MTA Számitástechnikai és Automatizálási Kutató Intézete, Közlemények 18/1977.

DETERMINISTIC SEQUENCING: COMPLEXITY AND OPTIMAL ALGORITHMS

E. G. Coffman, Jr.*
The Pennsylvania State University

I. Introduction

In the past several years interest and new results in the theory of deterministic scheduling have mounted at an increasing rate. This paper is an attempt to represent the current position of the field in terms of research into schedule length and mean flow time minimization problems. We describe theoretical results for sequencing problems arising mainly in computer and job-shop environments. However, the models are simple in structure and are consequently meaningful in a very large variety of applications.

Briefly, the general model studied assumes a set of tasks or jobs and a set of resources to be used in their execution or servicing. In all cases the models are deterministic in the sense that the information describing tasks is assumed known in advance. This information includes task execution times, operational precedence constraints, deferral costs, and resource requirements. The sequencing problems examined include not only the minimization of schedule-lengths and mean time-in-system (weighted by deferral costs), but also a number of closely related problems such as scheduling to meet due-dates or deadlines. The results presented include efficient optimal algorithms and mathematical descriptions of the complexity of sequencing problems.

*This research was partially supported by Institut de Recherche d'Informatique et d'Automatique, France, and by NSF grant GJ-28290.

Computers arise in the subject matter in at least three ways. Firstly, they represent an almost universal job-shop for our purposes. The appearance of virtually all of the problems we analyze can be observed or envisioned in the design or operation of general-purpose computer systems, although the prime importance of specific problems may exist in other applications. Secondly, computers must be considered in the implementation of the enumerative and iterative approaches to sequencing problems. Finally, the field of Computer Science is the origin of the complexity theory which we apply to problems of sequence.

We emphasize that our interest is almost wholly mathematical, with very little recourse to discussions of pragmatics. The applicability (and, of course, inapplicability) of the results will be quite evident in virtually all cases, owing primarily to the simplicity of the models. The reader is referred to [CMM] for insights into the general problems in practice and the many features of such problems that extend the models examined here but for which comparable results are not known (see also [Ba]).

This paper presents virtually all the theoretical results in [Ba] and [CMM] that concern our problems of deterministic scheduling theory. They form the background for the new results. Additional background material concerned with computer sequencing problems can be found in a more recent text [CD] on operating-systems theory. For recent survey papers dealing with many of the subjects of this paper, the reader is referred to [G3], [C1], [B1], and [BLR].

II. A General Model

The scheduling model, from which subsequent problems are drawn, is described by considering in sequence the resources, task systems, sequencing constraints, and performance measures.

Resources. In the majority of the models studied, the resources consist simply of a set $P = \{P_1,\ldots,P_m\}$ of processors. Depending on the specific problem, they are either identical, identical in functional capability but different in speed, or different in both function and speed.

In the most general model there is also a set of additional resource types $\mathcal{R} = \{R_1,\ldots,R_s\}$, some (possibly empty) subset of which is required during the entire execution of a task on some processor. The total amount of resource of type R_j is given by the positive integer m_j. In the computer application, for example, such resources may represent primary or secondary storage, input/output devices, or subroutine libraries. Although it is possible to include the processors in $\mathcal{R}$, it is more convenient to treat them separately because

1) they will constitute a resource type necessarily in common with all tasks (although two different tasks need not require the same processors), and

2) they are discretized with the restriction that a task can execute on at most one processor at a time.

Task Systems. A general task system for a given set of resources can be defined as the system $(\mathcal{T},\prec,[\tau_{ij}],\{\mathcal{R}_j\},\{w_j\})$ as follows:

1. $\mathcal{T} = \{T_1,\ldots,T_n\}$ is a set of tasks to be executed.
2. $\prec$ is an (irreflexive) partial order defined on $\mathcal{T}$ which specifies operational precedence constraints. That is, $T_i \prec T_j$ signifies that T_i must be completed before T_j can begin.
3. $[\tau_{ij}]$ is an $m \times n$ matrix of execution times, where $\tau_{ij} > 0$ is the

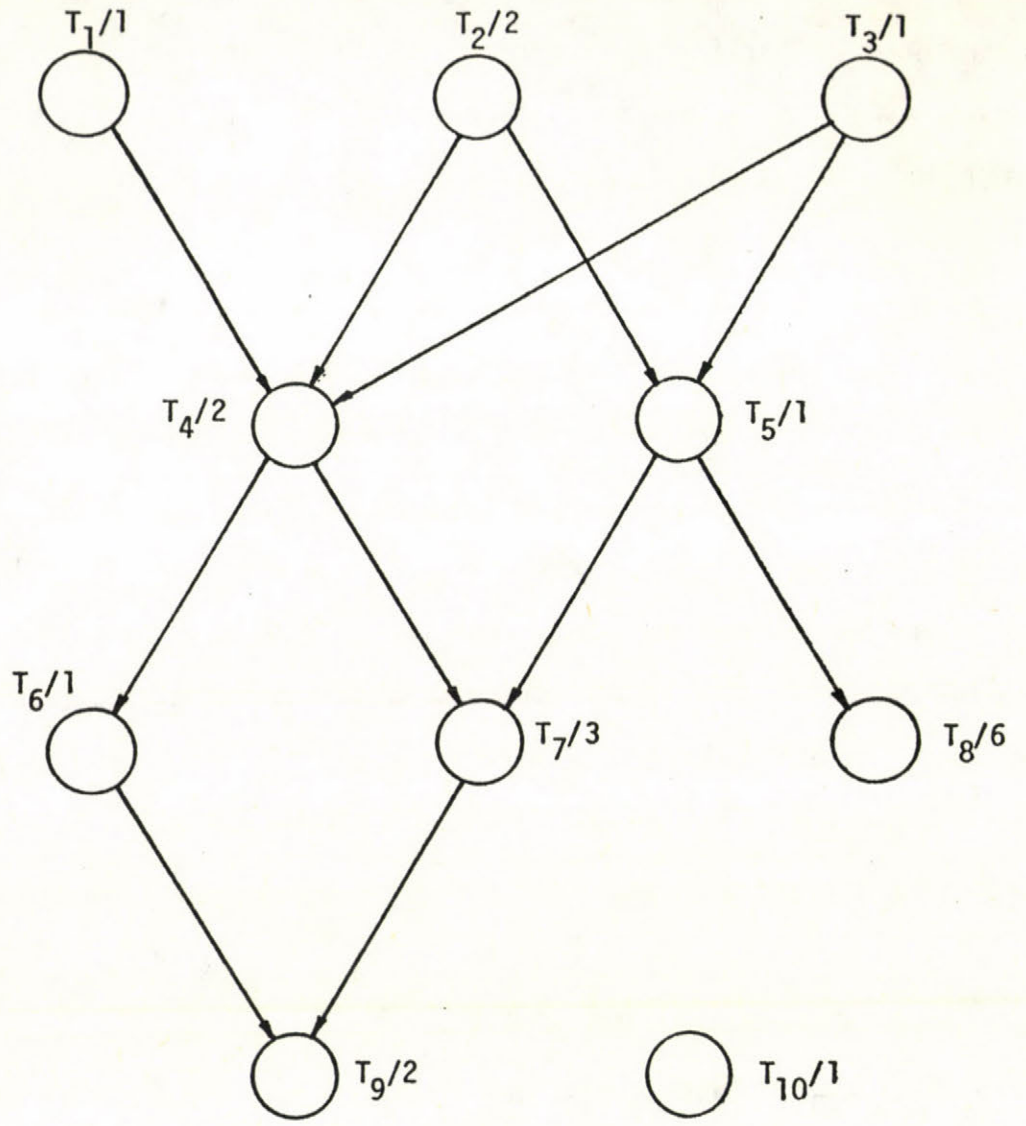
time required to execute T_j, $1 \le j \le n$, on processor P_i, $1 \le i \le m$. We suppose that $\tau_{ij} = \infty$ signifies that T_j cannot be executed on P_i and that for each j there exists at least one i such that $\tau_{ij} < \infty$. When all processors are identical we let τ_j denote the execution time of T_j common to each processor.

4. $\mathcal{R}_j = [R_1(T_j),\dots,R_s(T_j)]$, $1 \le j \le n$, specifies in the ith component, the amount of resource type R_i required throughout the execution of T_j. We always assume $R_i(T_j) \le m_i$ for all i and j.

5. The weights w_i, $1 \le i \le n$, are interpreted as deferral costs (or more exactly cost rates), which in general may be arbitrary functions of schedule properties influencing T_i. However, the w_i are taken as constants in the models we consider. Thus the "cost" of finishing T_i at time t is simply $w_i t$.

This formulation contains far more generality than we intend to embrace, but each problem studied can be represented as a special case of the model. One particular restriction worth noting is the limitation on operational precedence. We cannot, for example, represent loops in computer programs modeled as task systems. Note that the partial order $\prec$ is conveniently represented as a directed, acyclic graph (or dag) with no (redundant) transitive arcs. Unless stated otherwise, we assume $\prec$ is given as a list of arcs in such a graph. In general, however, the way in which a partial order is specified in a given problem may influence the complexity of its solution. (We return to this point later.)

In Figure 1, an example system is shown, where the notation T_i/τ_i is introduced for labeling vertices. As one might expect, heavy use is made of graphical methods for defining task systems, rather than defining them as appropriate five-tuples.



Task/execution time, identical processors

Figure 1. A dag representation of $(\mathcal{T}, \prec, \{\tau_i\})$.

Notation and properties:

1. Acyclic.
2. No transitive edges: (T_1,T_6) would be such an edge.
3. T_1,T_2,T_3,T_{10} are initial vertices; T_8,T_9,T_{10} are terminal vertices.
4. For example, T_7 is a successor of T_1,T_2,T_3,T_4,T_5 but an immediate successor of only T_4,T_5; T_5 is a predecessor of T_7,T_8,T_9 but an immediate predecessor of only T_7,T_8.
5. Levels:

8	9	8	7	7	3	5	6	2	1
T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}

6. Critical paths: T_2,T_5,T_8 and T_2,T_4,T_7,T_9.

In the following we use a number of more or less common terms concerning dags. In particular, a path of length k from T to T' in a given graph G is a sequence of vertices (tasks) $T_{i_1},\ldots,T_{i_k}$ such that $T = T_{i_1}, T' = T_{i_k}$ $(k \geq 1)$ and $(T_{i_j}, T_{i_{j+1}})$ is an arc in G for all $1 \leq j \leq k-1$. Moreover, if such a path exists, T will be called a predecessor of T' and T' a successor of T. If $k = 2$ the terms immediate predecessor and immediate successor will be used. Initial vertices are those with no predecessors, and terminal vertices are those with no successors. The graph forms a forest if either each vertex has at most one predecessor, or each vertex has at most one successor. If a forest has in the first case exactly one vertex with no predecessors, or in the second case, exactly one vertex with no successors, it is also called a tree. In either case, the terms root and leaf have the usual meaning. The level of a vertex T is the sum of the execution times associated with the vertices in a path from T to a terminal vertex such that this sum is maximal. Such a path is called a critical path if the vertex T is at the highest level in the graph.

Sequencing Constraints. By "constraint" we mean here a restriction of scheduling algorithms to specific (though broad) classes. Two main restrictions are considered.

1. Nonpreemptive scheduling: with this restriction a task cannot be interrupted once it has begun execution; that is, it must be allowed to run to completion. In general, preemptive scheduling permits a task to be interrupted and removed from the processor under the assumption that it will eventually receive all its required execution time, and there is no loss of execution time due to preemptions (i.e., preempted tasks resume execution from the point at which they were last preempted).

2. List scheduling [G1]: in this type of scheduling an ordered list of the tasks in $\mathcal{T}$ is assumed or constructed beforehand. This list is often called the priority list. The sequence by which tasks are assigned to processors is then decided by a repeated scan of the list. Specifically, when a processor becomes free for assignment, the list is scanned until the first unexecuted task T is found which is ready to be executed; that is, the task can be executed on the given processor, all predecessors of T have been completed, and sufficient resources exist to satisfy $R_i(T)$ for each $1 \le i \le s$. This task is then assigned to execute on the available processor. We assume the scan takes place instantaneously, and if more than one processor is ready for assignment at the same time, they are assigned available tasks in the order P_1 before P_2 before P_3, etc.

Before discussing performance measures, let us illustrate the means by which schedules are usually represented graphically, assuming $s = 0$. We use the type of timing diagram illustrated in Fig. 2 for the task system shown in Fig. 1. In the obvious way the number of processors determines the number of horizontal lines which denote time axes. The hatching shown in the figure represents periods during which processors are idle. The symbols $s_i(S)$ and $f_i(S)$ will denote, respectively, the start and finishing times of T_i.

For problems assuming additional resources, we will have occasion to draw timing diagrams only for $s = 1$. In this case the vertical axis denotes the amount of additional resource required, the number of vertical segments being bounded by the given number of processors. An example is given shortly (Fig. 4b).

The timing diagrams of Fig. 2 give an informal and intuitive notion of schedule. Somewhat more formally, a schedule can be defined as a suitable mapping that in general assigns a sequence of one or more disjoint execution intervals in $[0,\infty)$ to each task such that

1. Exactly one processor is assigned to each interval.
2. The sum of the intervals is precisely the execution time of the task, taking into account, if necessary, different processing rates on different processors.
3. No two execution intervals of different tasks assigned to the same processor overlap.
4. Precedence and additional-resource usage constraints are observed.
5. There is no interval in $[0,\max\{f_i\}]$ during which no processor is assigned to some task (i.e., a schedule is never allowed to have all processors idle when uncompleted tasks exist).

For nonpreemptive schedules there is exactly one execution interval for each task, and for list schedules we further require that no processor can be idle if there is a task ready and able to execute on it. We do not attempt to further formalize the notion of schedules--a wholly mathematically definition is unnecessary and very elaborate for the general model.

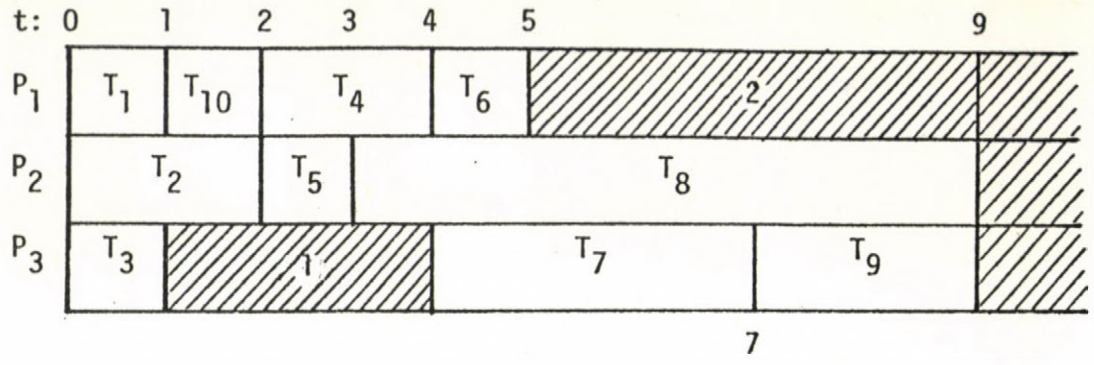


Figure 2. Example timing diagram for Fig. 1. Schedule with $m = 3$.

Performance Measures. Two principal measures of schedule performance that we consider are the schedule-length or maximum finishing (or flow) time

$$\omega(S) = \max_{1 \le i \le n}\{f_i(S)\} \tag{1}$$

and the mean weighted finishing (or flow) time

$$\bar{\omega}(S) = \frac{1}{n} \sum_{i=1}^{n} w_i f_i(S) \; . \tag{2}$$

The basic problems, therefore, are to find efficient algorithms for the minimization of these quantities over all schedules S, drawn perhaps from a specific class of schedules as defined earlier.

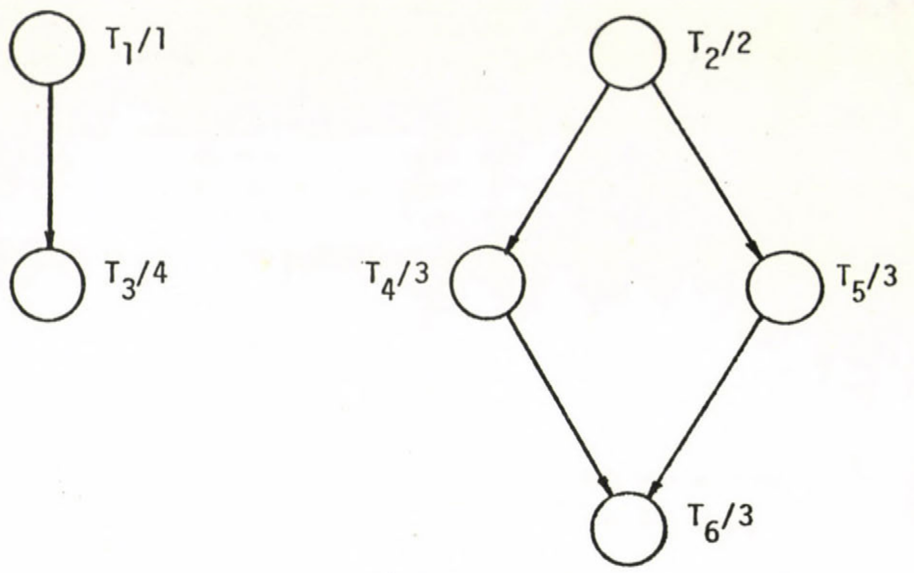
At this point it is convenient to illustrate that preemptive, nonpreemptive, and list scheduling disciplines can be distinct in terms of minimum schedule length and mean weighted flow time and that the "power" of these disciplines decreases in the order given. A graph of a task system, a minimum-length preemptive (i.e., unrestricted) schedule, a minimum-length nonpreemptive schedule, and a minimum-length list schedule appear in

Fig. 3a, b, c, and d, respectively. (Verifying the optimality of the schedules is not difficult). Note that for $w_i = 1$ $(1 \le i \le n)$ the mean weighted flow-time performance is optimal in each case and concurs with the ordering in terms of minimum schedule lengths just given.

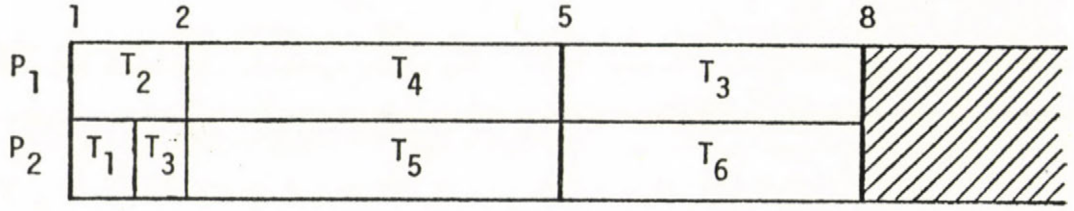
Even for $\prec$ empty, preemptive schedules can be shorter than non-preemptive schedules (e.g., consider $m = 2$ and three unit length tasks). However, for nonpreemptive scheduling on identical processors, a minimum-length list schedule is a minimum-length nonpreemptive schedule when $\prec$ is empty. For mean weighted flow-time (non-negative weights) on identical processors, the optimal list schedule is an optimal preemptive schedule when $\prec$ is empty, thus removing any distinction between the disciplines in these circumstances.

A good many of the results, especially those related to problem complexity, can be readily extended to a number of other performance measures of interest in job-shop or computer sequencing. For example, suppose we have identical processors and we define $W_i(S) = f_i(S) - \tau_i$ as the waiting time of T_i in S. Then it is easily seen that a schedule minimizing $\bar{\omega}$ also minimizes the mean, weighted waiting time.

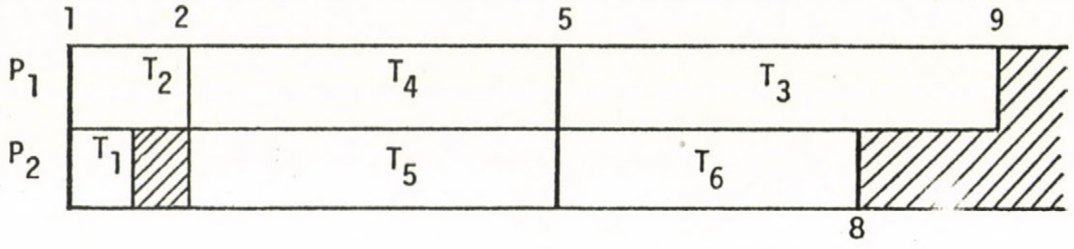
Now suppose the general model is extended to permit a positive number $d_i (1 \le i \le n)$ called the due date to be given for each task T_i. The due date expresses the time at which it is desired to have a task finished. Then the lateness of T_i in S is defined as $f_i(S) - d_i$, and the tardiness is defined as $\max\{0, f_i(S) - d_i\}$. The maxima and weighted means of lateness and tardiness are also interesting performance measures. As with waiting times, it is easily seen that sequences mini-



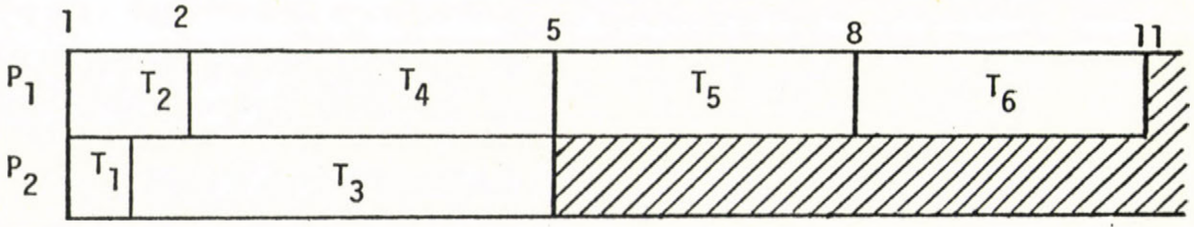
(a)



(b)



(c)



(d)

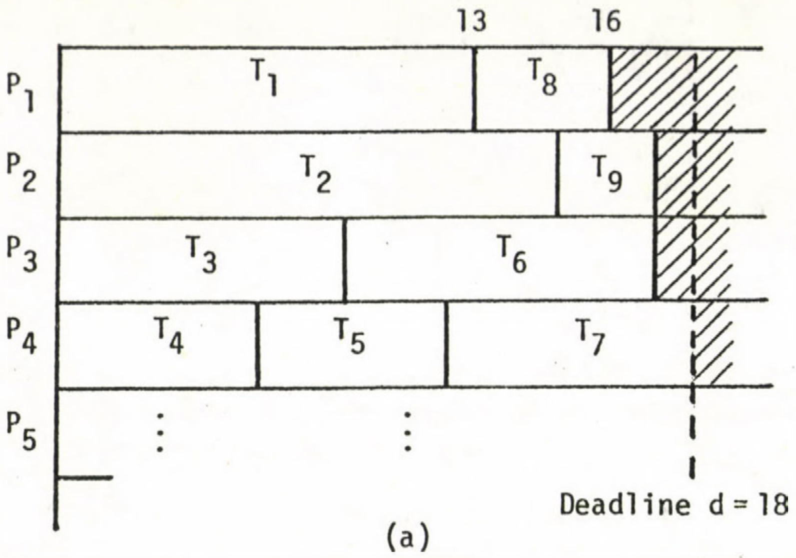
Figure 3. Discipline hierarchy. (a) $m=2$ identical processors, $s=0$, $w_i=1\,(1\le i\le n=6)$. (b) Optimal preemptive schedule, $\omega=8$, $\bar{\omega}=29/6$. (c) Optimal nonpreemptive schedule, $\omega=9$, $\bar{\omega}=30/6$. (d) Optimal list schedule, $\omega=11$, $\bar{\omega}=32/6$.

mizing mean weighted flow time also minimize mean weighted lateness; however, this is not true for the mean weighted tardiness.

When we consider problems in which due dates <u>must</u> be respected (i.e., $f_i(S) \le d_i$, $1 \le i \le n$), the due dates are also called <u>deadlines</u>. One such problem to which we devote considerable attention assumes $d_i = d$, $1 \le i \le n$, $\prec$ is empty, $s = 0$, identical processors, and seeks to minimize the number of processors required to meet the common deadline d. This problem is referred to as the bin-packing problem. Interestingly, we find that this problem is equivalent to the schedule-length minimization problem for identical processors, $\prec$ empty, $\tau_i = 1$ $(1 \le i \le n)$, $m \ge n$, and $s = 1$. Figures 4a and b illustrate this equivalence. Note that if we remove the restriction $m \ge n$ in the parameter list, we have an equivalence to the problem of scheduling to meet a common deadline with the constraint of at most m tasks per processor and the objective of minimizing the number of processors. Although there was no need to introduce deadlines as another component in the general model, the augmented system with an arbitrary set $\{d_i\}$ is discussed briefly in the next section on single-machine results.

III. <u>Background in Single-Processor Results</u>

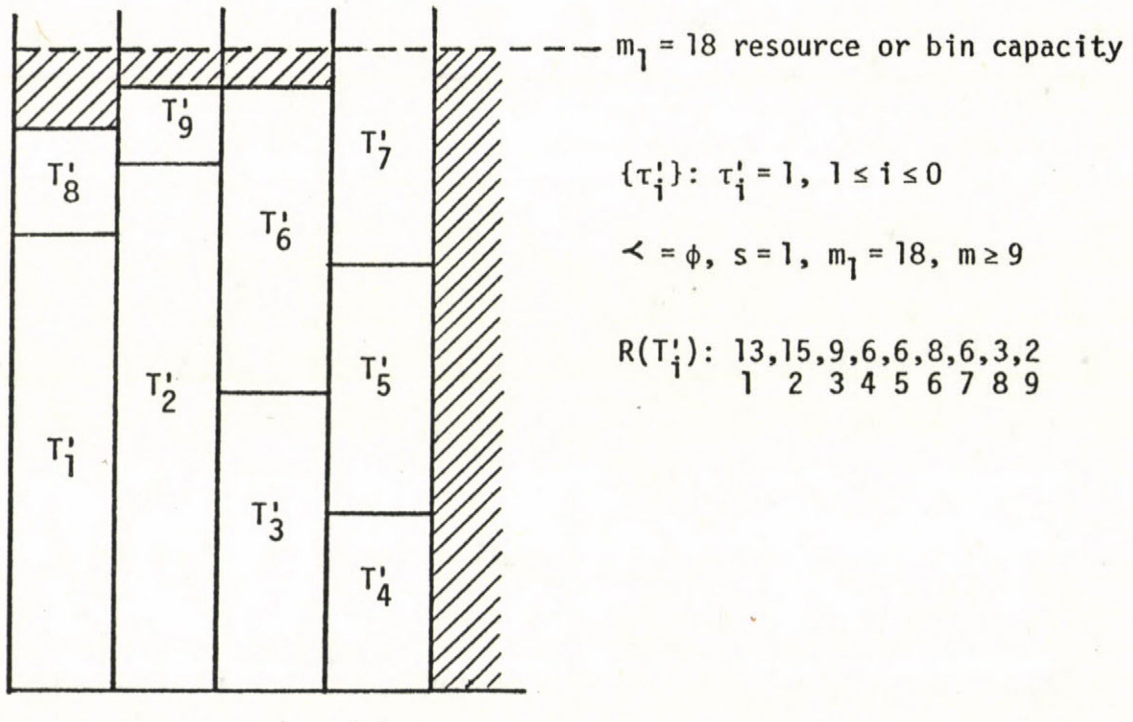
In this section we cover classical results for the special case of one processor $(m = 1)$, $\prec$ empty, and $s = 0$. For $m = 1$ the problem of minimizing $\max\{f_i\}$ vanishes; hence we are concerned only with the other measures mentioned in the previous section. The important results are summarized in the following theorems.



$\mathcal{T} = \{T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8, T_9\}$

$\{\tau_i\} = \{13,15,9,6,6,8,6,3,2\}$

$\prec = \phi$, $s = 0$



$\{\tau'_i\}$: $\tau'_i = 1$, $1 \le i \le 0$

$\prec = \phi$, $s = 1$, $m_1 = 18$, $m \ge 9$

$R(T'_i)$: 13,15,9,6,6,8,6,3,2
1 2 3 4 5 6 7 8 9

Figure 4. An equivalence of two sequencing problems. (a) Scheduling $\{T_1, \ldots, T_9\}$ to meet deadline $d = 18$ on minimum number (4) of processors. (b) Scheduling unit-length tasks with single-resource requirements to minimize schedule length.

Theorem 1 [Sm] The mean weighted flow time, lateness, and waiting time are minimized by sequencing the tasks in an order of nondecreasing ratio τ_i/w_i.

Now suppose our system includes a set of due dates for the tasks, and our problem is to optimize performance with respect to the lateness and tardiness measures. We have the following result.

Theorem 2 [Ja2] The maximum lateness and maximum tardiness are minimized by sequencing the tasks in an order of nondecreasing due dates.

Finally, for the due-date problem, let us define the slack time of T_i at time t in a schedule as $d_i - \tau_i - t$. The result of sequencing by the intuitively appealing rule of nondecreasing slack time is given by the following somewhat unexpected result.

Theorem 3 [CMM] Sequencing in an order of nondecreasing slack time maximizes the minimum lateness and minimum tardiness.

Single machine results for problems involving due-dates and the tardiness performance measure have received considerable attention. See [La1], [HK], and [RLL] (which contains a recent survey) for results related to efficient iterative and enumerative approaches. In a later section we discuss the complexity of these problems.

A well-known related result in [Mo] concerns the problem of finding a sequence that minimizes the number of late tasks. An optimal algorithm can be described as follows. First, the tasks are sequenced in the order of Theorem 2. The algorithm then finds the first late task T in this sequence and eliminates that task T' in the initial subsequence terminated by T, which has the largest execution time. The process is then iterated on the resulting sequence (with T' removed) until a sequence is obtained with no late tasks. Any sequence beginning with the initial subsequence determined by the above process minimizes the number of late tasks. Extensions have been examined in [Si2].

Another problem using as a performance measure the maximum of the weighted finishing times has been studied in [La2]. A simple algorithm is produced for $\prec$ arbitrary which minimizes $\max\{w_i f_i\}$ on one processor.

It is clearly interesting to assess the importance of sequencing algorithms by comparing their performance with that achieved under a random selection procedure. Moreover, it is also of considerable interest to analyze an algorithm that is optimal under the assumptions of known execution times, under weaker conditions in which only partial information (specifically, a probability distribution) is available. Examination of the difficulties presented by such an analysis of the models in subsequent sections reveals the general intractability of this approach in virtually all cases of interest. However, [CMM] present results for single-processor problems which illustrate the value of probability models.

IV. Results in Schedule-Length and Mean Flow-Time Minimization

We begin with some remarks on measures of complexity. These comments are rather brief, but should be sufficient to permit appreciation of the results described in this section. A careful presentation of these notions appears [U].

In general, the "complexity" of an algorithm solving a given problem refers only to its execution time, expressed as a function of the input-length; i.e., the number of bits needed to describe an instance of the problem. For our purposes we will usually specify complexity as a function of basic problem parameters, primarily the number n of tasks. In some cases, this is a considerable simplification, but not an inappropriate one for our purposes.

The detailed specification of the functions representing complexity depend on the details of algorithm and data structure design in which we take no special interest. Thus we use the order-of-magnitude notation $O(.)$, which concentrates on the terms of a function that dominate its behavior. Thus if we say that an algorithm has complexity $O(n^2)$, we simply mean that there exists a constant c such that the function cn^2 bounds the execution time as a function of n. As a specific example, if a sequencing rule depends essentially on the ordering of an arbitrary permutation of n execution times, we know that algorithms exist, using binary comparison operations only, whose complexity is $O(n \log_2 n)$. But the specific functions of execution time, for different algorithms, usually consist also of terms $O(n)$, $O(\log_2 n)$, or $O(1)$ and may differ in the coefficient of the $n \log_2 n$ term.

Any sequencing algorithm whose complexity is bounded by a polynomial in n is called a polynomial-time algorithm or an algorithm that runs in polynomial time. The corresponding problem is said to have a polynomial-time solution (algorithm). Henceforth we consider an algorithm to be efficient if it runs in polynomial time. The practical motivations of this terminology are strong, especially for large n. In this respect we should note that the sequencing problems we consider are or can be reduced to finite problems, in the sense that the solution space is finite.

Thus our notion of efficient algorithm can be associated with nonenumerative algorithms; "inefficient" algorithms are those which effectively require a search (enumeration) of the solution space and have a complexity that is at least some exponential in n.

There exists a set of problems, each of which is called NP-complete (sometimes called polynomial complete, or simply complete), which includes many classical, hard problems, Such problems include the traveling salesman problem, finding the chromatic number of a graph, and the knapsack problem, just to mention a few. It is known that in terms of complexity, all the NP-complete problems are equivalent in the following intuitive sense. If one can find a polynomial-time algorithm to solve one of these problems, one can find a polynomial-time algorithm for every other problem in the class, according to a procedure that normally varies from one problem to another. Thus either there exist polynomial-time algorithms for all the NP-complete problems, or none of them has a polynomial-time algorithm; we do not know which of these two assertions is the correct one. However, despite the lack of an answer to this "ultimate" question, there is strong evidence to suggest that all NP-complete problems are inherently intractable. As we shall see, almost all sequencing problems stated in complete generality are NP-complete.

The above intuitive discussion of course is very informal. More formally, one must proceed by defining a common mathematical basis for representing algorithms and instances of problems, defining problem complexity in terms of the model, and defining the transformation (or reducibility) of one problem to another. With such a formalism one can address for specific problems such questions as deciding their complexity in terms of desirable problem parameters, the tradeoff between storage and execution-time complexity, the essential aspect of a problem that contributes to its complexity, and so on. Later we shall illustrate the techniques for taking a new combinatorial problem and showing that it admits of a polynomial-time solution only if some known NP-complete problem has such a solution.

Schedule-Length Minimization Problems. Table 1 summarizes the majority of results presently known for these problems. To help interpret the table, the following remarks are in order.

1. The columns correspond to the possible parameters of an algorithm that solves a problem defined by the assumptions given in a row of the table. Those entries in which a value is specified eliminate a "free" parameter. For example, in problem (row) 2 we find that m is not a parameter but is fixed at $m = 2$. Similarly, $\{\tau_i\}$ is not a parameter, for the specific common value of the τ_i does not influence the algorithm. The partial order is a free parameter, but no preemptions are allowed and there is no additional resource requirement that can be specified. Here and in the sequel, n is always a free parameter.

2. The entries in the column on complexity given as "open" simply mean that the question of the complexity of the corresponding problem has not been resolved.

3. For each of the nonpreemptive problems for which polynomial-time optimal algorithms are known, there in fact exists an optimal list-scheduling algorithm.

4. An important point concerning the NP-complete problems indicated in Table 1 is that they represent the simplest cases for which NP-completeness is known. Thus the reader can and should make appropriate inferences

regarding more general problems. Generalizing any one of the parameter restrictions in a given problem (e.g., assuming as appropriate, non-identical processors, nonempty partial orders, additional resources, etc.) obviously produces a problem at least as hard as the original one.

One important observation in this respect concerns a comparison of rows such as 6 and 4. Since problem 6 is NP-complete, it is easy to see that so also is the problem for m a free parameter. However, the converse is not necessarily true. Problem 4 is NP-complete, but it is not known whether for any fixed $m \geq 3$ the corresponding problem (problem 3) is NP-complete (and in fact a considerable effort has been invested in resolving this problem for $m = 3$).

5. Regarding polynomial-time algorithms, we do not in all cases claim that the complexity shown is minimal. As an analysis of the sequencing problems indicates, however, there is virtually conclusive evidence to this fact in a number of cases. For example, algorithms necessarily depending on an ordering of execution times must surely be of at least $O(n \log_2 n)$ complexity, assuming an unordered list to begin with. Algorithms depending on precedence constraint structure must surely have a complexity at least $O(n^2)$, since there are $O(n^2)$ edges in a general precedence graph (see problem 2, e.g.). However, complexity must also depend on data structures. For example, if a graph is specified by an edge list containing redundant transitive edges, the complexity may well increase. Problem 2 is a relevant example, for the complexity shown depends on the absence of transitive edges; the problem of removing such edges from a dag edge list has $O(n^{2.8})$ complexity according to the best algorithm currently known [AGU]. Thus without the assumption of the absence of transitive edges, problem 2 would have a complexity dominated by this latter algorithm.

Table 1. Results for Minimizing $\omega = \max\{f_i\}$

	Problem complexity	m	$\{\tau_i\}$*	$\prec$	Rule†	Resources s	Resources $\{m_i\}$	References
1.	$O(n)$	-	Equal	Forest	Nonpr	0		[H]
2.	$O(n^2)$	2	Equal	-	Nonpr	0		[CG]
3.	Open	Fixed $m \geq 3$	Equal	-	Nonpr	0		
4.	NP-complete	-	Equal	-	Nonpr	0		[U]
5.	NP-complete	Fixed $m \geq 2$	$\tau_i=1$ or 2 for all i	-	Nonpr	0		[U]
6.	NP-complete	Fixed $m \geq 2$	-	ϕ	Nonpr	0		
7.	$O(n \log_2 n)$	-	-	Forest	Pr	0		[MC]
8.	$O(n^2)$	2	-	-	Pr	0		[MC]
9.	Open	Fixed $m \geq 3$	-	-	Pr	0		
10.	NP-complete	-	-	-	Pr	0		[U]
11.	$O(n^3)$	2	Equal	ϕ	Nonpr	-	-	[GJ1]
12.	NP-complete	Fixed $m \geq 2$	Equal	Forest	Nonpr	1	-	[GJ1]
13.	NP-complete	Fixed $m \geq 2$	Equal	-	Nonpr	1	$m_1=1$	[U]
14.	NP-complete	Fixed $m \geq 3$	Equal	ϕ	Nonpr	1	-	[GJ1]
15.	$O(n \log_2 n)$	2	Flow shop		Nonpr	0		[Jo]
16.	NP-complete	Fixed $m \geq 3$	Flow shop		Nonpr	0		[GJS]
17.	NP-complete	Fixed $m \geq 2$	Job shop		Nonpr	0		[GJS]
18.	NP-complete	Fixed $m \geq 2$	-	ϕ	Σf_i is minimum	0		[CS]

*Identical processors assumed throughout, except for problems 15-17.

†Nonpr and pr are abbreviations for nonpreemptive and preemptive, respectively.

The optimization algorithms for problems 1, 2, 7, and 8 are essentially critical path algorithms (also called level-by-level algorithms), since the scheduling priority at any point is given sequentially to the remaining tasks at the highest level. (Actually, in problem 2 this criterion must be augmented by another property of tasks which, in the sense of the precedence graph, is locally computable.) Clearly, the complexity of these algorithms is dominated by the complexity of the graph operations needed to find critical paths. Note that "first-order" generalizations of these problems are NP-complete as shown in problems 4, 5 and 10. For special cases applicable to the case of two non-identical processors, see [B2].

Problem 15 is solved essentially by an appropriate ordering of the tasks; hence the complexity is determined by the complexity of sorting. This is the earliest result (1954) in schedule-length minimization. Apart from problem 1 (1961) and special cases of problem 16, the remaining results date from 1968.

The flow-shop problem referred to in problems 15 and 16 is defined as follows. Each task system consists of a set of n/m chains of length m, usually called jobs in the literature, with the restriction that the ith task in a chain must be executed on processor P_i (n is a multiple of m). In terms of $[\tau_{ij}]$ in our original model, columns $km+i$, $k=0,1,\ldots,n/m-1$, correspond to tasks that must execute on P_i, so that $\tau_{j,km+i}=\infty$ for all $j \neq i$. The T_{km+i}, $i=1,2,\ldots,m$, will correspond to the sequence of tasks in job k.

Problem 17 concerns the complexity of the simple job-shop problem, described as follows. As in the flow-shop problem $\prec$ is a set of chains, each of arbitrary length, called jobs. (In the general job-shop problem there is no constraint on $\prec$.) Each job J_i can be characterized by a sequence of pairs (a_{ij}, P_{ij}), where a_{ij} is the execution time of the jth task of J_i and P_{ij} is the processor on which it must execute. The general problem is NP-complete for all $m \geq 2$. (Later in this section we illustrate a technique that can be used for this result.) However, there is an interesting special case, which we now describe, that admits of a polynomial-time solution for $m = 2$ [Ja1].

Each job J_i is characterized simply by a single operation on each machine (i.e., each J_i is a two-task chain): each J_i either executes a task first on P_1 then on P_2 or it executes a task first on P_2 then on P_1. Let (x_i, y_i) be the execution times for the tasks of J_i. Construct two ordered sets C_1 and C_2 such that all J_i whose first task executes on P_1 (respectively P_2) are elements of C_1 (respectively C_2) and such that if (x_i, y_i) and (x_j, y_j) are both in C_1, then $\min(x_i, y_i) < \min(x_j, y_j)$ implies that J_i precedes J_j in the ordering. (This is the rule used in problem 15.) Order C_2 in the same way. According to this ordering, assign tasks on P_1 first from C_1 then C_2, observing precedence constraints. Similarly, assign tasks on P_2 first from C_2 then C_1. Note that x_i or y_i can be zero, in which case we effectively account for jobs having only a single task for one machine or the other.

The basic intractability of problems in which there are additional resources is made clear in problems 12 to 14. Problem 11, not discussed subsequently, can be viewed as an application of the maximum matching [E] problem. One may construct in less than $O(n^3)$ time an undirected graph G such that $(T_i,T_j) \in G$ if and only if T_i and T_j are compatible: that is, $R_k(T_i) + R_k(T_j) \leq m_k$, for all k; hence T_i and T_j can execute in parallel. A maximum matching of G provides a shortest-length schedule and can be found in $O(n^3)$ time.

Problem 7 with $\prec = \phi$ has a well-known $O(n)$ algorithm [Mc], which we reconsider later.

We conclude this subsection with a brief description of recent results bearing on deadline problems relevant to the problems we have been discussing. Consider the problem of scheduling equal-execution-time tasks on two identical processors when there is an arbitrary partial order and each task has an individual deadline to meet. If the partial order is given in transitively closed form, it is known [GJ2] how to determine in time $O(n^2)$ whether a valid schedule exists which meets all the deadlines, and if so, generate one. If it is desired to know whether a valid schedule exists that violates at most k deadlines, where k is fixed, this can be done in time $O(n^{k+2})$. However, the problem of the existence of a valid schedule that violates at most k deadlines, k variable, is NP-complete, even for only one processor.

Mean Flow-Time Results. We now present recent results generalizing those in the previous section for single-machine systems. Additional resources are not included because corresponding results do not exist. Discipline constraints are also not considered, since nonpreemptive scheduling is assumed throughout for the same reason. (Recall, however, that a preemptive capability does not accomplish anything more in the case of identical processors, non-negative weights, and an empty partial order.) Comments 1, 2, 4, and 5 describing Table 1 also apply to Table 2 in which we summarize most of the results.

Table 2. Results for Minimizing $\Sigma_i w_i f_i$

	Parameters: n and				
Problem complexity	$\prec$	$\{w_i\}$	$\{\tau_{ij}\}$	m	References
1. $O(n^3)$	ϕ	Equal	-	-	[BCS2,Ho2]
2. NP-complete	Set of chains	Equal	Identical processors	-	[BSe]
3. $O(n^2)$	Forest	-	Identical processors	1	[Ho1](See also [Ga,Si1])
4. Open	-	-	Identical processors	1	
5. NP-complete	ϕ	-	Identical processors	Fixed $m \geq 2$	[BCS1]
6. NP-complete	Flow shop			Fixed $m \geq 2$	[GJS]

For problem 1 of Table 2 the existence of a polynomial-time algorithm is indicated for the most general case of independent tasks and equal deferral costs. Special cases of this result have been known for some time, in particular the result for SPT sequencing introduced earlier. The result of problem 1 is based on a formulation that identifies it as a special case of the general transportation problem.

The solution of problem 3 results from a general formulation that proceeds by identifying a locally computable cost function (sequencing criterion) to be associated with each vertex (subtree) of the forest. This function is in fact a generalization of the function τ_i/w_i introduced in Theorem 1. Using an ordering obtained from these computed costs, the optimal sequence is achieved by recursively identifying the subtrees of the forest that are to be executed next.

Problem 6 of Table 2 concerns a flow-shop problem that has received considerable attention. This NP-complete problem forms the basis of an illustration in [KS] of enumerative and approximate techniques applied to sequencing problems.

The problem of minimizing mean tardiness (see previous section) is also relevant to Table 2, with due dates also a parameter. It has been shown that the minimum mean weighted tardiness problem is NP-complete for all $m \geq 1$ [BLR]. An interesting open problem is the complexity of the minimum mean tardiness problem (equal weights) when $m = 1$.

Further Remarks on the Complexity of Sequencing Problems. In [U] a formal approach to the definition of problem complexity is introduced and used in demonstrations of the NP-completeness of a large variety of sequencing problems, either explicitly or implicitly. Such an approach begins with the description of a computer model by which algorithms for combinatorial problems can be commonly posed, in a manner that simplifies their analysis and comparison with respect to their complexity. The notion of polynomial-time nondeterministic algorithms is defined, a notion which identifies with problems admitting of an enumerative solution describable by a polynomial-depth search tree. As a matter of definition, the existence of such an algorithm is necessary for the NP-completeness of a sequencing problem. Finally, the concept of polynomial reducibility among combinatorial problems is defined, a definition used in deciding that a problem has a polynomial-time solution if and only if some NP-complete problem has such a solution.

In connection with Table 1, comment 4 is worth repeating at this point. Namely, the reader will be able to infer the NP-completeness of new problems from similar results shown for other problems. For example, the NP-complete problem 6 of Table 1 is known to have a polynomial-time solution if and only if a similar solution exists for the classical deadline problem [U]; that is, given a deadline d common to all (independent) tasks, is there a schedule on m identical processors such that all tasks finish no later than d? The consequent NP-completeness of this problem in effect accounts in a straightforward manner for the NP-completeness of the general additional-resource $(s \geq 1)$ and bin-packing problems [G3].

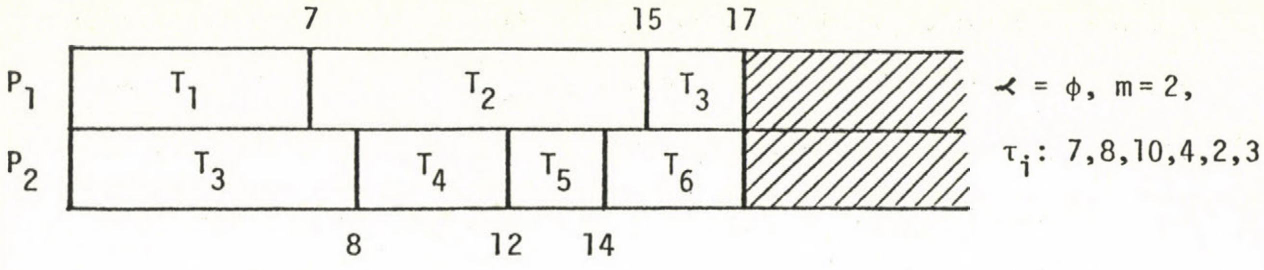
We can illustrate informally but effectively the process of reducing one combinatorial problem to another by the following examples. Consider as a basis the simple version of the knapsack problem, stated as follows. Given as parameters a set $X = \{a_1, \ldots, a_n\}$ of n positive integers and an additional integer b, is there a subset of X whose elements sum exactly to b? That is, does the equation $\sum_{i=1}^{n} c_i a_i = b$ have a 0-1 solution in the c_i's? This problem is well known to be NP-complete [Ka]. We now make use of this problem in assessing the complexity of the following sequencing problems.

1. Consider problem 7 of Table 1 with $\prec = \phi$, for which it is readily verified that in general there can be $O(n!)$ solutions. Suppose we add in the "Discipline" column that the number of preemptions must be minimized. Call this the PM(m) problem for m processors.

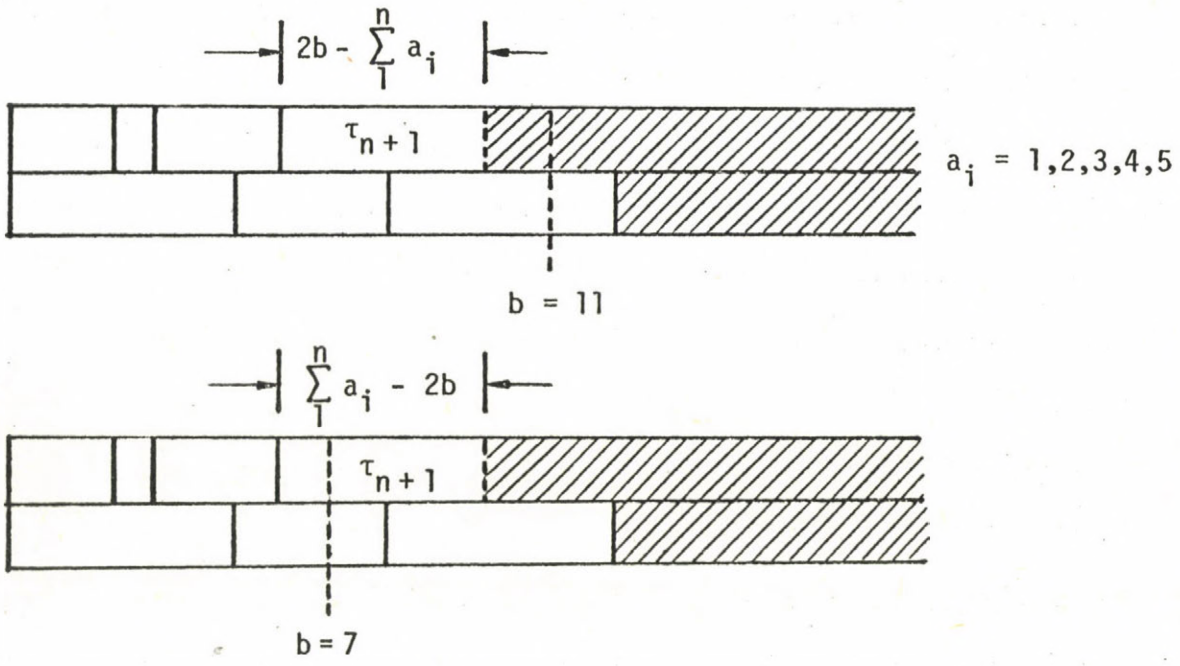
An example of the case for $m = 2$ and $\max\{\tau_i\} < \sum_{i=1}^{n} \tau_i/2$ is shown in Fig. 5a. Note that any sequence of tasks can be used in carrying out the assignment; one simply must insert a preemption when necessary for the last task scheduled on P_1 (first task scheduled on P_2). Thus at most one preemption is necessary, and its necessity depends on finding a subset of $\{\tau_i\}$ that adds up to exactly $\sum_{i=1}^{n}\tau_i/2$. Therefore, we can reduce a knapsack problem to a PM(2) problem as follows. Let b and $\{a_i\}_{i=1}^{n}$ be an instance of the knapsack problem. Consider the instance of the PM(2) problem given by $\{\tau_i\}_{i=1}^{n+1}$ where $\tau_i = a_i$, $1 \le i \le n$, and $\tau_{n+1} = |2b - \sum_{i=1}^{n} a_i|$. It is easy to see that this PM(2) problem has a solution with no preemptions if and only if the original knapsack problem has a solution. Figure 5b provides examples. Clearly, because of the simplicity of the reduction (calculating $|2b - \sum_{i=1}^{n} a_i|$), we expect the PM(2) problem to be at least as hard as the knapsack problem. We make the same statement for the general case $m \ge 3$, since we can add $m-2$ tasks of length $\sum_{i=1}^{n}\tau_i/2$ to reduce the PM(2) problem to the PM(m) problem.

2. Consider problem 16 of Table 1, and let the jobs (i.e., three-task chains) be specified by the triples (x_i, y_i, z_i), $1 \le i \le n$, where x_i (respectively, y_i and z_i) is the time required for the first (respectively, second and third) task to execute on P_1 (respectively, P_2 and P_3). We can provide as follows the basic observation in a proof of NP-completeness.

From an instance b, $\{a_i\}$ of the knapsack problem, let the execution times in an instance of problem 16 for $m = 3$ be given by $(0, a_1, 0), \ldots, (0, a_n, 0)$, $(b, 1, \sum_{i=1}^{n} a_i - b)$. Figure 6 gives an example schedule. Note that a minimal-length schedule is at least $\sum_{i=1}^{n} a_i + 1$ long. But as illustrated, we can know whether we have a minimal-length schedule only if we can answer the question: is there a subset of the a_i's which sum exactly to b? Thus by a simple algorithm we can transform an instance of the knapsack problem to an instance of the $(m = 3)$



(a)



(b)

Figure 5. Preemptive scheduling

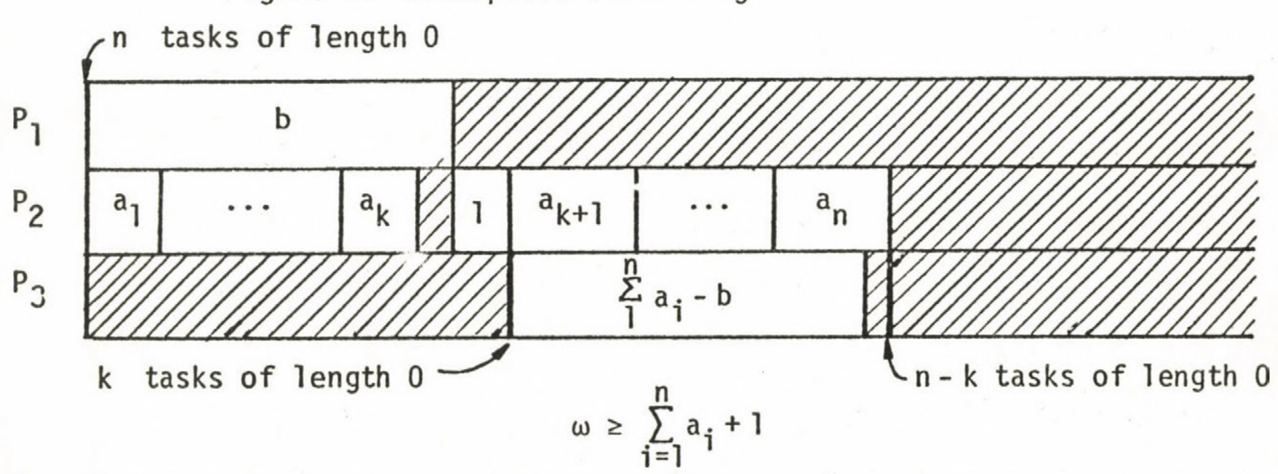


Figure 6. A flow-shop schedule, m = 3

flow-shop problem such that an algorithm for the flow-shop problem finds a minimal solution if and only if the original knapsack problem has a solution. As before, it is not difficult to render a general case for $m > 3$ identical to a problem for $m = 3$. It follows that we can expect the problem to be at least as hard as the knapsack problem.

A similar type of argument can be easily constructed for the result of problem 17, taking advantage of the absence of the "processor ordering" implicit in the flow-shop problem. It is important to note that the knapsack problem can be solved in time polynomial in the desired sum b (this does not contradict the NP-completeness of the knapsack problem since b can be expressed using $\log_2 b$ bits). Therefore, if the complexity parameter of interest is the sum of task execution times we can not infer from the above arguments that these problems are intractable. For example, Problem 6 in Table 1 can be solved in polynomial time for fixed $m \geq 2$ for this less stringent input measure. However, Problems 16 and 17 remain NP-complete even for the latter measure [GJS].

Clearly, we have only shown these problems to be at least as hard as NP-complete problems; we have not verified that they are no harder. At this point it is best to leave full proofs of NP-completeness to those familiar with [U]. We might note here, however, that the problem of showing reducibility has no generally applicable structure (thus the open problems of Tables 1 and 2), and solutions to these problems can be quite involved.

We conclude this discussion of complexity with the warning that having a specific problem in hand, one should be fully aware of any special features or constraints that might exist. For in this case the

NP-completeness of the general problem to which it corresponds should not immediately cause us to abandon hope for an efficient optimal algorithm. Specific limitations on the problem may enable us to design an algorithm whose execution time is bounded by a polynomial in the basic characteristics of the problem.

V. Concluding Remarks

We have limited our discussion to efficient optimization algorithms and to problem complexity. The results on complexity demonstrate rather clearly that much of the progress in the theory of scheduling is very likely to result from the study of fast heuristics (approximate algorithms) and effective enumerative procedures. In [G2] and [G3] performance bounds are presented and derived for a variety of sequencing and allocation heuristics whose complexity is at most $O(n^2)$. We refer to [KS], [CMM] and [Ba] for extensive treatments of enumerative techniques including dynamic programming and local neighborhood search.

One performance measure of broad interest that we have neglected, and for which results have been obtained very recently, is the mean number of tasks, $\bar{N}(S)$, in the system calculated over the interval $[0,\omega(S)]$. In general, such a measure is useful in expressing expected inventory or storage requirements for tasks in the job shop or computer. We can write

$$\bar{N}(S) = \frac{1}{\omega(S)} \int_0^{\omega(S)} N(t)dt$$

where $N(t)$ is the number of uncompleted tasks in the system at time t. Clearly $N(t)$ takes the form of a decreasing staircase function with changes in value occurring at task completion times (subsets of which may, of course, be coincident). Assuming unit weights in (2), one may show rather easily

$$\bar{N}(S) = n \frac{\bar{\omega}(S)}{\omega(S)}$$

which is obtained independently of scheduling disciplines, number of processors, or any property other than finishing times. From this expression we note immediately that for $m=1$ processors $\bar{N}(S)$ is minimized when $\bar{\omega}(S)$ is minimized, since $\omega(S)$ is in fact not a function of S for $m=1$. For the case $m \geq 2$ it is known [CL] that the problem of finding a sequence minimizing $\bar{N}$ is NP-complete.

A recent generalization of our general model for which an efficient optimal algorithm has been found is described as follows. With respect to problem 3 of Table 2 we once again assume a forest and a single processor, but each tree (called a job) in the forest is now regarded as a decision tree in the following sense. There is an arbitrary (discrete) probability distribution associated with each vertex governing the decision as to which task (vertex) is to be executed next in the job. Thus the eventual execution of a job consists of a sequence or chain of tasks beginning with the root and ending at a descendant vertex, and the chain can be interrupted only at task termination times. These sequences and their total execution times are random variables, hence the problem becomes one of minimizing the expected value of the mean weighted flow time. A sequencing criterion for deciding sequentially which job is next to have a task executed has been developed along the lines of problem 3. The complexity question is studied, with the result that an $O(n^2)$ algorithm is produced [BCJ].

References

[AGU] Aho, A. V., M. R. Garey, and J. D. Ullman, "The Transitive Reduction of a Directed Graph," SIAM Journal on Computing, 1 (1972), 131-137.

[B1] Baer, J. L., "A Survey of Some Theoretical Aspects of Multiprocessing," Computing Surveys, 5, 1 (1973).

[B2] Baer, J. L., "Optimal Scheduling on Two Processors of Different Speeds," Computer Architectures and Networks, E. Gelenbe and R. Mahl (Eds.), North Holland Publishing Company, 1974, 27-45.

[Ba] Baker, K., Introduction to Sequencing and Scheduling, John Wiley and Sons, 1974.

[BCJ] Bruno, J., E. G. Coffman, Jr., and D. B. Johnson, "On Batch Scheduling of Jobs with Stochastic Service Times and Cost Structures on a Single Server," Technical Report No. 154, Computer Science Dept., Pennsylvania State University, August 1974, Journal of Computer and System Sciences (to appear).

[BCS1] Bruno, J., E. G. Coffman, Jr., and Ravi Sethi, "Scheduling Independent Tasks to Reduce Mean Finishing Time," Communications of the ACM, 17, 7 (1974), 382-387.

[BCS2] Bruno, J., E. G. Coffman, Jr., and Ravi Sethi, "Algorithms for Minimizing Mean Flow Time," Proceedings, IFIPS Congress, North Holland Publishing Co., August 1974, 504-510.

[BLR] Brucker, P., K. K. Lenstra, and A. H. G. Rinooy Kan, "Complexity of Machine Scheduling Problems," Technical Report BW 43/75, Mathematisch Centrum, Amsterdam, 1975, Operations Research (to appear).

[BSe] Bruno, J., and R. Sethi, "On the Complexity of Mean Flow-Time Scheduling," Technical Report, Computer Science Dept., Pennsylvania State University, 1975.

[C1] Coffman, E. G., Jr., "A Survey of Mathematical Results in Flow-Time Scheduling for Computer Systems," Proceedings, GI 73, Hamburg, Springer-Verlag, 1973, 25-46.

[C2] Coffman, E. G., Jr., (Editor) Computer and Job-Shop Scheduling Theory, John Wiley and Sons, 1975.

[CD] Coffman, E. G., Jr., and P. J. Denning, Operating Systems Theory, Prentice Hall, Englewood Cliffs, N. J., 1973.

[CG] Coffman, E. G., Jr., and R. L. Graham, "Optimal Scheduling for Two Processor Systems," Acta Informatica, 1, 3 (1972), 200-213.

[CL] Coffman, E. G., Jr., and J. Labetoulle, "Deterministic Scheduling to Minimize Mean Number in System," Proceedings, HICCS 9, University of Hawaii (to appear Jan. 1976).

[CMM] Conway, R. W., W. L. Maxwell, and L. W. Miller, Theory of Scheduling, Addison-Wesley, Reading, Mass., 1967.

[CS] Coffman, E. G., Jr., and R. Sethi, "Algorithms Minimizing Mean-Flow-Time: Schedule-Length Properties," Technical Report, Computer Science Dept., Pennsylvania State University, 1973, Acta Informatica (to appear).

[E] Edmonds, J., "Paths, Trees, and Flowers," Canadian Journal of Mathematics, 17 (1965), 449-467.

[G1] Graham, R. L., "Bounds on Multiprocessing Timing Anomalies," SIAM Journal on Applied Math., 17 (1969), 416-429.

[G2] Graham, R. L., (Chapter 5 of [C2]).

[G3] Graham, R. L., "Bounds on Multiprocessing Anomalies and Related Packing Algorithms," Proceedings, AFIPS Conference, 40 (1972), 205-217.

[Ga] Garey, M. R., "Optimal Task Sequencing with Precedence Constraints," Discrete Math., 4 (1973) 37-56.

[GJ1] Garey, M. R. and D. S. Johnson, "Complexity Results for Multiprocessor Scheduling Under Resource Constraints," Proceedings, 8th Annual Princeton Conference on Information Sciences and Systems, 1974.

[GJ2] Garey, M. R. and D. S. Johnson, "Deadline Scheduling of Equal Execution Time Tasks on Two Processors," Technical Report, Bell Laboratories, Murray Hill, N. J., 1975.

[GJS] Garey, M. R., D. S. Johnson, and R. Sethi, "The Complexity of Flow Shop and Job Shop Scheduling," Technical Report No. 168, Computer Science Dept., The Pennsylvania State University, 1975.

[H] Hu, T. C., "Parallel Sequencing and Assembly Line Problems," Operations Research 9, 6 (1961), 841-848.

[HK] Held, M. and R. Karp, "A Dynamic Programming Approach to Sequencing Problems," SIAM Journal on Applied Math., 10, 11 (1962), 196-210.

[Ho1] Horn, W. A., "Single-Machine Job Sequencing with Treelike Precedence Ordering and Linear Delay Penalties," SIAM Journal on Applied Math., 23 (1972) 189-202.

[Ho2] Horn, W. A., "Minimizing Average Flow Time with Parallel Machines Operations Research, 21 (1973), 846-847.

[Ja1] Jackson, J. R., "An Extension of Johnson's Results on Job-Lot Scheduling," Naval Research and Logistics Quarterly, 3, 3 (1956).

[Ja2] Jackson, J. R., "Scheduling a Production Line to Minimize Maximum Tardiness," Research Report No. 43, Management Sciences Research Project, UCLA, January 1955.

[Jo] Johnson, S. M., "Optimal Two- and Three-Stage Production Schedules," Naval Research and Logistics Quarterly, 1, 1 (1954).

[Ka] Karp, R. M., "Reducibility Among Combinatorial Problems," Complexity of Computer Computation, R. E. Miller and J. W. Thatcher (Eds.), Plenum Press, New York, 1972, 85-104.

[KS] Kohler, W. H. and K. Steiglitz (Chapter 6 in [C2].)

[La1] Lawler, E. L., "On Scheduling Problems with Deferral Costs," Management Science, 11 (1964), 280-288.

[La2] Lawler, E. L., "Optimal Sequencing of a Single Machine Subject to Precedence Constraints," Management Science, 14 (1973), 544-546.

[Mc] McNaughton, R., "Scheduling with Deadlines and Loss Functions," Management Science, 12, 7 (1959).

[MC1] Muntz, R. R. and E. G. Coffman, Jr., "Preemptive Scheduling of Real Time Tasks on Multiprocessor Systems," Journal of the ACM, 17, 2 (1970), 324-338.

[MC2] Muntz, R. R. and E. G. Coffman, Jr., "Optimal Preemptive Scheduling on Two-Processor Systems," IEEE Transactions on Computers, C-18, 11 (1969), 1014-1020.

[Mo] Moore, J. M., "An n Job, One Machine Sequencing Algorithm for Minimizing the Number of Late Jobs," Management Science, 15, (1968), 102-109.

[RLL] Rinnooy Kan, A. H. G., B. J. Lageweg, and J. K. Lenstra, "Minimizing Total Costs in One-Machine Scheduling," Technical Report No. BW33/74, Mathomatisch Centrum, Amsterdam, 1974.

[Si1] Sidney, J. B., "One Machine Sequencing with Precedence Relations and Deferral Costs - Part I.," Working Paper No. 124, "Part II," Working Paper No. 125, Faculty of Commerce and Business Administration, University of British Columbia, 1972.

[Si2] Sidney, J. B., "An Extension of Moore's Due-Date Algorithm," Symp. on Theory of Scheduling and Its Applications, S. M. Elmagrabhy (Ed.), Springer-Verlag, 1973, 393-398.

[Sm] Smith, W. E., "Various Optimizers for Single-Stage Production," Naval Research and Logistics Quarterly, 3, 1 (1956).

[U] Ullman, J. D., (Chapter 4 in [C2].)

Összefoglaló

Determinisztikus ütemezés: "komplexitás" és optimális algoritmusok

Coffman E. G.

A dolgozat korlátokat ad a hatékonyságra különféle ütemezések esetén, ha a "komplexitás" legfeljebb $0(n^2)$. Optimális ütemezési algoritmusokat ad preemptiv és nempreemptiv esetekben a részben rendezett "task"- rendszerek számára. Részletesen elemzi a "polinomálisan teljes" feladatok esetét.

Р Е З Ю М Е

ДЕТЕРМИНИЧЕСКОЕ РАСПИСАНИЕ: "СЛОЖНОСТЬ" И ОПТИМАЛЬНЫЕ АЛГОРИТМЫ

Коффман Э.Г.

В работе даны ограничения для эффективности, в случае разных расписаний, если сложность имеет порядок 0 (n^2). Даются оптимальные алгоритмы расписания в "преэмптивном" и непреэмтивном" случаях для упорядоченных систем тасков.

Подробно изучается случай полиномиально полной задачи.