Fast algorithms for preemptive scheduling of equal-length jobs on a single and identical processors to minimize the number of late jobs

Nodari Vakhania*

Abstract— We consider the preemptive scheduling of n equal-length jobs with release times and due dates with the objective to minimize the number of late jobs on a single and parallel identical processors. Our algorithm for the single-processor version is on-line and runs in time $O(n \log n)$. It produces an optimal schedule. Our multiprocessor algorithm is off-line, it has the time complexity of $O(n^2)$ and produces a sub-optimal schedule. No optimal polynomial time algorithm for this problem is known yet.

Keywords–algorithm, due date, release time, scheduling

1 Introduction

In this paper we deal with two versions of the preemptive scheduling problem with the objective to minimize the number of late jobs. This objective function is motivated by applications in real-time overloaded systems where job due dates are crucial so that if a job is late then it might rather be removed completely or postponed for an undefined period of time in favor of other jobs which might be completed on time on the same processor.

We consider single-processor and multiprocessor versions. In the first model, we have a single processor that has to perform jobs which arrive on-line over time. Each job j is characterized by a *due date* d_j which is the desirable time for its completion on the processor. We aim to minimize the number of *late jobs*, that is, ones completed after their due dates. In our model, the *processing time* of all jobs is a given integer p. Each j becomes available (and known) at its integer *release time* r_j and, as already noted, has the due date d_j which is also an integer. The processor can handle at most one job at a time. As above noted, our problem is *preemptive*, i.e., we may split a job in a finite number of portions and assign the portions at different time moments to the processor. This assignment is feasible if no two job portions overlap in time. Thus a *feasible schedule* is a mapping which assigns each job j a unique specified time intervals with the total length p on the processor, such that the earliest such an interval starts at time no earlier than r_j. A job in a schedule S is *late* (*on time*, respectively) if it is completed after (at or before, respectively) its due date in S. Again, our objective is to find a feasible schedule with the minimal number of late jobs, equivalently, with the maximal number of on time jobs.

The multiprocessor model is similar to the above one, with the only different that instead of a single processor, each job might be performed on any (but only one) processor from the given set of m parallel identical processors.

According to the commonly used scheduling notations, our single-processor and multiprocessor problems are abbreviated as $1/pmtn, p_j = p, r_j/\sum U_j$ and $P/pmtn, p_j = p, r_j/\sum U_j$, respectively (here U_j is a 0-1 function with $U_j = 1$ iff j is late). The single-processor version is known to be polynomial due to Lawler [4]. Lawler's algorithm is off-line, that is, it needs all the problem data in advance. In many practical situations with real-time jobs (part of) the problem data are not known in advance. For example, the jobs may arrive on-line over time (at their release dates). Hence, the complete set of jobs might not be known in advance. Alternatively, the set of jobs might be given in advance but (some or all) job parameters may become known only upon the arrival of each job.

Our single-processor algorithm works in either of the above circumstances on-line. It has the time complexity of $O(n \log n)$. The algorithm uses preemptive Earliest Due-date Heuristic (*ED-heuristic*) and decides on-line whether to preempt the currently executed job and include the next arrived job into the constructed schedule or take any other actions.

Some know results for the related single-processor versions are as follows. When all the jobs are released simultaneously, the non-preemptive single-machine problem $1//\sum U_j$ can be solved off-line in polynomial time Moore [6] and Lawler [5]. If we have release times and allow preemptions then the problem $1/pmtn, r_j/\sum U_j$ can also be solved off-line in polynomial time Baptiste [1]. Without preemptions and with equal-length jobs, the problem $1/p_j = p, r_j/\sum U_j$ is also polynomial Chrobak et al. [2] and Vakhania [7].

No polynomial time algorithm producing optimal schedules is known for the multiprocessor problem, neither for the preemptive, nor for the non-preemptive version. The time complexity status of these problems remains open. We propose an $O(n...)$ algorithm for the preemptive version producing sub-optimal solutions. If jobs have arbitrary processing times, already scheduling on 2 identical processors

*State University of Morelos, Mexico. Inst. of Computational Math., Tbilisi, Georgia. E-mail: nodari@uaem.mx. Partially supported by CONACyT grant 48433

even with preemptions $P2/pmtn, r_j/\sum U_j$ is NP-hard Du et al. [3]. The weighted version of our problem even without release times $P/pmtn, p_j = p/\sum w_j U_j$ is also NP-hard Brucker & Kravchenko [**?**], but its non-preemptive version $P/p_j = p/\sum w_j U_j$ is surprisingly polynomial. Without release times, if we allow preemptions and fix the number of processors, then $Qm/pmtn/\sum U_j$ and $Pm/pmtn, p_j = p/\sum w_j U_j$ are also polynomially solvable.

2 The single-processor case

2.1 Preliminaries

Due to the nature of our objective function, if a job is late then it can obviously be scheduled arbitrarily late without affecting our objective function. Suppose S is a feasible schedule with all its jobs being included on-time, and we can assert that we have included the maximal possible number of jobs in it. Then we can append all the omitted jobs in an arbitrary feasible fashion at the end of S, in linear time.

Because of the above observation, from now on, we shall take care only on on time scheduling of jobs replacing due dates by *deadlines*: unlike due dates, deadlines are strict dates, i.e., each job is to be finished no later than its deadline in any feasible schedule. Then our task reduces to finding a feasible schedule which contains the maximal possible number of jobs.

As already noted in the Introduction, our algorithm employs on-line preemptive ED-heuristic. Initially, the current scheduling time is defined by the minimal job release time. Iteratively, among all released jobs by the current scheduling time t, ED-heuristic schedules a job with the smallest deadline (ties can be broken arbitrarily). If during the execution of a job another job with a smaller deadline is released, the former job is interrupted and the latter job is initiated. If all earlier assigned jobs are completed by the current time t and no new job is released, the algorithm waits till the next job is released. Thus each new value of t is either a job release time or (and) the completion time of the latest scheduled job so far. In the former case, if the machine is not idle and the current job is not finished, it is interrupted by a newly released job with a smaller deadline.

We call the job selected at the current scheduling time t the *incoming job* at that time and denote it by $i(t)$ or just by i when this will cause no confusion.

It is straightforward to see that the described algorithm will give the minimal overall completion time for the selected (included) set of jobs.

We denote by $S = S(t)$ our ED-schedule constructed to the current scheduling time t, and by $p_l^*(S)$ yet unscheduled part of job l in S (we may omit the argument S when this will cause no confusion).

An ED-schedule (one constructed by ED-heuristic) consists of a number of $blocks$, that is, a sequence of jobs scheduled in turn without any idle time (a gap) in between. Thus any gap in an ED-schedule is left outside any block and it arises only if there is no released job that can be scheduled within that gap.

As it is not difficult to see, not necessarily the ED-heuristic gives an optimal schedule. Consider the following problem instance with 3 jobs with the processing time $p = 10$, and with $r_1 = 4,\ d_1 = 14$, $r_2 = 2,\ d_2 = 15$ and $r_3 = 0,\ d_3 = 20$. The ED-heuristic will schedule job 3 at time 0, will interrupt this job at time 2 scheduling job 2 and will interrupt the latter job at time 4 scheduling job 1. The latter job completes at time 14. But now neither job 2 nor job 3 can be completed (scheduled) on time. We have $\sum U_j = 2$.

At the same time, two optimal schedules exist for the above problem instance. The first one does not interrupt job 2, completes it at time 12 and schedules job 3 at time 12 completing job 3 by its deadline 20. The second optimal schedule ignores job 2. It processes the first portion of job 3 in the interval $[0, 4)$. Then schedules (already released) job 1 at time 4 till its completion at time 14, and resumes job 3 at that time till its completion that occurs by time 20. For both above schedules, $\sum U_j = 1$ (for the first and the second schedules jobs 1 and 2, respectively, contribute in this sum).

2.2 The algorithm

In this section, we describe our algorithm and prove its soundness and time complexity.

In the previous section, we have seen that ED-heuristic does not guarantee the optimality of the solution that it generates. Let j be a job that could not have been included on-time by ED-heuristic. j might be a new incoming job or it might also be a partially scheduled job which unscheduled part cannot be completed on time (the second portion of job 2 in our example from the previous section).

In general, each current schedule S contains a (possibly empty) set of the partially scheduled jobs. Due to ED-heuristic, they are scheduled by decreasing order of their deadlines. We shall construct S in such a way that each job from the above set can be completed on time if all these jobs are completely scheduled.

Let us say that the incoming job $i(t)$ can be *perfectly scheduled* at time t if job $i(t)$ and all partially scheduled jobs in $S(t)$ can be completed on time.

It can be checked whether i can be perfectly scheduled in a constant time as follows. A temporary schedule $\sigma = \sigma(S)$, generated just for this checking purposes, keeps the track of the latest possible starting time of all uncompleted job portions of all the partially scheduled jobs from S in a feasible schedule (σ is an auxiliary schedule generated just for the testing purpose). The processor is assigned jobs (more precisely, their unscheduled in S parts) backwards so that they are completed on time as late as possible without overlapping with the earlier included job parts in σ. Due to ED-heuristic, each incoming job i has a deadline, smaller than that of any uncompleted preempted job in S (if there is such a job). It follows that the preempted uncompleted jobs in S are scheduled by the decreasing order of their deadlines. Symmetrically, the unscheduled portions of these jobs are included in the same order in σ, but backwards (from right to left).

Now we may "join" S with σ and see whether there occurs an intersection; if it does not occur, i can be perfectly scheduled, otherwise i cannot be perfectly scheduled. We describe this in more details below.

Initially, $\sigma = \emptyset$. The current scheduling time τ in σ is the starting time of the latest scheduled job in σ; initially, $\tau = +\infty$. Assume job $l \in S$ is to be interrupted by the incoming job i and we are to check whether i can be perfectly scheduled. Yet unscheduled part of job l is included in σ at time $d_l - p^*_l$ if $d_l \leq \tau$ (note that this will be the case if $\tau = +\infty$), otherwise we schedule l at time $\tau - p^*_j$ (recall that the deadline of each newly included job in σ is less than that of all the earlier included ones). We update current τ each time we schedule a job in σ. Whenever any preempted job from σ is completed, it is removed from the current σ and τ is respectively updated. Since the latest scheduled jobs in σ (the ones scheduled at earlier time moments in σ) will be completed first, such removal can cause no gap in σ. We will denote by S_+ the ED-schedule obtained from S by completing all partially scheduled jobs, as described above.

The next lemma immediately follows.

Lemma 1 *The incoming job $i(t)$ can be perfectly scheduled in $S(t)$ iff $\tau \geq t + p$ and $t + p \leq d_i$, and it takes a constant time to verify this.*

Now we are ready to give our algorithm. It basically verifies whether the next incoming job i can be perfectly scheduled. If yes, i is included, otherwise i is discarded.

```
ALGORITHM SINGLE_PROCESSOR

Initial Settings:

$S :=$ empty schedule;
$t :=$ the release time of the earliest arrived job

REPEAT

from all jobs released by time $t$ select job $i$ with
the earliest deadline (break ties by selecting the
partially scheduled job)

IF $i$ can be perfectly scheduled

THEN schedule $i$ in $S$ till the next job with a
smaller deadline arrives or $i$ is completed;
set $t$ to the completion/interaption time of $i$

ELSE {$i$ cannot be perfectly scheduled}
disregard job $j$ and all (current/future) jobs with
the same deadline

UNTIL there are no more jobs
```

The next lemma immediately follows from the fact that each job is perfectly scheduled in S:

Lemma 2 *Any partially scheduled job can be completed on time in S.*

Theorem 3 *Algorithm SINGLE_PROCESSOR finds an optimal schedule in time $O(n \log n)$.*

Proof. Let t be the earliest scheduling time such that the incoming job $i = i(t)$ cannot be perfectly scheduled. We first show that there is an optimal schedule in which all partially scheduled jobs in S are completely scheduled. To see this, assume l is a partially scheduled job in S and we apply the ED-heuristic to the jobs from $S \backslash l$. In the resulting schedule either there will arise a new gap or (and) another job with a no smaller (the same) processing time and with a no smaller deadline will occupy the liberated by job l time interval(s). It is straightforward to see that this cannot lead us to an increased number of the on time scheduled jobs.

Job i cannot be perfectly scheduled because of one, or both of the following reasons. (1) $t + p > d_i$ or/and (2) $t + p > \tau$. By ED-heuristic, in case (1) there is no job $j \in S$ scheduled after time r_i with $d_j > d_i$. Hence i cannot be completed on time without the removal of some job scheduled at or after time r_i. Obviously, such a schedule alteration cannot lead us to a greater number of included jobs.

In case (2), To complete on time all partially scheduled jobs, either (i) job i together with all unscheduled jobs with the same deadline have to be disregarded or (ii) a job from S has to be removed.

We show that option (i) will always work. Note that with this option all partially scheduled jobs in S will be scheduled on time (since the previous incoming job was perfectly scheduled). If the current block B can be completed without disregarding any other job, the number of on time jobs scheduled by the completion time of that block is clearly maximal (the disregarding of job i raised no gap between the current scheduling time t and the completion time of B). Otherwise, assume that the incoming job $j = i(t')$ with $d_j > d_i$ cannot be perfectly scheduled at the scheduling time $t' \geq t$ in B. Notice that by the disregarding job i at time t we have shifted to the left by p all successively scheduled jobs in B (observe that no gap may occur before the completion of B). On the other hand, the removal of any other job of S at time t could have shifted these jobs of B by no more than p. Then we would be forced to remove some other job before or at point t' ("instead" of j). Let τ' be the completion time of the latest scheduled job in $S(t')_+$. Then it follows that by time τ', any feasible schedule can contain no more on time scheduled jobs than $S(t')_+$ does, and we can complete the proof by repeatedly applying the same reasoning up to the end of the current block.

Since whether next incoming job can be perfectly scheduled or not can be checked in a constant time (Lemma 1, the time complexity of the algorithm is the same as that of the ED-heuristics, i.e., $O(n \log n)$. □

3 Multiprocessor algorithm

3.1 Some additional preliminaries

Unlike the single-processor algorithm, for multiprocessor case the next incoming job i which cannot be perfectly scheduled not necessarily is disregarded: we may include i into the current schedule $S = S_t$ at the expense of moving some already scheduled jobs (with no less deadline) still keeping these jobs scheduled on time.

Let us first describe the multiprocessor version of the ED-heuristic (MED-heuristic) that we employ. A new incoming job i is again considered at each scheduling time t defined now as follows. Initially, when all processors are idle, $t := \min\{r_j | j \in J\}$. Iteratively, among all unscheduled jobs released by time t, a job i with the minimal d_i is determined. If there is an idle processor at time r_i or (and) there is a processor handling at moment r_i a job with a deadline, greater than d_i then $t := r_i$ (in the former case ties are broken by scheduling i on the processor with the minimal index; in the latter case, ties are broken by scheduling i on the processor with the minimal index among the processors handling the job with the maximal deadline). Job i is schedules at time t on the selected processor. If neither of the above two conditions hold, no new job is scheduled at the current time, t is set to the next to r_i minimal job release time and the checking is repeated until one of the conditions hold. The MED-heuristic stops when no unscheduled job is left (as before, i and S stand for the next incoming job and the current schedule, respectively).

Recall that we were able to check in a constant time whether each incoming job can be perfectly scheduled for the single-processor case. This checking becomes essentially more complicated for the multiprocessor case. A straightforward extension of the single-processor procedure does not lead to an optimal strategy. In particular, such a procedure may fail to include $i(t)$ but it still might be possible to include $i(t)$ perfectly, i.e., there may exist a partial schedule with job $i(t)$ and all the partially scheduled jobs feasibly included. We were not able yet to derive a reasonable polynomial time procedure for verifying whether $i(t)$ can be perfectly scheduled. Instead, we use the following $O(m)$ procedure INCLUDE(i) for deciding whether to include $i(t)$. If INCLUDE(i) returns a "yes" answer then it succeeds in accommodation of job $i(t)$ feasibly; otherwise, it gives a "no" answer in which case $i(t)$ might be discarded. INCLUDE(i) may return a "no" answer but it still might be to schedule $i(t)$ perfectly. That is why our multiprocessor algorithm produces sub-optimal solutions.

INCLUDE(i) is as follows. A temporary schedule $\sigma = \sigma(S)$, generated as in the single-processor case just for the checking purposes, keeps the track of the latest possible starting times of the uncompleted job portions of all partially scheduled jobs from S in any feasible schedule (again, σ is just a fictitious schedule generated for the testing purpose, i.e., a job (portion) is not actually scheduled while included in σ). The processors are assigned jobs (more precisely, their unscheduled in S parts) backwards so that they are completed on time as late as possible without overlapping with the earlier included jobs in σ. Because of the MED-heuristic, each incoming job i has a deadline, smaller than that of any uncompleted preempted job in S (if there exists such a job). It follows that the preempted uncompleted jobs in S are scheduled by the non-increasing order of their deadlines. Symmetrically, the unscheduled portions of these jobs will be included in the same order in σ, but backwards (from right to left, or from up to down).

Initially, $\sigma = \emptyset$. Assume job $l \in S$ is to be interrupted by the incoming job i and we are to check whether i can be perfectly scheduled. Yet unscheduled part of job l is included in σ at the latest possible time so that l finishes by its deadline and it does not overlap with any job currently in σ; if there is more than one processor on which this time is reached, ties are broken by scheduling l on the processor with the minimal index. Whenever any preempted job from σ is completed, it is removed from the current σ. Since the latest scheduled jobs in σ (the ones scheduled at earlier time moments in σ) will be completed first, such a removal can cause no gap in σ. Hence, since $d_l \leq d_j$ for any $j \in \sigma$, l will be completed no later than any earlier included job in σ.

Let us call the *reversed completion time* of a processor in σ the starting time of the latest scheduled job on that processor. The processors in σ and S are "matched" as follows. A processor with the minimal reversed completion time in σ is matched with a processor with the maximal completion time in S, a processor with the next minimal reversed completion time in σ is matched with a processor with the next maximal completion time in S, and so on, a processor with the maximal reserved completion time in σ is matched with a processor with the minimal completion time in S (ties are broken arbitrarily). This matching defines an order on the processor couples. Note that the overlapping on the first couple of the matched processors can be no less than that on the second couple, in general, the overlapping on the $k-1$th couple of processors can be no less than that of the kth couple.

Note that if there occurs no overlapping in time on the first couple of the matched processors then obviously i can be perfectly scheduled in S (note that in this case no job from σ starts earlier than at its release time). At the same time, i cannot be perfectly scheduled if there occurs the overlapping on the last couple of processors. For the intermediate cases when there occurs the overlapping on the first μ ($1 < \mu < m$) couple of processors, INCLUDE(i) will still return a "no" answer (but it might be the case that i can be perfectly scheduled).

3.2 The algorithm

Let us call an incoming job i *deserted* if $r_i < t_S$ and it INCLUDE(i) ="no". It follows that no part of i could have been earlier scheduled in S, hence $t_S + p > d_i$. Likewise, there can be no interrupted and uncompleted (by time t) job in S. Since $r_i < t$, job i could have been potentially scheduled earlier at the expense of the shifting some job(s) from S to the right. Next we define such jobs.

Let us call a job processed in S within the time interval $[r_i, t_S]$ and completed strictly before its deadline (i.e., at least one time unit before) a *standby* job for i. Potentially, standby jobs and only these can be rescheduled later in favor of the deserted job i that might be possible to insert into the released time intervals within $[r_i, t_S]$. The lower and upper limits, respectively, for the number of the standby jobs for i is 0 and the total number of jobs processed in S within the time interval $[r_i, t_S]$, respectively. Notice that we can have no gap in S within this time interval, since otherwise job i would have been considered by the MED-heuristic at the beginning of such a gap (which, as we have already noted, cannot be the case). Thus if we wish to shift to the right a standby job without delaying successively scheduled jobs in S then it can only be rescheduled within the external gaps which arise after time t_S in S: an *external gap* on machine M in S is an open idle-time interval $(t_S(M), \infty)$ in M, where $t_S(M)$ is the completion time of machine M in S (note that $t = \min_M\{t_S(M)\}$). But since the first part of job i is to be inserted into the interval $[t, d_i]$, a standby job is to be rescheduled after the time moment t'_S, t'_S being the second element in the non-decreasing list of machine completion times in S. Hence, from now on, we require that the deadline of any standby job in S is greater than t'_S.

Let $\mathcal{S}$ be a schedule with a partially scheduled job i in it. The *deficit* of job i in $\mathcal{S}$, $def(i, \mathcal{S})$ is yet unscheduled part of job i in $\mathcal{S}$. Initially, $def(i, S') = t+p-d_j$, S' being the extension of S with job i scheduled into the internal $[t, d_i]$.

The procedure INSERT is called by the main algorithm MULTIPROCESSOR whenever the next incoming job i is deserted. Initially, INSERT assigns to job i the time interval $[t, d_i]$ (here and further ties are broken by selecting the processor with the minimal index). It inserts the rest of job i iteratively, portion-by-portion. The first part of i is inserted at the earliest time moment no less than r_i at which a standby job k is processed in S. A part of job k is extracted and rescheduled from the time moment t'_S on the corresponding processor and the equal part of job i is inserted into the released time interval. We denote the resultant schedule by $S_{(k,i)}$. The length of the rescheduled part of job k in $S_{(k,i)}$ is determined in such a way that it is completed at the maximal time moment τ with $\tau \le d_k$ and $\tau - t'(S) \le def(j, S')$. Let $c'_j(S_{(k,j)})$ be the completion time of the inserted portion of job i in $S_{(k,j)}$. If $c'_j(S_{(k,j)}) > t_S$, then INSERT returns with a "no" answer. Otherwise, if $def(j, S_{(k,j)}) > 0$, on the next iteration, another part of job i is inserted at the earliest time moment no less than $c'_j(S_{(k,j)})$ at which a standby job k' is processed in $S_{(k,j)}$; if there exists no k' INSERT returns again a "no" answer. Otherwise, INSERT returns with the earliest generated schedule in which the deficit of job i is 0 and the main procedure MULTIPROCESSOR is resumed (the next scheduling time is determined and the next incoming job is considered). Whenever INSERT returns "no" MULTIPROCESSOR disregards job i and all possible unscheduled jobs with the same deadline and considers the next incoming job (with the deadline more than d_i) at time t_S.

At each iteration in INSERT, ties are broken by selecting a standby job with the earliest starting time, called an *active standby job* (further ties are broken arbitrarily). In the description below k is an active standby job.

```
ALGORITHM MULTIPROCESSOR
FUNCTION INSERT($j, S$)
BEGIN {INSERT}
IF there exists no (active standby job) $k$ in $S$ or
$c'_j(S_{(k,j)}) > t_S$ THEN INSERT:="no";
        RETURN
ELSE IF $def(j, S_{(k,j)}) = 0$ THEN INSERT:= $S_{(k,j)}$;
RETURN ELSE INSERT($j, S_{(k,j)}$)
END {INSERT}
BEGIN {MULTIPROCESSOR}
$S := \emptyset$; $t_S := \min\{r_j | j \in J\}$;
**basic step:** From all jobs released by time $t_S$, select
job $i$ with the earliest deadline (break ties by selecting a
partially scheduled job)
IF INCLUDE($i$) ="yes"
 THEN schedule $i$ in $S$ until its full completion if no
job with a smaller deadline is meanwhile released, otherwise interrupt $i$ at the release time of the latter job;
$S := S_{+j}$
 ELSE {INCLUDE($i$) ="no"}
  IF $i$ is not deserted
   THEN disregard job $i$ and all (current/future) jobs
with the same deadline
   ELSE {$i$ is deserted} assign $i$ the time interval
$[t_S, d_j]$ in $S$; INSERT($j, S$)
      IF INSERT="no"
      THEN disregard job $i$ an all (current/future) jobs
with the same deadline
        ELSE $S$ :=INSERT
IF there is an unprocessed job left
THEN REPEAT **basic step** ELSE Stop
```

END. {MULTIPROCESSOR}

Theorem 4 *The time complexity of the algorithm MULTI-PROCESSOR is* $O(n^2)$.

Proof. The time complexity of MED-heuristic is $O(n \log n)$, that of INSERT(j, S) is $O(n)$ and INCLUDE(i) takes time $O(m)$. The number of scheduling times in obviously $O(n)$. Hence, the worst-case time complexity is $O(n \log n) + O(n)(O(n) + O(m)) = O(n^2)$ (assuming without the loss of generality that $m \leq n$).

Acknowledgement

The author is grateful to Wojciech Jawor for his useful discussions and comments.

References:

[1] P. Baptiste. ”An $O(n^4)$ algorithm for preemptive scheduling of a single machine to minimize the number of late jobs”. *Operations Research Letters* 24, 175-180 (1999)

[2] M. Chrobak, C. Durr, W. Jawor, L. Kowalik and M. Kurowski. “A note on scheduling equal-length jobs to maximize throughput”. *Journal of Scheduling 9*, 71-73 (2006)

[3] J.Du, J.Y. Leung and C.S. Wong. ”Minimizing the number of late jobs with release time constraint”. *Journal of Combinatorial Mathematics and Combinatorial Computing* 11, 97-107 (1992)

[4] E.L. Lawler. ”A dynamic programming algorithm for preemptive scheduling of a single machine to minimize the number of late jobs”. *Annals of Operations Research* 26, 125-133 (1990)

[5] E.L. Lawler. ”Knapsack-like scheduling problems, the Moore-Hodgson algorithm and the tower of sets property”. *Mathematical Computer Modelling* 20, 91-106 (1994)

[6] J.M. Moore. ”An n job, one machine sequencing algorithm for minimizing the number of late jobs.” *Management Science* 15, 102-109 (1968)

[7] N.Vakhania. “Scheduling unit-length jobs to minimize the number of late jobs on a single-machine”. *Proceeding of 7th Workshop on Models and Algorithms for Planning and Scheduling Problems MAPSP 2005*, 273-276 (2005).