SIAM J. COMPUT.
Vol. 6, No. 3, September 1977

TWO-PROCESSOR SCHEDULING WITH START-TIMES AND DEADLINES*

M. R. GAREY AND D. S. JOHNSON†

Abstract. Given a set $\mathcal{T}=\{T_1, T_2, \cdots, T_n\}$ of tasks, each T_i having execution time 1, an integer start-time $s_i \geqq 0$ and a deadline $d_i > 0$, along with precedence constraints among the tasks, we examine the problem of determining whether there exists a schedule on two identical processors that executes each task in the time interval between its start-time and deadline. We present an $O(n^3)$ algorithm that constructs such a schedule whenever one exists. The algorithm may also be used in a binary search mode to find the shortest such schedule or to find a schedule that minimizes maximum "tardiness". A number of natural extensions of this problem are seen to be *NP*-complete and hence probably intractable.

Key words. multiprocessing systems, scheduling algorithms, *NP*-complete problems

1. Introduction. Since publication of the book *Theory of Scheduling* [4] by Conway, Maxwell, and Miller in 1967, considerable progress has been made in the mathematical analysis of abstract multiprocessing systems. One combinatorial model which is central to much of this work consists of a number m of identical, independent *processors*, a finite set $\mathcal{T}=\{T_1, T_2, \cdots, T_n\}$ of *tasks* to be executed, an *execution time* $\tau_i > 0$ for each $T_i \in \mathcal{T}$, and a *partial order* $\prec$ on $\mathcal{T}$. The partial order describes precedence constraints between the tasks, restricting allowable schedules to those in which, whenever $T_i \prec T_j$, the task T_j does not begin executing until T_i has been completed. The primary goal of scheduling is usually to minimize either the mean-time-in-system (mean flow time) or the maximum-time-in-system (maximum finishing time). Unfortunately these goals can be quite difficult to achieve and, in fact, most classes of scheduling problems appear to be computationally intractable [8], [10], [12], [15]. A notable exception to this state of affairs is the case of minimizing maximum finishing time when $m = 2$ and each $\tau_i = 1$, $1 \leqq i \leqq n$. Efficient scheduling algorithms for this case have been described in [3], [6], [7], [13]. These results have been extended in [9], which presents an efficient scheduling algorithm for the more complicated version of this problem in which each task $T_i \in \mathcal{T}$ also has a *deadline* $d_i > 0$ by which time its execution must be completed. In this paper we further extend these results to the situation in which each task $T_i \in \mathcal{T}$ has not only a deadline $d_i > 0$ but also an integer *start-time* s_i, $0 \leqq s_i \leqq d_i$, such that T_i must be executed entirely in the time interval $[s_i, d_i]$. We describe an $O(n^3)$ algorithm which determines whether there exists a schedule meeting all start-time, deadline, and precedence constraints and which constructs such a schedule if one exists.

In § 2 of this paper, we will describe the basic ideas behind the algorithm and show why it works. In § 3 we provide the details as to how it can be implemented to run in time $O(n^3)$. In § 4 we show how this basic feasibility algorithm can be used in iterative procedures to find schedules that minimize maximum finishing time or maximum tardiness. We also briefly examine the computational complexity of some related problems.

* Received by the editors January 26, 1976, and in revised form June 9, 1976.

† Bell Laboratories, Murray Hill, New Jersey 07974.

We conclude this section with a few preliminary definitions. For the sake of generality, we state them in terms of arbitrary m, $\{\tau_i\}$, and $\{s_i\}$.

A task T_i is called a *predecessor* of task T_j (and T_j is a *successor* of T_i) whenever there exists a sequence of tasks $T'_1, T'_2, \cdots, T'_k$, $k \geqq 1$, such that $T_i < T'_1 < T'_2 < \cdots < T'_k = T_j$. Given m, $\mathcal{T}$, $\{\tau_i\}$, $\{s_i\}$, $\{d_i\}$, and the partial order $<$, a *valid schedule* is a total function $\sigma : \mathcal{T} \to [0, \infty)$ which satisfies the following three properties:

(i) For all $t \in [0, \infty)$, $|\{T_i \in \mathcal{T} : \sigma(T_i) \leqq t < \sigma(T_i) + \tau_i\}| \leqq m$;
(ii) Whenever $T_i < T_j$, $\sigma(T_i) + \tau_i \leqq \sigma(T_j)$;
(iii) For each $T_i \in \mathcal{T}$, $\sigma(T_i) \geqq s_i$.

In plain language, the function σ assigns a starting time for execution to each task in $\mathcal{T}$, property (i) states that no more than m tasks are ever executed simultaneously, property (ii) ensures that the partial order is respected, and property (iii) ensures that the start-time constraints are not violated (although the deadlines may be). Note that the processing is assumed to be *nonpreemptive* in that once a task is initiated it continues executing until its completion. A valid schedule σ is said to *meet all the deadlines* if, for each $T_i \in \mathcal{T}$, $\sigma(T_i) + \tau_i \leqq d_i$. The *finishing time* ω for a valid schedule σ is given by $\omega = \max \{\sigma(T_i) + \tau_i : T_i \in \mathcal{T}\}$.

Unless stated otherwise, we shall assume henceforth that $m = 2$, each $\tau_i = 1$, and each s_i is an integer. Notice that, when minimizing maximum finishing time under these assumptions, there is no loss of generality in restricting consideration to schedules σ which map $\mathcal{T}$ into the nonnegative integers. Thus we may assume also that each deadline d_i is a positive integer.

2. The basic scheduling algorithm. In this section we describe the basic ideas behind our algorithm, which finds a valid schedule meeting all deadlines whenever one exists. The algorithm is similar to that of [9] in that it may be thought of as finding a priority list for directing the scheduling process. A *priority list* L is a permutation of the tasks in $\mathcal{T}$ which is used to define a valid schedule f in the following intuitive manner: Initially, all processors are idle. At any time t at which a processor is idle, the processor instantaneously scans L from the beginning and selects the first task T_k (if any) which may validly be executed, i.e., $s_k \leqq t$, all predecessors of T_k have been completed, and T_k itself has not yet been started. In case of a tie, T_k is assigned to the idle processor with lowest index and the remaining processors continue scanning the list. In our formal notation $\sigma(T_k)$ is set equal to that time t at which T_k is selected for execution by one of the processors. The reader should have no difficulty in specifying an $O(n^2)$ algorithm for computing σ from the list L.

Our algorithm will determine a specific priority list L for scheduling the tasks by this method. As in [9] the key idea involves a special modification of the task deadlines, having the property that a valid schedule meets all the modified deadlines if and only if it meets all the original deadlines. However the addition of task start-time constraints considerably complicates the necessary deadline modifications.

In order to state the lemma on which our deadline modifications are based, we require a few preliminary definitions. For any task T_i and integers s, d

satisfying $s_i \leqq s \leqq d_i \leqq d$, we define $S(i, s, d)$ to be the set of all tasks T_j $(j \neq i)$ which have $d_j \leqq d$ and either are successors of T_i or have $s_j \geqq s$. Let $N(i, s, d)$ denote the number of tasks in $S(i, s, d)$. We use $\lceil x \rceil$ to denote the least integer no less than x.

LEMMA 1. *For any task T_i and integers s, d satisfying $s_i \leqq s \leqq d_i \leqq d$, if $N(i, s, d) \geqq 2(d-s)$, then T_i must be completed by time $d - \lceil N(i, s, d)/2 \rceil$ in any valid schedule that meets all task deadlines.*

Proof. Suppose $N(i, s, d) \geqq 2(d-s)$ and let σ be any valid schedule that meets all task deadlines. We divide the proof into two cases, depending on whether $N(i, s, d)$ equals or exceeds $2(d-s)$.

First suppose $N(i, s, d) > 2(d-s)$. Since all tasks in $S(i, s, d)$ must be completed by time d and there are only two processors, operating nonpreemptively, there must be some task $T_j \in S(i, s, d)$ for which $\sigma(T_j) \leqq d - \lceil N(i, s, d)/2 \rceil < s$. Since $\sigma(T_j) < s$, the definition of $S(i, s, d)$ implies that T_j must be a successor of T_i. Hence T_i must be completed when T_j starts at time $\sigma(T_j)$ and the desired result follows.

Now suppose $N(i, s, d) = 2(d-s)$. Then we have $\lceil (N(i, s, d)+1)/2 \rceil = \lceil N(i, s, d)/2 \rceil + 1$. We parallel the previous argument using the set $S = S(i, s, d) \cup \{T_i\}$ instead of $S(i, s, d)$. Since all tasks in S must be completed by time d and there are only two processors, operating nonpreemptively, there must be some task $T_j \in S$ for which $\sigma(T_j) \leqq d - \lceil (N(i, s, d)+1)/2 \rceil = s - 1$. Since $\sigma(T_j) < s$, the definition of $S(i, s, d)$ implies that either $T_j = T_i$ or T_j is a successor of T_i. In either case we have

$$\sigma(T_i) + 1 \leqq \sigma(T_j) + 1 \leqq s = d - \lceil N(i, s, d)/2 \rceil$$

as desired. □

The significance of Lemma 1 is that, when the described conditions are met, it gives an additional constraint on the latest possible finishing time for T_i. Thus, if $N(i, s, d) \geqq 2(d-s)$ and $d - \lceil N(i, s, d)/2 \rceil < d_i$, we may set d_i equal to $d - \lceil N(i, s, d)/2 \rceil$ without foreclosing any possible valid schedules that meet all task deadlines. That is, a valid schedule meets all the original task deadlines if and only if it meets the new set of deadlines obtained by making this single change to d_i. In fact, such modifications may be performed repeatedly until either no further modifications are possible or we have some $d_i < s_i + 1$, in which case no valid schedule can possibly meet all the deadlines. In a later section we shall describe an algorithm for performing these successive modifications in an organized and efficient manner.

Motivated by the preceding discussion, we will call the deadlines *internally consistent* whenever the following conditions hold for every task $T_i \in \mathcal{T}$:

1) $d_i \geqq s_i + 1$;
2) For every pair of integers s, d satisfying $s_i \leqq s \leqq d_i \leqq d$, if $N(i, s, d) \geqq 2(d-s)$, then $d_i \leqq d - \lceil N(i, s, d)/2 \rceil$.

Two basic facts which follow from internal consistency will prove useful.

FACT 1. *If the deadlines are internally consistent, then $T_i < T_j$ implies $d_i < d_j$.*

Proof. Suppose we had both $T_i < T_j$ and $d_i \geqq d_j$. Then T_j belongs to $S(i, d_i, d_i)$ and hence $N(i, d_i, d_i) > 2(d_i - d_i) = 0$. Property 2) of internal consistency then requires that we have $d_i \leqq d_i - \lceil N(i, d_i, d_i)/2 \rceil \leqq d_i - 1$, a contradiction. □

FACT 2. *If the deadlines are internally consistent, then* $s \leqq d$ *implies*

$$|\{T_i \in \mathcal{T} : s \leqq s_i \text{ and } d_i \leqq d\}| \leqq 2(d-s).$$

Proof. Suppose for some $s \leqq d$ the set $S = \{T_i \in \mathcal{T} : s \leqq s_i \text{ and } d_i \leqq d\}$ had $|S| > 2(d-s)$. Let T_k be a task in S having the earliest start-time. Then $S - \{T_k\} \subseteq S(k, s_k, d)$ and hence $N(k, s_k, d) \geqq 2(d-s) \geqq 2(d-s_k)$. Property 2) of internal consistency then requires that

$$d_k \leqq d - \lceil N(k, s_k, d)/2 \rceil \leqq s.$$

But since $s_k \geqq s$, this implies that $s_k \geqq d_k$, contradicting property 1) of internal consistency. □

We now are prepared to state our main result.

THEOREM 1. *Let* $L = (T_1, T_2, \cdots, T_n)$ *be any priority list such that* $d_i \leqq d_{i+1}$ *for* $1 \leqq i \leqq n-1$. *If the deadlines are internally consistent, then the valid schedule defined by* L *meets all task deadlines.*

Proof. Suppose that the valid schedule σ constructed from L fails to meet the task deadlines. Since each $t_i = 1$ and all start-times and deadlines are integers, σ assigns an integer starting time to each task. Let T_j be a task with minimum $\sigma(T_j)$ which fails to meet its deadline, i.e., $\sigma(T_j)+1 > d_j$ and hence by integrality $\sigma(T_j) \geqq d_j$. Let s be the greatest integer time, $0 \leqq s \leqq \sigma(T_j)$, for which the set $P(s) = \{T_i \in \mathcal{T} : \sigma(T_i) = s-1 \text{ and } d_i \leqq d_j\}$ satisfies $|P(s)| < 2$. Defining $S = \{T_i \in \mathcal{T} : s \leqq \sigma(T_i) < \sigma(T_j)\} \cup \{T_j\}$, we observe that $|S| = 2(\sigma(T_j) - s) + 1$ and each $T_i \in S$ has $d_i \leqq d_j$. We divide the proof into two cases depending on whether $|P(s)| = 0$ or $|P(s)| = 1$.

First, suppose $|P(s)| = 0$. By the definition of S, any tasks scheduled to begin execution at time $s-1$ must have followed all the tasks in S on the priority list L. Thus none of the tasks in S were ready to begin execution at time $s-1$, and any tasks that *were* executed at that time could not have been predecessors of any task in S (by Fact 1 about internally consistent deadlines). Therefore every task in S must have start-time exceeding $s-1$ and hence, by integrality, has start-time s or greater. (All these observations hold trivially if $s = 0$). This implies that

$$S \subseteq \{T_i \in \mathcal{T} : s \leqq s_i \text{ and } d_i \leqq d_j\},$$

from which it follows that

$$\begin{aligned} |\{T_i \in \mathcal{T} : s \leqq s_i \text{ and } d_i \leqq d_j\}| &\geqq |S| \\ &= 2(\sigma(T_j) - s) + 1 \geqq 2(d_j - s) + 1 \\ &> 2(d_j - s). \end{aligned}$$

This last inequality contradicts Fact 2, proving the desired result when $|P(s)| = 0$.

Now suppose $|P(s)| = 1$. Let T_k be the single task in $P(s)$. Again, any other task scheduled to begin execution at time $s-1$ must have followed all tasks in S on the priority list L, and hence was not a predecessor of any task in S, while no task in S was ready to begin executing at time $s-1$. Therefore every task in S either has start-time at least s or is a successor of T_k. This implies that $S \subseteq S(k, s, d_j)$, from

which it follows that

$$N(k, s, d_j) \geqq |S| = 2(\sigma(T_j) - s) + 1$$
$$\geqq 2(d_j - s) + 1 > 2(d_j - s).$$

From property (2) of internal consistency we then must have

$$d_k \leqq d_j - \lceil N(k, s, d_j)/2 \rceil \leqq d_j - (d_j - s + 1) = s - 1.$$

Therefore T_k failed to meet its deadline, contradicting the choice of T_j as the earliest such task. This proves the desired result when $|P(s)| = 1$, completing the proof. □

We now summarize our basic algorithm, which determines whether a valid schedule meeting all deadlines exists and, if so, constructs one.

Step 1. Successively modify the deadlines using Lemma 1 until either (a) the deadlines are internally consistent or (b) some $s_i \geqq d_i$. In case (b), report that no schedule exists and halt.

Step 2. Form the priority list L by sorting the tasks in order of nondecreasing modified deadlines.

Step 3. Compute the valid schedule defined by priority list L. By Theorem 1 it meets all the task deadlines. □

By our previous comments on the complexity of Step 3, and the fact that all Step 2 involves is a simple sorting process, we see that the number of operations required by this algorithm is $O(n^2)$ plus the number of operations required for Step 1. In the next section we describe a method for performing Step 1 using $O(n^3)$ operations.

3. A deadline modification algorithm. In this section we describe an algorithm for successively modifying the task deadlines using Lemma 1. A straightforward approach to doing this would repeatedly examine $N(i, s, d)$ for all appropriate values of i, s, and d, modifying deadlines when required, until either the deadlines are internally consistent or some task deadline no longer exceeds the corresponding start-time. Though this algorithm will certainly terminate with the correct result, its computation time is potentially rather large. One reason for this is that the same $N(i, s, d)$ may have to be considered many times, since modifying task deadlines can change previously examined $N(i, s, d)$ values. A second reason is that the number of integer values for s and d which must be considered may be quite large. We now describe how a more careful approach avoids these difficulties.

Our algorithm is structured as three nested loops, each selecting successive values of one of the three parameters i, s, and d. The outer loop selects values of d in decreasing order. For each d, the next loop selects values of i in increasing order, skipping those values of i for which $d_i > d$. Finally, for fixed i and d, the inner loop selects appropriate values of s in increasing order and modifies d_i if required by the value of $N(i, s, d)$.

We first observe that this loop structure for choosing i, s, and d allows each triple to be considered only once. This is, of course, desirable, but can be justified only if it insures that no possible deadline modification is missed. To see that this is the case, we first observe that the value of $N(i, s, d)$ and the new deadline that may

be imposed on task T_i if $N(i, s, d) \geqq 2(d-s)$ do not depend on the value of d_i itself. Thus as long as the value of $N(i, s, d)$ remains unchanged, extra considerations of the triple (i, s, d) cannot lead to modifications that were not made the first time that triple was considered. Next, we observe that the value of $N(i, s, d)$ changes only when a task T_j with deadline $d_j > d$ has its deadline modified to be less than or equal to d. However, d_j can be so modified only when examining $N(j, s', d')$ for some $d' \geqq d_j > d$. Considering values of d in decreasing order insures that all such modifications affecting the value of $N(i, s, d)$ have been made before $N(i, s, d)$ is examined. Thus we never need to consider a triple (i, s, d) more than once when using our loop structure.

Our next step is to bound the *number* of triples considered. If our algorithm is to be $O(n^3)$, we clearly can consider only $O(n^3)$ such triples, but at present we only have a bound of (number of d's considered) · (number of i's considered for each d) · (number of s's considered for each i and d). The middle factor is at most n, since there are only n tasks, but the remaining factors might be considerably larger, since individual tasks might originally have start-times and/or deadlines which are much larger than n.

We first examine the parameter s, and show that, for fixed i and d, the inner loop need never consider more than $n+1$ possible values for s. In particular, we show that the only values for s that need to be considered are those integers s, $s_i \leqq s \leqq d_i$, which are task start-times or d_i itself. Suppose there is some s not of this form for which $N(i, s, d)$ requires that d_i be modified. Letting s' be the minimum of d_i and the least start-time exceeding s, we have $N(i, s', d) = N(i, s, d) \geqq 2(d-s) > 2(d-s')$. Thus the same modification forced by $N(i, s, d)$ will be forced by $N(i, s', d)$ and s' belongs to our more restricted set of choices. Furthermore, once d_i has been modified, no more choices for s need be considered for these fixed values of i and d. To see this, let s^* denote the least value of s such that $N(i, s^*, d)$ forces d_i to be modified. Since the modified value of d_i must be less than or equal to s^* and since values for s were selected in increasing order, it follows that all values for s which remain relevant (s such that $s_i \leqq s \leqq s^*$) have already been considered. Thus, for fixed i and d, at most $n+1$ values for s need be considered and (a remark for future reference) at most one deadline modification occurs.

To show that at most n values of d need be considered in the outer loop involves a more complicated argument. The basic idea is that essentially every value of d considered will be guaranteed to be a final deadline for some one of the n tasks. In order to insure this, however, we need to be a bit more careful in the algorithm than we have indicated so far.

First, it will be useful to have the property that whenever T_j is a predecessor of T_i, $d_j \leqq d_i$. Note that reducing d_j so that this is the case will never preclude any possible valid schedules that meet the original deadlines. (In fact we could even reduce d_j to $d_i - 1$, but the weaker condition is sufficient for our purposes and is easier to maintain.) Some preprocessing is required to make this property hold initially; we postpone a detailed explanation of how this is done. Then, whenever a deadline d_i is modified in the algorithm, we merely need to examine each predecessor T_j of T_i and set d_j equal to the smaller of its current value and the new

value for d_i. This ensures that the desired property will hold throughout execution of the algorithm. (Note also that these extra modifications do not affect the property that permitted us to consider each $N(i, s, d)$ at most once, namely, that no deadline is ever modified when considering a value of d less than that deadline.)

Having the abovementioned property, we always select the next value of d to be the largest current task deadline which is less than all previously selected values of d. This immediately insures that no required modifications will be overlooked, since if d' is any integer between d and the previous value for d, we have $N(i, s, d) = N(i, s, d')$ for all appropriate i and s. Now suppose that, after considering all values of i and s for some d, no task remains with deadline d. We show that this implies no valid schedule can possibly meet all the deadlines. Let j be the largest task index such that, when d was selected, T_j had deadline d and had no successors with deadline d (at least one such task must exist). By our property for deadlines, T_j also has no successor with deadline less than d. Thus $S(j, s, d)$ contains no successors of T_j, $s_j \leqq s \leqq d$. Since d_j was modified, there must have been some s such that $N(j, s, d) \geqq 2(d-s)$. By our choice of T_j and the above remark, when $N(j, s, d)$ was examined, every task in $S(j, s, d)$ had start-time s or larger and either had deadline less than or equal to $d-1$ or else was a predecessor of T_j with deadline d. Thus all tasks in $S(j, s, d)$ must in fact be executed in the time interval $[s, d-1]$ in any valid schedule that meets all the deadlines. However, by choice of s, we have $N(j, s, d) \geqq 2(d-s) > 2(d-1-s)$, and so $S(j, s, d)$ contains more tasks than can *possibly* be executed in the time interval $[s, d-1]$. Hence no valid schedule can meet all the deadlines. Therefore, if we finish processing a deadline d and have no task left with that deadline, we know that no such valid schedule can exist. Hence, if we terminate whenever this situation arises, we will not be overlooking any modification which can lead to a set of internally consistent deadlines. Moreover, by terminating in this fashion, we insure that at most n values of d need be considered.

The above arguments show that at most $O(n^3)$ triples (i, s, d) need be considered. It remains to be shown that the various costs associated with preprocessing and with updating the values of i, s, d and $N(i, s, d)$ are also $O(n^3)$. In order to do this, we must provide more specific details on how the algorithm is to be implemented.

First we describe and discuss the preprocessing that must be done. The first step is to sort and re-index the tasks so that $s_1 \leqq s_2 \leqq \cdots \leqq s_n$. This takes time $O(n \cdot \log n)$. Next we compute the transitive closure of the partial order so that, in constant time, we can determine whether or not T_i precedes T_j, $1 \leqq i, j \leqq n$. This can be accomplished with $O(n^{2.81})$ operations using [5] or $O(n^3)$ operations using any of [2], [14], [16]. Then we perform the preliminary deadline modifications to ensure that $d_i \leqq d_j$ whenever T_i precedes T_j. This can be done in $O(n^2)$ operations by working "backwards" in the partial order, examining a task T_i only after examining all its successors and then setting d_i to the minimum value in $\{d_i\} \cup \{d_j : T_j \text{ is a successor of } T_i\}$. Finally we initialize the variable d to a value that exceeds the largest task deadline. The algorithm then proceeds as follows:

Step 1. If any task T_i has $d_i \leqq s_i$, halt (no schedule is possible). If no task has a deadline less than d, halt (the current deadlines are internally consistent). Other-

wise set d to the largest task deadline less than d and set i to the least task index for which d_i is less than or equal to the new value of d.

Step 2. Scan the task list to compute $N(i, s_i, d)$. Set COUNT $\leftarrow N(i, s_i, d)$, $s \leftarrow s_i$, and set $k \leftarrow$ least j such that $s_j = s_i$.

Step 3. If COUNT $\geqq 2(d-s)$ and $d_i > d - \lceil \text{COUNT}/2 \rceil$, set $d_i \leftarrow d - \lceil \text{COUNT}/2 \rceil$ and, for each predecessor T_j of T_i whose deadline exceeds the new d_i, set $d_j \leftarrow d_i$.

Step 4. If $s \geqq d_i$, go to Step 5. Otherwise increment k by 1 until either $k > n$ or $s_k > s$. During this scan, subtract 1 from COUNT for each T_j (original $k \leqq j <$ new k) which is not a successor of T_i and which satisfies $d_j \leqq d$, $s_j = s$, and $j \neq i$. If $k = n+1$ or $s_k > d_i$, set $s \leftarrow d_i$ and go to Step 3. Otherwise set $s \leftarrow s_k$ and go to Step 3.

Step 5. Find the least $j > i$ such that $d_j \leqq d$. If such a j exists, set $i \leftarrow j$ and go to Step 2. If no such j exists and some task has current deadline d, go to Step 1. Otherwise halt (no schedule is possible).

Step 1 checks two termination conditions, both of which can be verified in $O(n)$ operations by a simple scan through all the tasks. If both conditions fail, it then selects the next value for d and the first relevant value of i for d, again accomplished easily by simple scans in $O(n)$ operations. Since, by our previous comments, at most n values of d will be selected, this step will be entered at most $n+1$ times, for a total contribution of at most $O(n^2)$ operations.

Step 2 computes $N(i, s_i, d)$, stores it in COUNT, initializes s to s_i, and initializes the variable k which will be used in updating COUNT as s changes. All of these can again be accomplished by simply scanning through all n tasks, with a constant number of operations for each task (for instance, to determine whether it meets the membership conditions for $S(i, s_i, d)$). Thus each entry of this step uses $O(n)$ operations. Because it is entered at most n times for each d, always with a new value for i, the total contribution to the algorithm is at most $O(n^3)$ operations.

Step 3 checks the internal consistency conditions for $N(i, s, d) = \text{COUNT}$ and, if necessary, modifies the appropriate deadlines. This step is entered at most $O(n^3)$ times, once for each choice of the three parameters i, s, and d. It requires only constant time unless deadline modification is necessary, in which case $O(n)$ operations may be required. However, since d_i is modified at most once for a particular value of d, those $O(n)$ modification operations are required at most n^2 times. Thus this step contributes a total of at most $O(n^3)$ operations.

Step 4 first checks whether all relevant values of s have been considered for the current i (i.e., either d_i has been modified or the last s was equal to d_i). If not, it continues scanning the task list from T_k to find the next relevant s. During this scan it continually updates COUNT, subtracting 1 for each task which belonged to the previous $S(i, s, d)$ but not the new $S(i, s, d)$. After resetting s, it returns to Step 3. This step is entered at most $O(n^3)$ times since it is entered only through Step 3. However, by using the variable k to resume scanning the task list from where it left off, it only scans the task list once for each choice of i and d. It follows that Step 4 contributes a total of at most $O(n^3)$ operations to the algorithm.

Finally, Step 5 selects the next relevant value of i, if any, and returns to Step 2. If all values of i for this d have been checked and some task deadline remained equal to d, it goes to Step 1 to determine the next d. Otherwise, by our previous

discussion, we know that no valid schedule can possibly meet all the deadlines and the algorithm halts. Since Step 5 is entered at most once for each choice of i and d, it is entered a total of at most $O(n^2)$ times. The scanning of the tasks is a process which occurs only once for each d, although it is interrupted each time i is updated. Thus this step contributes a total of at most $O(n^2)$ operations.

From our discussion preceding the algorithm, the reader should have little difficulty in verifying that the algorithm works properly. It terminates either with a set of internally consistent deadlines equivalent to the original deadlines, or with the conclusion that no valid schedule can meet all the deadlines. Furthermore, since each step contributes at most $O(n^3)$ operations to the total procedure, and all required preprocessing can be done in at most $O(n^3)$ operations, the algorithm takes total time $O(n^3)$ as claimed. Although we know of no method for doing this faster than proportional to n^3, we note that the algorithm as described could probably be improved by a constant factor in a careful implementation. We chose not to incorporate such details into the description of our algorithm since that would have served primarily to further complicate an already complicated algorithm, without substantially improving its performance.

4. Conclusion. The algorithm described in §§ 2 and 3 is designed only to test for feasible schedules and to generate such a schedule whenever one exists. Given that feasible schedules exist, however, one might wish to find such schedules that, for example, minimize maximum finishing time. This can be done using our algorithm in a simple iterative procedure as follows.

First observe that, for any integer D, we can decide whether there exists a valid schedule that meets all deadlines and has maximum finishing time at most D, by applying our algorithm after setting equal to D all deadlines that exceed D. The least possible maximum finishing time can then be found using a binary search on D. Since the only values of D that need be considered are those integers that exceed the largest start-time by no more than n, this involves only $O(\log n)$ applications of our basic algorithm, and the complete procedure requires $O(n^3 \log n)$ operations. In case all start-times or all deadlines are the same, the simpler procedure described in [9] can be used.

A similar approach, using binary search, can be used to find a valid schedule that minimizes maximum tardiness (the *tardiness* of a task in a schedule is the maximum of zero and its finishing time minus its deadline), in case no valid schedule meeting all the deadlines is possible. To check whether a valid schedule exists with maximum tardiness D or less, merely replace each deadline d_i by $D + d_i$ and apply our basic algorithm.

Unfortunately, the problem of minimizing the *number* of tardy tasks, even if we have only one processor and all start-times (or all deadlines) are the same, is *NP*-complete [9] and hence probably computationally intractable (see [1], [11], [12] for comprehensive treatments of "*NP*-completeness"). A number of other simple generalizations of our scheduling problem are also *NP*-complete.

First let us consider relaxing the constraint that all $\tau_i = 1$ by allowing $\tau_i \in \{1, 2\}$. In this case, with all $s_i = 0$ and all $d_i = D$, Ullman [15] has shown that the problem of deciding whether there exists a valid schedule meeting all deadlines is

NP-complete. If we further relax the constraint on task times to allow them to be arbitrary integers, then the problem of deciding whether there exists a valid schedule meeting all deadlines is *NP*-complete even for one processor and no precedence constraints. This result has not appeared previously but can be proved easily from the *NP*-complete 3-PARTITION problem [8], [10]. The 3-PARTITION problem is, given $3n$ integers $a_1, a_2, \cdots, a_{3n}$ and an integer B such that each a_i satisfies $B/4 < a_i < B/2$, to determine whether the $\{a_i\}$ can be partitioned into n 3-element sets which each sum exactly to B. It follows, as in [10], that this scheduling problem is *NP*-complete even if input size is measured by the sum of the task execution times, rather than the number of bits required to encode them. This means that any algorithm that always finds the desired schedule, if it exists, will probably require time exponential in the sum of execution times rather than just exponential in the number of tasks, a significant difference when tasks with large execution times are present.

Returning now to our original problem, suppose we relax the constraint that all start-times must be integers. (Allowing arbitrary rational start-times is equivalent to allowing all tasks to have arbitrary, but identical, integer execution times). In this case little is yet known. In fact, the complexity of determining the existence of valid schedules meeting all deadlines for the case of one processor, unit execution times, no precedence constraints, and arbitrary start-times and deadlines is still open. (In the very special case with all s_i multiples of $\frac{1}{2}$, we *can* give a polynomial-time algorithm).

Returning again to our original problem with integer start-times (and hence integer deadlines), consider the effect of increasing the number of processors. If the number of processors is arbitrary, the problem is *NP*-complete even with all $s_i = 0$ and all $d_i = D$ [15]. It is not known whether this problem is *NP*-complete for any *fixed* number m of processors, although $m = 2$ is the largest value for which a polynomial time algorithm is known. Our algorithm *can* be used to solve a significant special case of the 3-processor problem (with all $s_i = 0$ and all $d_i = D$) in which we ask whether there exists a valid schedule with maximum finishing time D, where D is the length of the longest chain ("critical path") in the partial order. Choosing one such chain, there is no loss of generality in assigning all tasks in the chain, in order, to the third processor. Then all remaining tasks must be executed on the remaining two processors, with the precedence constraints between these tasks and the tasks on the third processor serving merely to impose individual integer start-times and deadlines on the remaining tasks. Hence we are reduced to precisely the problem our algorithm was designed to solve. The solvability of this special case is particularly interesting because the authors (and others) have made numerous attempts to prove the general 3-processor problem *NP*-complete by proving that this special case was *NP*-complete. If, as is generally believed, the *NP*-complete problems are intractable, this particular approach was doomed to fail.

REFERENCES

[1] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974, Chap. 10.

[2] V. L. ARLAZAROV, E. A. DINIC, M. A. KRONOD AND I. A. FARADZEV, *On economical construction of the transitive closure of an oriented graph*, Dokl. Akad. Nauk SSSR, 11 (1970), pp. 1209–1210.
[3] E. G. COFFMAN AND R. L. GRAHAM, *Optimal scheduling for two-processor systems*, Acta. Informatica, 1 (1972), pp. 200–213.
[4] R. W. CONWAY, W. L. MAXWELL AND L. W. MILLER, *Theory of Scheduling*, Addison-Wesley, Reading, MA, 1967.
[5] M. J. FISCHER AND A. R. MEYER, *Boolean matrix multiplication and transitive closure*, 12th Ann. IEEE Symp. on Switching and Automata Theory, East Lansing, MI, 1971, pp. 129–131.
[6] M. FUJII, T. KASAMI AND K. NINOMIYA, *Optimal sequencing on two equivalent processors*, SIAM J. Appl. Math., 17 (1969), pp. 784–789.
[7] ———, *Erratum*, Ibid., 20 (1971), p. 141.
[8] M. R. GAREY AND D. S. JOHNSON, *Complexity results for multiprocessor scheduling under resource constraints*, this Journal, 4 (1975), pp. 397–411.
[9] ———, *Scheduling tasks with nonuniform deadlines on two processors*, J. Assoc. Comput. Mach., 23 (1976), pp. 461–467.
[10] M. R. GAREY, D. S. JOHNSON AND R. SETHI, *Complexity of flowshop and jobshop scheduling*, Math. Operations Res., 1 (1976), pp. 117–129.
[11] R. M. KARP, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. M. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–104.
[12] ———, *On the complexity of combinatorial problems*, Networks, 5 (1975), pp. 45–68.
[13] Y. MURAOKA, *Parallelism, exposure and exploitation in programs*, Ph.D. thesis, Computer Sci. Dept., Univ. of Illinois, 1971.
[14] P. PURDOM, *A transitive closure algorithm*, BIT, 10 (1970), pp. 76–94.
[15] J. D. ULLMAN, *NP-complete scheduling problems*, J. Comput. System Sci., 10 (1975), pp. 384–393.
[16] S. WARSHALL, *A theorem on Boolean matrices*, J. Assoc. Comput. Mach., 9 (1962), pp. 11–12.