A binary search algorithm for a special case of minimizing the lateness on a single machine

Nodari Vakhania*

Abstract— We study the problem of scheduling jobs with release times and due-dates on a single machine with the objective to minimize the maximal job lateness. This problem is strongly NP-hard, however it is known to be polynomially solvable for the case when the processing times of some jobs are restricted to either p or $2p$, for some integer p. We present a polynomial-time algorithm based on binary search when job processing times are less restricted; in particular, when they are mutually divisible. We first consider the case when the following condition holds: for any pair of jobs, if one is longer than another then the due-date of the former job is no larger than that of the latter one. We also study cases when a slight modification of our algorithm gives an optimal solution for the version without the restriction on job due-dates.

Keywords– algorithm, scheduling, single processor, release date, due-date, lateness

1 Introduction

The *scheduling problems* deal with a finite set of requests called *jobs* or *tasks* which have to be performed on a finite set of resources called *machines* or *processors*. A job in a factory or a program in a computer system or a lesson in a school are examples of requests. A machine in a factory or a processor in a computer system or a teacher in a school are examples of resources. Each job has its *processing requirement*, i.e., it needs a prescribed time on a machine, and usually a machine cannot handle more than one request at a time (for example, a teacher cannot give two lessons simultaneously). Besides, there are a limited number of machines and also time is limited, so we need to arrange an order in which the jobs are handled by the machines to make the total elapsed time as small as possible. We are given a non-decreasing time *objective function* which we wish to minimize.

Here we consider a problem of the above type in which additionally each job has the *release time* and the *due-date*: a job cannot be started before its release time and it is desirable to complete it by its due-date. We may have a single machine or a set of the *parallel machines* available for scheduling the jobs.

A more formal description of our problem, which, in scheduling literature is commonly abbreviated as $1/r_j/L_{max}$, is as follows. We consider the case when there is a single machine available for scheduling the jobs from the set $I = \{1, 2, ..., n\}$. The nature of the problem is such that each job $j \in I$ becomes available at its release time r_j and it needs a continuous processing time p_j on the machine (no job preemption is allowed). d_j is the due-date of j (these parameters are non-negative integral numbers). A *feasible schedule* S 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 in 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. We denote by $c_j(S) = t_j(S) + p_j$ the completion time of job j. We aim to find out if there is a schedule which meets all job due-dates, i.e., every 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* $L_{\max} = \max\{j|c_j - d_j\}$.

$1/r_j/L_{max}$ is strongly NP-hard Garey & Johnson [5]. Thus there is not much hope to find an efficient algorithm for the general setting. Baker and Zaw–Sing [1], Bratley et al [2], Carlier [3] and McMahon and Florian [8] have suggested exponential enumerative algorithms. The algorithms from [8] and [3] have a good practical performance.

Without release times, scheduling jobs in order of non-decreasing due-dates gives an optimal schedule in $O(n \log n)$ time Jackson [7]; similarly, if all d_js are equal, then scheduling jobs in order of non-decreasing release times is optimal. Other restrictions concern job processing times. Garey et. al. [6] have developed a subtle $O(n \log n)$ algorithm for the case when all processing times are equal. In this algorithm a concept of the so-called "forbidden region" is introduced. A forbidden region is a time interval in a schedule in which it is forbidden to start any job. In the algorithm the forbidden regions are defined, and a failure is declared if there exists no feasible schedule. Otherwise, a feasible schedule is generated, using the earliest deadline scheduling heuristic and the declared forbidden regions.

The above result was generalized in Vakhania [13] for the case of two processing times p and $2p$ (for some integer p) with an $O(n^2 \log n \log p)$ algorithm. The algorithm is obtained as a consequence of two enumerative algorithms for the version of the problem with arbitrary processing times. The first algorithm is exponential, whereas the second one, which is a restriction of the first one, runs in $O(n^2 \log n)$ time if certain conditions during its execution are satisfied. Otherwise, a special procedure, called the Balancing Procedure, is to be applied to guarantee the optimality. As it is shown, there is an $O(n^2 \log n \log p)$ implementation of the Balancing Procedure for the version of the problem when job processing times can take values p and $2p$, which yields an algorithm with the same time complexity for this problem.

The multiprocessor case with m parallel identical processors is much more complicated. However, it can also be solved polynomially if job processing times are equal. Simons & Warmuth [10] have suggested an $O(n^2 m)$ algorithm which uses the earlier mentioned forbidden regions from [6]. It applies the earliest deadline scheduling rule for the construction of feasible schedules, while each time the next scheduled job fails to meet its deadline, backtracking is performed and a new forbidden region is declared. It is easily seen that the minimization problem can be solved by the repeated application of an algorithm for the corresponding feasibility problem: we iteratively increase the due-dates of all jobs until we find a feasible schedule with the modified data. Since

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

the maximum tardiness will depend on the job processing time and the number of jobs, we may need to apply such an algorithm np/m times (p is the job processing time). This number can be reduced to $\log(np/m)$ if we use a binary search.

Vakhania [11] has proposed an $O(mn \log n)$ algorithm for the same problem when the maximal job due-date $d_{\max}$ has a given upper limit. The practical performance of this algorithm remains good when there is no restriction on the maximal job due-date: based on the computational experiments, for arbitrarily large due-dates, the running time of the algorithm, in practice, does not depend on $d_{\max}$. The algorithm from Vakhania [12] has the time complexity of $O(d_{\max}mn \log n + O(m\kappa n))$ (for the case of unrestricted due-dates), where $\kappa < n$ is a parameter which is known only after the termination of the algorithm; in practice, κ turns out to be a small positive integer. This algorithm enumerates a special type of feasible schedules called the *complementary schedules* on a search tree. A complementary schedule is constructed by the earliest due-date heuristic (ED-heuristic), proposed earlier in citeJ and [9]. This heuristic is repeatedly applied to the iteratively modified problem instances and different complementary schedules are produced. In each generated complementary schedule a new gap (a time interval in a schedule, not occupied by a job) arises. These gaps are similar to the earlier mentioned forbidden regions, though we spend no special computational effort for their construction. The so called *overflow job* realizes the maximal value of our objective function in a schedule. The *behavior alternatives* reflect all possible ways of altering of the overflow job in a newly generated complementary schedule. The algorithm is based on a special analysis of behavior alternatives.

In this paper we consider the single-machine version relaxing restrictions on job processing times: we deal with the case when job processing times are *mutually divisible*, that is, they can be ordered in a non-decreasing sequence such that for each two neighboring elements, the second exactly divides the first. First we restrict our attention to problem instances such that for jobs i, j with $p_i \geq p_j$, $d_i \leq d_j$. In other words, longer jobs are more urgent than shorter ones. This might well be the case in some industries where the manufacturer gives a higher priority to larger orders wishing to complete them ahead smaller ones (as larger orders give more profit). An example of a practical problem in which item sizes are mutually divisible is a computer memory allocation in which block sizes are restricted to powers of 2.

We present an efficient polynomial $O(n^2 \log n \log p_{max})$ algorithm, where $p_{max} = \max\{p_j | j = 1, 2, \ldots, n\}$. The algorithm uses some concepts introduced earlier in references [11], [12] and [13] incorporating though a different type of a search. In particular, the algorithm does not conduct the search on a solution tree and uses binary search. The whole set of jobs is partitioned into two kinds of subsets, non-critical and critical. The non-critical subsets contain jobs that might be flexibly moved within the schedule, whereas the critical sets contain the subsets of jobs which form tight sequences in the sense that the delay of the earliest scheduled job from the subset cannot exceed some calculated parameter between (including) 0 and p_{max}. Whenever the delay of the latter job is 0, the lateness of the latest scheduled job from the set defines a valid lower bound on the optimal value.

We first define the above job partition. Then we determine the above lower bounds yielded by each critical partition. The maximum among them is a valid lower bound for the problem, and it also determines a delay that might be imposed to other critical subsets without increasing the maximum lateness. Having at hand all the above magnitudes, we try to distribute jobs from the non-critical subsets in order to utilize the intervals in between the critical sequences, we call them *bins*, in the optimal way.

In this way a variation of a bin packing problem with different bin capacities when the objective is to find whether there exists a packing (solution) which includes all the items arises. Using the binary search, we find the minimal possible delay for the critical sequences that results in an optimal schedule.

A venerable paper by Coffman, Garey & Johnson [4] studies different versions of bin packing problem when the item sizes (job processing times in our case) are mutually divisible. Coffman, Garey & Johnson show that these versions can be solved in polynomial time.

As already noted, the case when job processing times are mutually divisible turns out also useful for our scheduling problem $1/r_j/L_{\max}$. In fact, it is sufficient to consider the version of our general problem in which the jobs from only non-critical subsets have mutually divisible processing times. We abbreviate the corresponding problem as $1/p_j : divisible, r_j/L_{\max}$.

We stress again that the processing times of the jobs from the critical subsets can be arbitrary. In some applications it is predictable which jobs will form a critical sequence so that the processing times of such jobs can be set without any restriction.

Finally, we study some cases when a slight modification of our above algorithm gives an optimal solution for the version without the restriction on job due-dates.

The paper is organized as follows. In section 2 we give basic concepts and notions and some useful properties of ED-schedules which are beneficially used later on in our derivations. Section 3 is devoted to the description of our algorithm. First we define the basis for the binary search procedure. In Section 3.1. we describe the AED-Algorithm which we use for scheduling each bin. In Section 3.1. we describe the overall algorithm and prove its correctness. Section 4 deals with the general setting with mutually divisible job processing times. We study a number of cases when a slight modification of our main algorithm still gives an optimal solution for this version without the restriction on job due-dates.

2 Some relevant properties of ED-schedules

We denote by $f(S)$ ($f_S(j)$, respectively) the maximal lateness in the schedule S (the lateness of $j \in S$, respectively). We construct our first and further feasible schedules using *Earliest Due-date* heuristic (ED-heuristic) suggested by Jackson [7] in early 1955 and later also used by Schrage [9]. The preemptive (on-line) version of this heuristic optimally solves the preemptive version of our (general) problem: Starting from time 0, at any time t, among all available jobs, schedule next one with the smallest due-date; if a job with a due-date, smaller than the currently processed job becomes available, interrupt the latter job and schedule the former job (note that the resulting schedule will have no machine idle time unless no job is available).

Assume job i is processed at time r_j, where $d_j < d_i$. If i cannot be preempted then it will delay the starting of job j. Note however that this delay will be less than p_i. The delay of job j in S (equal to the magnitude $t_j^S - r_j$) may result a non-optimal schedule. Indeed, it may turn out that job j has a tight due-date and had to be scheduled without or with a less delay in an optimal schedule.

Let t be the maximum between the minimal release time of yet unscheduled job and the time when the machine completes

the latest scheduled job (0 if no job is yet scheduled). Initially, we have no scheduled jobs. Iteratively, ED-heuristic (the non-preemptive version of the above mentioned preemptive heuristic), among all available jobs at time t, schedules a job with the smallest due-date breaking ties by selecting a longest job. The *initial ED-schedule* σ is the one, generated by ED-heuristic for the originally given problem instance. As we have noted, the value of our objective function for the latter ED-schedule may be $p_{max} = max\{p_i | i = 1, 2, ..., n\} - 1$ more than the optimal value (although this absolute error may seem to be not so essential, it 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 σ. This we do by simply verifying $f_\sigma(j)$ for each included j. In general, we call a job o in an ED-schedule S an *overflow job* if $f_S(o) = max\{f(j) | j \in S\}$ (we note that $f_S(o)$ might be positive or non-positive magnitude: in the former case there arises a late job in S).

We call a *kernel* the maximal job sequence/set in S ending with an overflow job o such that no job from this sequence has a due-date more than d_o (if there are several successively scheduled overflow jobs then o is the latest one).

Let us make the following easily seen but important observation: if the earliest scheduled kernel job in σ starts at its release time then there is no feasible schedule S' with $f(S') < f_\sigma(o)$ and σ is an optimal schedule:

Observation 1 *σ is optimal if it contains a kernel with its earliest scheduled job starting at its release time.*

Assume in this rest of the paper that the condition in the above Observation does not hold. Then there might be possible to restart kernel jobs earlier and decrease $f(\sigma)$. We dedicate to this task the rest of the paper. We introduce some necessary definitions first. Suppose i precedes j in an ED-schedule S. We will say that *i pushes j* in S if job 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 l with $d_l > d_o$. In general, we may have more than one such an l scheduled before kernel K in the block containing K. We call such a job an *emerging job* for K, and the latest scheduled emerging job (one scheduled immediately before the earliest kernel job) the *delaying* emerging job for that kernel.

Although the emerging jobs are initially non-critical, they may become critical if their scheduling is postponed for an "inadmissible" amount of time. With the simplest scenario, we may just postpone the scheduling of every delaying job. However, this may not result in an optimal way of using the intervals in between the kernels, and as a consequence, some of these delaying jobs may become critical. We concern this question little later in more details. Thus in aggregated terms, we aim to distribute emerging jobs in between the kernels in an optimal way. We shall postpone the scheduling of an emerging job e rescheduling it after K and call it the *activation* of e for K. Two or more emerging jobs might be activated for K and the same emerging job might also be activated for two or more successive kernels.

It easily follows from ED-heuristic that if we activate the delaying job and will not include any other non-urgent job before K, then the earliest scheduled kernel job will start it its release time (we denote this magnitude by $r(K) = \min_{i \in K}\{r_i\}$). Let us re-calculate the values $f(i)$ in the resultant partial ED-schedule containing only the kernel jobs, for each $i \in K$, and let $L(K) = max_{i \in K}\{f(i)\}$. Then clearly, $L(K)$ is a valid lower bound on the value of the optimal schedule.

Let $K_1, K_2, \ldots, K_k$ be all the formed kernels while generating σ. For any feasible S, $f(S) \geq L^{max} = max_\kappa\{L(K_\kappa)\}$ is a valid lower bound. Furthermore, if $\delta(K_\iota) = L^{max} - L(K_\iota)$, then clearly, in any feasible S we may allow the delay of $\delta(K_\iota) \geq 0$ without increasing the maximal lateness, for every K_ι; i.e., the earliest scheduled job of every K can be started at time $r(K) + \delta(K)$.

Note that $f(o)$ is an obvious upper bound, where o is an overflow job in σ. Let us now define the trial interval of the length $\Delta = f(o) - L^\sigma$, $[r(K_\kappa) + \delta(K_\kappa), r(K_\kappa) + \delta(K_\kappa) + \Delta]$, for every kernel K_κ. Either there exists an optimal schedule S_{opt} in which each kernel K_κ starts no later than at time $r(K_\kappa) + \delta(K_\iota)$ or not. In the former case, the lower bound L^σ is attainable, and in the latter case it is not. We shall carry out a binary search within the trial intervals for all kernels to determine the minimal δ such that each kernel K starts no later than at time $r(K) + \delta(K) + \delta$ in S_{opt}. To each δ its own set of kernels denoted by $\mathcal{K}_\delta$ corresponds (obtained from the earlier set by a possible addition of new kernel(s)). We respectively redefine the delaying job for $K \in \mathcal{K}_\delta$ as one that completes after time $r(K) + \delta(K) + \delta$ and will refer to that job as the δ-delaying job for K. Respecting a complete feasible ED-schedule, a total order can be defined on the set $\mathcal{K}_\delta$: we will write $K \prec K'$ if kernel K' immediately succeeds K.

3 The binary search procedure

Let us call a *δ-balanced* schedule a feasible schedule with the value of at most $L^{max} + \delta$ (the so-called *δ-boundary*), where $0 \leq \delta \leq \Delta$ (we already know that there exists a Δ-balanced schedule). Our binary search procedure finds the minimal such δ. It is easy to see that δ might be set equal for all the kernels. 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 is estimated to be $\log p_{max}$ multiplied by the cost for the generation of a δ-balanced schedule or the verification that it does not exist.

While scheduling the jobs with the current δ, respecting the early starting time of each kernel determined by δ we try to fill in maximally the space in between each neighboring pair of kernels and that before the first kernel. We will give a bit later a detailed description of our algorithm distributing emerging (non-kernel) jobs within the intervals in between the kernels when job processing times of these jobs are mutually divisible (we don't have such an optimal procedure for the arbitrary processing times: not surprising as the respective problem is NP-hard). For now assume we already have this algorithm that schedules emerging jobs in between the kernels.

At an iteration with current δ in the binary search procedure, no job from any $K \in \mathcal{K}_\delta$ will surpass the δ-boundary if K starts exactly at time $r(K) + \delta(K) + \delta$. Besides, no gap may occur in

between the jobs of K in this case. Suppose the earliest job of K starts earlier at moment $r(K) + \delta(K) + \delta - \epsilon$, for some integer $\epsilon \geq 1$. Then while scheduling jobs of K by ED-heuristic, 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 may put a restriction on the total length of such external jobs which 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 jobs of K occurs then this magnitude cannot be more than ϵ, otherwise it cannot exceed ϵ minus the total length of the gap(s) occurred. We call this restriction the *fitness* rule for K. The following lemma is now apparent:

Lemma 2 *For any δ in the binary search, the fitness rule for any kernel K is to be respected. In this case no job from K can surpass the δ-boundary.*

Defining new kernels. From now on, assume the fitness rule is respected while constructing our next δ-balanced schedule for the current set of kernels from $\mathcal{K}_\delta$. Let K be the latest scheduled so far kernel from $\mathcal{K}_\delta$, and there occurs a job l surpassing the δ-boundary before the scheduling of the next to K kernel K', $K \prec K'$, is started. l does not belong to any kernel from the current $\mathcal{K}_\delta$. Furthermore, if l is scheduled immediately after (the latest job of) K then it must be a (former) emerging/delaying job activated for K and/or some preceding kernel(s) from $\mathcal{K}_\delta$. Obviously, l cannot be scheduled after K in any δ-balanced schedule in this case.

Now in general, suppose while we apply ED-heuristic, there arises a non-kernel job j surpassing the current δ-boundary. If there are two or more such jobs arisen in a row, let j be the latest one. Obviously, either j is of type l or a job scheduled after all jobs of K must be pushing j. Furthermore, that job is one activated for K. So either j is an activated former emerging job or/and it is pushed by such a job e. For the latter case, consider the longest sequence of jobs containing j among which none is an activated job (a segment from the current ED-schedule). As we have observed, job e immediately preceding the earliest job from the sequence is an activating job. If e is also an emerging job for j then the latter sequence defines a new kernel for the current δ and e is the δ-delaying job for this new kernel. Otherwise no new kernel can be defined: either e is not an emerging job for j, or j itself is an activated former emerging job. In these cases will say that an *instance of alternative (b)* (IA(b) for short) with job e (job j) occurs. Thus intuitively, IA(b) covers the situation when while scheduling the next bin the due-date of an activated (former) emerging job turns out to be "insufficiently small".

Whenever a new (non-kernel and non-emerging) job surpasses the current δ-boundary and a new kernel can be defined (no IA(b) occurs) it is added to the current $\mathcal{K}_\delta$ (its parameters being defined according the the earlier specified riles) and the construction is resumed respecting the newly added kernel for the current δ. The other case we treat in Lemma 3 and Theorem 6.

The first iteration in our binary search is carried out for $\delta = \Delta$ when we generate our initial ED-schedule σ and obtain our first lower and upper bounds. We try to start each $K \in \mathcal{K}_\delta$ no later than at time $r(K) + \delta(K)$ on the second iteration with $\delta = 0$. To achieve this, the total processing time of jobs scheduled before K is to be reduced compared to that on the first iteration. In general, this is the case whenever δ is reduced from one iteration to the next one. If δ is increased, then the total processing time of jobs scheduled before K can be increased correspondingly. The change from larger to smaller value of δ is carried out if we have succeeded to generate a δ-balance schedule. Otherwise, we switch to the next iteration with a larger δ given that no δ-balance schedule exists. We will first describe how we try to generate a δ-balanced schedule, and then give some conditions when we can assert that no such a schedule may exist.

We may just brutally postpone the scheduling of the delaying job of each kernel while applying ED-heuristic. But evidently, one cannot guarantee that the obtained schedule is δ-balanced: we might be left with a batch of the rescheduled emerging jobs which no more can be completed at a due-time. That is why we need to use the space before every kernel (between two neighboring kernels) as much as possible, i.e., include non-kernel jobs with the maximal total length. We shall call the corresponding interval in between the preceding to K kernel and kernel K the *bin* defined by kernel K and denote it by B_K (the space before K if it is the earliest kernel in $\mathcal{K}_\delta$).

Thus we came to a version of bin packing problem: we have a fixed number of bins of different capacities and we wish to know if the given items can be distributed into these bins (our task is more complicated though: due to the release and due-dates, not all items can be placed into all the bins). A simplest instance of this problem is a well-known NP-complete subset sum problem. And it is clear why we do not expect to obtain a precise criterion for non-existence of a δ-balance schedule for the general setting. Coffman, Garey & Johnson [4] have shown that some versions of bin packing problem can be efficiently solved whenever item sizes are mutually divisible. We find such a restriction on job processing times also useful for our problem. From now on, we restrict the processing times of *(only)* non-kernel jobs to mutually divisible times. As powers of 2 is the most "dance" such a set, without loss of generality, we assume that the processing times of non-kernel jobs are powers of 2.

3.1 The AED-Algorithm

We apply two passes for scheduling each bin. On the first pass we use ED-heuristic, and we use some its extension on the second pass. Suppose e is the δ-delaying job for $K \in \mathcal{K}_\delta$ that occurs on the first pass, and let c be the length of the interval between the completion time of the job immediately preceding e and time moment $r(K) + \delta(K) + \delta$. No job from K will surpass the current δ-boundary if this interval is completely filled in. For this reason, we call it a *pseudo gap*. From the mutual divisibility of job processing times, this pseudo-gap can be reduced by at most p, where p is the largest job processing time such that $p \leq c$ (recall that $c < p_{max}$). Then while rescheduling bin B_K, we may allow the gaps with the total length of at most $c - p$ between the jobs scheduled in B_K, whereas we might be able to increase the total length of the jobs scheduled in B_K by at most p.

We call jobs with processing time $2p$ or more *long*. As longer jobs are more urgent, every long job l might be delayed by at most one shorter job k. Suppose l does not *fit* into the bin, *i.e.*, cannot be completed before time $r(K) + \delta(K) + \delta$. Then we postpone the inclusion of job k incorporating a waiting strategy into ED-heuristic: The *Augmented ED-algorithm* (AED-algorithm) postpones the scheduling of k and waits for at most $c - p$ time units for l. If l is released within that interval and it fits into B_K, then it is scheduled. Otherwise, the inclusion of l in B_K will yield a non-δ-balanced schedule. In other words, in no δ-balanced schedule l can be included into B_K. Hence, l might only be included into one of the following bins. In this case AED-algorithm does not

include l in B_k and does not postpone the scheduling of k, i.e., it proceeds as ED-algorithm.

Suppose while scheduling one of the following bins job l surpasses the current δ-boundary. By ED-heuristic, all jobs that have been scheduled in these bins are more urgent than l. Then it follows that no δ-balanced schedule including job l together with all the other jobs exists. Hence, we can cease the binary search procedure for the current δ and switch to the next iteration with the next larger value of δ. We summarize our derivations in the following lemma:

Lemma 3 *Suppose AED-algorithm did not include a long job l into B_K. Then l cannot be scheduled in B_K in any δ-balanced schedule. Moreover, if while scheduling one of the following bins IA(b) with job l occurs then there exists no δ-balanced schedule.*

We need another auxiliary lemma. It shows that the current gap in a partial schedule for B_K cannot be reduced by replacing a long job with shorter but also long jobs. Suppose AED-Algorithm leaves the (pseudo) gaps with the total length $c' > c - p$ such that $c \geq p_{\min}$ (if the latter inequality does not hold then c' cannot be further reduced). And let, similarly as before, p' be the maximal job processing time such that $p' \leq c'$. Suppose we have no short jobs, i.e., ones with processing time p' or less. So, we cannot reduce the (pseudo) gap by including a short job. In general, one could expect to reduce the gap by selecting another sequence of jobs in B_K. This would yield the replacement of a long job l by a sequence of shorter jobs having the processing times larger than p'. But such a replacement cannot exist if jobs are mutually divisible. This is stated in the next lemma which straightforwardly follows:

Lemma 4 *Suppose we have a collection of jobs with processing times $2p', 4p', 8p', \ldots, kp'$, where $k \geq 2$ is a power of 2. Then no sequence containing some subset of these jobs scheduled without a gap can be completed at time $(k-1)p'$, more generally, at any time moment $\iota p'$, for an odd ι.*

Theorem 5 *Among all feasible assignments of the available jobs to B_K, AED-Algorithm leaves the (pseudo) gaps with the minimal total length.*

Proof. Let p' be defined as above. We know that jobs with the total length no more than p' can potentially be appended to the partial schedule for B_K constructed by AED-algorithm (one job with the duration p', or two jobs with the duration $p'/2$, etc.). No such a job j was available once the latest included job in B_K was completed as otherwise either it would have been included or would have define a new kernel. It follows that no j with $p_j \leq p'$ could remain available for scheduling in B_K. But from Lemma 4, no feasible partial schedule for B_K can reduce the existing c' without including j with $p_j \leq p'$, which proves the theorem. □

3.2 Summarizing the algorithm

Let us first summarize our algorithm and its soundness proof. As we already know, the binary search procedure starts with $\delta = 0$ and the initial set of kernels and bins determined in σ. These sets are updated while the next bin from the current set of bins is being scheduled during the first pass, as earlier described. In general, if there are left untested trial values for δ, it is decreased whenever a δ-balanced schedule was generated for the latest δ. Otherwise, there exists no δ-balanced schedule (see the proof of the theorem below) and the current δ is increased on the next iteration.

Theorem 6 *The algorithm above finds an optimal schedule in time $O(n^2 \log n \log p_{max})$.*

Proof. For the soundness part, it clearly suffices to show that there exists no δ-balanced schedule if AED-algorithm did not generate it for the current δ. Indeed, suppose bin B_K could not be successfully scheduled, i.e., there arises some job o surpassing the current δ-boundary. Either job o was available while one of the preceding kernels was scheduled or not. In the latter case it immediately follows from Lemma 5 that o cannot be restarted earlier for the current δ and hence no δ-balanced schedule may exist. In the former case, if o is a long job then we just apply Lemma 3. Otherwise, let $B_{K'}$ be any preceding (already scheduled) bin (if there is no such a bin then we are clearly done). At least one $j \in B_{K'}$ is to be removed in order to include job o in $B_{K'}$. Moreover, by AED-Algorithm, $p_j > p_o$, hence $d_j \leq d_o$. Then clearly, the maximal lateness cannot be reduced by rescheduling job o to an earlier bin. This completes the soundness part.

As to the time complexity, we already have noted that the number of iterations for the binary search procedure is bounded by $\log p_{max}$. For each of these iterations we use AED-Algorithm for scheduling bins with a possible update of the current set of kernels $\mathcal{K}_\delta$. The running time of AED-Algorithm applied to all bins is $O(n \log n)$ (that of ED-heuristic) multiplied by the number of times a new kernel arises. The latter is clearly bounded by n. Hence, AED-Algorithm needs time $O(n^2 \log n)$ and the overall time complexity is $O(n^2 \log n \log p_{max})$. □

4 The general setting with mutually divisible job processing times

Throughout this section we relax our restriction that $p_i \geq p_j$ implies $d_i \leq d_j$, i.e., we consider the version of the general setting $1/r_j/L_{\max}$ with mutually divisible processing times $1/p_j : divisible, r_j/L_{\max}$.

We first observe that Lemmas 4 and 5 still hold. However, Lemma 3 and hence Theorem 6 may not hold. Indeed, a long job l might be pushed not by a single (as before) but by 2 or more shorter jobs (since all of them might be more urgent that l). Then, if we postpone just one such a short job, the long job l still may not be restarted early enough by its release time.

Now we formulate a sufficient condition when the algorithm from the previous section will still produce an optimal solution for this extended case.

Theorem 7 *The algorithm from the previous section is optimal if no IA(b) during its execution occurs.*

Proof. From the condition and the definition of the algorithm it immediately follows that it will find the minimal δ for which there exists a δ-balanced schedule, and the theorem follows. □

In the rest of the paper we study the case when the condition in the above theorem does not hold. Remind that while scheduling a bin B, longer jobs are included in B ahead shorter ones by AED-heuristic (the ED-rule is no more conserved). In this way shorter jobs are forced to be rescheduled later, may be still within B or after B within some succeeding bin. Although such a short job i might be more urgent than a longer job l already included in B. Whenever this is the case, we will say that l is a *big brother* of i. Job i may have one or more big brothers, have been scheduled in B or in some successively scheduled bin $B' \succ B$.

Whenever i cannot be included properly (without surpassing the current δ-boundary) IA(b) with i will occur. The following lemma is obvious:

Lemma 8 *Suppose IA(b) with job i occurs. Then there exists no δ-balanced schedule if i has no big bother.*

Thus assume that i has at least one big brother l. Suppose l is the only big brother of i. Then clearly, if there exists a δ-balanced schedule, job l is to be rescheduled after job i in that schedule. I.e., it will suffice to consider feasible schedules with this property:

Lemma 9 *Suppose IA(b) with job i occurs and $l \in B$ is the only big brother of i. Then in any δ-balanced schedule i is to be included ahead job l in B.*

Thus the above lemma imposes some corrections in AED-heuristic. In particular, we modify it so that, whenever the job selected next by AED-heuristic is l, its scheduling is postponed until i gets included. This rule is conserved, in general, for any pair (l, i). We shall refer to that modification of AED-heuristic as AED-algorithm_1. The next lemma follows from earlier lemmas and definitions:

Lemma 10 *Suppose, while AED-algorithm_1 is applied for scheduling each bin, there occurs IA(b) with a (former) big bother l. Then there exists no δ-balanced schedule.*

Recall that whenever we know that no δ-balanced schedule for the current δ exists, we can switch to the next larger value for δ in our binary search procedure thus continuing our search for an optimal solution.

If i has two or more big brothers then we need to determine which of them are to be postponed. In this way, the number of possible rearrangements that our algorithm may need to consider will somehow depend on the number of big brothers of i. It might be possible to discard some possibilities to come to a polynomial-time solution. This can be a subject of a further research (see also Section 5).

5 Conclusions

We have proposed an exact polynomial-time algorithm for finding an optimal solution for a version of a classical NP-hard scheduling problem with a single-machine when jobs have release times and due-dates and the objective is to minimize the maximum lateness. To came to a polynomial-time solution of the problem, we have restricted job processing times to mutually divisible times and have tied job due-dates with job processing times. We have also studied some cases when the proposed algorithm remains optimal for the version in which job due-dates and processing times are not tied.

Two main research directions remain. First, does there exist a polynomial-time solution for the version in which job processing times are mutually divisible but job due-dates and processing times are not tied? Second, if an answer on the latter question is positive, is the corresponding problem a maximal polynomially solvable case with restricted job processing times (i.e., any setting with a less restricted times becomes NP-hard)?

References:

[1] K.R. Baker and Zaw–Sing Su (1974). Sequencing with due-dates and early start times to minimize maximum tardiness. *Naval Res. logist. Quart.* 21, 171–177.

[2] P. Bratley, M. Florian and P. Robillard (1973). On sequencing with earliest start times and due–dates with application to computing bounds for (*n/m/G/F_{max}*) problem. *Naval Res. logist. Quart*. 20, 57–67.

[3] J. Carlier (1982). The one–machine sequencing problem. *European J. of Operational Research*. 11, 42–47.

[4] Coffman E.G. Jr., M.R. Garey and D.S. Johnson. Bin packing with divisible item sizes. *Journal of Complexity* 3, 406–428 (1987)

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

[6] Garey M.R., 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)

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

[8] G. McMahon and M. Florian. On scheduling with ready times and due-dates to minimize maximum lateness. *Operations Research.* 23, 475–482 (1975)

[9] Schrage L. Obtaining optimal solutions to resource constrained network scheduling problems, unpublished manuscript (march, 1971)

[10] Simons B., M. Warmuth. A fast algorithm for multiprocessor scheduling of unit-length jobs. *SIAM J. Comput.* 18, 690-710 (1989)

[11] Vakhania N. Scheduling equal-length jobs with delivery times on identical processors. *International Journal of Computer Mathematics*, 79(6), p. 715-728 (2002)

[12] Vakhania N. A better algorithm for sequencing with release and delivery times on identical processors. *Journal of Algorithms* 48, p.273-293 (2003)

[13] Vakhania N. "Single-Machine Scheduling with Release Times and Tails". *Annals of Operations Research*, 129, p.253-271 (2004)