two-and-Three-Stage Production Schedules with Setup Times Included," *Naval Res. Log. Quart.* **1,** 61–68 (1954).
28. D. G. Kafura, "Analysis of Scheduling Algorithms for a Model of a Multiprocessing Computer System," doctoral thesis, Purdue University, 1974.
29. D. G. Kafura and V. Y. Shen, "Scheduling Independent Processors with Different Storage Capacities," *Proceedings of the 1974 ACM National Conference*, pp. 161–166, Association for Computing Machinery, New York, 1974.
30. R. M. Karp, "On the Computational Complexity of Combinatorial Problems," *Networks* **5,** 45–68 (1975).
31. R. M. Karp, "The Fast Approximate Solution of Hard Combinatorial Problems," *Proceedings of the 6th Southeastern Conference on Combinatorics, Graph Theory, and Computing*, pp. 15–34, Utilitas Mathematica Publishing Inc., Winnipeg, 1975.
32. M. T. Kaufman, "An Almost-Optimal Algorithm for the Assembly Line Scheduling Problem," *IEEE Trans. Comput.* **C-23,** 1169–1174 (1974).
33. K. L. Krause, "Analysis of Computer Scheduling with Memory Constraints," doctoral thesis, Purdue University, 1973.
34. K. L. Krause, V. Y. Shen, and H. D. Schwetman, "Analysis of Several Task-Scheduling Algorithms for a Model of Multiprogramming Computer Systems," *J. Assoc. Comput. Machinery* **22,** 522–550 (1975).
35. J. K. Lenstra and A. H. G. Rinnooy Kan, "Complexity of Scheduling under Precedence Constraints," *Opns. Res.* **26,** 22–35 (1978).
36. J. K. Lenstra, A. H. G. Rinnooy Kan, and P. Brucker, "Complexity of Machine Scheduling Problems," Report BW 43/75, Mathematisch Centrum, Amsterdam (1975), (to appear in Annals of Discrete Math.)
37. C. L. Liu, "Approximation Algorithms for Discrete Optimization Problems," in *Proceedings of the 4th Texas Conference on Computing Systems*, pp. 4B1.1–4B1.5, IEEE Computer Society, Long Beach, Calif. 1975.
38. J. W. S. Liu and C. L. Liu, "Bounds on Scheduling Algorithms for Heterogeneous Computing Systems," in *Proceedings of the 1974 IFIP Congress*, pp. 349–353, North-Holland, Amsterdam, 1974.

39. G. L. NEMHAUSER AND P. L. YU, "A Problem in Bulk Service Scheduling," *Opns. Res.* **20,** 813–819 (1972).
40. S. SAHNI, "Approximate Algorithms for the 0–1 Knapsack Problem," *J. Assoc. Comput. Machinery* **22,** 115–124 (1975).
41. S. SAHNI, "Algorithms for Scheduling Independent Tasks," *J. Assoc. Comput. Machinery* **23,** 116–127 (1976).
42. H. S. STONE AND S. H. FULLER, "On the Near-Optimality of the Shortest-Latency-First Drum Scheduling Discipline," *Comm. ACM* **16,** 352–353 (1973).