# Preemptive Scheduling with Due Dates

SARTAJ SAHNI

*University of Minnesota, Minneapolis, Minnesota*

(Received July 1977; accepted August 1978)

An $O(n \log mn)$ algorithm is presented to preemptively schedule $n$ tasks on $m$ identical machines. The tasks are assumed to have due dates. All tasks are initially available. The objective is to obtain a preemptive schedule that meets all due dates (when possible). Our algorithm generates schedules with at most $n-2$ preemptions. The algorithm may also be used to schedule a set of $n$ tasks all having the same due date but having different release dates.

WE STUDY the problem of scheduling $n$ independent jobs on $m$ identical and parallel machines (processors). Each job $i$ has associated with it a release date $r_i$, $r_i \geq 0$, a due date $d_i$ and a processing requirement $t_i$, $t_i \geq 0$. The processing of job $i$ cannot begin until its release date $r_i$ and must be completed by its due date $d_i$. It is assumed that $r_i + t_i \leq d_i$, $1 \leq i \leq n$. We also assume that there is no cost associated with a preemption. In addition, no machine can simultaneously process two jobs. No job can simultaneously be processed by more than one machine. The reader is referred to [3] for a formal definition of a preemptive schedule.

DEFINITION. *A DD-schedule for a set $J$ of $n$ jobs is a schedule in which all jobs are completed by their due date.*

Our main concern is the development of a fast algorithm to obtain preemptive DD-schedules when either all jobs have the same release time or all jobs have the same due time. Before going into the algorithm, we briefly review some related results.

Lenstra et al. [6] have shown that deciding whether or not a set of jobs has a nonpreemptive DD-schedule is NP-Complete even when $m = 1$. Hence, it appears unlikely that a polynomial time algorithm for this problem exists. Bratley et al. [1] present a branch-and-bound algorithm to obtain nonpreemptive schedules. This algorithm claims to have a good chance of solving problems with up to 100 jobs in a "reasonable" amount of time. The problem of obtaining preemptive DD-schedules (when such schedules exist) appears to be much simpler. Horn [4] presents an $0(n^3)$ algorithm for this case. His algorithm essentially reduces the scheduling

0030-364X/79/2705-0925 $01.25

problem to a network flow problem. Bruno and Gonzalez [2] extend this idea to the case of two uniform machines.

When attempting to construct an algorithm to obtain a preemptive (or nonpreemptive) DD-schedule, one can distinguish among three types of algorithms: on line, nearly on line and off line.

DEFINITION. *Let $J$ be a set of $n$ jobs with $k$ distinct release times. An on line algorithm to construct a DD-schedule is one which generates the schedule from* 0 *to any time $\tau$ without knowledge of the jobs released at or after time $\tau$, and without knowledge of the remaining release times. A nearly on line algorithm is one which constructs the schedule from* 0 *to $r_i$ without knowledge of the jobs released at $r_i, r_{i+1}, \cdots, 1 \leq i \leq k$. The next release date is the only information available to a nearly on line algorithm that is not available to an on line algorithm. An off line algorithm has available to it at time* 0 *all job times, release dates and due dates.*

We can see that in some scheduling environments an on line algorithm is essential, while in others a nearly on line algorithm will do, and, in yet other situations, an off line algorithm will suffice. Also, from the definition of the terms it follows that if there is no off line algorithm to obtain DD-schedules, then there can be no nearly on line algorithm. Further, if there is no nearly on line algorithm, then there can be no on line algorithm. The following lemma shows that there is no nearly on line (and hence no on line) algorithm to obtain preemptive DD-schedules when $m > 1$.

LEMMA 1. *When there are two or more parallel machines, there exists no nearly on line algorithm to construct DD-schedules.*

*Proof.* Assume $m = 2$ and that at time 0 three jobs $A$, $B$ and $C$, are released. $t_A = t_B = 5$, $t_C = 10$, $d_A = 5$, $d_B = 15$ and $d_C = 24$. The next release time is 5. Every nearly on line algorithm must build the schedule from 0 to 5 before it can learn the due dates and processing requirements of jobs to be released at time 5. In every DD-schedule job $A$ must be scheduled from 0 to 5. Without loss of generality we may assume it is scheduled on machine 1. Further, we may assume that $B$ is scheduled on machine 2 for some time $\Delta$, $0 \leq \Delta \leq 5$ and $C$ is also scheduled on this machine for time $5 - \Delta$ (see Figure 1(a)). If $\Delta = 0$ then assume that at time 5 jobs $D$ and $E$ with $d_D = d_E = 11$ and $t_D = t_E = 6$ are released. Now, the DD-schedule must have the form shown in Figure 1(b). Clearly, there is no way to schedule $B$ to finish by its due time. On the other hand, had we chosen $\Delta = 5$, then all five jobs could have been processed by their due times (Figure 1(c)).

Next, assume that we had scheduled with $\Delta \neq 0$. Then, assume that at time 5 jobs $D$ and $E$ are released. $t_D = t_E = 5$, $d_D = d_E = 10$. Also, assume that at time 15 jobs $F$ and $G$ are released with $t_F = t_G = 9$, $d_F = d_G = 24$.

Any DD-schedule must schedule *D*, *E*, *F* and *G* as in Figure 1(d). Clearly, there is no way to schedule *C* to complete by its due date of 24. If, we had chosen $\Delta = 0$, then a DD-schedule could have been constructed (Figure 1(e)).

The same example can be used for $m > 2$ by introducing $m - 2$ jobs released at time 0 with due times 24 and processing requirement 24.

For the case $m = 1$, Horn [4] presents an $0(n \log n)$ nearly on line algorithm to obtain a DD-schedule. This algorithm constructs DD-schedules with at most $n - 1$ preemptions. One may easily establish $n - 1$ as the lower bound on the minimum number of preemptions in every DD-schedule for some set of $n$ jobs.

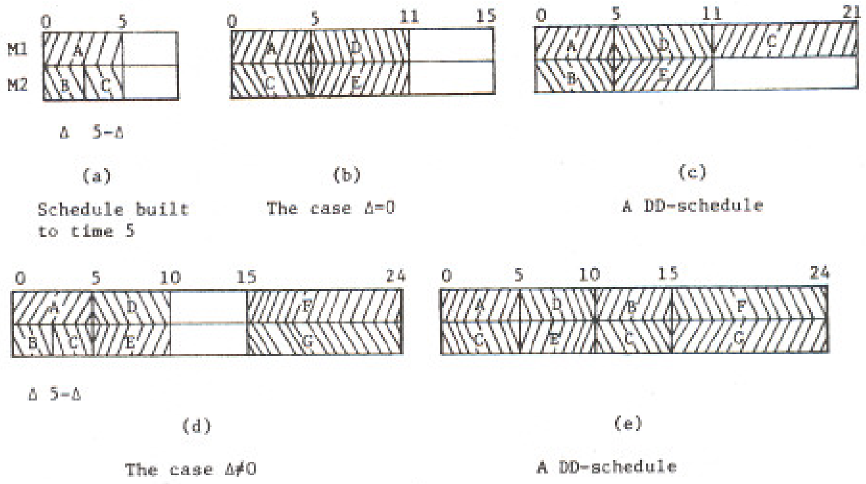


**Figure 1**

LEMMA 2. *For the case $m = 1$, there exist job sets $J$ of size $n$ for which every DD-schedule has at least $n - 1$ preemptions.*

*Proof.* Consider the job set $(r_1, t_1, d_1) = (0, n, 2n - 1)$, $(r_j, t_j, d_j) = (2j - 3, 1, 2j - 2)$, $2 \le j \le n$. Clearly, in every DD-schedule, job $j$ must be scheduled from time $2j - 3$ to time $2j - 2$, $2 \le j \le n$. The only way to complete job 1 is by scheduling it in the slots $(2i, 2i + 1)$, $0 \le i < n$. This introduces $n - 1$ preemptions.

We obtain an algorithm that constructs DD-schedules for job sets in which all jobs have the same release date. For this case, all algorithms are on line. In [4] Horn presents necessary and sufficient conditions for the existence of a DD-schedule for the case when all jobs have the same release date. If the $n$ jobs have $k$ distinct due dates, it takes $0(kn)$ time to test these conditions. In the worst case $0(kn) = 0(n^2)$. Since Horn's proof for the sufficiency of these conditions is constructive, an algorithm can be obtained from this proof. The proof constructs the DD-schedule, interval by interval. For each distinct due date $0(kn)$ time is needed. The total time needed by the resulting algorithm is $0(k^2n) = 0(n^3)$. Schedules resulting from Horn's algorithm may have $0(mn)$ preemptions. Our algorithm is of complexity $0(n \log mn)$ and, in addition, constructs schedules with at most $n - 2$ preemptions.

## THE ALGORITHM

When all jobs become available for processing at time 0, it is possible to construct a DD-schedule (if one exists) by considering jobs in order of nondecreasing due dates. Let $J$ be a set of $n$ jobs and let $\delta_i$, $1 \leq i \leq k$ be the $k$ distinct due dates. For convenience, we introduce $\delta_0$, $\delta_0 = 0$. Let $n_i$, $1 \leq i \leq k$ be the number of jobs with due date $\delta_i$. Clearly, $\sum n_i = n$ and we may assume $n_i \geq 1$, $1 \leq i \leq k$. We first present our scheduling algorithm and then show that it works.

The algorithm schedules the $n$ jobs in $k$ stages. In stage $i$ all jobs with due time $\delta_i$ are scheduled. The scheduling of any job $r$ with due time $\delta_i$ is carried out as follows:

1. Let RPT$(j)$ be the processing time available on machine $j$ from time 0 to time $\delta_i$, $1 \leq j \leq m$.
2. If $t_r$ is greater than RPT$(j)$ for all $j$ then job $r$ cannot be scheduled to meet its due time and the algorithm terminates.
3. If $t_r$ is less than or equal to RPT$(j)$ for all $j$, such that RPT$(j) \neq 0$, then job $r$ is scheduled on the machine with least nonzero RPT. The algorithm proceeds to schedule the next job.
4. If neither 2 nor 3 hold, then we determine the machine $x$ with maximum RPT and $t_r \geq$ RPT$(x)$ and also the machine $y$ with least RPT (and RPT $\neq 0$) and $t_r <$ RPT$(y)$. Note that machine $y$ may not exist. If $y$ exists then job $r$ is scheduled to use up all the RPT of $x$ and any additional time needed from $y$. If $y$ does not exist then we must have $t_r =$ RPT$(x)$ and $r$ is scheduled to use up all the RPT on $x$. The algorithm proceeds to schedule the remaining jobs.

The above ideas are formalized in procedure ONERT (Figure 2). Procedure ONERT assumes that the jobs have been ordered such that $d_i \leq d_{i+1}$, $1 \leq i \leq n$. $F(i)$ gives the time at which machine $i$ will complete processing jobs (or parts of jobs) already assigned to it. Initially, no processing is assigned to any machine and so $F(i) = 0$, $1 \leq i \leq m$ (lines 1-3). In iteration $i$ of the loop of lines 5-20, all jobs with due date $\delta_i$ are scheduled. These jobs have index $l$ through $u$. The loop of lines 7 to 18 does the actual scheduling. A subalgorithm FIND$(t, x, y)$ is used that has one input $t$ and two outputs $x$ and $y$.

Let $i_1, i_2, \cdots, i_w$ be the processors with some idle time preceding $\delta_i$

when job $r$ is to be scheduled. Assume $F(i_1) \geq F(i_2) \geq \cdots \geq F(i_w)$. Algorithm FIND does the following. If $t_r \leq \delta_i - F(i_1)$ then it returns $x = 0$ and $y = i_1$. If $t_r > \delta_i - F(i_w)$ it returns $y = m + 2$. If neither of these cases holds then FIND determines the largest $b$ such that $\delta_i - F(i_b) \leq t_r$ and the smallest $c$ such that $\delta_i - F(i_c) > t_r$. If there is no such $c$ then let $i_c = m + 1$. The algorithm returns $x = i_b$ and $y = i_c$. Based upon the output from FIND, job $r$ is either scheduled as in lines 11–16 or there exists no

```
line  procedure ONERT($J, p, k, m$)
        //obtain a DD-schedule for job set $J$. The jobs are ordered such that $d_i$
          $\leq d_{i+1}$, $1 \leq i < n$. $\delta_i$, $1 \leq i \leq k$ are the distinct due times. $m \geq 1$ is the
          number of processors//
          declare $F(m)$
  1     for $i \leftarrow 1$ to $m$ do //all processors initially free//
  2       $F(i) \leftarrow 0$
  3     repeat
  4     $l \leftarrow 1$
  5     for $i \leftarrow$ to $k$ do //schedule jobs with due time $\delta_i$//
  6       $u \leftarrow$ maximum $r$ such that $l \leq r$ and $d_r = \delta_i$
  7       for $r \leftarrow l$ to $u$ do
  8         call FIND($t_r, x, y$)
  9         case
 10           :$y = m + 2$:  print, ('no DD-schedule'); stop
 11           :$x = 0$:      print, ('scheule', $r$, 'on machine', $y$, 'from', $F(y)$, 'to',
                               $F(y) + t_r$)
 12                          $F(y) \leftarrow F(y) + t_r$
 13           :else:         print, ('schedule', $r$, 'on machine', $x$, 'from', $F(x)$, 'to',
                               $\delta_i$)
 14                          if $y \neq m + 1$ then[ print, ('schedule', $r$, 'on machine',
                               $y$, 'from', $F(y)$, 'to', $F(x) + F(y) + t_r - \delta_i$)
 15                          $F(y) \leftarrow F(x) + F(y) + t_r - \delta_i$]
 16                          $F(x) \leftarrow \delta_i$
 17         endcase
 18       repeat
 19       $l \leftarrow u + 1$
 20     repeat
 21   end ONERT
```

**Figure 2**

DD-schedule (line 10). It should be clear that ONERT generates valid schedules. Note that the loops of lines 5 and 7 may be combined into one: *for* $i \leftarrow 1$ *to* $n$ *do*. We choose the two loop format as this corresponds to our correctness proof for this algorithm.

We shall first prove that ONERT obtains a DD-schedule for every jobset $J$ for which such a schedule exists.

LEMMA 3. *Let $F(j)$, $1 \leq j \leq m$ be the time at which machine $j$ becomes free. Let $F(j) \geq F(j+1)$, $1 \leq j < m$. Let $R$ be a set of jobs that can be scheduled to finish by $\delta_i$ on these $m$ machines. A schedule for $R$ may be constructed by assigning jobs in $R$ (in any order) to the machines as in lines 7-18 of procedure ONERT.*

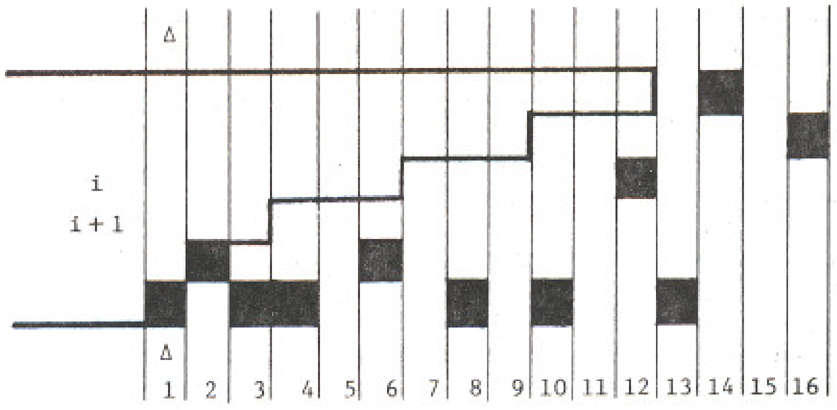
*Proof.* The proof is by induction on the number of jobs, $w$, in $R$. The lemma is clearly true when $w = 1$. Assume it is true for all $w$, $1 \leq w < z$. We shall show it true for $w = z$. Let $S$ be any schedule for $R$. Divide the interval from $F(m)$ to $\delta_i$ into intervals of size $\Delta$ such that no job is preempted on any processor within such an interval and also such that each $F(i)$ is the start of a new interval. (Since $S$ has a finite number of preemptions, it follows that such a $\Delta$ exists.) Number these intervals 1 to $q$ from left to right. Let $t_1$ be the processing requirement of the first job in $R$. Assume that ONERT assigns this job to machines $i$ and $i + 1$ (note that because of the assumption $F(i) \geq F(i+1)$, $y = x + 1$. Also, note that if $t_1$ is assigned by line 11 then no processing is assigned to $i + 1$). Rearrange the processing of job 1 in $S$ as follows. If job 1 is being processed in a $\Delta$ interval that includes processing time on machine $i$, then interchange the processing of job 1 in this interval with that of the job scheduled on machine $i$ in this interval. If job 1 is scheduled in a $\Delta$ interval including no processing time on machine $i$, then interchange the processing of job 1 in this interval with that of the job scheduled in this interval on the processor with least index (see Figure 3).

We define two types of interval interchanges. A type 1 interchange takes place between two intervals both of which include the same number of machines (for example intervals 7 and 8 or 15 and 16 of Figure 3(b)). Clearly, the interchange of these two intervals (i.e., replace 7 by 8 and 8 by 7) leaves behind a valid schedule. An interchange of type 2 takes place between two intervals $\alpha$ and $\beta$ that include a different number of machines (for example, intervals 1 and 7 or 4 and 9). Job 1 is being processed in the interval that includes a smaller number of machines and is not being processed in the other interval. Without loss of generality, we may assume $\alpha < \beta$ and $\alpha$ includes fewer machines than $\beta$. Hence, there exists a job $s$ being processed in $\beta$ but not in $\alpha$. Job 1 may be interchanged with this job $s$. The resulting schedule is valid as neither job $s$ nor job 1 is being processed by two machines in any given interval. Now, job 1 is interchanged with the job on machine $i$ in interval $\beta$.

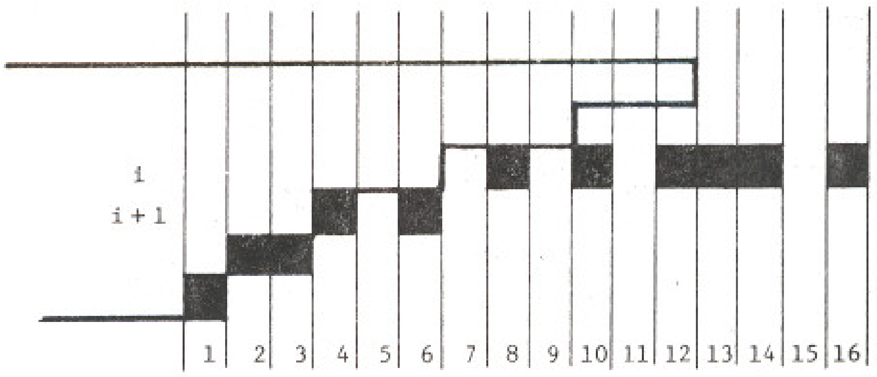
Now suppose there is an interval on machine $i$ (Figure 3(b)) in which job 1 is not being processed. We have one of two cases.

*Case 1.* $t_1 < \delta_i - F(i)$. In this case, $i = 1$ and using interchanges of type 2, all of job 1's processing may be assigned to machine 1. Since, $i = 1$, interchanges of type 1 can be used so that job 1 is assigned on machine 1 to the interval $[F(1), F(1) + t_1]$.

*Case 2.* $t_1 \geq \delta_i - F(i)$. In this case, we can use interchanges of type 2 to completely fill the interval $[F(i), \delta_i]$ on machine $i$. If there is any processing remaining on machines $i + 1, \cdots, m$ then this may be assigned to machine $i + 1$ in the interval $[F(i + 1), F(i + 1) + F(i) + t_1 - \delta_i]$. Note that by definition of $i$ and $i + 1$, $F(i + 1) + F(i) + t_1 - \delta_i < F(i)$. Hence,



(a) Schedule S



(b) Rearrangement of job 1

**Figure 3.** Shaded areas are processing job 1.

this rearrangement may be carried out in a manner similar to that of Case 1.

Following the rearrangement described above, set $F(j) = G(j)$, where $G(j)$ is the availability time for machine $j$ following the scheduling of job

1 by ONERT. The machines may be reindexed so that $F(j) \geq F(j + 1)$, $1 \leq j < m$. If $F(1) = \delta_i$ then machine 1 may be discarded and the remaining machines reindexed 1 through $m - 1$. We are left with $u - 1$ jobs. There exists a schedule for these jobs that finishes by $\delta_i$. From the induction hypothesis, it follows that lines 7–18 of ONERT will generate a schedule for them.

THEOREM 1. *Procedure ONERT generates a DD-schedule for every jobset J for which there exists such a schedule.*

*Proof.* Let $S$ be a DD-schedule for $J$. Let $R$ be the set of jobs (or parts of jobs) scheduled in the interval $[0, \delta_1]$. From McNaughton's algorithm [7] it follows that jobs in $R$ may be rescheduled in the interval $[0, \delta_1]$ so that jobs with due time $\delta_1$ are scheduled exactly as they would be using ONERT with the input including only these jobs. Let $F(i)$ be the time at which machine $i$ finishes processing jobs assigned to it (in the interval $[0, \delta_1]$) with due time $\delta_1$. Consider the schedule $S'$ resulting from this rearrangement in $[0, \delta_1]$. Let $T$ be the set of jobs with due time $> \delta_1$ and scheduled in $S'$ in the interval $[0, \delta_2]$. From Lemma 3 it follows that these jobs may be arranged so that the schedule for jobs with due times $\delta_1$ and $\delta_2$ is identical to what would be obtained if only these jobs were input to ONERT. Repeating this argument, we see that the original DD-schedule $S$ may be transformed into a DD-schedule that will be generated by ONERT. Hence, if there is a DD-schedule then ONERT will generate such a schedule.

## ANALYSIS OF ONERT

*Number of Preemptions.* It is easy to verify that job 1 is scheduled with no preemption and that the remaining jobs may have at most one preemption each. Hence, the maximum number of preemption in DD-schedules generated by ONERT is $n - 1$. This figure can be easily reduced to $n - 2$ by using ONERT to schedule only the first $n - 1$ jobs and then assigning job $n$ to the processor with least $F(i)$.

### Time Complexity

The time complexity of ONERT depends on the complexity of algorithm FIND $(t, x, y)$. Clearly, FIND can be implemented to run in time $0(m)$. This would result in a complexity of $0(nm)$ for ONERT. We can do much better than this by using balanced search trees, either AVL trees or 2–3 trees [5]. Each node in the search tree will contain a distinct $F$ value together with a list of all machines having this $F$ value. Since the search tree will contain at most $m$ nodes, insertions, deletions and searches in the tree can be carried out in $0(\log m)$ time. Initially the search tree contains only one node. This node has an $F$ value 0 and the list (1, 2, 3, $\cdots$, $m$). This list represents the fact that all machines have an $F$ value 0 to begin with. The special cases $t \leq \delta_i - F(i_1)$ and $t > \delta_i - F(i_k)$ may be tested by extracting from the search tree the largest $F$ value smaller than $\delta_i$ and the smallest $F$ value respectively. If neither of these cases holds then we need the largest $F$ value that is less than $\delta_i - t$ and the smallest $F$ value that is greater than or equal to $\delta_i - t$. Clearly, both these may also be determined from the search tree in $0(\log m)$ time. One machine corresponding to each of these $F$ values may be deleted from the corresponding nodes. This gives $x$ and $y$. The deletion may require the physical deletion of one or two nodes from the search tree but this can be done in $0(\log m)$ time. Following the scheduling of a job on $x$ and $y$, their $F$ values change. So $x$ and $y$ are reinserted into the tree. In case there is already a node in the tree corresponding to the $F$ value of $x$ (or $y$) then $x$ (or $y$) is added to the machine list for this node; otherwise a new node is inserted into the tree. This together with any needed rebalancing takes only $0(\log m)$ time. Hence, FIND together with the updates needed following the scheduling in lines 10–16 of ONERT takes only $0(\log m)$ time. The total time for ONERT is therefore $0(n \log m)$. If we include the time needed to initially sort all jobs by due dates then the total time needed becomes $0(n \log nm)$.

We conclude by noting that algorithm ONERT may also be used as an off line algorithm for the case when all jobs have the same due time. Let $J$ be a set of $n$ jobs with release times $r_i$ and processing requirements $t_i$. Let $d$ be the common due time. Construct a new instance $J'$ with $r_i' = 0$, $d_i' = d - r_i$ and $t_i' = t_i$, $1 \leq i \leq n$. This is an instance with just one release time. It is easy to see that $J'$ has a DD schedule if and only if $J$ does. Also, from a DD schedule for $J'$ one may readily obtain a DD schedule for $J$. An algorithm based on this construction is not an on line algorithm, however. Horn [4] has also pointed out the symmetry between the case when all jobs have the same due time and the case when all jobs have the same release time.

## ACKNOWLEDGMENT

This research was supported in part by NSF grant MCS 76-21024.

## REFERENCES

1. P. Bratley, M. Florian and P. Robillard, "Scheduling with Earliest Start and Due Date Constraints," *Naval Res. Logist. Quart.* **18**, 511–517 (1971).
2. J. Bruno and T. Gonzalez, "Scheduling Independent Tasks with Release Dates and Due Dates on Parallel Machines," Technical Report No. 213, The Pennsylvania State University, December 1976.
3. E. G. Coffman, Jr., *Computer and Job Shop Scheduling*, John Wiley & Sons, New York, 1975.

4. W. HORN, "Some Simple Scheduling Algorithms," *Naval Res. Logist. Quart.* **21,** 177–185 (1974).
5. E. HOROWITZ AND S. SAHNI, *Fundamentals of Data Structures*, Computer Science Press, Potomac, Md., 1976.
6. J. LENSTRA, A. RINNOOY KAN AND P. BRUCKER, "Complexity of Machine Scheduling Problems," *Ann. Discrete Math.* **1,** 343–362 (1977).
7. R. MCNAUGHTON, "Scheduling with Deadlines and Loss Functions," *Mgmt. Sci.* **12,** 1–12 (1959).