SIAM J. COMPUT.
Vol. 10, No. 2, May 1981

0097-5397/81/1002-0003 $01.00/0

SCHEDULING UNIT-TIME TASKS WITH ARBITRARY RELEASE TIMES AND DEADLINES*

M. R. GAREY†, D. S. JOHNSON†, B. B. SIMONS‡ AND R. E. TARJAN§

Abstract. The basic problem considered is that of scheduling n unit-time tasks, with arbitrary release times and deadlines, so as to minimize the maximum task completion time. Previous work has shown that this problem can be solved rather easily when all release times are integers. We are concerned with the general case in which noninteger release times are allowed, a generalization that considerably increases the difficulty of the problem even for only a single processor. Our results are for the one-processor case, where we provide an $O(n \log n)$ algorithm based on the concept of "forbidden regions".

Key words. scheduling, release time, deadline, computational complexity

1. Introduction.

The scheduling problems we will be considering in this paper are all special cases of the following general scheduling problem. We are given n tasks, $T_1, T_2, \cdots, T_n$, each requiring one unit of execution time. Each task T_i has associated with it an arbitrary release time $r_i \geqq 0$ and a deadline $d_i \geqq r_i + 1$. In addition, there may be a partial order $<$ imposed on the tasks. We wish to schedule the given tasks nonpreemptively on m identical processors so that

(i) Each task T_i is started no earlier than its release time r_i and is completed no later than its deadline d_i.

(ii) Whenever $T_i < T_j$, T_j does not start before T_i has been completed.

(iii) The maximum completion time (or *makespan*) is minimized.

Previous results on related problems include the following. If the tasks are allowed to have unequal lengths, a simple transformation from the 3-PARTITION problem [4] shows that the problem is NP-complete in the strong sense [4], even for one processor and integer release times and deadlines. If the partial order is allowed to be arbitrary, then the problem with unit-time tasks and a variable number of processors is NP-complete [10], even if all release times are 0 and there is only a single overall task deadline. On the other hand, good algorithms are known for the following special cases: If no partial order is imposed and the release times are all integers, then the "earliest deadline scheduling rule" [5], [7] can be used to solve the problem in O($n \log n$) time for any number of processors. Indeed, this method can be used for one processor even with an arbitrary partial order, since (as we observe in § 2) the presence of a partial order is essentially irrelevant to the one-processor case. For two processors, integer release times and an arbitrary partial order, an $O(n^3 \log n)$ algorithm is given in [3].

As observed in [3], these problems seem to be considerably more difficult when the release times are not required to be integers (i.e., are not multiples of the common task length), even when there is only a single processor. In this paper we shall consider the version in which *arbitrary* release times are allowed, concentrating on the one-processor case. The first polynomial time algorithm for this problem was obtained recently by Simons [8], and it has time complexity $O(n^2 \log n)$. An alternative

* Received by the editors July 19, 1979, and in final form May 5, 1980.

† Bell Laboratories, Murray Hill, New Jersey 07974.

‡ Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, California; now at IBM, San Jose, California. Much of the work of this author was done while she was a graduate student at University of California, Berkeley, and partially supported by the National Science Foundation under grant MCS 77-09906.

§ Computer Science Department, Stanford University, Stanford, California 94305. The work of this author was supported in part by the National Science Foundation under grant MCS-7826858 and by the U.S. Office of Naval Research under contract N00014-76-C-0330. Some of this research was done while the author was visiting Bell Laboratories.

algorithm with the same time complexity has since been obtained by Carlier [2]. Using the new concept of "forbidden regions", we shall describe an algorithm which, when suitably modified to use appropriate data structures, runs in time $O(n \log n)$ and space $O(n)$.

The paper is divided into five sections. In § 2 we make some simple preliminary observations, including a "normalization" lemma and a lemma showing that partial orders are essentially irrelevant when there is only one processor. § 3 gives an $O(n^2)$ algorithm for the one processor problem based on "forbidden regions", and § 4 improves the algorithm to $O(n \log n)$ through the careful use of appropriate data structures and several new ideas. Finally, § 5 concludes the paper by mentioning several problems that remain open, particularly with regard to the multi-processor case.

2. Preliminary observations.

We shall represent a schedule (or a partial schedule) by giving a starting time s_i for each task T_i. Sometimes we will use $f_i = s_i + 1$ to denote the finishing time for T_i. A schedule is *feasible* if it satisfies the release times and deadlines (i.e., $r_i \leqq s_i \leqq d_i - 1$ for all i), obeys the partial order constraints (i.e., $T_i < T_j$ implies $f_i \leqq s_j$), and executes at most m tasks at a time (i.e., for any time t, there are at most m tasks T_i for which t belongs to the execution interval $[s_i, f_i)$).

A task T_i is *ready* at time t if $r_i \leqq t$. We shall say that a schedule is *normal* if, for any two tasks T_i and T_j, $s_i < s_j$ implies that either $d_i \leqq d_j$ or $r_j > s_i$. In other words, a normal schedule has the property that, whenever one or more tasks begin execution at some time t, those tasks have the earliest deadlines among all remaining tasks that are ready at t. The following lemma can be proved by straightforward interchange arguments.

LEMMA 1. *For any $m \geqq 1$, if there are no partial order constraints, then the existence of a feasible m-processor schedule implies the existence of a schedule that both minimizes maximum completion time and is normal.*

Lemma 1 tells us that in the absence of a partial order we can restrict our attention to normal schedules. The next lemma will show how, for $m = 1$, we can restrict ourselves to normal schedules even in the presence of a partial order.

Given a partial order $<$ on the tasks, we say the release times and deadlines are *consistent* with the partial order if $T_i < T_j$ implies $r_i + 1 \leqq r_j$ and $d_i \leqq d_j - 1$. We can make release times and deadlines consistent with the partial order by processing the tasks once in topological order [6] assigning $r_j \leftarrow \max(\{r_j\} \cup \{r_i + 1: T_i < T_j\})$ and once in reverse topological order assigning $d_i \leftarrow \min(\{d_i\} \cup \{d_j - 1: T_i < T_j\})$. This requires time linear in the size of the partial order and does not alter the feasibility of any schedule. Furthermore, in the one-processor case it allows us subsequently to ignore the partial order constraints. □

LEMMA 2. *If the release times and deadlines are consistent with a partial order, then any normal one-processor schedule that satisfies the release times and deadlines must also obey the partial order.*

Proof. Consider any normal one-processor schedule, and suppose that $T_i < T_j$ but that $s_j < f_i$ (which, since there is only one processor, implies $s_j < s_i$). By the consistency assumption we have $r_i < r_j$ and $d_i < d_j$. However, these, together with $s_j < f_i$, cause a violation of the assumption that the schedule is normal, a contradiction from which the result follows. □

Lemma 2 means that a partial order is essentially irrelevant when scheduling on one processor. Henceforth we shall assume that no partial order is imposed, and we will consider only normal schedules.

3. One-processor scheduling using forbidden regions.

Our one-processor scheduling algorithms depend upon discovering "forbidden regions". A *forbidden region* is an interval of time (open both on the left and right) during which *no* task can start if the schedule is to be feasible.

The following algorithm forms a basic building block of our main algorithm. Suppose we are given k unit-time tasks, all of which must be scheduled to finish by some time d, and a finite collection of forbidden regions F_j. Ignoring the individual release times and deadlines of the tasks, we would like to find the latest time by which the first such task must start if all of them are to be completed by time d (without starting any of them in a forbidden region).

We do this using the following naive algorithm: Order the tasks arbitrarily as $T_1, T_2, \cdots, T_k$ and schedule them from the back of the schedule in order of *decreasing* index. When scheduling task T_i, start it at the latest time less than or equal to $s_{i+1}-1$ (or $d-1$, if $i=k$) which does not fall in a forbidden region. We call this the *Backscheduling Algorithm.*

LEMMA 3. *The starting times s_1 found for T_1 by the Backscheduling Algorithm is such that, if all the given tasks were to start at times strictly greater than s_1, with none of them starting in one of the given forbidden regions, then at least one of them would not be completed by time d.*

Proof. Consider the schedule found by the Backscheduling Algorithm. Let $h_0 = s_1$, let $h_1, h_2, \cdots, h_j$ be the starting times of the idle periods (if any) in the schedule and let $h_{j+1} = d$. See Fig. 1.

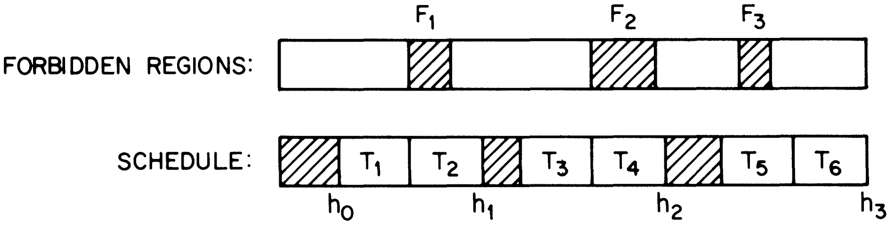


FIG. 1. *Scheduling by the Backscheduling Algorithm to avoid forbidden regions.*

Notice that whenever (t_1, t_2) is an idle period, it must be the case that $(t_1-1, t_2-1]$ is part of some forbidden region, for otherwise the Backscheduling Algorithm would have scheduled some task to overlap or finish during $(t_1, t_2]$. Now consider any interval $(h_i, h_{i+1}]$, $0 \leqq i \leqq j$. By the preceding observation, no task can possibly be scheduled to finish after h_i but before (or at) the starting time of the first task in the interval. Furthermore, by definition of the $\{h_i\}$, the tasks that are finished in the interval are scheduled with no idle periods separating them and with the rightmost one finishing at time h_{i+1}. It follows that the Backscheduling Algorithm finishes the *maximum* possible number of tasks in each interval $(h_i, h_{i+1}]$. Since there is no idle time in the schedule during $[h_0, h_1]$, any other schedule that started all the tasks later than time s_1 and finished them all by time d would have to exceed this maximum number of tasks in some interval $(h_i, h_{i+1}]$, $1 \leqq i \leqq j$, a contradiction. □

We shall use the Backscheduling Algorithm as follows. Consider any task ready time r_i and any task deadline $d_j \geqq d_i$. Suppose that we have already found a collection of forbidden regions in the interval $[r_i, d_j]$ and that we then apply the Backscheduling Algorithm, with $d = d_j$ and with the forbidden regions we have already found, to the set of all tasks T_k satisfying $r_i \leqq r_k \leqq d_k \leqq d_j$. Let s be the latest possible start time found by the Backscheduling Algorithm in this case. There are two possibilities which are of interest. First, if $s < r_i$, then we know that there can be *no* feasible schedule, since all these tasks must be completed by time d, none of them can be started before r_i, but at least one *must* be started by time $s < r_i$ if all are to be completed by d. Second, if $r_i \leqq s < r_i + 1$, then we know that $(s-1, r_i)$ can be declared to be a forbidden region, since any task started in that region would not belong to our set (its release time is less than r_i) and it would force the first task of our set to be started later than s, thus preventing these tasks from being completed by d.

Our first algorithm for the one-processor problem essentially applies the Backscheduling Algorithm to *all* such pairs of release times and deadlines, in such a manner as to find forbidden regions from right to left. We do this by processing the release times in order from largest to smallest. To process a release time r_i, we determine for each deadline $d_j \geqq d_i$ the number of tasks which cannot start before r_i and which must be completed by d_j. We then use the Backscheduling Algorithm with $d = d_j$ to determine the latest time at which the earliest such task can start. This time is called the *critical time* c_j for deadline d_j (with respect to r_i). Letting c denote the minimum of all these critical times with respect to r_i, we then declare failure if $c < r_i$ or declare $(c-1, r_i)$ to be a forbidden region if $r_i \leqq c < r_i + 1$. Notice that by processing release times from largest to smallest, all forbidden regions to the right of r_i will have been found by the time that r_i is processed. In order to make the entire process more efficient, we do not completely recompute the critical time for a deadline when a new release time is processed, but instead we update the old value.

Once we have found forbidden regions in this way, we schedule the full set of tasks forward from time 0 using the "earliest deadline scheduling rule". This proceeds by initially setting t to the least nonnegative time not in a forbidden region and then assigning start time t to a task with lowest deadline among those ready at t. At each subsequent step, we first update t to the least time which is greater than or equal to the finishing time of the last scheduled task, greater than or equal to the earliest ready time of an unscheduled task, and which does not fall in a forbidden region, and we then assign start time t to a task with lowest deadline among those ready (but not previously scheduled) at t. The entire algorithm is specified below.

ALGORITHM A. Index the tasks (arbitrarily) so that $r_1 \leqq r_2 \leqq \cdots \leqq r_n$.

Part I. (Forbidden Region Declaration). Initially no forbidden regions have been declared. For each task T_i, in order of decreasing index, perform the following two steps:

```
Step 1. For each task $T_j$ with $d_j \geqq d_i$, update its critical time $c_j$ as follows:
    1a. If $c_j$ is undefined, set $c_j \leftarrow d_j - 1$; otherwise set $c_j \leftarrow c_j - 1$.
    1b. While $c_j \in F$ for some declared forbidden region $F$, set $c_j \leftarrow \inf(F)$.
Step 2. If $i = 1$ or $r_{i-1} < r_i$, set $c \leftarrow \min\{c_j : c_j \text{ is defined}\}$ and proceed as follows:
    2a. If $c < r_i$, declare failure and halt.
    2b. If $r_i \leqq c < r_i + 1$, declare $(c-1, r_i)$ to be a forbidden region.
```

Part II. (Schedule Generation). Initially no tasks are scheduled and $t = 0$. Repeat the following three steps until all tasks have been scheduled.

```
Step 1. If no unscheduled task is ready at time $t$, set $t \leftarrow \min\{r_i : T_i \text{ has not yet been scheduled}\}$.
Step 2. While $t \in F$ for some forbidden region $F$, set $t \leftarrow \sup(F)$.
Step 3. Select an unscheduled task $T_j$ that has the least deadline among all such tasks that are ready at $t$. Set $s_j \leftarrow t$ and set $t \leftarrow t + 1$.
```

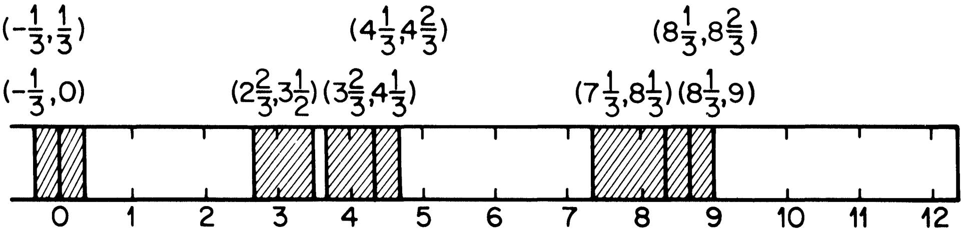
Fig. 2 illustrates the application of this algorithm to the tasks described by Table 1.

THEOREM 1. *In any feasible schedule, no task starts at a time that Algorithm* A *declares to be forbidden. If Algorithm* A *declares failure, then there is no feasible schedule.*

RELEASE TIMES

DEADLINES	9	$8\frac{2}{3}$	$8\frac{1}{3}$	5	$4\frac{2}{3}$	$4\frac{1}{3}$	$3\frac{1}{2}$	$1\frac{2}{3}$	$\frac{2}{3}$	$\frac{1}{3}$	0
$5\frac{2}{3}$									$4\frac{2}{3}$	$4\frac{2}{3}$	$4\frac{2}{3}$
6								5	$3\frac{2}{3}$	$3\frac{2}{3}$	$3\frac{2}{3}$
$6\frac{1}{3}$					$5\frac{1}{3}$	$5\frac{1}{3}$	$5\frac{1}{3}$	$4\frac{1}{3}$	$2\frac{2}{3}$	$2\frac{2}{3}$	$2\frac{2}{3}$
$6\frac{2}{3}$					$5\frac{2}{3}$	$4\frac{2}{3}$	$4\frac{2}{3}$	$3\frac{2}{3}$	$2\frac{2}{3}$	$2\frac{2}{3}$	$2\frac{2}{3}$
$7\frac{2}{3}$					$6\frac{2}{3}$	$5\frac{2}{3}$	$4\frac{2}{3}$	$3\frac{2}{3}$	$2\frac{2}{3}$	$2\frac{2}{3}$	$2\frac{2}{3}$
8				7	6	5	$3\frac{2}{3}$	$2\frac{2}{3}$	$1\frac{2}{3}$	$1\frac{2}{3}$	$1\frac{2}{3}$
10				9	$7\frac{1}{3}$	$6\frac{1}{3}$	$5\frac{1}{3}$	$4\frac{1}{3}$	$2\frac{2}{3}$	$1\frac{2}{3}$	$1\frac{2}{3}$
$10\frac{1}{3}$	$9\frac{1}{3}$	$9\frac{1}{3}$	$9\frac{1}{3}$	$8\frac{1}{3}$	$7\frac{1}{3}$	$6\frac{1}{3}$	$5\frac{1}{3}$	$4\frac{1}{3}$	$2\frac{2}{3}$	$1\frac{2}{3}$	$1\frac{2}{3}$
$11\frac{1}{3}$	$10\frac{1}{3}$	$9\frac{1}{3}$	$8\frac{1}{3}$	$7\frac{1}{3}$	$6\frac{1}{3}$	$5\frac{1}{3}$	$4\frac{1}{3}$	$2\frac{2}{3}$	$1\frac{2}{3}$	$\frac{2}{3}$	$\frac{2}{3}$
$12\frac{1}{3}$	$11\frac{1}{3}$	$10\frac{1}{3}$	$9\frac{1}{3}$	$8\frac{1}{3}$	$7\frac{1}{3}$	$6\frac{1}{3}$	$5\frac{1}{3}$	$4\frac{1}{3}$	$2\frac{2}{3}$	$1\frac{2}{3}$	$\frac{2}{3}$

FIG. 2a. *Table of critical times.*



$(-\frac{1}{3},\frac{1}{3})$
$(-\frac{1}{3},0)$
$(4\frac{1}{3},4\frac{2}{3})$
$(2\frac{2}{3},3\frac{1}{2})(3\frac{2}{3},4\frac{1}{3})$
$(8\frac{1}{3},8\frac{2}{3})$
$(7\frac{1}{3},8\frac{1}{3})(8\frac{1}{3},9)$
0
1
2
3
4
5
6
7
8
9
10
11
12

FIG. 2b. *Forbidden regions.*

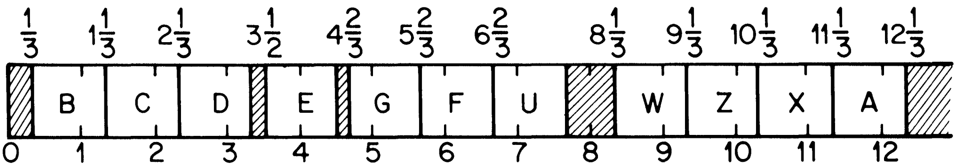


FIG. 2c. *Schedule generated for Algorithm* A *for tasks in Table 1.*

TABLE 1.
A set of tasks to schedule. Capital letters represent tasks, with release time first and deadline second.

A:	$0, 12\frac{1}{3}$	B:	$\frac{1}{3}, 10$	C:	$\frac{2}{3}, 5\frac{2}{3}$	D:	$1\frac{2}{3}, 6$
E:	$3\frac{1}{2}, 7\frac{2}{3}$	F:	$4\frac{1}{3}, 6\frac{2}{3}$	G:	$4\frac{2}{3}, 6\frac{1}{3}$	U:	$5, 8$
W:	$8\frac{1}{3}, 11\frac{1}{3}$	X:	$8\frac{2}{3}, 11\frac{1}{3}$	Z:	$9, 10\frac{1}{3}$		

Proof. The proof is by a straightforward induction on the number of forbidden regions, using Lemma 3, and is omitted. □

THEOREM 2. *If Algorithm* A *does not declare failure, then it finds a feasible schedule.*

Proof. Suppose we have a counterexample to the theorem, and let T_j be the first (earliest scheduled) task that fails to meet its deadline. Without loss of generality, we may assume that all idle times in $[0, s_j]$ belong to forbidden regions. (Otherwise, let $t' = \sup \{t : t \text{ is an idle time in } [0, s_j] \text{ that does not belong to a forbidden region}\}$, and let r denote the smallest release time among tasks started at time t' or later. Then, by the earliest deadline scheduling rule, $r \geqq t'$ and there are no nonforbidden idle times in $[r, s_j]$. Furthermore, all tasks completed before t' must have release times less than $r-1$, so they played no role in determining the forbidden regions after r. Thus we can obtain a new counterexample with the desired property by deleting all tasks completed before r and then subtracting r from the release times and deadlines of all remaining tasks.) We now consider two cases:

Case 1. Some task scheduled before T_j has a deadline later than d_j. Let T_i be such a task with maximum starting time, and let r denote the smallest release time among the tasks that start in $(s_i, s_j]$. By the earliest deadline scheduling rule, we know that r must exceed s_i. Index the tasks that start in $(s_i, s_j]$, in order of increasing starting times, as T'_1, $T'_2, \cdots, T'_k$ (note that $T'_k = T_j$), and consider the result of applying the Backscheduling Algorithm to these tasks with the given indexing and with $d = d_j$.

We claim that the Backscheduling Algorithm will assign each of these tasks a starting time that is strictly less than its starting time in the original schedule. This is clearly true for $T'_k = T_j$, since the Backscheduling Algorithm assigns it a starting time less than or equal to $d_j - 1$ (depending on whether or not $d_j - 1$ falls in a forbidden region). Inductively, suppose for some $l > 1$ that the claim holds for $T'_l, \cdots, T'_k$ and consider how the Backscheduling Algorithm schedules T'_{l-1}. If in the original schedule T'_l started immediately at the time T'_{l-1} finished, the fact that T'_l is started earlier by the Backscheduling Algorithm trivially implies that T'_{l-1} must also be started earlier. On the other hand, if in the original schedule T'_l was separated from T'_{l-1} by a block of idle time $[a, b)$, the fact that all idle times in the block belong to forbidden regions (by our choice of counterexample) implies that the Backscheduling Algorithm must have assigned T'_l a starting time strictly before a, and hence before the old finishing time of T'_{l-1}. Once again it follows that the Backscheduling Algorithm must start T'_{l-1} earlier than in the original schedule, and the claim follows by induction.

The import of the claim is that the critical time c_j (and hence the minimum critical time c) found by Algorithm A when processing release time r must have been strictly less than the starting time assigned to T'_1 by Algorithm A. Furthermore, since T'_1 started in that schedule at the first time not in a forbidden region in the interval $[f_i, s_j]$, it must be the case that $c \leqq c_j < f_i$. If $c < r$, then Algorithm A would have declared failure, a contradiction. If $c \geqq r$, then we have

$$s_i < r \leqq c < f_i = s_i + 1 < r + 1,$$

which implies that Algorithm A would have declared a forbidden region $(c-1, r)$ containing s_i, contradicting the fact that Algorithm A never starts a task in a forbidden region. It follows that Case 1 cannot occur.

Case 2. All tasks scheduled before T_j have deadlines less than or equal to d_j. Let r be the smallest release time among the tasks started in $[0, s_j]$, and let $T'_1, T'_2, \cdots, T'_k$ be the collection of all such tasks, indexed in order of increasing starting times (again note that $T'_k = T_j$). As in the previous case, if the Backscheduling Algorithm is applied to these tasks with the given indexing and with $d = d_j$, it will assign each such task a starting time strictly less than its starting time in the original schedule. Since T'_1 started in the original schedule at the earliest time not in a forbidden region, it follows that the minimum critical time c found by Algorithm A when processing release time r must be less than 0. Therefore, since $r \geqq 0$, Algorithm A would have declared failure, a contradiction to Case 2.

Since both Cases 1 and 2 result in contradictions, and since they include all possibilities, it follows that the assumed counterexample cannot exist, and Theorem 2 is proved. □

THEOREM 3. *If Algorithm* A *finds a schedule, that schedule has minimum makespan among all feasible schedules.*

Proof. Suppose we have a counterexample, and let T_i be the first (earliest scheduled) task to be completed after the minimum makespan d^*. As in the previous proof, we may assume that all idle times in $[0, s_i]$ belong to forbidden regions. Let $T'_1, T'_2, \cdots, T'_k$ denote the tasks that start in the interval $[0, s_i]$, indexed in order of increasing starting times. The same reasoning as in the previous proof shows that, if the Backscheduling Algorithm were applied to these tasks with the given indexing and with $d = d^*$, it would assign each of the tasks a starting time that is strictly less than its original starting time. Indeed, since in the original schedule T'_1 started at the *earliest* time not in a forbidden region, the Backscheduling Algorithm will assign T'_1 a starting time that is strictly less than 0. But, by Lemma 3, this says that no feasible schedule can possibly complete this many tasks by time d^*, contradicting the fact that d^* is the (achievable) minimum makespan for the assumed counterexample. The theorem follows.

Thus Algorithm A will find a feasible schedule with minimum makespan whenever there is a feasible schedule, and otherwise will correctly declare that no feasible schedule exists..

In implementing Algorithm A, we note that, although the forbidden regions found in Part I may overlap, each region has left and right endpoints no greater than the corresponding endpoints for the region declared just previous to it. Thus we can maintain the forbidden regions in a stack, combining overlapping regions as they occur. For each deadline d_j we maintain a pointer into the stack which indicates the latest forbidden region that precedes the critical time c_j. It is then easy to see that there is an overall time bound of $O(n^2)$ on each of steps 1 and 2, and hence on Part I. Part II requires at most $O(n \log n)$ additional time, for a total of $O(n^2)$ time. In the next section we shall see how to reduce this bound of $O(n^2)$ to $O(n \log n)$ by modifying the algorithm to make more sophisticated use of data structures.

4. Improving the algorithm to $O(n \log n)$.

Examination of Algorithm A reveals three places where $\Omega(n^2)$ time might be used, all of them in Part I. In the process of updating critical times c_j with respect to a new task T_i, each of steps 1a and 1b can contribute $\Omega(n)$ time, giving a total of $\Omega(n^2)$ time. Each computation of $c \leftarrow \min \{c_j : c_j$ is defined$\}$ in step 2 can also contribute $\Omega(n)$ time, again for a total of $\Omega(n^2)$ time.

The key to obtaining a speed-up from $\Omega(n^2)$ to $O(n \log n)$ involves a basic shift in the way we deal with critical times. Instead of keeping track of each c_j individually, so that the current value of any c_j can be found in constant time (the approach of Algorithm A), we shall keep track of a smaller amount of information, which will be sufficient for determining the current value of any c_j in time $O(\log n)$. This will permit us to use more efficient procedures for organizing and updating the data structures neded for computing the c_j values.

We first observe that each critical time can be decomposed into two components. After some task T_i is processed, each critical time c_j is smaller than the corresponding deadline d_j by an amount that depends on both the number of tasks seen so far that have deadlines d_j or less and the locations of previously declared forbidden regions. We can keep track of these factors through the *task load* n_j and the *offset* o_j. The task load n_j is essentially the contribution to c_j from step 1a of Algorithm A; that is, n_j is the number of tasks T_k with $k \geqq i$ for which $d_k \leqq d_j$. (Note that a critical time c_j is "defined" if and only if $n_j > 0$.) The *offset* o_j is the contribution to c_j of step 1b; that is, the total distance c_j has been moved to keep it from being in a forbidden region. Thus we can compute c_j from these two components by the formula $c_j = d_j - (n_j + o_j)$.

The task loads can be maintained easily within an overall time bound of $O(n \log n)$, primarily because whenever we add 1 to n_j we must also add 1 to *every* n_k such that $d_k \geqq d_j$. We store the task loads in a *task load tree*, which is a binary search tree [1, p. 115] having a vertex corresponding to each deadline d_j. In addition, we associate a numerical value with each vertex in such a way that the task load corresponding to any d_j is obtained by summing the values along the path from the root to the vertex for d_j (e.g., see [1, p. 141]). By the choice of an appropriate underlying data structure we can insure that no such path has length exceeding $O(\log n)$, and hence the time for determining the value of any n_j will be $O(\log n)$. Similarly, the cost of updating the tree when a new task is processed (which can involve changes to $\Omega(n)$ c_j values) will be only $O(\log n)$, for an overall updating cost of $O(n \log n)$, as claimed.

Maintaining the offsets is somewhat more complicated. In fact, we will not keep track of the offsets themselves, but rather certain related quantities which we will call *pseudo-offsets*. These are defined as follows:

Suppose that F is the set of forbidden regions declared before the start of processing for task T_i. For any deadline d_j and nonnegative integer n, let $b_j(n)$ denote the earliest starting time that would be assigned if the Backscheduling Algorithm of § 3 were applied to n tasks with deadline d_j and with the forbidden regions in F. (If $n = 0$, we let $b_j(n) = d_j$). For each n, let

$$o'_j(n) = d_j - b_j(n) - n,$$

and define the *pseudo-offset* o'_j by $o'_j = \lim_{n\to\infty} o'_j(n)$. Observe that o'_j is well defined and finite, since as soon as $b_j(n) < r_i$ we must have $o'_j(n+1) = o'_j(n)$.

Notice that, unlike the offset o_j, the pseudo-offset o'_j does not depend on the current value of the task load n_j. This is the property that will allow us to maintain the pseudo-offsets efficiently. We postpone for the moment the discussion of how pseudo-offsets can be used in place of offsets for finding the same forbidden regions. Instead we first fill in the details of how the pseudo-offsets can be maintained (first-time readers may wish to skip over these details for now).

Pseudo-offsets are all initially 0 and change only when a new forbidden region (a, b) is declared. In this case the pseudo-offset for a given deadline $d_j > b$ changes if and only if there exists a nonnegative integer n such that $d_j - o'_j - n \in (a, b)$. If this occurs, the increase in the pseudo-offset is exactly $d_j - o'_j - n - a$. Thus the change depends only on the *fractional offset* $q_j = (d_j - o'_j)$ [mod 1], i.e., the fractional part of $d_j - o'_j$.

To keep track of the pseudo-offsets, we use two data structures: a *fractional offset tree* and a *pseudo-offset forest*. The former is just a binary search tree with a vertex for each distinct fractional offset value. The latter is a standard union-find data structure (see [1]) with a vertex for each deadline d_j and with deadlines having the same fractional offset value belonging to the same tree in the forest. Each vertex in the fractional offset tree will contain a pointer to the root of the union-find tree for that fractional offset value. Each vertex in the pseudo-offset forest will contain a numerical value such that the sum of the values along the path from a vertex to the corresponding root is exactly the pseudo-offset for the deadline represented by that vertex. Initially both data structures are empty.

When a new forbidden region (a, b) is created, we update these data structures as follows: First, if there is any deadline $d_j > a+1$ not represented in the data structures, we add it with pseudo-offset $o'_j = 0$ and fractional offset value $q_j = d_j[\text{mod } 1]$, merging as necessary. Second, we determine the set $Q(a, b)$ of offsets affected by (a, b) which is given by

$$Q(a,b)=\begin{cases}\{q_j\colon q_j\in(a[\text{mod }1], b[\text{mod }1])\} & \text{if } a[\text{mod }1]<b[\text{mod }1],\\ \{q_j\colon q_j\in(0, b[\text{mod }1])\cup(a[\text{mod }1], 1)\} & \text{otherwise.}\end{cases}$$

If $Q(a, b)$ is empty, no pseudo-offset is affected by (a, b) and the two data structures can remain unchanged. If $Q(a, b)$ is nonempty, then all the corresponding entries in the fractional offset tree are replaced by a single entry with value $a[\text{mod } 1]$, and all trees in the pseudo-offset forest that correspond to the fractional offset value $a[\text{mod } 1]$ are merged together. Finally, the auxiliary values in the pseudo-offset forest are changed to reflect the corresponding changes in the pseudo-offsets of the members of $Q(a, b)$. The appropriate amount of this change is $q_j - a[\text{mod } 1]$ if $a[\text{mod } 1] < q_j$, or $1 + q_j - a[\text{mod } 1]$ if $a[\text{mod } 1] > q_j$.

With appropriate data structures for the fractional offset tree and the pseudo-offset forest (again, see [1] and also [9]), the overall time bound for maintaining this information will be $O(n \log n)$. Each deadline is added to this structure once at a cost of $O(\log n)$. The construction of each set $Q(a, b)$ requires time $O((|Q(a, b)|+1)\cdot \log n)$, and this will be $O(n \log n)$ overall since the sum over all $Q(a, b)$ of $|Q(a, b)|$ is bounded by $2n$. (Once two deadlines are merged because they have the same fractional offset value, they stay merged henceforth.) The time for merging two trees in the pseudo-offset forest is $O(1)$ per merge and hence $O(n)$ overall. Moreover, this can be done so that no tree ever has depth exceeding $O(\log n)$, so any particular pseudo-offset can be computed in time $O(\log n)$, as required. Finally, the changes in pseudo-offsets caused by a set $Q(a, b)$ can be incorporated in time $O(|Q(a, b)|)$ and hence $O(n)$ overall. Thus, as claimed, the overall time required for maintaining the pseudo-offset data structures is $O(n \log n)$. Specific details of the implementation are left to the reader.

We now wish to argue that we can still identify the same forbidden regions by using the pseudo-offsets. Recall that the critical time for deadline d_j is defined to be $d_j - o_j - n_j$. Define the *pseudo-critical time* c'_j to be $c'_j = d_j - o'_j - n_j$, and observe that the pseudo-critical time for a deadline never exceeds its critical time, though it may be smaller. In particular, if the *pseudo-task load* $n'_j = \min\{n\colon o'_j(n) = o'_j\}$ exceeds the task load n_j, then $c'_j < c_j$. The following lemma shows that we can use pseudo-critical times in place of critical times in step 2 of Algorithm A and still compute the same forbidden regions.

LEMMA 4. *If, after task T_i is processed, the minimum pseudo-critical time $c' = \min\{c'_j\colon n_j > 0\}$ satisfies $c' < r_i + 1$, then the minimum critical time $c = \min\{c_j\colon n_j > 0\}$ equals c'.*

Proof. The proof is by induction on the number of tasks processed. The lemma clearly holds if no tasks have been processed, since initially there are no forbidden regions and all critical times and pseudo-critical times equal their corresponding deadlines by definition. Suppose the lemma holds after processing task T_{i+1} but not after processing task T_i (recall that we process tasks in order of decreasing index). Then after processing task T_i there must be a deadline d_j such that $c'_j < r_i + 1$ is the minimum pseudo-critical time, but the minimum critical time c exceeds c'_j. In particular, this means that the minimum critical time c_0 after T_{i+1} is processed must obey $c_0 \geqq c > c'_j$.

We also must have $c_j > c'_j$, and hence $n_j < n'_j$ (and hence $n'_j > 0$). This means that

$$\begin{aligned} c'_j = d_j - (o'_j + n_j) &\geqq d_j - (o'_j(n'_j) + n'_j) + 1 \\ &\geqq b_j(n'_j) + 1. \end{aligned}$$

Since $n'_j > 0$, we must have $o'_j(n'_j) > o'_j(n'_j - 1)$ by definition. Therefore $b_j(n'_j - 1) - 1$ must be in a forbidden region and $b_j(n'_j)$ must be the left endpoint of a forbidden region, by the operation of the Backscheduling Algorithm. Thus, if l is the left endpoint of the leftmost forbidden region while T_i is being processed, we must have

$$c'_j \geqq b_j(n'_j) + 1 \geqq l + 1.$$

But, by the way forbidden regions are defined in step 2 of Algorithm A, we must have $l \geqq c_0 - 1$. Hence $c'_j \geqq c_0$, a contradiction. Thus the lemma does indeed hold after processing T_i, and, by induction holds after processing any task.

As a consequence of Lemma 4, we know that if critical times are replaced by pseudo-critical times in step 2 of Algorithm A, the same forbidden regions will be declared. Hence, by replacing steps 1a and 1b by the computation of task loads and pseudo-offsets, we will not affect the critical regions and we will reduce two of the potential sources of $\Omega(n^2)$ computation steps to $O(n \log n)$. The remaining potential difficulty is in the calculation of $c' = \min \{c'_j : n_j > 0\}$ in step 2. If we had to look at all the c'_j each time we calculated c', this could now conceivably take time $\Omega(n^2 \log n)$. Fortunately, we do not need to do this. The key observation is contained in the following lemma.

LEMMA 5. *If, after task T_i is processed, we have $c'_j \leqq c'_k$ for some deadlines $d_k \leqq d_j$, then at all times in the future we will have $c'_j \leqq c'_k$.*

Proof. The pseudo-critical time for a given deadline changes only when either (a) the task load changes or (b) a new forbidden region is defined and changes the pseudo-offset. Since $d_k \leqq d_j$, each change in the pseudo-critical time for d_k due to an increase in task load must be balanced by an identical change for d_j. Thus (b) is all that we need consider. The only way that a pseudo-critical time c'_k can be altered by a new forbidden region (a, b) is if there is some integer n such that $c'_k - n \in (a, b)$, in which case the pseudo-offset increases by $c'_k - n - a$ and the new pseudo-critical time becomes $c'_k - (c'_k - n - a) = n + a$. However, since $c'_j \leqq c'_k$, we must have either $c'_j - n \leqq a$, in which case $c'_j \leqq n + a$, or else $c'_j - n \in (a, b)$ and hence c'_j also becomes $n + a$. In either case we have that the new values obey $c'_j \leqq n + a = c'_k$, as claimed.

Thus, as our algorithm proceeds, certain pseudo-critical times become unnecessary for our computations of $c' = \min \{c'_j : n_j > 0\}$. To formalize this idea, let us say that a deadline d_i is *relevant* at a point in the updating process if for no d_j with $d_j \geqq d_i$ is $c'_j < c'_i$. A deadline is *irrelevant* if it is not relevant. Initially all deadlines are relevant, though some may become irrelevant as the computation proceeds. Note that if we sort the relevant deadlines into nondecreasing order, their pseudo-critical times must also be in nondecreasing order. If we maintain a pointer into this list to the first deadline with a "defined" pseudo-critical time (i.e., with $n_j > 0$), then at any time we can determine c' in time $O(\log n)$ by merely computing the pseudo-critical time for the deadline to which the pointer points, using the data structures for task loads and pseudo-offsets discussed earlier. The total time for computing values of c' will thus be $O(n \log n)$. Moreover, since we keep the relevant deadlines sorted by deadline rather than pseudo-critical time, we need not do any reordering except to delete newly irrelevant deadlines. This will allow us to maintain data structures for the relevant deadlines in a way that also obeys an $O(n \log n)$ bound.

We store the relevant deadlines in a *relevant deadline tree*, which is a binary search tree structured to allow deletions in $O(\log n)$ time and can itself be initialized in time $O(n \log n)$—again, see [1]. The pointer to the first relevant deadline with nonzero task load is initially undefined. In updating we make use of the following lemma, whose proof is much like that of Lemma 5 and is omitted.

LEMMA 6. *A relevant deadline can only become irrelevant as a result of the change of task loads during the processing of a task T_i, and not as the result of a change in pseudo-offsets during the processing of a forbidden region. If d_j is the minimum relevant deadline not less than d_i when T_i is processed, then the deadlines that become irrelevant are precisely those relevant deadlines $d_k < d_j$ whose old pseudo-critical times exceed the new value of the pseudo-critical time c'_j.*

Thus, after updating the task load tree, we need only identify d_j (in time $O(\log n)$), compute its pseudo-critical time (again in time $O(\log n)$), and then begin comparing this to the pseudo-critical times for those relevant deadlines d_k with $d_k < d_j$, in order, starting with the latest and ending as soon as one is found with $c'_k \leqq c'_j$. All those with $c'_k > c'_j$ are deleted, at a cost of $O(\log n)$ per deletion. Since a deadline can only be deleted once, the overall cost for comparisons and deletions will then be $O(n \log n)$ as claimed. In addition, during this process the pointer to the first relevant deadline with nonzero task load can be updated if necessary at an overall cost of $O(n)$. Thus step 2 of Algorithm A can be replaced by a procedure which accomplishes the same task in a running time of $O(n \log n)$.

This guarantees that the overall algorithm can be made to run in time $O(n \log n)$. We shall call the revised algorithm Algorithm B. It differs from Algorithm A only in Part I, as Part II already runs in time $O(n \log n)$. The revised Part I proceeds as follows:

ALGORITHM B.

Part I. (Forbidden Region Declaration). Initially, there are no forbidden regions, the relevant deadline tree contains all deadlines, its pointer is undefined, the task load tree contains all deadlines with their task loads initialized to 0 and the two offset data structures are empty. For each task T_i, in order of decreasing index, we then perform the following steps:

Step 1.

1a. Modify the task load tree to add 1 to the task load for each deadline $d_j \geqq d_i$.

1b. Set $d' \leftarrow \min \{d_j : d_j$ is relevant and $d_j \geqq d_i\}$. While the relevant deadline d'' that immediately precedes d' in the sorted order of deadlines has a pseudo-critical time exceeding that for d', delete d'' from the relevant deadline tree and, if the pointer was undefined or pointed to d'', reset the pointer to point to d'.

Step 2.

If $i = 1$ or $r_{i-1} < r_i$, set $c' \leftarrow \min \{c'_j : d_j$ is relevant and $n_j > 0\}$ and proceed as follows:

2a. If $c' < r_i$, declare failure and halt.

2b. If $r_i \leqq c' < r_i + 1$, declare $(c' - 1, r_i)$ to be a forbidden region and update the data structures as follows:

Add to the fractional offset tree and the pseudo-offset forest all deadlines $d_j \geqq c'$ that are not currently represented, merging if necessary. Compute $Q(c' - 1, r_i)$ and replace all corresponding entries in the fractional offset tree by a single entry with value $c' - 1$. Merge all the corresponding sets in the pseudo-offset forest and update the pseudo-critical times appropriately.

The reader should be able to verify from the preceding discussion that the algorithm does indeed compute the same forbidden regions as Algorithm A, but now does it in an overall time bound of $O(n \log n)$. Certain additional efficiencies can be obtained, for instance by using path compression in the pseudo-offset forest (which is not required for the $O(n \log n)$ bound) and by combining the relevant deadline tree and the task load tree into a single data structure, but these will not yield any improvement in the basic $O(n \log n)$ bound and so we leave such details to those readers interested in actually implementing the procedure. We also leave to the reader the straightforward verification of the fact that none of the data structures used by this algorithm (or Algorithm A) require more than linear space. Fig. 3 illustrates the application of Algorithm B to the tasks of Table 1, and can be compared to the analogous Fig. 2 for Algorithm A.

5. Conclusion. In this paper we have studied the problem of scheduling unit-time tasks with arbitrary release times and deadlines, and we have showed how the idea of "forbidden regions" could be used to construct an algorithm which solved the problem of minimizing makespan on one processor in time $O(n \log n)$.

Several interesting open problems remain. Carlier [2] has recently shown that the general m-processor case can be solved in time $O(n^{m+1} \log n)$. Can the general problem be solved in time polynomial in *both* m and n, perhaps by a suitable generalization of the concept of forbidden regions (which we were unable to find)? What happens if we add precedence constraints to the problem? We have already seen that this does not affect the one-processor algorithm, but what of the case of two processors, where a polynomial time algorithm *is* known for the case when all release times are integers [3]?

	RELEASE TIMES										
DEADLINES	9	$8\frac{2}{3}$	$8\frac{1}{3}$	5	$4\frac{2}{3}$	$4\frac{1}{3}$	$3\frac{1}{2}$	$1\frac{2}{3}$	$\frac{2}{3}$	$\frac{1}{3}$	0
$5\frac{2}{3}$	0	0	0	0	0	0	0	0	1	1	1
6	0	0	0	0	0	0	0	1	2	2	2
$6\frac{1}{3}$	0	0	0	0	1	1	1	2	3	3	3
$6\frac{2}{3}$	0	0	0	0	1	2	2	3	4	4	4
$7\frac{2}{3}$	0	0	0	0	1	2	3	4	5	5	5
8	0	0	0	1	2	3	4	5	6	6	6
10	0	0	0	1	2	3	4	5	6	7	7
$10\frac{1}{3}$	1	1	1	2	3	4	5	6	7	8	8
$11\frac{1}{3}$	1	2	3	4	5	6	7	8	9	10	10
$12\frac{1}{3}$	1	2	3	4	5	6	7	8	9	10	11

FIG. 3a. *Table of task loads.*

DEADLINES	FORBIDDEN REGIONS							
	$(8\frac{1}{3},9)$	$(8\frac{1}{3},8\frac{2}{3})$	$(7\frac{1}{3},8\frac{1}{3})$	$(4\frac{1}{3},4\frac{2}{3})$	$(3\frac{2}{3},4\frac{1}{3})$	$(2\frac{2}{3},3\frac{1}{2})$	$(-\frac{1}{3},\frac{1}{3})$	$(-\frac{1}{3},0)$
$5\frac{2}{3}$	-	-	-	$\frac{2}{3}$ 0	$\frac{2}{3}$ 0	$\frac{2}{3}$ 0	$\frac{2}{3}$ 0	$\frac{2}{3}$ 0
6	-	-	-	0 0	$\frac{2}{3}$ $\frac{1}{3}$	$\frac{2}{3}$ $\frac{1}{3}$	$\frac{2}{3}$ $\frac{1}{3}$	$\frac{2}{3}$ $\frac{1}{3}$
$6\frac{1}{3}$	-	-	-	$\frac{1}{3}$ 0	$\frac{1}{3}$ 0	$\frac{2}{3}$ $\frac{2}{3}$	$\frac{2}{3}$ $\frac{2}{3}$	$\frac{2}{3}$ $\frac{2}{3}$
$6\frac{2}{3}$	-	-	-	$\frac{2}{3}$ 0	$\frac{2}{3}$ 0	$\frac{2}{3}$ 0	$\frac{2}{3}$ 0	$\frac{2}{3}$ 0
$7\frac{2}{3}$	-	-	-	$\frac{2}{3}$ 0	$\frac{2}{3}$ 0	$\frac{2}{3}$ 0	$\frac{2}{3}$ 0	$\frac{2}{3}$ 0
8	-	-	-	0 0	$\frac{2}{3}$ $\frac{1}{3}$	$\frac{2}{3}$ $\frac{1}{3}$	$\frac{2}{3}$ $\frac{1}{3}$	$\frac{2}{3}$ $\frac{1}{3}$
10	0 0	0 0	$\frac{1}{3}$ $\frac{2}{3}$	$\frac{1}{3}$ $\frac{2}{3}$	$\frac{1}{3}$ $\frac{2}{3}$	$\frac{2}{3}$ $\frac{4}{3}$	$\frac{2}{3}$ $\frac{4}{3}$	$\frac{2}{3}$ $\frac{4}{3}$
$10\frac{1}{3}$	$\frac{1}{3}$ 0	$\frac{1}{3}$ 0	$\frac{1}{3}$ 0	$\frac{1}{3}$ 0	$\frac{1}{3}$ 0	$\frac{2}{3}$ $\frac{2}{3}$	$\frac{2}{3}$ $\frac{2}{3}$	$\frac{2}{3}$ $\frac{2}{3}$
$11\frac{1}{3}$	$\frac{1}{3}$ 0	$\frac{1}{3}$ 0	$\frac{1}{3}$ 0	$\frac{1}{3}$ 0	$\frac{1}{3}$ 0	$\frac{2}{3}$ $\frac{2}{3}$	$\frac{2}{3}$ $\frac{2}{3}$	$\frac{2}{3}$ $\frac{2}{3}$
$12\frac{1}{3}$	$\frac{1}{3}$ 0	$\frac{1}{3}$ 0	$\frac{1}{3}$ 0	$\frac{1}{3}$ 0	$\frac{1}{3}$ 0	$\frac{2}{3}$ $\frac{2}{3}$	$\frac{2}{3}$ $\frac{2}{3}$	$\frac{2}{3}$ $\frac{2}{3}$

FIG. 3b. *Table of fractional and pseudo-offsets.*

	9	$8\frac{2}{3}$	$8\frac{1}{3}$	5	$4\frac{2}{3}$	$4\frac{1}{3}$	$3\frac{1}{2}$	$1\frac{2}{3}$	$\frac{2}{3}$	$\frac{1}{3}$	0
$5\frac{2}{3}$	$5\frac{2}{3}$	$5\frac{2}{3}$	$5\frac{2}{3}$	$5\frac{2}{3}$	▨	▨	▨	▨	▨	▨	▨
6	6	6	6	6	▨	▨	▨	▨	▨	▨	▨
$6\frac{1}{3}$	$6\frac{1}{3}$	$6\frac{1}{3}$	$6\frac{1}{3}$	$6\frac{1}{3}$	$5\frac{1}{3}$	▨	▨	▨	▨	▨	▨
$6\frac{2}{3}$	$6\frac{2}{3}$	$6\frac{2}{3}$	$6\frac{2}{3}$	$6\frac{2}{3}$	$5\frac{2}{3}$	$4\frac{2}{3}$	▨	▨	▨	▨	▨
$7\frac{2}{3}$	$7\frac{2}{3}$	$7\frac{2}{3}$	$7\frac{2}{3}$	▨	▨	▨	▨	▨	▨	▨	▨
8	8	8	8	7	6	5	$3\frac{2}{3}$	$2\frac{2}{3}$	$1\frac{2}{3}$	▨	▨
10	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
$10\frac{1}{3}$	$9\frac{1}{3}$	$9\frac{1}{3}$	▨	▨	▨	▨	▨	▨	▨	▨	▨
$11\frac{1}{3}$	$10\frac{1}{3}$	$9\frac{1}{3}$	$8\frac{1}{3}$	$7\frac{1}{3}$	$6\frac{1}{3}$	$5\frac{1}{3}$	$4\frac{1}{3}$	$2\frac{2}{3}$	$1\frac{2}{3}$	$\frac{2}{3}$	$\frac{2}{3}$
$12\frac{1}{3}$	$11\frac{1}{3}$	$10\frac{1}{3}$	$9\frac{1}{3}$	$8\frac{1}{3}$	$7\frac{1}{3}$	$5\frac{1}{3}$	$5\frac{1}{3}$	$3\frac{2}{3}$	$2\frac{2}{3}$	$1\frac{2}{3}$	$\frac{2}{3}$

FIG. 3c. *Table of pseudo-critical times for relevant deadlines when Algorithm* B *is applied to tasks in Table* 1. *Circled entries are the minimum pseudo-critical times for deadlines with nonzero task loads. Cross-hatched regions are for irrelevant deadlines.*

Note added in proof. B. Simons has recently resolved one of our open problems by showing that the general m-processor case (with no procedure constraints) can be solved in time $O(n^3 \log n)$. [*A fast algorithm for multiprocessor scheduling*, IEEE 21st Annual Symposium on Foundations of Computer Science, Long Beach, California, 1980, pp. 50–53.]

REFERENCES

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

[2] J. CARLIER, *Problème à une machine dans le cas où les taches ont des durées égales*, Technical Report (1979) Institut de Programmation, Université Paris VI, Paris.

[3] M. R. GAREY AND D. S. JOHNSON, *Two-processor scheduling with start-times and deadlines*, this Journal, 6 (1977), pp. 416–426.

[4] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of* NP-*Completeness*, W. H. Freeman, San Francisco, California, 1979.

[5] J. R. JACKSON, *Scheduling a production line to minimize maximum tardiness*, Research Report 43 (1955), Management Science Research Project, University of California at Los Angeles.

[6] D. E. KNUTH, *The Art of Computer Programming*, Vol. 1: *Fundamental Algorithms*, Addison-Wesley, Reading, Mass., 1968.

[7] T. LANG AND E. B. FERNÁNDEZ, *Scheduling of unit-length independent tasks with execution constraints*, Information Processing Letters, 4 (1976), pp. 95–98.

[8] B. SIMONS, *A fast algorithm for single processor scheduling*, IEEE 19th Annual Symposium on Foundations of Computer Science, Long Beach, California, 1978, pp. 246–252.

[9] R. E. TARJAN, *Applications of path compression on balanced trees*, J. Assoc. Comput. Mach., submitted.

[10] J. D. ULLMAN, *NP-complete scheduling problems*, J. Comput. System Sci., 10 (1975), pp. 384–393.