# A Polynomially Solvable Case of a Single Machine Scheduling Problem When the Maximal Job Processing Time is a Constant

Nodari Vakhania
Science Faculty, State University of Morelos
Av. Universidad 1001, Cuernavaca 62210, Morelos, Mexico
Inst. of Computational Math., Akuri 8, Tbilisi 93, 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

October 7, 2011

**Abstract:** We consider the problem of scheduling jobs with given release times and due dates on a single machine to minimize the maximal job lateness. This problem is NP-hard, and its version when the job processing times are restricted to $p, 2p, 3p, 4p, \ldots$, for an integer $p$, is also NP-hard. We consider the case when the maximal job processing time is $kp$, for any constant $k$, and propose its polynomial-time solution. We easily establish that the version of this problem with unrestricted $k$ is NP-hard. Moreover, it is strongly NP-hard if $p$ has no exponential-time dependence on the maximal job due date. From a practical point of view, this is a realistic assumption.

**Keywords:** scheduling, single machine, release time, due date, lateness

## 1 Introduction

We consider a *scheduling* problem when $n$ given jobs 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 $S$. The first inequality says that a job cannot be started before its release time, and the second one reflects the restriction that the machine can handle only one job at any time. $c_j(S) = t_j(S) + p_j$ is the completion time of job $j$. We 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, then we aim to find an optimal schedule, i.e., one minimizing the maximal *lateness* which is the difference between the actual completion time and the due date of the job. The maximal lateness is defined as $L_{\max} = \max\{j | c_j - d_j\}$.

In the scheduling literature, this problem is commonly abbreviated by $1/r_j/L_{\max}$. It is known to be strongly NP-hard [2]. Hence, it is unlikely that it can be solved in reasonable time in the general setting. The problem can naturally be simplified by imposing some restrictions on the job processing times. Two such versions were known to be polynomially solvable: When all jobs have equal integer length $p$ (problem $1/p_j = p, r_j/L_{\max}$) Garey et al. [3] and when a job processing time can be either $p$ or $2p$ (problem $1/p_j \in \{p, 2p\}, r_j/L_{\max}$) [8] (for a recent overview on single and parallel machine problems with equal processing times, we refer the interested reader to [5]). Quite recently, another such version when the job processing times are mutually divisible, abbreviated by $1/p_j : divisible, r_j/L_{\max}$, was shown to be polynomially solvable [9]. Without release times, scheduling the jobs in order of non-decreasing due dates gives an optimal schedule in $O(n \log n)$ time [4]. Similarly, if all $d_j$'s are equal, then scheduling jobs in order of non-decreasing release times is optimal.

In this paper, we continue to study the polynomially solvable versions of problem $1/r_j/L_{\max}$. We now consider the case when the job processing times are multiples of an integer $p$, i.e., they can take the values $p, 2p, \ldots, kp$, where $p$ is the minimal job processing time and $k$ is a constant. Based on the commonly used notation, we abbreviate this problem by $1/p_j \in \{p, \ldots, kp\}, r_j/L_{\max}$. The algorithm uses the approach from [9] that applies a binary search procedure and reduces the problem to some version of the bin packing problem. It also uses some relevant concepts from [7] and [8] for a comparative structural analysis of the created schedules on each iteration in the binary search. The so-called *behavior alternatives* reflect all crucial ways in which a newly created feasible solution may vary from the earlier created ones. On the bases of the analysis of the behavior alternatives, we may claim the desired schedule properties.

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 $p_{\max}$. First, we define the initial set of kernels applying a commonly used ED 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 utilize the intervals in between kernels (the bins) in an optimal way so that no non-kernel job causes the lateness more than that of a kernel job. The minimum of such a $\delta$ yields an optimal schedule.

Thus, roughly, the problem considered reduces to an optimal distribution of the non-kernel jobs within the bins. Now, the maximal job processing time $p_{\max} = kp$, which in our setting is a constant, restricts 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. Our algorithm runs in time $O(n^2 \log n \log p_{\max})$. We have an alternative pseudo-polynomial time bound of $O(d_{\max} n \log n \log p_{\max})$, where $d_{\max}$ is the maximal job due date (this latter bound may be more attractable for some practical applications).

The version of the considered problem when $k$ is not a constant is NP-hard. This easily follows from the known reduction from the PARTITION problem to the general problem $1/r_j/L_{\max}$, given in [2]. 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 of problem $1/p_j \in \{p, \ldots, kp\}, r_j/L_{\max}$ when $k$ is not a constant. Furthermore, this version is strongly NP-hard whenever $p$ has a polynomial-time dependence on $d_{\max}$, i.e., when it is not an exponent of $d_{\max}$. This also easily follows from the transformation given in [2] 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 $d_{\max}$, which shows the strong NP-hardness of this version. Let us mention that this restriction on $p$ is natural from a practical pint of view (it is highly unlikely that a job may need a processing time which is an exponent of its due date).

In the next section, we describe some basic concepts. In Section 3, we define our lower and upper bounds and the interval for the binary search procedure. In Section 4, we describe how to verify the existence of a feasible schedule in which the lateness of no job is more than the maximal allowable lateness defined by the current $\delta$ from the above interval. We define the derived subsets of jobs which are used for possible job rearrangements and show a bound on their total number.

# 2 Basics

The *Earliest Due date* heuristic (ED-H) suggested by Jackson [4] in early 1955 is commonly used for the construction of feasible schedules. Starting from time 0, at the current scheduling time $t$, among all released jobs it schedules one with the smallest due date, and updates the current scheduling time by adding the processing time of the scheduled job. It is easily seen why the ED-heuristic is not optimal: Assume that at time moment $r_j$, job $i$ is processed where $d_j < d_i$. Hence, job $i$ will delay the starting of job $j$, whereas job $j$ was an urgent job that needed to be scheduled with no or less delay in an optimal schedule.

The *initial ED-schedule* $\sigma$ is the one generated by ED-H for the originally given problem instance. As we have noted, the value of the objective function for the latter ED-schedule may be $p_{\max} = \max\{p_i | i = 1, 2, ..., n\} - 1$ more than the optimal value (this absolute error delineates the frontier between the polynomial solvability and strong NP-completeness).

A *gap* in a schedule is a maximal consecutive time interval in which no job is processed by

the machine. By our convention, there occurs a 0-length gap $(c_j, t_i)$ whenever job $i$ starts at its release time immediately after the completion of job $j$. A *block* in an ED-schedule $S$ is its consecutive part preceded and succeeded by a (possibly a 0-length) gap.

Now we define critical and non-critical jobs introduced informally in the introduction. Our primary goal now is to determine rigid (critical) segments in our initial ED-schedule $\sigma$. We do this by simply verifying our objective function for each included $j$.

We call a job $o$ from an ED-schedule $S$ an *overflow job* if

$$f_S(o) = \max\{f(j)|j \in S\},$$

where $f_S$ ($f_S(j)$, respectively) is the maximal lateness in $S$ (the lateness of $j \in S$, respectively). Note that $f_S(o)$ might be a positive or non-positive magnitude: in the former case, there arises a late job in $S$.

We call a *kernel* a maximal job sequence/set in $S$ ending with an overflow job $o$ such that no job from this sequence has a due date larger than $d_o$; if there are several successively scheduled overflow jobs, then $o$ is the latest one. Note that kernel jobs imply rigid segments in a feasible schedule, i.e., they are to be scheduled within a highly restricted time interval. Let $\mathcal{K}$ be the set of all kernels

$$K_1 \prec K_2 \prec \ldots \prec K_k$$

arisen in $\sigma$, and let $r(K) = \min_{i \in K}\{r_i\}$, for a kernel $K$. If the earliest scheduled kernel job starts at time $r(K)$, then there is no feasible schedule $S'$ with $f(S') < f_S(o)$ (as reordering the kernel jobs cannot reduce the lateness) and $S$ is clearly an optimal schedule. Otherwise, there is a job scheduled before all kernel jobs that delays kernel jobs including the overflow job. By removing such a job, the kernel jobs might be restarted earlier reducing in this way $f_{\max}$. To be more formal, let us introduce some definitions. Suppose that $i$ precedes $j$ in an ED-schedule $S$. We will say that $i$ *pushes* $j$ in $S$, if $j$ gets rescheduled earlier whenever $i$ is removed and the succeeding jobs from $S$ are rescheduled as early as possible respecting the order in $S$.

It follows from our assumption and definitions that the earliest scheduled job of every kernel is immediately preceded and pushed by a job $e$ with $d_e > d_o$. Clearly, we may have more than one such job scheduled before the kernel $K$ in the block containing $K$. We call such a job an *emerging job* for $K$, and the latest scheduled one (job $e$ above) the *delaying* emerging job. We shall later refer to job $j$ with $d_j > d_o$ and $r_j < r(K)$ scheduled after $K$ as a *passive emerging job* for $K$.

To restart the kernel jobs earlier, we *activate* an emerging job $e$ for $K$, *that is*, we force it and all passive emerging jobs to be rescheduled after $K$ by increasing their release times to a sufficiently large magnitude, say $r(K)$ (the latter jobs also are said to be activated for $K$). Then, when ED-H is again applied, neither job $e$ nor any passive emerging job will surpass any kernel job and hence, the earliest job in $K$ *will start* at $r(K)$. (More than one emerging job can be activated for $K$ and the same emerging job may be activated for two or more successive kernels.)

# 3 Defining the Boundaries

**$\delta$-balanced schedules.** Consider an (incomplete) ED-schedule $\sigma^{**}$ in which the delay job of every $K \in \mathcal{K}$ is just omitted, and let $f'(i)$ be the new (reduced) value of the lateness of each kernel job $i$ in $\sigma^{**}$. Since every $K$ is (re)started at time $r(K)$ in $\sigma^{**}$,

$$L(K) = \max_{i \in K}\{f'(i)\}$$

is a lower bound on the value of the optimal schedule. For any feasible $S$,

$$f(S) \geq L^* = \max_{\kappa}\{L(K_\kappa)\}$$

is a stronger lower bound. Furthermore, if $\delta(K) = L^* - L(K)$, then clearly, in any feasible $S$ we may allow the delay of $\delta(K) \geq 0$ without increasing the current maximal lateness, for every $K$.

We still need to accommodate every delaying job omitted in $\sigma^{**}$. Assume, instead of omitting, the delaying job is activated for each $K$. This simple solution might be too rough as due to the gaps left before each kernel, no available space might be left for scheduling the activated jobs together with the rest of the jobs. So the gap(s) before each kernel are to be somehow beneficially used. We use a binary search procedure. We already know that the earliest scheduled job of every $K$ can be started at time $r(K) + \delta(K)$. In an optimal schedule $S_{opt}$, either each kernel $K$ starts no later than at time $r(K) + \delta(K)$ or $K$ is to be delayed by some $\delta$, $0 \leq \delta \leq \Delta$, where $\Delta = f(o) - L^*$. Observe that, if the earliest job of every $K$ starts no later than at time $r(K) + \delta(K) + \delta$, then $f(i) \leq L^* + \delta$, for any $i \in K$. More generally, we call a feasible schedule $S(\delta)$ with

$$f(S(\delta)) \leq L^* + \delta$$

*$\delta$-balanced* (note that $\sigma = S(\Delta)$). The magnitude $L^* + \delta$ is the *$\delta$-boundary*; job $j$ *surpasses* the $\delta$-boundary if $f(j) > L^* + \delta$.

Thus, we allow an extra delay of $\delta$ for any (former) overflow job in $S(\delta)$, whereas the first job of every kernel $K \in S(\delta)$ starts no later than at time $r(K) + \delta(K) + \delta$ (we respectively redefine the delaying job for a kernel $K$ as one that completes *after* time $r(K) + \delta(K) + \delta$). Though, this does not yet guarantee that no job from $K$ will surpass the $\delta$-boundary when ED-H is again applied. This will be clearly so if $K$ starts at moment $r(K) + \delta(K) + \delta$.

Suppose that 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 the jobs of $K$ by ED-H, there may occur a time moment at which some kernel job completes but no yet unscheduled job from $K$ is released. Then some external job might be included at that (or later) moment. We put a restriction on the total length of such external jobs that might be included in between the jobs of $K$ (more formally, in any feasible $S$ with

$$f(S) \leq L(K) + \delta(K) + \delta$$

the above length is to be restricted as follows): clearly, if no gap in between the jobs of $K$ occurs, then the above magnitude cannot be more than $\epsilon$, otherwise it cannot exceed $\epsilon$ minus

the total length of the gap(s) occurred. We shall refer to this restriction as the *fitness* rule for $K$. Based on the next lemma which immediately follows, we further assume that this rule is respected while scheduling each kernel in $S(\delta)$:

**Lemma 1** *No job from any kernel $K$ will surpass the $\delta$-boundary if the fitness rule is respected.*

Although no job from the current set of kernels may surpass the current $\delta$-boundary, there may arise some other job that surpasses it. We wish to find out if there exists such a $S(\delta)$. At the first iteration of the binary search procedure when we generate $\sigma = S(\Delta)$, $\delta = \Delta$. The next value for $\delta$ is 0. If there exists no $S(0)$, then the next value of $\delta$ is $[\Delta/2]$. So $\delta$ 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 respectively in the next iteration. The following observation is apparent:

**Observation 1** *$S(\delta)$ with a minimal possible $\delta$ is optimal.*

Observe now that the problem is already solved given that we have a procedure that either constructs a $S(\delta)$ or asserts that it does not exist. As $\delta < p_{\max}$, the number of iterations for the binary search procedure is bounded by $\log p_{\max}$. Then, roughly, the running time of the overall algorithm would be $\log p_{\max}$ multiplied by the time complexity of the above procedure.

# 4 Seeking After $S(\delta)$

The procedure SEEK($S(\delta)$) that seeks after $S(\delta)$ consists of several passes. It again uses the ED-rule (in which ties are broken by selecting a longest job).

Let $\mathcal{K}_\delta$ be the set of kernels corresponding to $\delta$ in the binary search procedure (the initial set of kernels $\mathcal{K} = \mathcal{K}_\Delta$). For $\delta' < \delta''$, $|\mathcal{K}_{\delta'}| \geq |\mathcal{K}_{\delta''}|$. Whenever the binary search procedure resumes with a new $\delta$, SEEK($S(\delta)$) sets initially $\mathcal{K}_\delta$ to $\mathcal{K}'_\delta$, where $\delta' > \delta$ is the smallest so far encountered value with an existing $S(\delta')$. As SEEK($S(\delta)$) advances in forming a complete schedule, the current $\mathcal{K}_\delta$ might be completed by a new kernel. Recall that each $K \in \mathcal{K}_\delta$ starts no later than at time $r(K) + \delta(K) + \delta$ in $S(\delta)$. SEEK($S(\delta)$) respects this restriction. Hence, if there arises a job $j$ surpassing the $\delta$-boundary, then it cannot belong to any kernel in $\mathcal{K}_\delta$ (see also Lemma 1).

We shall refer to the interval before each $K \in \mathcal{K}_\delta$ as the *bin* defined by $K$ and denote it by $B_K$. Note that we may have a non-idle time interval after the latest kernel of $\mathcal{K}_\delta$ that we also call bin.

Observe that, if SEEK($S(\delta)$) has succeeded to construct a complete schedule so that no bin job has surpassed the $\delta$-boundary, then this schedule is $S(\delta)$. 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 a former emerging job (one activated for $K$ or/and some preceding kernel), then we will say that an *instance of alternative* (b1) (IA(b1)) with job $j$ occurs.

If $j$ is not a former emerging job, then an activated (former emerging) job must be pushing $j$. If among such jobs there is an emerging job for $j$, let $e$ be the latest scheduled one. If job $e$

was included before $K$, then the jobs from $K$ together with $j$ and all jobs that were included after $e$ (before job $j$ has occurred) define a new kernel, also denoted by $K$. If $e$ was included after $K$, then the sequence of jobs in-between $e$ and $j$ (including $j$) forms a new kernel $K'$ for the current $\delta$. SEEK($S(\delta)$) updates the current $\mathcal{K}_\delta$ respectively.

If no new kernel can be defined, i.e., there is no job $e$, let $i$ be an activated (former emerging) job pushing $j$. Then an *instance of alternative* (b2) (IA(b2)) with job $i$ is said to occur. The next observation immediately follows:

**Observation 2** *If in the block containing $K$ there arises a non-kernel job surpassing the $\delta$-boundary, then there must be occurring an IA(b1/b2) with an activated job.*

So, if no IA(b1/b2) occurs, then we already have a correct answer (for the general problem $1/r_j/L_{\max}$). Otherwise, to complete the description of SEEK($S(\delta)$), we need to describe how we rearrange non-kernel jobs for an IA(b1/b2). Clearly, at least one passive emerging job $q$ for $K$ is to be rescheduled before $K$. This will not be possible (in $S(\delta)$), unless some job $i$ scheduled in $B_K$ or some earlier bin is rescheduled after $K$ (if this was possible, ED-H would include $q$ in $B_K$). In general, suppose that we impose the delay of jobs $i_1, i_2, \ldots, i_\iota$ so that they are not scheduled before $K$. In the resultant schedule portion before $K$, let $\gamma$ be a gap occurring after time $r_q$, and $\lambda$ be the total length of all gaps including $\gamma$ and all succeeding gaps before $K$. Then $\gamma$ is *(potentially) valid* for $q$ (subject to $i_1, i_2, \ldots, i_\iota$) if $p_q < \lambda$. We call a job $s$ pushing $q$ a *substitution* job for $q$ if it is an emerging job for $K$ ($s$ is from $B_K$ or some earlier bin). Suppose that we activate some substitution job(s) for $K$. If among the newly arisen gaps there is a valid one for $q$ subject to the above substitution job(s) then, and only then, it might be possible to avoid another IA(b1/b2):

**Observation 3** *Suppose that there occurs an IA(b1/b2) behind the kernel $K$. Then there exists no $S(\delta)$ if there arises no valid gap for none of the passive emerging jobs for $K$ subject to some substitution jobs.*

Proof. First, we show that the claim holds if there exists no substitution job for none of the passive emerging jobs. At least one passive emerging job $q$ for $K$ is to be rescheduled before $K$. Hence, some job $i$ scheduled before $K$ is to be moved after $K$. However, since $i$ is not a substitution job for $q$, it is not an emerging job for $K$. Then it will surpass the $\delta$-boundary. Now, if there arises no valid gap for $q$ subject to some substitution jobs, then similarly, the $\delta$-boundary will be surpassed. □

## 4.1 Arranging the Activations for Substitution Jobs

Now we describe how the activations for the substitution jobs for a given kernel $K \in \mathcal{K}_\delta$ are organized. By Observation 3, all SEEK($S(\delta)$) has to do is to activate substitution jobs for $K$ in a proper fashion, whenever IA(b1/b2) with a job $j$, activated for $K$ occurs. Suppose again IA(b1/b2) with a job $j$ activated for kernel $K$ arises. We know that we may avoid this occurrence of IA(b1/b2) only if we activate some substitution job(s) for $K$ so that $j$ gets rescheduled earlier. By ED-H, all potentially useful substitution jobs must have been scheduled in the same bin $B(K')$, each of them being started strictly before the moment $r_j$,

where $K' = K$ or $K \prec K'$. Denote this set of these substitution jobs by SUBST$(K, \delta)$, and denote the corresponding set of the passive emerging jobs by PASS$(K)$ (observe that $j \in$PASS$(K)$). Basically, we need to determine which jobs from SUBST$(K, \delta)$ are to be activated for $K$ (whereas ED-H will determine the job(s) in PASS$(K)$ which will be included within the released space in $B(K')$). The total length of these jobs must be "enough long" to provide a sufficient left-shift (to avoid another IA(b1/b2) with a job from PASS$(K)$).

By ED-H, job $j$ cannot be released before the latest scheduled job in SUBST$(K, \delta)$ is started. Hence, the jobs in SUBST$(K, \delta)$ are scheduled one after another as a single sequence. Denote the interval occupied by this sequence by $I($SUBST$(K, \delta))$. We may achieve a desired left-shift for the job(s) in PASS$(K)$ by rescheduling some job(s) from SUBST$(K, \delta)$ behind $K$. It follows that the activation of jobs from $\mathcal{S} \subset$SUBST$(K, \delta)$ is successful (*that is*, $S(\delta)$ includes these activations) unless an IA(b1/b2) with (1) a job from PASS$(K)$ or (2) $\mathcal{S}$ occurs (see Observation 2).

It is apparent that the total length of the jobs in $\mathcal{S}$ matters. Roughly, a subset of SUBST$(K, \delta)$ with a correct total length would work out: this length should be enough to left-shift job $j$ by a sufficient amount of time units. Though, if it is "too long", it may cause a space shortage at a later schedule portion. Since the ED-heuristic could not include any job from PASS$(K)$ in $B(K)$, the available space before time moment $r(K) + \delta(K) + \delta$ was less than $p_{\max}$. By the activation of some job(s) from SUBST$(K, \delta)$ such a space can be released. Though clearly, it does not need to be more than $p_{\max}$. That is, the total processing time of jobs in any $\mathcal{S}$ (that we may need to evaluate) is bounded by $p_{\max} = kp$. Then the constant $k$ also bounds to a constant the total number of subsets that SEEK$(S(\delta))$ may need to evaluate (we give more details on this later).

We call two subsets of jobs from SUBST$(K, \delta)$ *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. Two congruent subsets have the same total job length, though the corresponding left-shift (which is determined by the resultant newly released space within $I($SUBST$(K, \delta))$) may be different. Intuitively, it is not difficult to see why the rescheduling of later scheduled jobs gives no-less left-shift than that of the earlier scheduled jobs. A formal proof of a similar statement can be found in [7]. Thus, while forming the next (yet untried) subset, among all same-length jobs (the candidates to be included into the subset) we always break ties by selecting the latest scheduled (the closest to the corresponding kernel) job. With this kind of tie-breaking rule, the formed subset will always provide the maximal possible left-shift among all yet untried congruent subsets. However, because of the variation of job due dates in two congruent subsets, it may still be necessary to deal with more than one congruent subset.

Although the total number of non-congruent subsets is a constant, the number of elements in each class of congruent subsets is not a constant. However, as we will see later, the total number of congruent subsets that might SEEK$(S(\delta))$ may need to evaluate can be bounded from above by any of the magnitudes $n$ or $d_{\max}$.

Assume that there occurs IA(b) either (1) with a job of PASS$(K)$ or (2) with a job of SUBST$(K, \delta)$ (and recall that these are the only possibilities). In case (1), the resultant

left-shift after the activation of jobs in $\mathcal{S}$ turns out to be insufficient. Then clearly, no other subset congruent to $\mathcal{S}$ can work out. We proceed with the next (non-congruent to $\mathcal{S}$) subset of SUBST$(K, \delta)$. If no such untried subset remains, then there exists no $\delta$-balanced schedule and SEEK$(S(\delta))$ can stop with a "no" answer.

Unlike case (1), in case (2) a suitable subset of SUBST$(K, \delta)$ may be congruent to $\mathcal{S}$. However, in such a job set there must be at least one job $i$ with a larger due date than the corresponding job from $\mathcal{S}$, as otherwise it is easily seen that the outcome would be the same. Thus, from the remaining jobs in $I$(SUBST$(K, \delta)$, the latest scheduled yet untried job is looked for such that it has the same processing time as a job in $\mathcal{S}$ but has a larger due date. By interchanging the two jobs, we create the new subset (congruent to $\mathcal{S}$), to be tried the next.

## 4.2 Summarizing the Algorithm

In the previous section, we have described how the activation of jobs in the subsets of SUBST$(K, \delta)$, for a kernel $K \in \mathcal{K}_\delta$, is organized. Whenever for the next tried $\mathcal{S} \subset$ SUBST$(K, \delta)$ no further IA(b1/b2) for a job from PASS$(K)$ or SUBST$(K, \delta)$ arises the arrangement of the bins preceding kernel $K$ is complete. SEEK$(S(\delta))$ continues the construction of the current schedule until a complete schedule $S(\delta)$ is obtained, or another IA(b1/b2) for a kernel succeeding $K$ occurs, in which case the procedure of the previous section for this new kernel is repeatedly applied.

To estimate the time complexity of SEEK$(S(\delta))$, we need to know the total number of subsets $\mathcal{S} \subset$ SUBST$(K, \delta)$ that SEEK$(S(\delta))$ may evaluate, for all $K \in \mathcal{K}_\delta$.

First, we show that the total number of non-congruent subsets is a constant, for any $K \in \mathcal{K}_\delta$. Indeed, since the total length of the jobs in any considered subset is bounded from above by $p_{\max}$, the number of possible subset lengths is also bounded by $p_{\max}$. For any fixed length $p^* \leq p_{\max}$, the number of possible non-congruent subsets equals to the number of combinations of different positive integers which sum to $p^*$ (without considering the sequence). As an example, for $p^* = 5$, we have $P(5) = 7$ since the following sum representations exist:

$$1+1+1+1,\ 1+1+1+2,\ 1+2+2,\ 1+1+3,\ 2+3,\ 1+4,\ 5.$$

This number, known as the *partition function* $P(p^*)$, is approximately equal to

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

for large numbers $p^*$ (note that there exists a rather complicated formula for the detailed computation of $P(p^*)$ given by Hans Rademacher). Therefore, the total number of all non-congruent subsets is bounded by $p_{\max}$ multiplied by the above magnitude $P(p^*)$, which is a constant in the problem under consideration.

We have two alternative bounds on the total number of congruent subsets we may need to evaluate, $d_{\max}$ and $n$. Indeed, since the due date of each newly selected job in each newly

formed congruent subset of SUBST$(K, \delta)$ is less than that of the replaced job, there is an obvious upper bound $d_{\max}$ on the total number of congruent subsets that SEEK$(S(\delta))$ may test, for each $K \in \mathcal{K}_\delta$. Moreover, this bound also applies to the total number of congruent subsets that SEEK$(S(\delta))$ may test for *all* $K \in \mathcal{K}_\delta$. Indeed, let $K \prec K'$, and $i$ be the latest replaced job in a congruent subset of SUBST$(K, \delta)$. Then note that $i$ cannot be an emerging job of $K'$. Hence, $d_i$ is less than the due date of any job in SUBST$(K', \delta)$. In other words, the due date of any replaced job from SUBST$(K', \delta)$ must be more than that of any replaced job from SUBST$(K, \delta)$, which shows our first claim for $d_{\max}$.

To see our second claim with the bound $n$, denote the number of jobs in SUBST$(K, \delta)$ by $\nu(K)$. Since the number of kernels is less than $n$, and for each tried $\mathcal{S} \subset$ SUBST$(K, \delta)$, for every $K \in \mathcal{K}_\delta$, we apply the $O(n \log n)$ ED-heuristic, the overall time spent by SEEK$(S(\delta))$ to build a complete schedule $S(\delta)$ (or establish that it does not exist) is

$$O\left(n \log n \sum_{K \in \mathcal{K}_\delta} \nu(K)\right) = O(n^2 \log n).$$

Similarly, based on our first claim, an alternative time complexity for SEEK$(S(\delta))$ is $O(d_{\max} n \log n)$. This completes the description of our algorithm and the proof of its correctness and time complexity:

**Theorem 1** *The binary search procedure finds an optimal schedule in time*

$$O(n^2 \log n \log p_{\max}) \quad \textit{or} \quad O(d_{\max} n \log n \log p_{\max}).$$

# 5 Concluding Remarks

We have proposed a polynomial-time solution of a restricted version of a strongly NP-hard problem $1/r_j/L_{\max}$. In our restriction the maximal job processing time is bounded from above by $kp$, where $p$ is an integer and $k$ is an integer constant. We believe that this is the maximal polynomially solvable special case of problem $1/r_j/L_{\max}$ with restricted job processing times.

# Acknowledgments

This work has been partially supported by Deutscher Akademischer Austauschdienst (DAAD) and by CONACyT grant 160162.

# References

[1] K.R. Baker and Z.-S. Su. Sequencing with due-dates and early start times to minimize maximum tardiness. *Naval Res. Logist. Quart.* 21:, 171–177, 1974.

[2] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP–completeness*, Freeman, San Francisco, 1979.

[3] M.R. Garey, D.S. Johnson, B.B. Simons and R.E. Tarjan. Scheduling unit–time tasks with arbitrary release times and deadlines. *SIAM J. Comput.*, 10: 256–269, 1981.

[4] J.R. Jackson. Scheduling a production line to minimize the maximum lateness, *Management Science Research Report* 43, University of California, Los Angeles, 1955.

[5] S.A. Kravchenko, F. Werner. Parallel machine problems with equal processing times. *Journal of Scheduling*, 14, 435 – 444, 2011.

[6] L. Schrage. Obtaining optimal solutions to resource constrained network scheduling problems, *Unpublished Manuscript*, March, 1971.

[7] N. Vakhania. A better algorithm for sequencing with release and delivery times on identical processors. *Journal of Algorithms*, 48, 273 –293, 2003.

[8] N. Vakhania. Single-machine scheduling with release times and tails. *Annals of Operations Research*, 129, 253 – 271, 2004

[9] N. Vakhania. Scheduling jobs with release times preemptively on a single machine to minimize the number of late jobs. *Proceedings of MISTA 2011*, 389 – 398, 2011.