Minimizing maximum lateness of jobs with naturally bounded job data on a single machine in polynomial time

Nodari Vakhania  
Facultad de Ciencias, UAEMor, Mexico  
Inst. of Computational Math., Georgia  
Email address: nodari@uaem.mx

Frank Werner  
Fakultät für Mathematik, Otto-von-Guericke-Universität  
PSF 4120, 39106 Magdeburg, Germany  
Email address: frank.werner@ovgu.de

Abstract: We consider the problem of scheduling jobs with given release times and due dates on a single machine to minimize the maximum job lateness. It is NP-hard and remains such if the maximal job processing time is unrestricted and there is no constant bound on the difference between any job release times. We give a polynomial-time solution of the version in which the maximal job processing time and the differences between the job release times are bounded by a constant, which are restrictions that naturally arise in practice. Our algorithm reveals the inherent structure of the problem and also gives conditions when it is able to find an optimal solution unconditionally.

Keywords: algorithm, scheduling single-machine, release time, lateness, binary search

1 Introduction

One of the main drawbacks in finding an optimal solution for a considerable part of combinatorial optimization problems are item parameters that may take arbitrary values. In practice however, often the arbitrary parameters do not occur. Probabilistic algorithms achieve good (average-case) performance by assuming some probabilistic distribution on item parameters. In the deterministic setting, it is natural to study types of restrictions on problem parameters (reasonable from the practical point of view) that may lead to an efficient solution method. This paper considers such a situation presenting an efficient solution for a strongly NP-hard scheduling problem. In that problem, the job processing times and the differences between their release times are bounded from above by a constant. Indeed, in practice, there always
exists such a bound.

In our single-machine scheduling problem, every job is characterized by its processing time, its release time and its due date (the two latter extra parameters make the problem complicated). With arbitrary job processing times, the feasibility problem where we are asked to find a schedule in which all jobs complete by their due dates is strongly NP-complete. This was shown in the late 70-s in Garey & Johnson [2], where a reduction from the 3-partition problem is used. A bit later, a very sophisticated $O(n \log n)$ algorithm for the equal-length version of this problem was proposed in Garey et al. [5].

An algorithm for the feasibility problem can be used for the corresponding minimization problem, in which we aim to minimize the maximum job lateness $L_{\text{max}}$ (the difference between its completion time and its due date): iteratively, we increase the due dates of all jobs until we find a feasible schedule with these modified due dates. Since the min-max job lateness obtained in this way depends on $p_{\text{max}}$ (the maximal job processing time) and $n$, we may need to apply a feasibility algorithm $np_{\text{max}}$ times. However, we can reduce the cost to $O(\log(n p_{\text{max}}))$ using a binary search.

Using the standard notation by Graham et al. [3] for scheduling problems, the above minimization problem and its equal-length restriction are abbreviated as $1/r_j/L_{\text{max}}$ and $1/p_j = p, r_j/L_{\text{max}}$, respectively. Besides the earlier mentioned algorithm for equal-length jobs, there are a few known polynomial special cases of the problem $1/r_j/L_{\text{max}}$. For example, without release times, it is easy to see that scheduling the jobs in order of non-decreasing due dates gives an optimal schedule in $O(n \log n)$ time, see Jackson [6]. Similarly, if all $d_j$ values are equal, then scheduling the jobs in order of non-decreasing release times is optimal. Hoogeveen [4] has established another polynomially solvable special case, in which the difference between every job due date and release time is tied with the processing time of that job in a special way. As it was shown, this version can be solved in $O(n \log n)$ time.

Other polynomially solvable special cases also involve restrictions on job processing times. In particular, the version $1/p_j \in \{p, 2p\}, r_j/L_{\text{max}}$ with only two processing times $p$ and $2p$ for an integer $p$ is solvable in time $O(n^2 \log n \log p)$ [11]. For equal-length jobs but on $m$ identical machines, problem $P/p_j = p, r_j/L_{\text{max}}$ can be solved in time $O(mn^2)$ [10]. (We refer the interested reader to [7] for a recent overview on single and parallel machine problems with equal job processing times.) Quite recently, another version when the job processing times are mutually divisible, abbreviated by $1/p_j : \text{divisible}, r_j/L_{\text{max}}$, was also shown to be polynomially solvable [12].

### 1.1 Intractability vs practicality

In the light of the results, it is natural to think about a generalization of the problem $1/p_j : \text{divisible}, r_j/L_{\text{max}}$ to the version in which the job processing times are allowed to take values from the set $\{p, 2p, 3p, \ldots\}$. It is not difficult to see that even the feasibility version of this problem is NP-complete. For this version but with arbitrary job processing times, Garey & Johnson [2] gave a reduction from the PARTITION problem showing that it is NP-complete. If in this transformation, we multiply the derived data for the corresponding scheduling problem
by p, we obtain another polynomial-time reduction from PARTITION to the version with times \{p, 2p, 3p, \ldots\}. Furthermore, this version is strongly NP-complete whenever p has a polynomial-time dependence on \(d_{\text{max}}\), i.e., when it is not an exponent of \(d_{\text{max}}\). This follows from another transformation in [2]. The latter transformation is from the strongly NP-hard 3-PARTITION problem. Similarly, multiplying the data of the derived scheduling instance by p, we obtain a reduction from 3-PARTITION to the problem with unrestricted \(k\). This transformation is pseudo-polynomial if p is a polynomial of \(d_{\text{max}}\), which shows the strong NP-hardness of this version. We note that this restriction on p is quite natural from a practical point of view, as it is highly unlikely that a job may need a processing time which is an exponent of its due date.

Based on these NP-completeness results, it is unlikely that our problem can efficiently be solved without bounding \(p_{\text{max}}\). Likewise, the above transformation from the PARTITION problem in [2] essentially uses the fact that there is one “separating” job the release time of which is “appropriately far away” from the release times of all other jobs: the difference between these times is dependent on \(n\). Similarly, in the above transformation from the 3-PARTITION problem, there are also such jobs. Hence, it is also unlikely that the problem can polynomially be solved without restricting also the difference between the job release times.

Due to the above observations, in the setting that we solve here, the only restriction on the job processing times is that \(p_{\text{max}}\) is bounded from above by a polynomial function on \(n, P(n)\); whereas the differences between the release times of any two jobs are bounded by a constant \(R\), i.e., they do not depend on \(n\). If \(P(n)\) is a constant or it is \(O(n)\), then the time complexity of our algorithm is \(O(n^2 \log n \log p_{\text{max}})\); for \(k \geq 2\), the time complexity is \(O(n^{k+1} \log n \log p_{\text{max}})\) when \(p_{\text{max}} = O(n^k)\).

Adopting the standard three-field notation, we abbreviate this problem as \(1/p_{\text{max}} < P(n), |r_j - r_i| < R/L_{\text{max}}\). One may expect our restrictions to be satisfied in most of the practical applications. Indeed, whenever a concrete practical (NP-hard) problem is concerned, in any case for carrying out the computational experiments, job data are drawn from a fixed interval. For example, for scheduling problems job processing times are taken from some interval \([p_{\text{min}}, p_{\text{max}}]\). The same observation applies to other problem parameters. In addition, in practice, problem parameters are naturally restricted (otherwise one cannot really afford to complete the task). For example, in scheduling problems with job release times, one usually considers a planning horizon for a set of jobs (e.g. a day, a week, a month); hence, the differences between job release times are naturally limited.

Because of these above considerations, one may think of \(1/p_{\text{max}} < P, |r_j - r_i| < R/L_{\text{max}}\) as a naturally restricted (from the practical point of view) version of (the strongly NP-hard) problem \(1/r_j/L_{\text{max}}\) with the bounded data. As long as the theoretical analysis is of interest, if we remove from our setting the restriction on \(p_{\text{max}}\), then our algorithm becomes pseudo-polynomial, whereas without the restriction on the job release times it becomes exponential. This is not surprising as by relaxing any of these two restrictions the problem becomes hard.
1.2 A brief look on the algorithm

Our algorithm is a direct combinatorial one and applies binary search, reducing the problem to some version of the bin packing problem (an approach similar to the one used for the problem $1/p_j : \text{divisible}, r_j/L_{\text{max}}$ in [12]). It reveals the inherent structure of the problem and gives the conditions when it is able to find an optimal solution for the general setting $1/r_j/L_{\text{max}}$. On each iteration of the binary search, the algorithm uses the behavior alternatives for the structural analysis of the solutions it creates. Intuitively, these alternatives characterize the “crucial” structural properties of these solutions (the behavior alternatives were introduced in [10] and have been used in various direct combinatorial algorithms called blesscmore algorithms, see [13] for a general description of these algorithms).

On the basis of the analysis of the behavior alternatives we claim the desired schedule properties, which lead us to an optimal decision. We distinguish two basic kinds of jobs, non-critical and critical ones. The non-critical jobs may be flexibly moved within a feasible schedule, whereas the critical jobs form tight sequences (called kernels), in the sense that the delay of the earliest scheduled job from the subset cannot exceed some precalculated parameter between (including) 0 and $p_{\text{max}}$.

We first define the initial set of kernels applying the earlier mentioned Jackson’s heuristic to the original problem instance. Then we determine lower and upper bounds on the allowable delay for each kernel and carry out the binary search within the corresponding interval. Each new derived value $\delta$ defines the maximal currently allowable delay for the kernels. For each $\delta$, we aim to distribute the non-kernel jobs in order to fill in the intervals in-between the kernels (the bins) “as much as possible”, so that no non-kernel job causes a lateness more than that of a kernel job. The minimum of such a value $\delta$ yields an optimal schedule.

In this way, the problem $1/r_j/L_{\text{max}}$ reduces to an optimal distribution of the non-kernel jobs within the bins. Our two restrictions in the problem $1/p_{\text{max}} < P(n), |r_j - r_i| < R/L_{\text{max}}$ conveniently bound the total number of job rearrangements which may potentially lead to an optimal solution. This, in turn, results in a polynomial-time behavior of our algorithm. Besides the earlier indicated polynomial time bound, the algorithm has an alternative pseudo-polynomial bound of $O(d_{\text{max}} n \log n \log p_{\text{max}})$ ($d_{\text{max}}$ is the maximum job due date) when $P(n) = O(n)$. For $k \geq 2$ with $P(n) = O(n^k)$, the latter estimation converts to $O(d_{\text{max}} n^k \log n \log p_{\text{max}})$ (and with bounded $d_{\text{max}}$ to $O(n^k \log n \log p_{\text{max}})$).

The paper is organized as follows. In the next section, we describe some common basic concepts for our blesscmore algorithm. In Section 3, we define basic parameters for the binary search procedure. In Section 4, we give the definitions, necessary for our procedure that verifies the existence of a feasible schedule in which the lateness of no job is more than the maximal allowable lateness defined by the current trial $\delta$ of the binary search procedure. We also give the conditions when our algorithm finds an optimal schedule for our general problem. Section 5 exclusively studies the cases when these conditions do not hold. Final remarks on the overall algorithm are given in Section 6.
2 Basic background

As earlier noted, the basic notions and framework for our algorithm, which, for the completeness of the presentation, are described in this and the following sections, have been introduced in [12].

The problem definition. Let us first state formally our problem in its general setting. We have \( n \) jobs \( \{1, 2, \ldots, n\} \) which are to be performed by a single machine. Each job \( j \) has three non-negative integer parameters, the processing time \( p_j \) which is the number of times units that \( j \) needs on the machine, the release time \( r_j \) which is the time moment when \( j \) becomes available, and the due date \( d_j \), the time moment by which it is desirable to complete \( j \). A feasible schedule \( S \) is a mapping that assigns to each job \( j \) a starting time \( t_j(S) \), such that \( t_j(S) \geq r_j \) and \( t_j(S) \geq t_k(S) + p_k \), for any job \( k \) included earlier into \( S \) (for the notational simplicity, we use \( S \) for both, a schedule and the corresponding job set). The first inequality says that a job cannot be started before its release time, and the second one reflecting the restriction that the machine can handle only one job at any time moment. \( c_j(S) = t_j(S) + p_j \) is the completion time of job \( j \).

We may have two objectives. First, we may wish to know if there is a schedule which meets all job due dates, i.e., every job \( j \) is completed by time \( d_j \). If there is no such schedule, we still wish to find an optimal schedule, i.e., one minimizing the maximum lateness \( L_{\text{max}} = \max_j \{ c_j - d_j \} \) (the difference between the actual completion time and the due date of the job). We denote by \( L(S) \) (\( L_j(S) \), respectively) the maximum lateness in \( S \) (the lateness of job \( j \) in \( S \), respectively).

Jackson’s heuristic. Now we describe in detail Jackson’s heuristic which is our main tool for the schedule generation, commonly referred to as the Earliest Due-date Heuristic (from now on, abbreviated as ED-H). ED-H distinguishes \( n \) scheduling times, the time moments at which a job is assigned to the machine. Initially, the earliest scheduling time is set to the minimum job release time. Among all jobs released by that time a job with the minimum due-date is assigned to the machine (ties being broken by selecting a longest job). Iteratively, the next scheduling time is either the completion time of the latest assigned so far job to the machine or the minimum release time of a yet unassigned job, whichever is more (as no job can be started before the machine gets idle neither before its release time). And again, among all jobs released by this scheduling time a job with the minimum due-date is assigned to the machine. Observe that the ED-H creates no gap that can be avoided always scheduling an already released job once the machine becomes idle, whereas among the yet unscheduled jobs released by each scheduling time it gives the priority to a most urgent one.

Our first feasible schedule, \( \sigma \), is obtained by applying ED-H to the original problem instance. We shall generate other feasible schedules with the same heuristic applied to some modified problem instances.

An ED-schedule may contain a gap, which is its maximal consecutive time interval in which the machine is idle. For convenience, we assume that there occurs a 0-length gap \( (c_j, t_i) \) if job \( i \) starts at its release time immediately after the completion of job \( j \).
A block in an ED-schedule is its consecutive part consisting of the successively scheduled jobs (without any gap in between), which is preceded and succeeded by a (possibly a 0-length) gap.

The kernels. As briefly noted in the introduction, in and ED-schedule $S$, we partition the whole set of jobs in two kinds of subsets, non-critical and critical. The non-critical subsets contain jobs that might be flexibly moved within our current feasible schedule, whereas the critical sets contain the subsets of jobs which form rigid sequences. Now we define critical and non-critical jobs formally.

We call a kernel a maximal (consecutive) job sequence in $S$ ending with job $o$ with $L_o(S) = \max_i \{L_i(S)\}$, such that no job from this sequence has a due-date more than $d_o$; job $o$ is called an overflow job in $S$. For a kernel $K$, we let $r(K) = \min_{i \in K} \{r_i\}$.

Note that since a kernel is a maximal job sequence possessing the above properties, it may not be immediately followed by another job realizing the value of the maximum lateness in $S$ (though the overflow job $o$ may be preceded in the kernel by another job $j$ with $L_j(S) = \max_i \{L_i(S)\}$). Note also that there may exist no gap within any kernel and that every kernel is contained in some block (whereas a block may contain more than one kernel); the total number of kernels in $S$ equals to the number of the overflow jobs in it.

Observation 1 ED-schedule $\sigma$ is optimal if it contains a kernel with its earliest scheduled job starting at time $r(K)$.

Proof. Let $K$ be a kernel in $\sigma$ with this property. Since all jobs in $K$ are no less urgent than the overflow job $o$ and $K$ contains no gap, no reordering of jobs in $K$ can reduce the current maximum lateness, which is $L_o(S)$. Then, since the earliest scheduled job in $K$ starts at its release time, there is no feasible schedule $S$ with $L(S) < L_o(\sigma)$ and $\sigma$ is optimal.

Emerging jobs. If the condition in the above observation does not hold, then there must exist a job, less urgent than $o$, scheduled before all jobs in $K$ that delays jobs in $K$ (the overflow job $o$). By rescheduling such a job to a later time moment behind $K$ the jobs in $K$ can be restarted earlier. We need some extra definitions to define this operation formally.

Suppose job $i$ precedes job $j$ in $S$. We will say that $i$ pushes $j$ in $S$ if ED-H may reschedule job $j$ earlier, under the restriction that it does not include job $i$ before job $j$.

Since the earliest scheduled job of kernel $K$ does not start at its release time (see Observation 1), it is immediately preceded and pushed by a job $l$ with $d_l > d_o$. In general, we may have more than one job scheduled before kernel $K$ (in the block containing $K$) pushing the earliest scheduled job of $K$. We call such a job an emerging job for $K$ (or for job $o$), and we call the latest scheduled one (job $l$ above) the delaying emerging job.

We shall later refer to $j$, with $d_j > d_o$ and $r_j < r(K)$, scheduled after $K$ in $S$ as a passive emerging job for kernel $K$ (or for job $o$) in $S$. Note that a passive emerging job is scheduled in the same block to which $K$ belongs, i.e., all emerging jobs and passive emerging jobs belong to the same block in $S$.

In the future, we shall refer to the first above kind of emerging jobs (including a delaying
emerging job) as just an emerging job, whereas we shall explicitly specify when if we mean a delaying or a passive emerging job.

Aiming in restarting the kernel jobs earlier, we may activate an emerging job \( e \) for \( K \); that is, we force \( e \) and all passive emerging jobs to be rescheduled after \( K \) (all these jobs are said to be in the state of activation for \( K \)). This we achieve by increasing the release times of all these jobs to a sufficiently large magnitude, say \( r(K) \), so that when ED-H is newly applied, neither job \( e \) nor any passive emerging job will surpass any kernel job, and hence the earliest job in \( K \) will start at time \( r(K) \). (We note that the same emerging job may be activated for two or more successive kernels.)

3 Defining the boundaries

Consider an auxiliary ED-schedule \( \sigma^* \) in which the delaying job of every kernel \( K \in \sigma \) is just omitted (\( \sigma^* \) is not a complete schedule and it is considered solely for the purpose of the following definitions). Let \( L_i^* \) be the new (reduced) value of the lateness of each kernel job \( i \) in \( \sigma^* \). Since every \( K \) is restarted at time \( r(K) \) in \( \sigma^* \), \( L^*(K) = \max_{i \in K} \{ L_i^* \} \), and hence \( L^* = \max_\sigma \{ L^*(K) \} \) are lower bounds on the objective value (see Observation 1).

Observation 2 Every \( K \) can be delayed by \( \delta(K) = L^* - L^*(K) \) without increasing the maximum lateness.

Proof. Let \( K' \) be a kernel that realizes \( \max_\sigma \{ L^*(K) \} \). From the definition of \( \delta(K) \), by augmenting the completion time of every job in \( K' \neq K' \) by \( \delta(K) \), none of the jobs in \( K \) will be completed later than a job realizing \( \max_{i \in K' \setminus K} \{ L_i^* \} \). This proves the observation.

By Observation 2, we may restrict our attention to schedules in which every kernel \( K \) starts either no later than at time \( r(K) + \delta(K) \) or is delayed by some \( \delta \geq 0 \) (the latter is due to the necessity of a proper accommodation all non-kernel jobs). Let \( \Delta = L_\sigma(\sigma) - L^* \), where \( o \) is an overflow job in \( \sigma \). Note that the maximum lateness in any feasible ED-schedule in which the delay of some kernel is more than \( \Delta \) is no less than that in \( \sigma \). Hence, the above threshold value \( \delta \) in our binary search (the extra delay for each kernel) can obviously be derived from the interval \([0, \Delta]\).

At the iteration in the binary search procedure with the threshold value \( \delta \), we refer to the magnitude \( L^* + \delta \) as the \( \delta \)-boundary. Job \( j \in S \) is said to surpass the \( \delta \)-boundary in \( S \) if \( L_j(S) > L^* + \delta \). We respectively redefine the delaying job for a kernel \( K \) in \( S \) as one that completes after time \( r(K) + \delta(K) + \delta \). We denote by \( K_\delta \) the current set of kernels corresponding to \( \delta \) (as we will see later, the initial kernel set in \( \sigma \), \( K_\Delta \), may be completed with new kernels as \( \delta \) in the binary search procedure becomes smaller).

As we will see in the next section, it is not difficult to maintain kernel jobs scheduled so that no kernel job surpasses \( \delta \)-boundary. At the same time, this task is more complicated for the non-kernel jobs. We call \( S \) a \( \delta \)-balanced schedule if no job (neither kernel, nor non-kernel) surpasses the \( \delta \)-boundary in \( S \) (i.e. \( L(S) \leq L^* + \delta \)). We denote a \( \delta \)-balanced schedule by \( S(\delta) \) (observe that \( \sigma = S(\Delta) \)).
On the way of the construction of \( S(\delta) \), we first define the intervals within which non-kernel jobs are to be included. On the iteration in the binary search procedure with the threshold value \( \delta \), we shall refer to the interval before each \( K \in \mathcal{K}_\delta \) up to the time moment \( r(K) + \delta(K) + \delta \) as the bin defined by \( K \) and denote it by \( B_K(\delta) \) (for notations implicitly, we may omit argument \( \delta \) in \( B_K(\delta) \)); a non-idle time interval after the latest kernel of \( \mathcal{K}_\delta \) is our last bin. The length of this bin is unrestricted subject to the condition that no non-kernel job scheduled in that bin surpasses the \( \delta \)-boundary.

Recall that in our auxiliary schedule \( \sigma^* \) we have just omitted each delaying job. There is an easy way to accommodate the delaying jobs omitted in \( \sigma^* \) within the bins: just activate each delaying job for the corresponding kernel. However, the resultant schedule may not be \( \delta \)-balanced: “too long” gaps might be left before some kernels, and not “enough” space may remain for the accommodation of all the activated jobs together with the rest of the bin jobs. Hence, the gaps before the kernels are to be used beneficially.

In the next section we describe how we construct a \( \delta \)-balanced schedule, for each \( \delta \) in the binary search, or establish that it does not exist. Our binary search procedure can be easily described. On its initial iteration \( \delta = \Delta \) and \( \sigma = S(\Delta) \) is generated. The next value for \( \delta \) is 0; if there exists no \( S(0) \) then the next value of \( \delta \) is \( \lfloor \Delta/2 \rfloor \). In general \( \delta \) is derived from the interval \([0, \Delta]\), whereas the change from larger to smaller value of \( \delta \) is carried out if a \( \delta \)-balanced schedule for the current \( \delta \) was successfully created; otherwise, \( \delta \) is increased respectively on the next iteration.

**Observation 3** \( S(\delta) \) with minimal possible \( \delta \) is optimal.

Proof. The minimal \( \delta \) yields the minimal possible lateness for the kernel jobs subject to the condition that no non-kernel (bin) job surpasses \( \delta \)-boundary. This clearly proves our claim. \( \square \)

Thus, all we need to do is to find the minimal \( \delta \) for which there exists a \( \delta \)-balanced schedule. For this, we apply binary search. At its first iteration \( \sigma = S(\Delta) \) with \( \delta = \Delta \) is generated. The next value for \( \delta \) is 0; if there exists no \( S(0) \), then the next value of \( \delta \) is \( \lfloor \Delta/2 \rfloor \). In general \( \delta \) is derived from the interval \([0, \Delta]\), whereas the change from a larger to a smaller value of \( \delta \) is carried out if a \( \delta \)-balanced schedule for the current \( \delta \) was successfully created; otherwise, \( \delta \) is increased accordingly at the next iteration.

## 4 Basics for verifying the existence of \( S(\delta) \)

Based on the results from the previous section, the solution of \( 1/r_j/L_{\max} \) is reduced to a procedure that either constructs a \( S(\delta) \) or asserts that it does not exist. In the rest of this paper, we describe this procedure called SEEK(\( S(\delta) \)), where \( \delta \) is the threshold value on the corresponding iteration of the binary search procedure. Let \( \delta' > \delta \) be the smallest threshold value encountered so far by the binary search procedure with an existing \( S(\delta') \). Procedure SEEK(\( S(\delta) \)) sets initially \( \mathcal{K}_\delta \) to \( \mathcal{K}_{\delta'} \). As SEEK(\( S(\delta) \)) advances in forming a complete schedule, \( \mathcal{K}_{\delta'} \) may be completed by new kernels.

SEEK(\( S(\delta) \)) applies a slightly modified version of ED-H while seeking for \( S(\delta) \). Recall from
the previous section that each \( K \in \mathcal{K}_\delta \) is to be started no later than at time \( r(K) + \delta(K) + \delta \). If the next job selected by ED-H completes by time \( r(K) + \delta(K) + \delta \), it is scheduled the next; otherwise, among the available jobs, one with the minimal due-date is similarly selected, until no released job fits into the bin \( B(K) \) (within still available interval before time moment \( r(K) + \delta(K) + \delta \)).

It follows that no job from \( K \) will surpass the \( \delta \)-boundary if \( \text{SEEK}(S(\delta)) \) has started the earliest job of \( K \) at moment \( r(K) + \delta(K) + \delta \). Suppose the earliest job of \( K \) starts strictly earlier at moment \( r(K) + \delta(K) + \delta - \epsilon \), for an integer \( \epsilon \geq 1 \). Then, when scheduling jobs in \( K \) by ED-H, there may occur a time moment at which some kernel job completes but no yet unscheduled job from \( K \) is released. Hence, ED-H may include some “external” job (one, not in \( K \)) at that (or later) moment. Roughly, such a job cannot be too long as otherwise, some kernel jobs will surpass the \( \delta \)-boundary.

\( \text{SEEK}(S(\delta)) \) imposes a restriction on the total length of such external jobs that might be included in between the jobs of \( K \). Clearly, it cannot exceed \( \epsilon \). In particular, while ED-H is scheduling the jobs of \( K \), consider any time moment \( t \) at which no job from \( K \) is yet released (this may happen because the earliest job of \( K \) now is restarted earlier), neither there is other “sufficiently short” (external) job released. Since no “long” external job can be included without forcing some job in \( K \) to surpass the \( \delta \)-boundary, we need to impose a gap starting at moment \( t \) until the earliest time moment at which either some (yet unscheduled) job in \( K \) is released or a sufficiently short external job is released.

Thus if \( S(\delta) \) exists, there occurs the above gap in between the jobs of \( K \) in it. Note that the total length of the non-kernel jobs that may further be included within the jobs of \( K \) is now restricted by \( \epsilon \) minus the total length of the already included external jobs minus the length of the newly arisen gap. If no gap at time \( t \) occurs (i.e., \( t \) is such that a sufficiently short external job is released at that time) then the latter magnitude is not subtracted. This update on the currently allowable total length for the external jobs is carried out at every scheduling time on which no (yet unscheduled) job from \( K \) is released.

The above modified ED-H is applied in \( \text{SEEK}(S(\delta)) \) for scheduling kernel jobs. The following observation is apparent from the above discussion.

**Observation 4** Any gap imposed in \( \text{SEEK}(S(\delta)) \) by the modified ED-H in between the jobs of any kernel \( K \) also occurs in \( S(\delta) \), whenever it exists. Furthermore, no kernel job in \( S \) may surpass the \( \delta \)-boundary in \( \text{SEEK}(S(\delta)) \) (given that the earliest job of every \( K \in \mathcal{K}_\delta \) starts no later than at time moment \( r(K) + \delta(K) + \delta \)).

It follows that, if \( \text{SEEK}(S(\delta)) \) has encountered a job surpassing the \( \delta \)-boundary, then it is a bin job. Before we describe how we deal with bin jobs surpassing the \( \delta \)-boundary, we introduce our two basic behavior alternatives and describe how \( \text{SEEK}(S(\delta)) \) complements the current set of kernels with a new kernel.

**Instances of alternative (b1).** Assume that \( K \) was the latest scheduled kernel from \( \mathcal{K}_\delta \) when there has occurred a non-kernel job \( j \) surpassing the \( \delta \)-boundary. If \( j \) is an ex emerging job (one activated for \( K \) or/and some preceding kernel(s)), then we will say that an *instance of alternative* (b1) (IA(b1)) with job \( j \) occurs.
Defining new kernels. Suppose that \( j \) is not an existing emerging job. Then there must exist an activated (existing emerging) job \( i \) that pushes job \( j \) (as \( j \) was not an overflow job in the previous iteration of the binary search). Suppose that among such \( i \)-s, there is an emerging job for job \( j \), and \( e \) is the latest scheduled one. Either (i) \( e \) was included before \( K \), or (ii) it was included after \( K \).

In both cases above, \( \text{SEEK}(S(\delta)) \) defines a new kernel. In case (i), this new kernel is formed by the jobs that were included after job \( e \) including the jobs in \( K \) and job \( j \). In case (ii), the new kernel is formed by the sequence of the jobs in-between the jobs \( e \) and \( j \) (including job \( j \) but not including job \( e \)).

\( \text{SEEK}(S(\delta)) \), updating the current \( \mathcal{K}_\delta \) with the new kernel (in each case correspondingly), (re)schedules the jobs from this kernel (respecting the earlier described restrictions).

Instances of alternative (b2). It remains to consider the case when among the (existing) emerging jobs pushing job \( j \), there is no emerging job for \( j \) (i.e., no new kernel can be defined). In this case, an instance of alternative (b2) (IA(b2)) with such an ex emerging job is said to occur. The next Observation immediately follows from the definitions:

Observation 5 Procedure \( \text{SEEK}(S(\delta)) \) finds \( S(\delta) \) for problem 1/\( r_j/L_{\text{max}} \) if no IA(b1/b2) during its execution occurs.

The following corollary is apparent.

Corollary 1 An optimal schedule for problem 1/\( r_j/L_{\text{max}} \) can be found in \( O(n \log n \log p_{\text{max}}) \) time if for all trial values \( \delta \), during the execution of \( \text{SEEK}(S(\delta)) \) no IA(b1/b2) occurs.

5 Coping with IA(b1/b2)

In this section we describe how IA(b1/b2) are dealt with by Procedure \( \text{SEEK}(S(\delta)) \). In other words, we study the only remained yet unconsidered by us case when while applying the modified ED-H there arises a job surpassing the \( \delta \)-boundary that neither belongs to any existing kernel nor yields a new kernel. Suppose that, while \( \text{SEEK}(S(\delta)) \) applies the modified ED-H, IA(b1/b2) behind kernel \( K \) arises, \( j \) being the corresponding job surpassing the \( \delta \)-boundary, say, by the amount \( \Delta \geq 1 \). Let \( S \) be the partial ED-schedule constructed by Procedure \( \text{SEEK}(S(\delta)) \) by the stage when IA(b1/b2) behind kernel has \( K \) occurred, i.e., \( j \) is the latest scheduled job in \( S \). Due to definition of \( S(\delta) \), the starting time of job \( j \) in \( S \) is to be decreased by at least \( \lambda \). This may only be possible if some passive emerging job(s) for \( K \) in \( S \) (with a total length no less than \( \lambda \)) are rescheduled before \( K \) (within \( B(K) \) or some preceding bin, where \( j \) may of course be such a job). At the same time, no such passive emerging job \( \pi \) can be rescheduled before \( K \) unless at least one job \( s \) from \( B_K \) or some preceding bin in \( S \) is, in turn, rescheduled after \( K \) (if this was possible, ED-H would include job \( \pi \) before \( K \)). We have arrived at the following definition.

Job \( s \in S \) is called a substitution job if it is an emerging job for \( K \); notice that once activated, \( s \) will not surpass the \( \delta \)-boundary. We denote the corresponding job set by \( \text{SUBST}(K, \delta) \), and we will accordingly use \( \text{PASS}(K, \delta) \) for the set of the passive emerging
jobs for $K$ scheduled before job $j$ in $S$ including job $j$ in the case when it is a passive emerging job for $K$. Observe that, if $j$ is not a passive emerging job, there exists no emerging job for $j$ (as otherwise a new kernel containing job $j$ would have been defined).

Let $\mathcal{J}(K)$ be the set of emerging jobs for $K$ (which includes $\text{PASS}(K, \delta) \cup \text{SUBST}(K, \delta)$ and the passive emerging jobs for $K$ not included into the schedule $S$, i.e., the ones which would have been scheduled after job $j$ by the extended ED-heuristic). Further, let $ST(K)$ be the length of the interval between the completion time of the latest scheduled non-kernel job before $K$ and the time moment $r(K) + \delta(K) + \delta$ in $S$.

Assume, for a moment, that we rearrange the jobs in $\mathcal{J}(K)$ so that the resultant new value of $ST(K)$ is the minimal possible one. Then this rearrangement is optimal if there occurs no further IA(b1/b2) with a job in $\mathcal{J}(K)$. Otherwise, while the current rearrangement is discarded, the next one is similarly tried, until all potentially useful rearrangements are checked. This type of procedure will have a polynomial-time cost for the version $1/p_{\max} < P(n), |r_j - r_i| < R/L_{\max}$. We need to introduce some new concepts and notations necessary for a detailed description of our job rearrangement.

5.1 Generating the subsets for filling the bins

Let $I(K)$ be the subinterval of the bin $B(K)$ from the starting time of the earliest scheduled job from $\text{SUBST}(K, \delta) \cap B(K)$ up to the time moment $r(K) + \delta(K) + \delta$.

**Observation 6** For the problem $1/p_{\max} < P, |r_j - r_i| < R/L_{\max}, |I(K)| < R + 2p_{\max}$, i.e., $|I(K)|$ is a constant.

Proof. First observe that the earliest scheduled job in $I(K)$ is started behind its original release time. Besides, there is at least one (kernel) job released at or after time $r(K)$. In fact, if $K$ is not the latest kernel in $\mathcal{K}_\delta$, then the latter job is released after time $r(K) + \delta(K) + \delta$; otherwise, since both $\delta(K)$ and $\delta$ are bounded by $p_{\max}$, this job is released after time $r(K) - 2p_{\max}$. Now our claim follows from the assumption that the maximal difference between the job release times $R$ and $p_{\max}$ are constants.

Procedure $\text{SEEK}(S(\delta))$ considers all (potentially useful) sets $J$ to fill in the interval $I(K)$, for all $K \in \mathcal{K}_\delta$. First, sets $J$ with a total job length of $l = |I(K)| - \alpha$ are tested, where $\alpha$ is the total length of the non-substitution jobs scheduled within $I(K)$ (and $|I(K)|$ is the length of $I(K)$); if there does not exist such a $J$ or none of them may lead to $S(\delta)$ (we give precise conditions when this happens a bit later), next $\text{SEEK}(S(\delta))$ looks for the sets with the total length $|I(K)| - \alpha - 1$, and so on. Since the (initial) $ST(K)$ is less than $p_{\max}$, the least trial value for $l$ is $l = |I(K)| - \alpha - p_{\max} + 1$.

**Congruent subsets.** We define an equivalence class in $\mathcal{J}(K)$ as follows. We call two subsets of $\mathcal{J}(K)$ congruent if there exists a one-to-one and onto mapping between these subsets, so that the image of every job in the first subset is a job of the same length from the second subset (note that two congruent subsets have the same total job length, but two subsets with the same total length are not necessarily congruent). Below we estimate the total number of all non-congruent subsets for each bin $B(K)$, and later we give such an estimation
for the congruent subsets.

**The number of non-congruent subsets.**

**Lemma 1** The total number of non-congruent subsets for the problem $1/p_{\text{max}} < P, |r_j - r_i| < R/L_{\text{max}}$ is a constant, for any bin $B(K), K \in K_\delta$.

Proof. For any set $J \subseteq \mathcal{J}(K)$, the total length of the jobs in $J$ and hence $l$ are constants (Observation 6). For each $l$, the number of possible non-congruent subsets is obviously bounded by the number of representations of $l$ as a sum of positive integers, which are the possible job processing times in our problem (without considering the order of terms of this sum). (As an example, for $l = 5$, we have $P(5) = 7$ since the following sum representations exist: $1 + 1 + 1 + 1 + 1, 1 + 1 + 1 + 2, 1 + 2 + 2, 1 + 1 + 3, 2 + 3, 1 + 4, 5$. Then we will have the same number of sets $J$ only if we have 4 jobs of the length 1 in $\mathcal{J}(K)$, 2 jobs with the length 2, and 1 job with the lengths 3, 4 and 5.) This number, known as the partition function $P(l)$, is approximately equal to

$$P(l) \sim \frac{\exp(\pi \sqrt{2l/3})}{4l^{\sqrt{3}}}$$

for large $l$ (there exists a rather complicated formula for the detailed computation of $P(l)$ given by Hans Rademacher [8]).

Since the number of possible numbers $l$ is less than $p_{\text{max}}$, the total number of all non-congruent subsets for $B(K)$ is bounded by $p_{\text{max}}$ multiplied by $P(R + 2p_{\text{max}})$ (see Observation 6). This product is a constant for the problem $1/p_{\text{max}} < P, |r_j - r_i| < R/L_{\text{max}}$ (as both multipliers are constants).

A note on the computational complexity of our algorithm. Instead of verifying all $p_{\text{max}} - 1$ possible values of $l$, one could be tempted to use a binary search to reduce the number of trial values for $l$ from $p_{\text{max}} - 1$ to $\log(p_{\text{max}} - 1)$. However, this is clearly not possible; otherwise, we would be able to remove the restriction $p_{\text{max}} < P$ from our problem, still solving it in polynomial time. Indeed, the term $p_{\text{max}}$ in the time complexity expression of our algorithm is due to the above estimated number of all non-congruent subsets for each bin $B(K)$. Therefore, our algorithm is pseudo-polynomial for the version $1/|r_j - r_i| < R/L_{\text{max}}$.

5.2 Subset sequences

Starting with $l = |I(K)| - \alpha$, and later for each particular $l$, the non-congruent subsets of $\mathcal{J}(K)$ are enumerated in a lexicographic order of the job processing times that sum up to $l$. SEEK($S(\delta)$) tests each next created subset $J$ if it is “satisfactory” or not (a bit later we give the precise conditions). Once and if this happens, the generation of subsets for the current bin is interrupted (it may later be resumed from that point of the search). Depending on current circumstances, SEEK($S(\delta)$) may also generate two or more congruent subsets (see the next subsection). Whenever a satisfactory subset $J$ is found, SEEK($S(\delta)$) incorporates the corresponding rearrangement that includes the jobs in $J$ within the interval $I(K)$ into the current $S$, repeating then the whole procedure for the next encountered kernel.
A satisfactory subset $J \subseteq \mathcal{J}(K)$ is incorporated into an auxiliary ED-schedule $S^{J,K}$ within the interval $I(K)$ using the ED-H as follows. From the beginning of the interval $I(K)$, the ED-H is applied to the job set $J \cup J'$ for scheduling jobs in this interval, where $J'$ is the set of the non-substitution jobs which were included into the interval $I(K)$ in the schedule $S$. Once the interval $I(K)$ is scheduled, the ED-H is applied to the jobs in $K$ (any job from the set $J$ which the ED-H could not include within $I(K)$ is forced to be scheduled after the jobs in $K$). Then the bin, immediately succeeding $K$, is completed by the ED-H (applied from the moment when the latest job in $K$ finishes).

We use $h$ for the current stage in $\text{SEEK}(S(\delta))$ when a satisfactory subset for the kernel $K_h \in \mathcal{K}_\delta$ (bin $B(K_h)$) is looked for. At the stage $h$, all satisfactory subsets $J_1, \ldots, J_h$ are incorporated within the bins $B(K_1) \prec \ldots \prec B(K_h)$ into the current schedule $S = S^{h,K_h}$; for the conciseness, we later use $S^h$ for $S^{J_h,K_h}$.

A satisfactory subset $J_h$ has to fulfill the two conditions, which are stated in the following two definitions.

The subset $J_h$ (generated for the bin $B(K_h)$) is said to be stable if all jobs in $S^h$ including the jobs from the bin $B_{h+1}$ (the one, immediately succeeding the kernel $K_h$) are scheduled so that no IA(b1/b2) occurs. Intuitively, a stable $S^h$ gives us a temporary (partial) solution for the jobs scheduled until the bin $B_{h+1}$.

We will say that a subset $J$ generated for the bin $B(K_h)$ conflicts with the subset $J < h$, if $J \cap J_i \neq \emptyset$ (we shall refer to the jobs in this intersection as the conflict jobs). Clearly, two or more conflict subsets cannot be feasibly incorporated.

**Theorem 1** Suppose that $K_1, K_2, \ldots, K_h$ are the kernels in $\mathcal{K}_\delta$ and that $J_1, J_2, \ldots, J_h, J_{h+1}$ is a corresponding sequence of stable subsets such that there is no conflict between any pair of these subsets. Then $S^h = S(\delta)$. Otherwise, if there exists no such sequence, then there exists no $S(\delta)$.

Proof. $S^h$ is a well-defined feasible (complete) schedule as there is no conflict between the incorporated subsets. Besides, since all the incorporated subsets are stable, no IA(b1/b2) in $S^h$ occurs. Hence, $S^h = S(\delta)$ by Observation 5. On the other hand, if there exists no sequence possessing with the claimed properties, then a feasible complete schedule in which no IA(b1/b2) occurs cannot exist. Hence, in any feasible schedule, there will be a job surpassing the $\delta$-boundary and therefore, there may exist no $S(\delta)$. □

Now it is clear that all $\text{SEEK}(S(\delta))$ has to do is to find a stable non-conflicting sequence of subsets (which can be done in polynomial time for our problem $1/p_{max} < P, |r_j - r_i| < R/L_{max}$, see Theorem 2).

### 5.3 Generating congruent subsets

For a given bin and kit of job processing times, two or more congruent subsets may exist. There are certain circumstances when it will be necessary to create a subset congruent to an earlier created one. However, there is a convenient bound on the total number of all
congruent subsets that SEEK(S, δ) may need to generate. The next observation immediately follows from the definitions.

**Observation 7** If a subset J_\text{h} is stable, then no subset congruent to J_\text{h} needs to be generated unless IA(b2) with a job from J_\text{h} in some S', \text{ } i > \text{h}, occurs.

Thus, the need to create a subset congruent to J = J_\text{h} may only arise when either J is non-stable or a corresponding IA(b2) occurs later on. We may have 3 distinct cases.

**Case 1.** J is non-stable and rigid, that is, at least one job from J is forced to be completed after the moment \( r(K) + \delta(K) + \delta \) (it does not fit before K; hence the modified ED-H may only include it after K).

Suppose that J is rigid. Since the total processing time of the jobs in J is no more than \(|I(K)|\), a new (non-existing in S) gap should have been arisen within I(K) in \( S^{J,K} \).

Let \( J' \) be any non-rigid subset, congruent to J. There is at least one job \( j' \in J' \) such that \( p_j = p_j \) for some \( j \in J \). Moreover, the job \( j' \) is (partially) scheduled within the above gap, \( r_j \) being smaller than \( r_j \) (otherwise, this gap in \( S^{J,K} \) could not have been avoided/reduced in \( S^{J',K} \)).

In **Case 2**, J is not rigid but remains non-stable. Either J conflicts with some earlier incorporated subsets or not.

Subcase 2.1. Suppose that there occurs a conflict and that there exists a subset congruent to J that does not conflict with any earlier incorporated subset. In that subset, every conflict job is replaced by an equal-length (non-conflict) job. Moreover, only the congruent subsets in which the due date of each newly replaced job is more than that of the replaced job need to be considered.

Subcase 2.2. Suppose that \( J = J_\text{h} \) does not conflict with any earlier incorporated subset. As J is not rigid, the jobs in J appropriately fill the corresponding interval before the kernel K. However, since J is not stable, there must be occurring an IA(b1/b2) with a job from \( J(K) \) scheduled in \( B_\text{h} \) or \( B_{\text{h}+1} \) (a job with an “insufficiently small” due date gets rescheduled later). Let \( j' \) be this job, and suppose there exists a subset \( J' \) congruent to J which is stable. The job \( j \in J \) with \( p_j = p_j \) replaces the job \( j' \) in \( J' \). Moreover, we have again \( d_j > d_j \) as otherwise IA(b1/b2) with \( j \) would similarly occur in \( S^{J',K} \).

In **Case 3**, the set \( J_\text{h} \) is stable, but an IA(b2) with a job in J occurs later at stage \( \text{i} > \text{h} \). This case clearly reduces to Case 2 (with stage \( \text{i} \) replacing the stage \( \text{h} \) in Case 2).

On the base of the analysis of the above cases, now we can estimate the total number of all congruent subsets.

**Lemma 2** The total number of congruent subsets that SEEK(S, δ) may need to generate for the problem \( 1/p_{\text{max}} < P, |r_j - r_i| < R/L_{\text{max}} \) is bounded by any of the magnitudes \( d_{\text{max}} \) or \( n \).

Proof. From the above analysis of Cases 1-3, it follows that the total number of congruent subsets that SEEK(S, δ) may need to generate for any \( K \in K_\delta \) is bounded by \( O(r_{\text{max}} + d_{\text{max}}) \), where \( r_{\text{max}} \) (\( d_{\text{max}} \)) is the maximum job release time (due date, respectively). This bound can also be written as \( O(d_{\text{max}}) \) as \( r_{\text{max}} < d_{\text{max}} \).
It is easy to see that the same bound applies to the total number of congruent subsets that SEEK($S(\delta)$) may test for all $K \in \mathcal{K}_\delta$. Indeed, consider kernels $K_\kappa$ and $K_h$ with $\kappa < h$. Let $i$ be the latest replaced job in a congruent subset $J_\kappa$ generated for the kernel $K_\kappa$. However, since job $i$ cannot be an emerging job for $K_h$ (its due date was already too small to be rescheduled after $K_\kappa$), $d_i$ is less than the due date of any job which may be used as a similar replacement for the kernel $K_h$. In other words, the due date of any replaced job for $K_h$ must be more than that of any replaced job for $K_\kappa$, which shows our claim.

Similarly, since no replaced job is again tried and there may exist no more than $n$ distinct emerging jobs for all kernels in $\mathcal{K}_\delta$, the bound $O(n)$ for the total number of the congruent subsets alternatively holds.

5.4 Time complexity of SEEK($S, \delta$)

**Theorem 2** For the problem $1/p_{\max} < P, |r_j - r_i| < R/L_{\max}$, Procedure SEEK($S, \delta$) creates $S(\delta)$ or establishes that it does not exist in the following two alternative time estimations $O(d_{\max} n \log n)$ and $O(n^2 \log n)$.

Proof. The soundness part immediately follows from the construction of SEEK($S, \delta$). Indeed, SEEK($S, \delta$) verifies the existence of a stable non-conflicting sequence by explicitly generating all possible non-conflict sequences (see Theorem 1). The total number of such sequences has a polynomial bound for the problem $1/p_{\max} < P, |r_j - r_i| < R/L_{\max}$.

First, we show that the total number of kernels that may arise is bounded by the constant $R$. Indeed, from the time moment $r_{\max}$, all yet unscheduled jobs can optimally be scheduled by the ED-H (this is a previously known fact, see Jackson [6]). Therefore, since the total length of the jobs in each kernel is at least 1, there may occur less than $r_{\max}$ kernels while scheduling jobs with the modified ED-H in SEEK($S, \delta$). Now our claim follows from the fact that $r_{\max} \leq R$.

Thus, the total number of kernels in $\mathcal{K}_\delta$ and the total number of sets $\mathcal{J}(K)$ that may arise is a constant. By Lemma 1, for each $K \in \mathcal{K}_\delta$, the total number of non-congruent subsets is also a constant. Hence, the total number of non-congruent subset sequences for all kernels is also a constant. By Lemma 2, the total number of congruent subsets that SEEK($S, \delta$) may need to generate for all the kernels is bounded by $d_{\max}$ and also by $n$. Hence, the total number of subset sequences for all kernels is also bounded by either $d_{\max}$ or $n$.

Procedure SEEK($S, \delta$) uses the modified $O(n \log n)$ ED-H for the construction of $S(\delta)$. If no IA(b1/b2) occurs, then SEEK($S, \delta$) builds $S(\delta)$ in $O(n \log n)$ time. Otherwise, the number of times IA(b1/b2) may occur during the execution of SEEK($S, \delta$) is obviously bounded by the total number of subset sequences for all kernels, which, as we have shown is either $d_{\max}$ or $n$. Hence, the two alternative bounds $O(n \log n) O(d_{\max}) = O(d_{\max} n \log n)$ and $O(n \log n) O(n) = O(n^2 \log n)$ hold.
6 Final remarks on the overall algorithm

**Theorem 3** The binary search procedure finds an optimal schedule for the problem \(1/p_{\text{max}} < P, |r_j - r_i| < R/L_{\text{max}}\) in

\[O(n^2 \log n \log p_{\text{max}}) \text{ or } O(d_{\text{max}} n \log n \log p_{\text{max}})\]

proof. By Theorem 2, for each trial value of \(\delta\), Procedure SEEK\((S, \delta)\) finds \(S(\delta)\) or gives the corresponding “no” answer in either \(O(d_{\text{max}} n \log n)\) and \(O(n^2 \log n)\) time. As \(\delta < p_{\text{max}}\), the number of iterations in the binary search procedure is bounded by \(\log p_{\text{max}}\). Then the running time of the overall algorithm is \(O(\log p_{\text{max}})\) multiplied by the time complexity of SEEK\((S(\delta))\).

In this way, the minimal \(\delta\) for which there exists a \(\delta\)-balanced schedule will be found in either of the time estimations \(O(d_{\text{max}} n \log n)\) and \(O(n^2 \log n)\). This schedule is optimal by Observation 3.

\[\square\]

**Acknowledgments**

The first author was partially supported by Deutscher Akademischer Austauschdienst (DAAD) grant and by CONACyT grant 160162.

**References**


