mathematics

MDPI

Article

Dynamic Restructuring Framework for Scheduling with Release Times and Due-Dates

Nodari Vakhania

Centro de Investigación en Ciencias, Universidad Autónoma del Estado de Morelos, Morelos 62209, Mexico; nodari@uaem.mx

Received: 8 October 2019; Accepted: 12 November 2019; Published: 14 November 2019

Abstract: Scheduling jobs with release and due dates on a single machine is a classical strongly NP-hard combination optimization problem. It has not only immediate real-life applications but also it is effectively used for the solution of more complex multiprocessor and shop scheduling problems. Here, we propose a general method that can be applied to the scheduling problems with job release times and due-dates. Based on this method, we carry out a detailed study of the single-machine scheduling problem, disclosing its useful structural properties. These properties give us more insight into the complex nature of the problem and its bottleneck feature that makes it intractable. This method also helps us to expose explicit conditions when the problem can be solved in polynomial time. In particular, we establish the complexity status of the special case of the problem in which job processing times are mutually divisible by constructing a polynomial-time algorithm that solves this setting. Apparently, this setting is a maximal polynomially solvable special case of the single-machine scheduling problem with non-arbitrary job processing times.

Keywords: scheduling algorithm; release-time; due-date; divisible numbers; lateness; bin packing; time complexity

1. Introduction

Scheduling jobs with release and due-dates on single machine is a classical strongly NP-hard combination optimization problem according to Garey and Johnson [1]. In many practical scheduling problems, jobs are released non-simultaneously and they have individual due-dates by which they ideally have to complete. Since the problem is NP-hard, the existing exact solution algorithms have an exponential worst-case behavior. The problem is important not only because of its immediate real-life applications, but also because it is effectively used as an auxiliary component for the solution of more complex multiprocessor and shop scheduling problems.

Here, we propose a method that can, in general, be applied to the scheduling problems with job release times and due-dates. Based on this method, we carry out a detailed study of the single-machine scheduling problem disclosing its useful structural properties. These properties give us more insight into the complex nature of the problem and its bottleneck feature that makes it intractable. At the same time, the method also helps us to expose explicit conditions when the problem can be solved in polynomial time. Using the method, we establish the complexity status of the special case of the problem in which job processing times are mutually divisible by constructing a polynomial-time algorithm that solves this setting. This setting is a most general polynomially solvable special case of the single-machine scheduling problem when jobs have restricted processing times but job parameters are not bounded: if job processing times are allowed to take arbitrary values from set $\{p, 2p, 3p, \dots\}$, for an integer p, the problem remains strongly NP-hard [2]. At the same time, the restricted setting may potentially have practical applications in operating systems (we address this issue in more detail in Section 12).

Problem description. Our problem, commonly abbreviated in the scheduling literature as $1|r_j|L_{\max}$ (the notation suggested by Graham et al. [3]), can be stated as follows. There are given n jobs $\{1,2,\ldots,n\}$ and a single machine. Each job j has (uninterruptible) *processing time* p_j, *release time* r_j and *due-date* d_j: p_j is the time required by job j on the machine; r_j is the time moment by which job j becomes available for scheduling on the machine; and d_j is the time moment, by which it is desirable to complete job j on the machine (informally, the smaller is job due-date, the more urgent it is, and the late completion is penalized by the objective function).

The problem restrictions are as follows. The first basic restriction is that the machine can handle at most one job at a time.

A *feasible schedule* S is a mapping that assigns to every job j its starting time $t_j(S)$ on the machine, such that

$$t_j(S) \geq r_j \tag{1}$$

and

$$t_j(S) \geq t_k(S) + p_k, \tag{2}$$

for any job k included earlier in S (for notational simplicity, we use S also for the corresponding job-set).

The inequality in Equation (1) ensures that no job is started before its release time, and the inequality in Equation (2) ensures that no two jobs overlap in time on the machine.

$$c_j(S) = t_j(S) + p_j$$

is the *completion time* of job j in schedule S.

The *delay* of job j in schedule S is

$$t_j(S) - r_j.$$

An optimal schedule is a feasible schedule minimizing the maximum job *lateness*

$$L_{\max} = \max\{j|c_j - d_j\}$$

(besides the lateness, there exist other due-date oriented objective functions). $L_{\max}(S)$ ($L_j(S)$, respectively) stands for the maximum job lateness in schedule S (the lateness of job j in S, respectively). The objective is to find an optimal schedule.

Adopting to the standard three-field scheduling notation, we abbreviate the special case of problem $1|r_j|L_{\max}$ with divisible job processing times by $1|p_j : divisible, r_j|L_{\max}$. In that setting, we restrict job processing times to the mutually divisible ones: given any two neighboring elements in a sequence of job processing times ordered non-decreasingly, the first one exactly divides the second one (this ratio may be 1). A typical such sequence is formed by the integers each of which is (an integer) power of 2 multiplied by an integer $p \geq 1$.

A brief introduction to our method. Job release times and due-dates with due-date orientated objective functions compose a sloppy combination for most of the scheduling problems in the sense that it basically contributes to their intractability. In such problems, the whole scheduling horizon can be partitioned, roughly, into two types of intervals, the rigid one and the flexible ones. In an optimal schedule, every rigid interval (that potentially may contribute to the optimal objective value) is occupied by a specific set of (urgent) jobs, whereas the flexible intervals can be filled out by the rest of the (non-urgent) jobs in different ways. Intuitively, the "urgency" of a job is determined by its due-date and the due-dates of close-by released jobs; a group of such jobs may form a rigid sequence in a feasible schedule if the differences between their due-dates are "small enough". The remaining jobs are to be "dispelled" in between the rigid sequences.

This kind of division of the scheduling horizon, which naturally arises in different machine environments, reveals an inherent relationship of the scheduling problems with a version of bin packing problem and gives some insight into a complicated nature of the scheduling problems with job release times and due-dates. As shown below, this relationship naturally yields a general algorithmic framework based on the binary search.

A bridge between the scheduling and the bin packing problems is constructed by a procedure that partitions the scheduling horizon into the rigid and the flexible intervals. Exploring a recurrent nature of the scheduling problem, we develop a polynomial-time recursive procedure that partitions the scheduling horizon into the rigid and flexible intervals. After this partition, the scheduling of the rigid intervals is easy but scheduling of the flexible intervals remains non-trivial. Optimal scheduling of the flexible intervals, despite the fact that these intervals are formed by non-urgent jobs, remains NP-hard. To this end, we establish further structural properties of the problem, which yield a general algorithmic framework that may require exponential time. Nevertheless, we derive a condition when the framework will find an optimal solution in polynomial time. This condition reveals a basic difficulty that would face any polynomial-time algorithm to create an optimal solution.

Some kind of compactness property for the flexible segments may be guaranteed if they are scheduled in some special way. In particular, we show that the compactness property can be achieved by an underlying algorithm that works for the mutually divisible job processing times. The algorithm employs some nice properties of a set of mutually divisible numbers.

In terms of time complexity, our algorithmic framework solves problem $1|r_j|L_{\max}$ in time $O(n^2 \log n \log p_{\max})$ if our optimality condition is satisfied. Whenever during the execution of the framework the condition is not satisfied, an additional implicit enumeration procedure can be incorporated (to maintain this work within a reasonable size, here we focus solely to exact polynomial-time algorithms). Our algorithm for problem $1|p_j : \mathit{divisible}, r_j|L_{\max}$ yields an additional factor of $O(n \log p_{\max})$, so its time complexity is $O(n^3 \log n \log p^2_{\max})$.

Some previous related work. Coffman, Garey and Johnson [4] previously showed that some special cases of a number of weakly NP-hard bin packing problems with divisible item sizes can be solved in polynomial time (note that our algorithm implies a similar result for a strongly NP-hard scheduling problem). We mention briefly some earlier results concerning our scheduling problem. As to the exponential-time algorithms, the performance of venerable implicit enumeration algorithms by McMahon and Florian [5] and Carlier [6] has not yet been surpassed. There is an easily seen polynomial special case of the problem when all job release times or due-dates are equal (Jackson [7]), or all jobs have unit processing times (Horn [8]). If all jobs have equal integer length p, the problem $1|p_j = p, r_j|L_{\max}$ can also be solved in polynomial time $O(n^2 \log n)$. Garey et al. [9] described how the union and find tree with path compression can be used to reduce the time complexity to $O(n \log n)$. The problem $1|p_j \in \{p, 2p\}, r_j|L_{\max}$, in which job processing times are restricted to p and $2p$, for an integer p, can also be solved in polynomial $O(n^2 \log n \log p)$ time [10]. If we bound the maximum job processing time p_{max} by a polynomial function in n, $P(n) = O(n^k)$, and the maximal difference between the job release times by a constant R, then the problem $1/p_{max} < P, |r_j - r_i| < R/L_{\max}$ remains polynomially solvable [2]. When $P(n)$ is a constant or it is $O(n)$, the time complexity of the algorithm by [2] is $O(n^2 \log n \log p_{\max})$; for $k \geq 2$, it is $O(n^{k+1} \log n \log p_{\max})$. The algorithm becomes pseudo-polynomial without the restriction on $p_{\max}$ and it becomes exponential without the restriction on job release times. In another polynomially solvable special case the jobs can be ordered so that $d_1 \leq \cdots \leq d_n$ and $d_1 - \alpha r_1 - \beta p_1 \geq \cdots \geq d_n - \alpha r_n - \beta p_n$, for some $\alpha \in [0, +\infty)$ and $\beta \in [0, 1]$ Lazarev and Arkhipov [11]. The problem allows fast $O(n \log n)$ solution if for any pair of jobs j, i with $r_i > r_j$ and $d_i < d_j$, $d_j - r_j - p_j \leq d_i - r_i - p_i$, and if $r_i + p_i \geq r_j + p_j$ then $d_i \geq d_j$ [12].

The structure of this work. This paper consists of two major parts. In Part 1, an algorithmic framework for a single machine environment and a common due-date oriented objective function, the maximum job lateness, is presented, whereas, in Part 2, the framework is finished to a polynomial-time algorithm for the special case of the problem with mutually divisible job processing times. In Section 2, we give a brief informal introduction to our method. Section 3 contains a brief overview of the basic concepts and some basic structural properties that posses the schedules enumerated in the framework. In Section 4, we study recurrent structural properties of our schedules, which permit the partitioning of the scheduling horizon into the two types of intervals. In Section 5, we describe how our general framework is incorporated into a binary search procedure. In Section 6, we give an aggregated description of our main framework based on the partitioning of the scheduling horizon into the flexible and the rigid segments, and show how the rigid segments are scheduled in an optimal solution. In Section 7, we describe a procedure which is in charge of the scheduling of the non-urgent segments, and formulate our condition when the main procedure will deliver an optimal solution. This completes Part 1. Part 2 consists of Sections 8–11, and is devoted to the version of the general single-machine scheduling problem with mutually divisible job processing times (under the assumption that the optimality condition of Section 7 is not satisfied). In Section 8, we study the properties of a set of mutually divisible numbers that we use to reduce the search space. Using these properties, we refine our search in Section 9. In Section 10, we give the final examples illustrating the algorithm for divisible job processing times. In Section 11, we complete the correctness proof of that algorithm. The conclusions in Section 12 contain final analysis, possible impact, extensions and practical applications of the proposed method and the algorithm for the divisible job processing times.

2. An Informal Description of the General Framework

In this section, we give a brief informal introduction to our method (the reader may choose to skip it and go to formal definitions of the next section). We mention above the ties of our scheduling problem with a version of bin packing problem, in which there is a fixed number of bins of different capacities and the objective is to find out if there is a feasible solution respecting all the bin capacities. To see the relationship between the bin packing and the scheduling problems, we analyze the structure of the schedules that we enumerate. In particular, the scheduling horizon will contain two types of sequences formed by the "urgent" jobs (that we call kernels) and the remaining sequences formed by the "non-urgent" jobs (that we call bins). A key observation is that a kernel may occupy a quite restricted time interval in any optimal schedule, whereas the bin intervals can be filled out by the non-urgent jobs in different ways. In other words, the urgent jobs are to be scheduled within the rigid time intervals, whereas non-urgent ones are to be dispelled within the flexible intervals. Furthermore, the time interval within which each kernel is to be scheduled can be "adjusted" in terms of the delay of its earliest scheduled job. In particular, it suffices to consider the feasible schedules in which the earliest job of a kernel K is delayed by at most some magnitude, e.g., δ_K; $\delta_K \in [0, \Delta_K]$, where Δ_K is the initial delay of the earliest scheduled job of that kernel (intuitively, Δ_K can be seen as an upper bound on the possible delay for kernel K, a magnitude, by which the earliest scheduled job of kernel K can be delayed without surpassing the minimal so far achieved maximum job lateness). As shown below, for any kernel K, $\Delta_K < p_{\max} = \max_j\{p_j\}$. Observe that, if $\delta_K = 0$, i.e., when we restrict our attention to the feasible schedules in which kernel K has no delay, the lateness of the latest scheduled job of that kernel is a lower bound on the optimal objective value. In this way, we can calculate the time intervals which are to be assigned to every kernel relatively easily. The bins are formed by the remaining time intervals. The length of a bin, i.e., that of the corresponding time interval, will not be prior fixed until the scheduling of that bin is complete (roughly, because there might be some valid range for the "correct" Δ_Ks).

Then, roughly, the scheduling problem reduces to finding out if all the non-kernel jobs can "fit" feasibly (with respect to their release times) into the bins without surpassing the currently allowable lateness for the kernel following that bin; recall that the "allowable lateness" of kernel K is determined by δ_K. We "unify" all the δ_Ks to a single δ (common for all the kernels), and carry out binary search to find an optimal δ within the interval $[0, \max_K \Delta_K$ (the minimum δ such that all the non-kernel jobs fit into the bins; the less is δ, the less is the imposed lateness for the kernel jobs).

Thus, there is a fixed number of bins of different capacities (which are the lengths of the corresponding intervals in our setting), and the items which are to be assigned to these bins are non-kernel jobs. We aim to find out if these items can feasibly be packed into these bins. A simplified version of this problem, in which no specified time interval with each bin is associated and the items can be packed in any bin, is NP-hard. In our version, whether a job can be assigned to a bin depends, in a straightforward way, on the interval of that bin and on the release time of that job (a feasible packing is determined according to these two parameters).

If the reader is not yet too confused, we finally note that the partition of jobs into kernel and non-kernel ones is somewhat non-permanent: during the execution of our framework, a non-kernel job may be "converted" into a kernel one. This kind of situation essentially complicates the solution process and needs an extra treatment. Informally, this causes the strong NP-hardness of the scheduling problem: our framework will find an optimal solution if no non-kernel job converts to a kernel one during its execution (the so-called instance of Alternative (b2)). We observe this important issue in later sections, starting from Section 7.

3. Basic Definitions

This subsection contains definitions which consequently gain in structural insight of problem $1|r_j|L_{\max}$ (see, for instance, [2,13]). First, we describe our main schedule generation tool. Jackson's extended heuristics (Jackson [7] and Schrage [14]), also referred to as the *Earliest Due-date* heuristics (ED-heuristics), is commonly used for scheduling problems with job release times and due-dates. ED-heuristics is characterized by n scheduling times: these are 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 a given scheduling time (the jobs available by that time moment), one with the minimum due-date is assigned to the machine (ties can be broken by selecting a longest job). Iteratively, the next scheduling time is the maximum between the completion time of the latest assigned so far job to the machine and the minimum release time of a yet unassigned job (note that no job can be started before the machine gets idle, and no job can be started before its release time). Among all jobs available by each scheduling time, a job with the minimum due-date is determined and is scheduled on the machine at that time. Thus, whenever the machine becomes idle, ED-heuristics schedules an available job giving the priority to a most urgent one. In this way, it creates no gap that can be avoided (by scheduling some already released job).

3.1. Structural Components in an ED-Schedule

While constructing an ED-schedule, a *gap* (an idle machine-time) may be created (a maximal consecutive time interval during which the machine is idle; by our convention, there occurs a 0-length gap (c_j, t_i) if job i is started at its release time immediately after the completion of job j.

An ED-schedule can be seen as a sequence of somewhat independent parts, the so-called *blocks*; each block is a consecutive part in that schedule that consists of a sequence of jobs successively scheduled on the machine without any gap in between any neighboring pair of them; a block is preceded and succeeded by a (possibly a 0-length) gap.

As shown below in this subsection, by modifying the release times of some jobs, ED-heuristics can be used to create different feasible solutions to problem $1|r_j|L_{\max}$. All feasible schedules that we consider are created by ED-heuristics, which we call ED-schedules. We construct our initial ED-schedule, denoted by σ, by applying ED-heuristics to the originally given problem instance. Then, we slightly modify the original problem instance to generate other feasible ED-schedules.

Kernels. Now, we define our kernels and the corresponding bins formally. Recall that kernel jobs may only occupy restricted intervals in an optimal schedule, whereas the remaining bin intervals are to be filled in by the rest of the jobs (the latter jobs are more flexible because they may be "moved freely" within the schedule, without affecting the objective value to a certain degree, as we show below).

Let $B(S)$ be a block in an ED-schedule S containing job o that realizes the maximum job lateness in that schedule, i.e.,

$$L_o(S) = \max_j\{L_j(S)\} = L_{\max}(S). \quad (3)$$

Among all jobs in block $B(S)$ satisfying Equation (3), the latest scheduled one is called an *overflow job* in schedule S.

A *kernel* in schedule S is a longest continuous job sequence ending with an overflow job o, such that no job from this sequence has a due-date greater than d_o (for notational simplicity, we use K also for the corresponding job-set). For a kernel K, we let $r(K) = \min_{i \in K}\{r_i\}$. We may observe that the number of kernels in schedule S equals to the number of the overflow jobs in it. Besides, since every kernel is contained within a single block, it may include no gap. We denote by $K(S)$ the earliest kernel in schedule S. The following proposition states an earlier known fact from [13]. Nevertheless, we also give its proof as it gains some intuition on the used here techniques.

Proposition 1. *The maximum lateness of a job of kernel K in ED-schedule S is the minimum possible if the earliest scheduled job of that kernel starts at time $r(K)$. Hence, if schedule S contains a kernel with this property, then it is optimal.*

Proof. By the definition, for any job $j \in K$, $d_j \leq d_o$ (job j is no-less urgent than the overflow job o), whereas note that the maximum lateness of a job of kernel K in schedule S is $L_o(S)$. At the same time, the jobs in kernel K form a tight (continuous) sequence without any gap. Let S' be a complete schedule in which the order of jobs of kernel K differs to that in schedule S and let job o' realizes the maximum lateness of a job of kernel K in schedule S'. Then, from the above observations and the fact that the earliest job of kernel K starts at its release time in schedule S, it follows that

$$L_o(S) \leq L_{o'}(S').$$

Hence,

$$L_{\max}(S') \geq L_o(S) = L_{\max}(S) \tag{4}$$

and schedule S is optimal. □

Emerging jobs. In the rest of this section, let S be an ED-schedule with kernel $K = K(S)$ and with the overflow job $o \in K$ such that the condition in Proposition 1 does not hold. That is, there exists job e with $d_e > d_o$ scheduled before all jobs of kernel K that imposes a forced delay (right-shift) for the jobs of that kernel. By creating an alternative feasible schedule in which job e is rescheduled after kernel K, this kernel may be (re)started earlier, i.e., the earliest scheduled job of kernel K may be restarted earlier than the earliest scheduled job of that kernel has started in schedule S. We need some extra definitions before we define the so-obtained alternative schedule formally.

Suppose job i precedes job j in ED-schedule S. We say that i *pushes* j in S if ED-heuristics may reschedule job j earlier if job i is forced to be scheduled after job j.

If (by the made assumption immediately behind Proposition 1) the earliest scheduled job of kernel K does not start at its release time, then it is immediately preceded and pushed by Job l with $d_l > d_o$, the so-called *delaying emerging* job for kernel K (we use l exclusively for the delaying emerging job).

Besides the delaying emerging job, there may exist job e with $d_e > d_o$ scheduled before kernel K (hence before Job l) in schedule S pushing jobs of kernel K in schedule S. Any such job as well as Job l is referred to as an *emerging job* for K.

We denote the set of emerging jobs for kernel K in schedule S by $E(K)$. Note that $l \in E(K)$ and since S is an ED-schedule, $r_e < r(K)$, for any $e \in E(K)$, as otherwise a job of kernel K with release time $r(K)$ would have been included at the starting time of job e in schedule S.

Besides jobs of set $E(K)$, schedule S may contain job j satisfying the same parametric conditions as an emerging job from set $E(K)$, i.e., $d_j > d_o$ and $r_j < r(K)$, but scheduled after kernel K. We call such a job a *passive emerging job* for kernel K (or for the overflow job o) in schedule S. We denote the set of all the passive emerging jobs for kernel $K = K(S)$ by $EP(K)$.

Note that any $j \in EP(K)$ is included in block $B(S)$ (the block in schedule S containing kernel K) in schedule S. Note also that, potentially, any job $j \in EP(K)$ can be feasibly scheduled before kernel K as well. A job not from set $E(K) \cup EP(K)$ is a *non-emerging* job in schedule S.

In summary, all jobs in $E(K) \cup EP(K)$ are less urgent than all jobs of kernel K and any of them may be included before or after that kernel within block $B(S)$. The following proposition is not difficult to prove (e.g., see [13]).

Proposition 2. *Let S' be a feasible schedule obtained from schedule S by the rescheduling a non-emerging job of schedule S after kernel K. Then, The inequality in Equation* (4) *holds.*

Activation of an emerging job. Because of the above proposition, it suffices to consider only the rearrangements in schedule S that involve the jobs from set $E(K) \cup EP(K)$. As the first pass, to restart kernel K earlier, we may create a new ED-schedule S_e obtained from schedule S by the rescheduling an emerging job $e \in E(K)$ after kernel K (we call this operation the *activation* of job e for kernel K). In ED-schedule S_e, besides job e, all jobs in $EP(K)$ are also scheduled (remain to be scheduled) after kernel K. Technically, we create schedule S_e by increasing the release times of job e and jobs in $EP(K)$ to a sufficiently large magnitude (e.g., the maximum job release time in kernel K), so that, when ED-heuristics is newly applied, neither job e nor any of the jobs in set $EP(K)$ will be scheduled before any job of kernel K.

It is easily seen that kernel K (regarded as a job-set) restarts earlier in ED-schedule S_e than it has started in schedule S. In particular, the earliest job of kernel K is immediately preceded by a gap and starts at time $r(K)$ in schedule S_l, whereas the earliest scheduled job of kernel K in schedule S starts after time $r(K)$ (the reader may have a look at the work of Vakhania, N. [13] for more details on the relevant issues).

***L*-schedules.** We call a complete feasible schedule S^L in which the lateness of no job is more than threshold L, an *L-schedule*. In schedule S, job i is said to *surpass the L-boundary* if $L_i(S) > L$.

The magnitude

$$\lambda_i(S) = L_i(S) - L \tag{5}$$

is called the *L-delay* of job i in schedule S.

3.2. Examples

We illustrate the above introduced notions in the following two examples.

Example 1. *We have a problem instance with four jobs $\{l, 1, 2, 3\}$, defined as follows:*
$r_l = 0,\ p_l = 16,\ d_l = 100,$
$r_1 = 5,\ p_1 = 2,\ d_1 = 8,$
$r_2 = 4,\ p_2 = 4,\ d_2 = 10,$
$r_3 = 3,\ p_3 = 8,\ d_3 = 12.$

The initial ED-schedule σ is illustrated in Figure 1. There is a single emerging job in that schedule, which is the delaying emerging Job l pushing the following scheduled Jobs 1–3, which constitute the kernel in σ; Job 3 is the overflow job o in schedule σ, which consists of a single block. $L_{\max}(\sigma) = L_3(\sigma) = 30 - 12 = 18$.

ED-schedule σ_l, depicted in Figure 2, is obtained by activating the delaying emerging Job l in schedule σ (the release time of Job l is set to that of job 1 and ED-heuristics is newly applied). Kernel in that schedule is formed by Jobs 1 and 2, Job 2 is the overflow job with $L_{\max}(\sigma_l) = L_2(\sigma_l) = 17 - 10 = 7$, whereas Job 3 becomes the delaying emerging job in schedule σ_l.

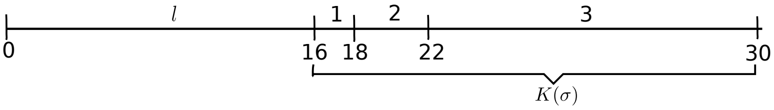

Figure 1. The initial ED-schedule σ for Example 1.

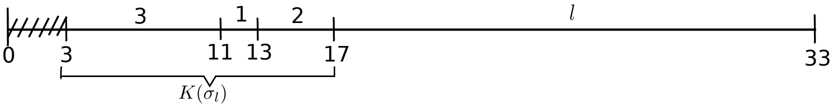

Figure 2. The ED-schedule σ_l for Example 1.

Example 2. *In our second (larger) problem instance, we have eight jobs* $\{l, 1, 2, \ldots, 7\}$*, defined as follows:*
$r_l = 0,\ p_l = 32,\ d_l = 50,$
$r_1 = 3,\ p_1 = 4,\ d_1 = 23,$
$r_2 = 10,\ p_2 = 2,\ d_2 = 22,$
$r_3 = 11,\ p_3 = 8,\ d_3 = 20,$
$r_4 = 0,\ p_4 = 8,\ d_4 = 67,$
$r_5 = 54,\ p_5 = 4,\ d_5 = 58,$
$r_6 = 54,\ p_6 = 4,\ d_6 = 58,$
$r_7 = 0,\ p_7 = 8,\ d_7 = 69.$

The initial ED-schedule σ is illustrated in Figure 3. Job l is the delaying emerging job, and Jobs 4 and 7 are passive emerging jobs. The kernel $K^1 = K(\sigma)$ is formed by Jobs 3, 2, and 1 (Job 1 being the overflow job).

ED-schedule σ_l is depicted in Figure 4. There arises a (new) kernel $K^2 = K(\sigma_l)$ formed by Jobs 5 and 6, whereas Job 4 is the delaying emerging job (Job 7 is the passive emerging job for both, kernels K^1 and K^2). Job 6 is the overflow job, with $L_{\max}(\sigma_l) = L_6(\sigma_l) = 68 - 58 = 10$.

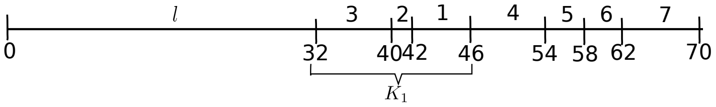

Figure 3. The initial ED-schedule σ for Example 2.

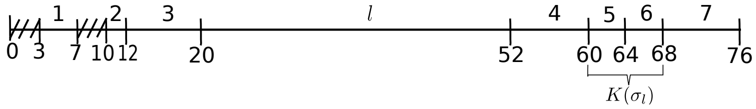

Figure 4. ED-schedule σ_l for Example 2.

4. Recurrent Substructures for Kernel Jobs

In this section, we describe a recursive procedure that permits us to determine the rigid intervals of a potentially optimal schedule (as we show below, these intervals not necessarily coincide with kernel intervals detected in ED-schedules). The procedure relies on an important recurrent substructure property, which is also helpful for the establishment of the ties of the scheduling problem with bin packing problems.

We explore the recurrent structure of our scheduling problem by analyzing ED-schedules. To start with, we observe that in ED-schedule S_l (where l is the delaying emerging job for kernel $K = K(S)$), the processing order of the jobs in kernel K can be altered compared to that in schedule S. Since the time interval that was occupied by Job l in schedule S gets released in schedule S_l, some jobs of kernel K may be scheduled within that interval (recall that by the construction, no job from $EP(K)$ may occupy that interval). In fact, the processing order of jobs of kernel K in schedules S and S_l might be different: recall from Section 3 that a job $j \in K$ with $r_j = r(K)$ will be included the first within the above interval in schedule S_l (whereas kernel K in schedule S is not necessarily initiated by job j; the reader may compare ED-schedules of Figures 1 and 2 and those of Figures 3 and 4 of Examples 1 and 2, respectively).

We call job $j \in K$ *anticipated* in schedule S_l if it is rescheduled to an earlier position in that schedule compared to its position in schedule S (in ED-schedules of Figures 2 and 4, Job 3 and Jobs 1 and 2, respectively, are the anticipated ones). In other words, job j surpasses at least one job i in schedule S_l such that i has surpassed j in schedule S (we may easily observe that, due to ED-heuristics, this may only happen if $q_j < q_i$, as otherwise job j would have been included before job i already in schedule S). Recall from Section 3 that the earliest scheduled job of kernel K is immediately preceded by a newly arisen gap in schedule S_l (in ED-schedules of Figures 2 and 4 it is the gap $[0, 3)$). Besides, a new gap in between the jobs of kernel K may also arise in schedule S_l if there exists an anticipated job since, while rescheduling the jobs of kernel K, there may occur a time moment at which some job of that kernel completes but no other job is available in schedule S_l. Such a time moment in ED-schedule of Figure 4 is 7, which is extending up to the release Time 10 of Job 2 resulting in a new gap $[7, 10)$ arising within the jobs of kernel K^1.

It is apparent now that jobs of kernel K (kernel K^1 in the above example) may be redistributed into several continuous parts separated by the gaps in schedule S_l (the first such part in ED-schedule of Figure 4. consists of the anticipated Job 1 and the second part consists of Jobs 2 and 3, where Job 2 is another anticipated job).

If there arises an anticipated job so that the jobs of kernel K are redistributed into one or more continuous parts in schedule S_l, then kernel K is said to *collapse*; if kernel K collapses into a single continuous part, then this continuous part and kernel K, considered as job-sets, are the same, but the corresponding job sequences are different because of an anticipated job. It follows that, if kernel K collapses, then there is at least one anticipated job in schedule S_l that converts to the delaying emerging job in that schedule (recall from Proposition 1 that schedule S is optimal if it possesses no delaying emerging job).

Throughout this section, we concentrate our attention to the part of schedule S_l initiating at the starting time of Job l in schedule S and containing all the newly arisen continuous parts of kernel K in that schedule that we denote by $S_l[K]$. We treat this part as an independent ED-schedule consisting of solely the jobs of the collapsed kernel K (recall that no job distinct from a job of kernel K may be included in schedule S_l until all jobs of kernel K are scheduled, by the definition of that schedule). For the instance of Example 1 with $S = \sigma$, schedule $\sigma_l[K]$ is the part of the ED-schedule of Figure 2 that initiates at at Time 0 and ends at Time 17. For the instance of Example 2, schedule $\sigma_l[K_1]$ starts at Time 0 and ends at Time 20 (see Figure 4).

We distinguish three different types of continuous parts in schedule $S_l[K]$. A continuous part that consists of only anticipated jobs (contains no anticipated job, respectively) is called an *anticipated* (*uniform*, respectively) continuous part. A continuous part which is neither anticipated nor uniform is called *mixed* (hence, mixed continuous part contains at least one anticipated and one non-anticipated job).

We observe that in ED-schedule of Figure 2 schedule $\sigma_l[K]$ consists of a single mixed continuous part with the anticipated Job 3, which becomes the new delaying emerging job in that schedule. Schedule $\sigma_l[K_1]$ of Example 2 (Figure 4) consists of two continuous parts, the first of which is anticipated with a single anticipated Job 1, and the second one is mixed with the anticipated Job 2. The latter job becomes the delaying emerging job in schedule $\sigma_l[K_1]$ and is followed by Job 3, which constitutes the unique kernel in schedule $\sigma_l[K_1]$.

Substructure Components

The decomposition of kernel K into the continues parts has the recurrent nature. Indeed, we easily observe that schedule $S_l[K]$ has its own kernel $K_1 = K((S_l)[K])$. If kernels K and K_1 (considered as sequences) are different, then the decomposition process naturally continues with kernel K_1 (otherwise, it ends by Point (4) of Proposition 3). For instance, in Example 1, kernel K_1 is constituted by Jobs 1 and 2 (Figure 2) and, in Example 2, it is constituted by Job 3 (see Figure 4) (in Lemma 4, we show that schedule $S_l[K]$ may contain only one kernel, which is from the last continuous part of that schedule). In turn, if kernel K_1 possesses the delaying emerging job, it may also collapse, and this process may recurrently be repeated. This gives the rise to a recurrent substructure decomposition of kernel K. The process continues as long as the next arisen kernel may again collapse, i.e., it possesses the delaying emerging job. Suppose there is the delaying emerging job l_1 for kernel K_1 in schedule $S_l[K]$. We recurrently define a (sub)schedule $S_{l,l_1}[K, K_1]$ of schedule $S_l[K]$ containing only jobs of kernel K_1 and in which the delaying emerging job l_1 is activated for that kernel, similarly to what is done for schedule $S_l[K]$. This substructure definition applies recursively as long as every newly derived (sub)schedule contains a kernel that may collapse, i.e., it possesses the delaying emerging job (this kernel belongs to the last continuous part of the (sub)schedule, as we prove in Lemma 4). This delaying emerging job is activated and the next (sub)schedule is similarly created.

We refer to the created is this (sub)schedules as the substructure *components* arisen as a result of the collapsing of kernel K and the following arisen kernels during the decomposition process. As already specified, the first component in the decomposition is $S_l[K]$ with kernel $K_1 = K(S_l[K])$, the second one is $S_{l,l_1}[K, K_1]$ with kernel $K_2 = K(S_{l,l_1}[K, K_1])$, the third one is $S_{l,l_1,l_2}[K, K_1, K_2]$, with kernel $K_3 = K(S_{l,l_1,l_2}[K, K_1, K_2])$, where l_2 is the delaying emerging job of kernel K_2, and so on, with the last *atomic component* being $S_{l,l_1,\ldots,l_k}[K, K_1, \ldots, K_k]$ such that the kernel $K^* = K(S_{l,l_1,\ldots,l_k}[K, K_1, \ldots, K_k])$ of that component has no delaying emerging job (here, l_k is the delaying emerging job of kernel K_k). Note that the successively created components during the decomposition form an embedded substructure in the sense that the set of jobs that contains each next generated component is a proper subset of that of the previously created one: substructure component $S_{l,l_1,\ldots,l_j}[K, K_1, \ldots, K_j]$, for any $j \leq k$, contains only jobs of kernel K_j, whereas clearly $|K_j| < |S_{l,l_1,\ldots,l_{j-1}}[K, K_1, \ldots, K_{j-1}]|$ (as kernel K_j does not contain, at least, job l_j, i.e., no activated delaying emerging job pertains to the next generated substructure component).

Below, we give a formal description of the procedure that generates the complete decomposition of kernel K, i.e., it creates all the substructure components of that kernel.

```
    PROCEDURE Decomposition$(S, K, l)$
{$S$ is an ED-schedule with kernel $K$ and delaying emerging Job $l$}
    WHILE $S_l[K]$ is not atomic DO
    BEGIN
        $S := S_l[K]$; $K :=$ the kernel in component $S_l[K]$;
        $l :=$ the delaying emerging job of component $S_l[K]$;
        CALL PROCEDURE Decomposition$(S, K, l)$
    END.
```

We illustrate the decomposition procedure on our two problem instances.

Example 1 (continuation). In the decomposition of kernel $K(\sigma)$ of Example 1, in ED-schedule of Figure 2, kernel K_1 of substructure component $S_l[K]$ consists of Jobs 1 and 2, and Job 3 is the corresponding delaying emerging job. Figure 5 illustrates schedule $\sigma_{l,3}$ obtained from schedule σ_l of Figure 2 by the activation of the (second) emerging Job 3 (which, in fact, is optimal for the instance of Example 1, with $L_{max}(\sigma_{l,3}) = L_3(\sigma_{l,3}) = 18 - 12 = 6$). A new substructure component $S_{l,3}[K, K_1]$ consisting of jobs of kernel K_1 is a mixed continuous part with the anticipated Job 2. Kernel K_2 of that component consists of Job 1, whereas Job 2 is the delaying emerging job for that sub-kernel ($L_1(\sigma_{l,3}) = 10 - 8 = 2$). Figure 6 illustrates ED-schedule $\sigma_{l,3,2}$ that contains the next substructure component $S_{l,3,2}[K, K_1, K_2]$ consisting of Job 1. Substructure component $S_{l,3,2}[K, K_1, K_2]$ is uniform and is the last atomic component in the decomposition, as it possesses no delaying emerging job and forms the last (atomic) kernel K_3 in the decomposition (with no delaying emerging job). $L_{max}(S_{l,3,2}[K, K_1, K_2]) = L_1(\sigma_{l,3,2}) = 7 - 8 = -1$. Note that the kernel in component $S_{l,3,2}[K, K_1, K_2]$

coincides with that component and is not a kernel in ED-schedule $\sigma_{l,3,2}$ (the overflow job in that schedule is Job 3 with $L_{max}(\sigma_{l,3,2}) = L_3(\sigma_{l,3,2}) = 19 - 12 = 7$).

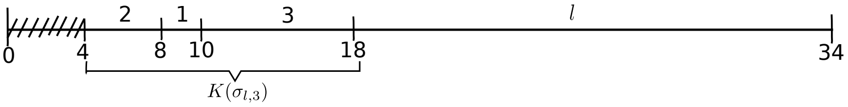

Figure 5. ED-schedule representing the second substructure component in the decomposition of kernel K for Example 1.

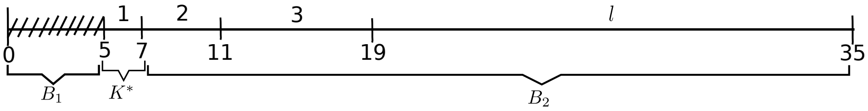

Figure 6. ED-schedule representing the third (atomic) substructure component in the decomposition of kernel K for Example 1.

Example 2 (continuation). Using this example, we illustrate the decomposition of two different kernels, which are denoted by K^1 and K^2 abvoe. In the decomposition of kernel K^1, in ED-schedule σ_l of Figure 4, we have two continuous parts in substructure component $S_l[K^1]$, the second of which contains kernel K^1_1 consisting of Job 3; the corresponding delaying emerging job is Job 2. The next substructure component $S_{l,2}[K^1, K^1_1]$ consisting of Job 3 (with the lateness $19 - 20 = -1$) is uniform and it is an atomic component that completes the decomposition of kernel K^1. This component can be seen in Figure 7 representing ED-schedule $\sigma_{l,2}$ obtained from schedule σ_l of Figure 4 by the activation of the emerging Job 2 for kernel K^1_1.

Once the decomposition of kernel K^1 is complete, we detect a new kernel K^2 consisting of Jobs 5 and 6 in the ED-schedule $\sigma_{l,2}$ depicted in Figure 7 (the same kernel is also represented in the ED-schedule σ_l of Figure 4). Kernel K^2 possesses the delaying emerging Job 4. The first substructure component $S_4[K^2]$ in the decomposition of kernel K^2 consists of a single uniform continuous part, which forms also the corresponding kernel K^2_1. The latter kernel has no delaying emerging job and the component $S_4[K^2]$ is atomic (see Figure 8).

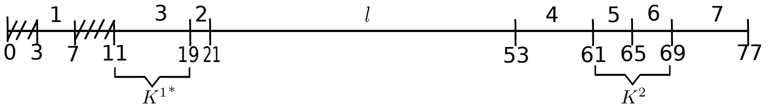

Figure 7. ED-schedule representing the second (atomic) substructure component in the decomposition of kernel K^1 and kernel K^2 for Example 2.

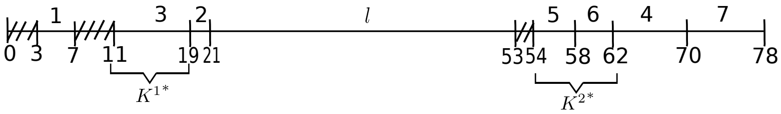

Figure 8. ED-schedule representing the atomic substructure components for kernels K^1 and K^2 for Example 2.

We need a few auxiliary lemmas to prove the validity of the decomposition procedure. For notational simplicity, we state them in terms of schedule S with kernel K and the component $S_l[K]$ (instead of referring to an arbitrary component $S_{l,l_1,\dots,l_j}[K, K_1, \dots, K_j]$ with kernel K_{j+1} and the following substructure component $S_{l,l_1,\dots,l_j,l_{j+1}}[K, K_1, \dots, K_j, K_{j+1}]$). The next proposition immediately follows from the definitions.

Proposition 3. *Suppose kernel K collapses. Then:*

(1) The anticipated jobs from kernel K are non-kernel jobs in schedule $S_l[K]$.
(2) Any continuous part in schedule $S_l[K]$ is either anticipated or uniform or mixed.

(3) *If schedule $S_l[K]$ consists of a single continuous part then it is mixed.*
(4) *If $K((S_l)[K]) = K$ (considering the kernels as job sequences), then schedule $S_l[K]$ consists of (a unique) uniform part that forms its kernel $K((S_l)[K])$. This kernel has no delaying emerging job and hence cannot be further decomposed.*

Lemma 1. *Let A be an anticipated continuous part in component $S_l[K]$. Then for any job $j \in A$,*

$$L_j(S_l[K]) < L_{\max}(S_l[K]),$$

i.e., an anticipated continuous part may not contain kernel $K(S_l[K])$.

Proof. Let G be the set of all jobs which have surpassed job j in schedule S and were surpassed by j in $S_l[K]$ (recall the definition of an anticipated part). For any job $i \in G$, $d_j \geq d_i$ since job j is released before jobs in set G and it is included after these jobs in ED-schedule S. This implies that $L_j(S_l[K]) < L_i(S_l[K]) \leq L_{\max}(S_l[K])$. The lemma is proved. □

Lemma 2. *A uniform continuous part U in component $S_l[K]$ (considered as an independent ED-schedule), may contain no delaying emerging job.*

Proof. Schedule U has no anticipated job, i.e., the processing order of jobs in U in both schedules S and $S_l[K]$ is the same. Observe that U constitutes a sub-sequence of kernel K in schedule S. However, kernel K has a single delaying emerging Job l that does not belong to schedule $S_l[K]$. Since U is part of $S_l[K]$ and it respects the same processing order as schedule S, it cannot contain the delaying emerging job. □

Lemma 3. *Suppose a uniform continuous part $U \in S_l[K]$ contains a job realizing the maximum lateness in component $S_l[K]$. Then,*

$$L_{\max}(U) \leq L_{\max}(S^{\text{opt}}), \tag{6}$$

i.e., the lateness of the corresponding overflow job is a lower bound on the optimal objective value.

Proof. Considering part U as an independent schedule, it may contain no emerging job (Lemma 2). At the same time, the earliest scheduled job in U starts at its release time since it is immediately preceded by a gap, and the lemma follows from Proposition 1. □

Lemma 4. *Only a job from the last continuous part $C \in S_l[K]$ may realize the maximum job lateness in schedule $S_l[K]$.*

Proof. The jobs in the continuous part C: (i) either were the latest scheduled ones from kernel K in schedule S; or (ii) the latest scheduled ones of schedule S have anticipated the corresponding jobs in C in schedule $S_l[K]$. In Case (ii), these anticipated jobs may form part of C or be part of a preceding continuous part P. In the latter sub-case, due to a gap in between the continuous parts in $S_l[K]$, the jobs of continuous part P should have been left-shifted in schedule $S_l[K]$ no less than the jobs in continuous part C and our claim follows. The former sub-case of Case (ii) is obviously trivial. In Case (i), similar to in the earlier sub-case, the jobs from the continuous parts preceding C in $S_l[K]$ should have been left-shifted in $S_l[K]$ no less than the jobs in C (again, due to the gap in between the continuous parts). Hence, none of them may have the lateness more than that of a job in continuous part C. □

Proposition 4. *PROCEDURE Decomposition(S, K, l) finds the atomic component of kernel K in less than $\kappa/2$ iterations, where κ is the number of jobs in kernel K. The kernel of that atomic component is formed by a uniform continuous part, which is the last continuous part of that component.*

Proof. With every newly created substructure component during the decomposition of a kernel with κ jobs, the corresponding delaying emerging job is associated. At every iteration of the procedure, the delaying emerging job is activated, and that job does not belong to the next generated component. Then, the first claim follows as every kernel contains at least one job. Hence, the total number of the created components during all calls of the collapsing stage is bounded above by $\kappa/2$.

Now, we show the second claim. From Lemma 4, the last continuous part of the atomic component contains the overflow job of that component. Clearly, the last continuous part of any component cannot be anticipated, whereas any mixed continuous part (seen as an independent schedule) contains an emerging job, hence a component with the last mixed continuous part cannot be atomic. Then, the last continuous part of the atomic component is uniform (see Point (2) in Proposition 3), and since it possesses no delaying emerging job (Lemma 2), it wholly constitutes the kernel of that component. □

From here on, let $K^* = K(S_{l,l_1,\dots,l_k}[K, K_1, \dots, K_k])$, where $S_{l,l_1,\dots,l_k}[K, K_1, \dots, K_k]$ is the atomic component in the decomposition of kernel K, and let ω^* be the overflow job in kernel K^*. By Proposition 4, K^* (the *atomic kernel* in the decomposition) is the only kernel in the atomic component $S_{l,l_1,\dots,l_k}[K, K_1, \dots, K_k]$ and is also the last uniform continuous part of that component.

Corollary 1. *There exists no L-schedule if*

$$L_{\max}(K^*) = L_{\omega^*}(S_{l,l_1,\dots,l_k}[K, K_1, \dots, K_k]) > L.$$

In particular, $L_{\max}(K^*)$ *is a lower bound on the optimum objective value.*

Proof. By Lemma 4 and Proposition 4, kernel K^* is the last continuous uniform part of the atomic component $S_{l,l_1,\dots,l_k}[K, K_1, \dots, K_k])$. Then, by Proposition 4 and the inequality in Equation (6),

$$L_{\max}(K^*) = L_{\max}(S_{l,l_1,\dots,l_k}[K, K_1, \dots, K_k] \le L_{\max}(S^{\text{opt}}).$$

□

Theorem 1. *PROCEDURE Decomposition*(S, K, l) *forms all substructure components of kernel K with the last atomic component and atomic kernel* K^* *in time* $O(\kappa^2 \log \kappa)$ *(where* κ *is the number of jobs in kernel K).*

Proof. First, observe that, for any non-atomic component $S_{l,l_1,\dots,l_j}[K, K_1, \dots, K_j]$ $(j < k)$ created by the procedure, the kernel $K_{j+1} = K(S_{l,l_1,\dots,l_j}[K, K_1, \dots, K_j])$ of that component is within its last continuous part (Lemma 4). This part cannot be anticipated or uniform (otherwise, it would not have been non-atomic). Thus, the last continuous part M in that component is mixed and hence it contains an anticipated job. The latest scheduled anticipated job in M is the delaying emerging job l_{j+1} for kernel K_{j+1} in the continuous part M. Then, the decomposition procedure creates the next component $S_{l,l_1,\dots,l_j,l_{j+1}}[K, K_1, \dots, K_j, K_{j+1}]$ in the decomposition (consisting of the jobs of kernel K_{j+1}) by activating job l_{j+1} for kernel K_{j+1}.

Consider now the last atomic component $S_{l,l_1,\dots,l_k}[K, K_1, \dots, K_k]$. By Proposition 4, atomic kernel K^* of component $S_{l,l_1,\dots,l_k}[K, K_1, \dots, K_k]$ is the last uniform continuous part in that component. By the inequality in Equation (6), $L_{\max}(S_{l,l_1,\dots,l_k}[K, K_1, \dots, K_k]) = L_{\max}(K^*)$ is a lower bound on the optimal objective value and hence the decomposition procedure may halt: the atomic kernel K^* cannot be decomposed and the maximum job completion time in that kernel cannot be further reduced. Furthermore, if $L_{\max}(K^*) > L$, then there exists no L-schedule (Corollary 1).

As to the time complexity, the total number of iterations (recursive calls of PROCEDURE Decomposition(S, K, l)) is bounded by $\kappa/2$ (where κ is the number of jobs in kernel K, see Proposition 4). At every iteration i, kernel K_{i+1} and job l_{i+1} can be detected in time linear in the number of jobs in component $S_{l,l_1,\dots,l_i}[K, K_1, \dots, K_i]$, and hence the condition in WHILE can be verified with the same cost. Besides, at iteration i, ED-heuristics with cost $O(\kappa \log \kappa)$ is applied, which yields the overall time complexity $O(\kappa^2 \log \kappa)$ of PROCEDURE Decomposition(S, K, l). □

Corollary 2. *The total cost of the calls of the decomposition procedure for all the arisen kernels in the framework is $O(n^2 \log n)$.*

Proof. Let $K_1, \ldots, K_k$ be all the kernels that arise in the framework. For the purpose of this estimation, assume κ, $k\kappa \leq n$, is the number of jobs in every kernel (this will give an amortized estimation). Since every kernel is processed only once, the the total cost of the calls of the decomposition procedure for kernels $K_1, \ldots, K_k$ is then

$$kO(\kappa^2 \log \kappa) \leq \frac{n}{\kappa} O(\kappa^2 \log \kappa) < O(n^2 \log n).$$

□

5. Binary Search

In this section, we describe how binary search can be beneficially used to solve problem $1|r_j|L_{\max}$. Recall from the previous section that PROCEDURE Decomposition(S, K, l) extracts the atomic kernel K^* from kernel K (recall that l is the corresponding delaying emerging job—without loss of generality, assume that it exists, as otherwise the schedule S with $K(S) = K$ is optimal by Proposition 1). Notice that, since the kernel of every created component in the decomposition is from its last continuous part (Lemma 4), there is no intersection between the continuous parts of different components excluding the last continuous part of each component. All the continuous parts of all the created components in the decomposition of kernel K except the last continuous part of each component are merged in time axes resulting in a partial ED-schedule which initiates at time $r(K)$ and has the number of gaps equal to the number of its continuous parts minus one (as every two neighboring continuous parts are separated by a gap). It includes (feasibly) all the jobs of kernel K except ones from the atomic kernel K^* (that constitutes the last continuous part of the atomic component, see Proposition 4). By merging this partial schedule with the atomic kernel K^*, we obtain another feasible partial ED-schedule consisting of all the jobs of kernel K, which we denote by $S^*[K]$. We extend PROCEDURE Decomposition(S, K, l) with this construction. It is easy to see that the time complexity of the procedure remains the same. Thus, from here on, we let the output of PROCEDURE Decomposition(S, K, l) be schedule $S^*[K]$.

Within the gaps in partial schedule $S^*[K]$, some *external* jobs for kernel K, ones not in schedule $S^*[K]$, will be included. During such an expansion of schedule $S^*[K]$ with the external jobs, the right-shift (a forced delay) of the jobs from that schedule by some constant units of time, which is determined by the current trial δ in the binary search procedure, will be allowed (in this section, we define the interval from which trial δs are taken).

At an iteration h of the binary search procedure with trial δ_h, one or more kernels may arise. Iteration h starts by determining the earliest arisen kernel, which, as we show below, depends on the value of trial δ_h. This kernel determines the initial partition of the scheduling horizon into one kernel and two non-kernel (bin) intervals. Repeatedly, during the scheduling of a non-kernel interval, a new kernel may arise, which is added to the current set of kernels at iteration h. Every newly arisen kernel is treated similarly in a recurrent fashion. We denote by $\mathcal{K}$ the set of kernels detected by the current state of computation at iteration h (omitting parameter h for notational simplicity). For every newly arisen kernel $K \in \mathcal{K}$, PROCEDURE Decomposition(S, K, l) is invoked and partial schedule $S^*[K]$ is expanded by external jobs. Destiny feasible schedule of iteration h contains all the extended schedules $S^*[K]$, $K \in \mathcal{K}$.

The next proposition easily follows from the construction of schedule $S^*[K]$, Lemma 4 and Corollary 1:

Proposition 5. $K^* = K(S^*[K])$ *(K^* is the only kernel in schedule $S^*[K]$) and*

$$L_{\max}(S^*[K]) = L_{\max}(K^*) \le L_{\max}(S^{\text{opt}}),$$

i.e., $L_{\max}(S^*[K])$ is a lower bound on the optimum objective value.

$$L^*_{\max} = \max_{K \in \mathcal{K}}\{L_{\max}(K^*)\}$$

is a stronger lower bound on the objective value.

Now, we define an important kernel parameter used in the binary search. Given kernel $K \in \mathcal{K}$, let

$$\delta(K^*) = L^*_{\max} - L_{\max}(K^*) \ge 0, \tag{7}$$

i.e., $\delta(K^*)$ is the amount of time by which the starting time of the earliest scheduled job of kernel K^* can be right-shifted (increased) without increasing lower bound $L^*_{\max}$. Note that for every $K \in \mathcal{K}$, $\delta(K^*) + L_{\max}(K^*)$ is the same magnitude.

Example 2 (continuation). For the problem instance of Example 2, $L_{\max}(K^{1^*}) = L_3(\sigma_{l,2}) = 19 - 20 = -1$, $L_{\max}(K^{2^*}) = L_6(\sigma_{l,2,4}) = 62 - 58 = 4$; hence, $\delta(K^{1^*}) = 5$ and $\delta(K^{2^*}) = 0$ (recall that atomic kernel K^{1^*} consists of a single Job 3, and atomic kernel K^{2^*} consists of Jobs 5 and 6; hence, the lower bound $L^*_{\max} = 4$ is realized by atomic kernel K^{2^*}).

Proposition 6. *Let S be a complete schedule and $\mathcal{K}$ be the set of the kernels detected prior to the creation of schedule S. The starting time of every atomic kernel K^*, $K \in \mathcal{K}$, can be increased by $\delta(K^*)$ time units (compared to its starting time in schedule $S^*[K]$) without increasing the maximum lateness $L_{\max}(S)$.*

Proof. Let $(K')^*$, $K' \in \mathcal{K}$, be an atomic kernel that achieves lower bound $L^*_{\max}$, i.e., $L_{\max}((K')^*) = L^*_{\max}$ (equivalently, $\delta((K')^*) = 0$). By Equation (7), if the completion time of every job in atomic kernel $K^* \neq (K')^*$ is increased by $\delta(K^*)$, the lateness of none of these jobs may become greater than that of the overflow job from kernel $(K')^*$, which proves the proposition as $L_{\max}(S) \ge L^*_{\max}$. □

We immediately obtain the following corollary:

Corollary 3. *In an optimal schedule S_{opt}, every atomic kernel K^*, $K \in \mathcal{K}$, starts either no later than at time $r(K^*) + \delta(K^*)$ or no later than at time $r(K^*) + \delta(K^*) + \delta$, for some $\delta \ge 0$.*

An extra delay δ might be unavoidable for a proper accommodation of the non-kernel jobs. Informally, δ is the maximum extra delay that we will allow for every atomic kernel in the iteration of the binary search procedure with trial value δ. For a given iteration in the binary search procedure with trial δ, the corresponding threshold, an upper limit on the currently allowable maximum job lateness, *L_δ-boundary* (or L-boundary) is

$$L_\delta = L^*_{\max} + \delta = L_{\max}(K^*) + \delta(K^*) + \delta \ (K \in \mathcal{K}). \tag{8}$$

We call $L_\delta - schedule$ a feasible schedule in which the maximum lateness of any job is at most $L_\delta = L^*_{\max} + \delta$ (see Equation (8)).

Note that, since to every iteration a particular δ corresponds, the maximum allowable lateness at different iterations is different. The concept of the overflow job at a given iteration is consequently redefined: such a job *must* have the lateness greater than L_δ. Note that this implicitly redefines also the notation of a kernel at that iteration of the binary search procedure.

It is not difficult to determine the time interval from which the trial δs can be derived. Let Δ be the delay of kernel $K(\sigma)$ imposed by the delaying emerging Job l in initial ED-schedule σ, i.e.,

$$\Delta = c_l(\sigma) - r(K(\sigma)). \tag{9}$$

Example 1 (continuation). For the problem instance of Example 1, for instance, $\Delta = 16 - 3 = 13$ (see Figure 1).

Proposition 7.

$$L_{max}(\sigma) - L^*_{\max} \leq \Delta. \tag{10}$$

Proof. This is a known property that easily follows from the fact that no job of kernel $K(\sigma)$ could have been released by the time $t_l(\sigma)$, as otherwise ED-heuristics would have been included the former job instead of Job l in schedule σ. □

Assume, for now, that we have a procedure that, for a given L-boundary (see Equation (8)), finds an L-schedule S^L if it exists, otherwise, it outputs a "no" answer.

Then, the binary search procedure incorporates the above verification procedure as follows. Initially, for $\delta = \Delta$, $L^*_{\max} + \Delta$-schedule σ already exists. For $\delta = 0$ with $L = L^*_{\max}$, if there exists no $L^*_{\max}$-schedule then the next value of δ is $\lceil \Delta/2 \rceil$. Iteratively, if an L-schedule with $L = L^*_{\max} + \delta$ for the current δ exists, the δ is increased correspondingly, otherwise it is decreased correspondingly in the binary search mode.

Proposition 8. *The L-schedule S^L corresponding to the minimum $L = L^*_{\max} + \delta$ found in the binary search procedure is optimal.*

Proof. First, we show that trial δs can be derived from the interval $[0, \Delta]$. Indeed, the left endpoint of this interval can clearly be 0 (potentially yielding a solution with the objective value $L^*_{\max}$). By the inequality in Equation (10), the maximum job lateness in any feasible ED-schedule in which the delay of some kernel is more than Δ would be no less than $L_{\max}(\sigma)$, which obviously proves the above claim.

Now note that the minimum L-boundary yields the minimal possible lateness for the kernel jobs subject to the condition that no non-kernel job surpasses L-boundary. This obviously proves the proposition. □

By Proposition 8, the problem $1|r_j|L_{\max}$ can be solved, given that there is a verification procedure that, for a given L-boundary, either constructs L_δ-schedule S^{L_δ} or answers correctly that it does not exist. The number of iterations in the binary search procedure is bounded by $\log p_{\max}$ as clearly, $\Delta < p_{\max}$. Then, note that the running time of our basic framework is $\log p_{\max}$ multiplied by the running time of the verification procedure. The rest of this paper is devoted to the construction of the verification procedure, invoked in the binary search procedure for trial δs.

6. The General Framework for Problem $1|r_j|L_{\max}$

In this section, we describe our main algorithmic framework which basic components form the binary search and the verification procedures. The framework is for the general setting $1|r_j|L_{\max}$ (in the next section, we give an explicit condition when the framework guarantees the optimal solution of the problem). At every iteration in the binary search procedure, we intend to keep the delay of jobs from each partial schedule $S^*[K]$, $K \in \mathcal{K}$ within the allowable margin determined by the current L_δ-boundary.

For a given threshold L_δ, we are concerned with the existence of a partial L_δ-schedule that includes all the jobs of schedule $S^*[K]$ and probably some external jobs. We refer to such partial schedule as an *augmented L_δ-schedule* for kernel K and denote it by $S^{L_\delta}[K]$ (we specify the scope of that schedule more accurately later in this section).

Due to the allowable maximum job lateness of $L_\delta \geq L_{\text{opt}}$ in schedule $S^{L_\delta}[K]$, in the case that the earliest scheduled job of kernel K^* gets pushed by some (external) job l^* in schedule $S^{L_\delta}[K]$, that job will be considered as the delaying emerging job iff

$$c_{l^*}(S^L[K]) \geq r(K^*) + \delta(K^*) + \delta.$$

For a given threshold $L = L_\delta$, the allowable L-*bias* for jobs of kernel K^* in schedule $S^L[K]$

$$\beta_L(K^*) = L - L_{\max}(K^*). \tag{11}$$

The intuition behind this definition is that the jobs of kernel K^* in schedule $S^*[K]$ can be right-shifted by $\beta_L(K^*)$ time units without surpassing the L-boundary (see Proposition 9 below).

Proposition 9. *In an L-schedule S^L, all the jobs of schedule $S^*[K]$ are included in the interval of schedule $S^*[K]$. Furthermore, any job in $S^*[K] \setminus K^*$ can be right-shifted provided that it remains scheduled before the jobs of kernel K^*, whereas the jobs from kernel K^* can be right-shifted by at most $\beta_L(K^*)$.*

Proof. Let j be the earliest scheduled job of atomic kernel K^* in schedule $S^*[K]$. By right-shifting job j by $\beta_L(K^*)$ time units (Equation (11)) we get a new (partial) schedule S' in which all the jobs are delayed by $\beta_L(K^*)$ time units with respect to schedule $S^*[K]$ (note that the processing order of the jobs of atomic kernel K^* need not be altered in schedule S' as the jobs of kernel K^* are scheduled in ED-order in schedule $S^*[K]$). Hence,

$$\max_{i \in S^*[K]} L_i(S') \leq \max_{i \in S^*[K]} \{L_i(S^*[K]) + \beta_L(K^*)\} = \max_{i \in S^*[K]} \{L_i(S^*[K])\} + \beta_L(K^*).$$

By substituting for $\beta_L(K^*)$ using Equation (11) and applying that $\max_{i \in S^*[K]}\{L_i(S^*[K])\} = L_{\max}(K^*)$, we obtain

$$\max_{i \in S^*[K]} L_i(S') \leq L.$$

Hence, the lateness of any job of atomic kernel K^* is no more than L. Likewise, any other job from schedule $S^*[K]$ can be right-shifted within the interval of $S^*[K]$ without surpassing the magnitude $L_{\max}(K^*) \leq L$ given that it is included before the jobs of kernel K^* (see the proof of Lemma 4). □

6.1. Partitioning the Scheduling Horizon into the Bin and Kernel Segments

By Proposition 9, all jobs from the atomic kernel K^* are to be included with a possible delay (right-shift) of at most $\beta_L(K^*)$ in L-schedule S^L. The rest of the jobs from schedule $S^*[K]$ are to "dispelled" before the jobs of K^* within the interval of that schedule. Since schedule $S^*[K]$ contains the gaps, some additional external jobs may also be included within the same time interval. According to this observation, we partition every complete feasible L-schedule into two types of segments, rigid and flexible ones. The rigid segments are to be occupied by the atomic kernels, and the rest of the (flexible) segments, which are called *bin* segments or intervals, are left for the rest of the jobs (we use term bin for both, the corresponding time interval and for the corresponding schedule portion interchangeably). For simplicity, we refer to the segments corresponding to the atomic kernels as kernel segments or intervals.

In general, we have a bin between two adjacent kernel intervals, and a bin before the first and after the last kernel interval. Because of the allowable right-shift $\beta_L(K^*)$ for the jobs of an atomic kernel K^*, the starting and completion times of the corresponding kernel and bin intervals are not priory fixed. We denote by $B^-(K)$ ($B^+(K)$, respectively) the bin before (after, respectively) the kernel interval corresponding to the atomic kernel K^* of kernel K. There are two bins in schedule $\sigma_{l,3,2}$, surrounding the atomic kernel consisting of Job 1 in Figure 6. We have three bins in schedules depicted in

Figures 8 and 9 for the problem instance of Example 2, $B_1 = B^-(K^1)$, $B_2 = B^+(K^1) = B^-(K^2)$ and $B_3 = B^+(K^2)$ (schedule of Figure 9 incorporates an optimal arrangement of jobs in these bins).

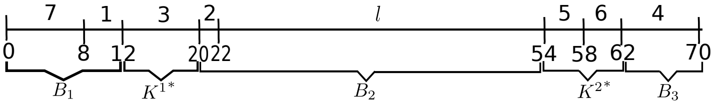

Figure 9. An optimal L-schedule for Example 2 with three bins ($L = L^*_{\max} = 4$).

The scope of augmented L-schedule $S^L[K]$ for kernel K includes that of bin $B^-(K)$ and that of the atomic kernel K^*. These two parts are scheduled independently. The construction of second part relies on the next proposition that easily follows from Proposition 9:

Proposition 10. *No job of the atomic kernel K^* will surpass the L-boundary if the latest scheduled job of bin $B^-(K)$ completes no later than at time moment*

$$\psi_L(K) = r(K^*) + \beta_L(K^*) \tag{12}$$

(the latest time moment when atomic kernel K^* may start in an L-schedule) and the jobs of that kernel are scheduled by ED-heuristics from time moment $\psi_L(K)$.

We easily arrange the second part of augmented schedule $S^L[K]$, i.e., one including the atomic kernel K^*, as specified in Proposition 10. Hence, from here on, we are solely concerned with the construction of the the first part, i.e., that of bin $B^-(K)$, which is a complicated task and basically contributes to the complexity status of problem $1|r_j|L_{\max}$.

We refer to a partial feasible L-schedule for the first part of schedule $S^L[K]$ (with its latest job completion time not exceeding the magnitude $\psi_L(K)$, at which the second part initiates) as a *preschedule* of kernel K and denote it by $PreS(K)$. Note that the time interval of preschedule $PreS(K)$ coincides with that of bin $B^-(K)$; in this sense, $PreS(K)$ is a schedule for bin $B^-(K)$.

Kernel preschedules are generated in Phase 1, described in Section 7. If Phase 1 fails to construct an L-preschedule for some kernel, then Phase 2 described in Section 9 is invoked (see Proposition 12 in Section 5). Phase 2 basically uses the construction procedure of Phase 1 for the new problem instances that it derives.

6.1.1. The Main Partitioning Procedure

Now, we describe the main procedure (PROCEDURE MAIN) of our algorithm, that is in charge of the partitioning of the scheduling horizon into the kernel and the corresponding bin intervals. This partition is dynamically changed and is updated in a recurrent fashion each time a new kernel arises. The occurrence of each new kernel K during the construction of a bin, the split of this bin into smaller bins and the collapsing of kernel K induce the recurrent nature in our method (not surprising, the recurrence is a common feature in the most common algorithmic frameworks such are dynamic programming and branch-and-bound).

Invoked for kernel K (K is a global variable), PROCEDURE MAIN first calls PROCEDURE Decomposition(S, K, l) that forms schedule $S^*[K]$ ending with the atomic kernel K^* (see the beginning of Section 5 and Propositions 5 and 9).

PROCEDURE MAIN incorporates properly kernel K into the current partition updating respectively the current *configuration* $\mathcal{C}(\delta, K)$ defined by a trial δ, the current set of kernels $\mathcal{K}$ together with the corresponding $\delta(M^*)$s (see Equation (7)) and the augmented schedules $S^{L_\delta}[M]$, for $M \in \mathcal{K}$, constructed so far.

Given trial δ and kernel K, the configuration $\mathcal{C}(\delta, K)$ is unique, and there is a unique corresponding schedule $\Sigma_{\mathcal{C}(\delta,K)}$ with $K = K(\Sigma_{\mathcal{C}(\delta,K)})$ that includes the latest generated (so far) augmented schedules $S^{L_\delta}[M]$, $M \in \mathcal{K}$.

PROCEDURE MAIN starts with the initial configuration $\mathcal{C}(\Delta, K)$ with $\delta = \Delta$, $K = K(\sigma)$, $\Sigma_{\mathcal{C}(\Delta,K)} = \sigma$ and $\mathcal{K} = \emptyset$ (no bin exists yet in that configuration).

Iteratively, PROCEDURE MAIN, invoked for kernel K, creates a new configuration $\mathcal{C}(\Delta, K)$ with two new surrounding bins $B^-(K)$ and $B^+(K)$ and the atomic kernel K^* in between these bins. These bins arise within a bin of the previous configuration (the later bin disappears in the updated configuration). Initially, atomic kernel $(K(\sigma))^*$ splits schedule σ in two bins $B^-(K(\sigma))$ and $B^+(K(\sigma))$.

Two (atomic) kernels in schedule $\Sigma_{\mathcal{C}(\delta,K)}$ are *tied* if they belong to the same block in that schedule.

Given configuration $\mathcal{C}(\delta, K)$, the longest sequence of the augmented L-schedules of the pairwise tied kernels in schedule $\Sigma_{\mathcal{C}(\delta,K)}$ is called a *secondary block*.

We basically deal with the secondary block containing kernel K and denote it by $\mathcal{B}_K$ (we may omit argument K when this is not important). An essential characteristic of a secondary block is that every job that pushes a job from that secondary block belongs to the same secondary block. Therefore, the configuration update in PROCEDURE MAIN can be carried out solely within the current secondary block $\mathcal{B}_K$.

As we show below, PROCEDURE MAIN will create an L-schedule for an instance of $1|p_j : divisible, r_j|L_{\max}$ whenever it exists (otherwise, it affirms that no L-schedule for that instance exists). The same outcome is not guaranteed for an instance of $1|r_j|L_{\max}$, in general. In Theorem 3, we give an explicit condition under which an L-schedule for an instance of $1|r_j|L_{\max}$ will always be created, yielding a polynomial-time solution for the general setting. Unfortunately, if the above condition is not satisfied, we cannot, in general, affirm that there exists no feasible L-augmented schedule, even if our framework fails to find it for an instance of problem $1|r_j|L_{\max}$.

6.1.2. PROCEDURE AUGMENTED(K, δ), Rise of New Kernels and Bin Split

PROCEDURE MAIN uses PROCEDURE AUGMENTED(K, δ) as a subroutine. PROCEDURE AUGMENTED(K, δ), called for kernel K with threshold L_δ, is in charge of the creation of an L_δ-augmented schedule $S^{L_\delta}[K]$ respecting the current configuration $\mathcal{C}(\delta, K)$. PROCEDURE AUGMENTED(K, δ) constructs the second part of schedule $S^{L_\delta}[K]$ (one including the atomic kernel K^*) directly as specified in Proposition 10. The most time consuming part of PROCEDURE AUGMENTED(K, δ) is that of the construction of the preschedule $PreS(K)$ of schedule $S^{L_\delta}[K]$. This construction is carried out at Phase 1 described in Section 7.

After a call of PROCEDURE AUGMENTED(K, δ), during the construction of an L-preschedule $PreS(K)$ at Phase 1, a new kernel K' may arise (the reader may have a look at Proposition 12 and Lemma 5 from the next section). Then, PROCEDURE AUGMENTED(K, δ) returns the newly arisen kernel K' and PROCEDURE MAIN, invoked for that kernel, updates the current configuration. Since the rise of kernel K' splits the earlier bin $B^-(K)$ into two new surrounding bins $B^-(K')$ and $B^+(K') = B^-(K)$ of the new configuration, the bin $B^-(K)$ of the previous configuration disappears and is "replaced" by a new bin $B^-(K) = B^+(K')$ of the new configuration. Correspondingly, the scope of a preschedule for kernel K is narrowed (the former bin $B^-(K)$ is "reduced" to the newly arisen bin $B^+(K') = B^-(K)$).

In this way, as a result of the rise of a new kernel within the (current) bin $B^-(K)$ and the resultant bin split, PROCEDURE AUGMENTED(K, δ) may be called more than once for different (gradually decreasing in size) bins: The initial bin $B^-(K)$ splits into two bins, the resultant new smaller bin $B^-(K)$ may again be split, and so on. Thus, to the first call of PROCEDURE AUGMENTED(K, δ) the largest bin $B^-(K)$ corresponds, and the interval of the new arisen bin for every next call of the procedure is a proper sub-interval of that of the bin corresponding to the previous call of the procedure. Note that each next created preschedule is composed of the jobs from the corresponding bin.

PROCEDURE AUGMENTED(K, δ) has three outcomes. If no new kernel during the construction of preschedule $PreS(K)$ respecting the current configuration arises, the procedure completes with the successful outcome generating an L-augmented schedule $S^{L_\delta}[K]$ respecting the current configuration (in this case, schedule $S^{L_\delta}[K]$ may form part of the complete L-augmented schedule if the later schedule exists). PROCEDURE MAIN incorporates L_δ-augmented schedule $S^{L_\delta}[K]$ into the current configuration (the first IF statement in the iterative step in the description of the next subsection).

With the second outcome, a new kernel K' during the construction of preschedule $PreS(K)$ within bin $B^-(K)$ arises (Proposition 12 and Lemma 5). Then, PROCEDURE AUGMENTED(K, δ) returns kernel K' and PROCEDURE MAIN is invoked for this newly arisen kernel and it updates the current configuration, respectively (see the iterative step in the description). Then, PROCEDURE MAIN calls recursively PROCEDURE AUGMENTED(K', δ) for kernel K' and the corresponding newly arisen bin $B^-(K')$ (this call is now in charge of the generation of an L-preschedule $PreS(K')$ for kernel K', see the second IF statement in the iterative step of the description in the next subsection).

With the third (failure) outcome, Phase 1 (invoked by PROCEDURE AUGMENTED(K, δ) for the creation of an L-preschedule $PreS(K)$) fails to create an L-preschedule respecting the current configuration (an IA(b2), defined in the next section, occurs (see Proposition 12). In this case, PROCEDURE MAIN invokes Phase 2. Phase 2 is described in Section 9. Nevertheless, the reader can see a brief description of that phase below:

Phase 2 uses two subroutines, PROCEDURE sl-SUBSTITUTION(K) and PROCEDURE ACTIVATE(s), where s is an emerging job. PROCEDURE sl-SUBSTITUTION(K) generates modified configurations with an attempt to create an L-preschedule $PreS(K)$ respecting a newly created configuration, in which some preschedules of the kernels, preceding kernel K in the secondary block $\mathcal{B}_K$ are reconstructed. These preschedules are reconstructed by the procedure of Phase 1, which is called by PROCEDURE ACTIVATE(s). PROCEDURE ACTIVATE(s), in turn, is repeatedly called by PROCEDURE sl-SUBSTITUTION(K) for different emerging jobs in the search of a proper configuration (each call of PROCEDURE ACTIVATE(s) creates a new configuration by a call of Phase 1). If at Phase 2 a configuration is generated for which Phase 1 succeeds to create an L-preschedule $PreS(K)$ respecting that configuration (the successful outcome), the augmented L-schedules corresponding to the reconstructed preschedules remain incorporated into the current schedule $\Sigma_{\mathcal{C}(\delta,K)}$.

6.1.3. Formal Description of PROCEDURE MAIN

The formal description of PROCEDURE MAIN below is completed by the descriptions of Phases 1 and 2 in the following sections. For notation simplicity, in set operations, we use schedule notation for the corresponding set of jobs. Given a set of jobs A, we denote by $ED(A)$ the ED-schedule obtained by the application of ED-heuristics to the jobs of set A.

Whenever a call of PROCEDURE MAIN for kernel K creates an augmented L-schedule $S^{L_\delta}[K]$, the procedure completes secondary block $\mathcal{B}_K$ by merely applying ED-heuristics to the remaining available jobs, ones to be included in that secondary block; i.e., partial ED-schedule $ED(\mathcal{B}_K \setminus \cup_{M \in \mathcal{B}_K}\{S^{L_\delta}[M]\})$ is generated and is merged with the already created part of block $\mathcal{B}_K$ to complete the block (the rest of the secondary blocks are left untouched in the updated schedule $\Sigma_{\mathcal{C}(\delta,K)}$).

PROCEDURE MAIN returns L_δ-schedule with the minimal δ, which is optimal by Lemma 8.

```
    PROCEDURE MAIN
    Initial step: {Determine the initial configuration $\mathcal{C}(\Delta, K)$, $K = K(\sigma)$}
    Start the binary search with trial $\delta = \Delta$
{see Equation (9) and the inequality in Equation (10)}
    $\Sigma_{\mathcal{C}(\Delta,K)} := \sigma$
{initialize the set of kernels}
    $K := K(\sigma)$; $\mathcal{K} := K$
{set the initial lower bound and the initial allowable delay for kernel $K$}
    $L^*_{\max} := L_{\max}(K^*)$; $\delta(K^*) := 0$
    IF schedule $\sigma$ contains no kernel with the delaying emerging job, output $\sigma$ and halt
{$\sigma$ is optimal by Proposition 1}

    Iterative step:
```

```
    {Update the current configuration $\mathcal{C}(\delta, K)$ with schedule $\Sigma_{\mathcal{C}(\delta,K)}$ as follows:}
    {update the current set of kernels}
    $\mathcal{K} := \mathcal{K} \cup K$;
    {update the current lower bound}
    $L^*_{\max} := \max\{L^*_{\max}, L_{\max}(K^*)\}$;
    {update the corresponding allowable kernel delays (see Equation (7))}
    $\delta(M^*) := L^*_{\max} - L_{\max}(M^*)$, for every kernel $M \in \mathcal{K}$
    Call PROCEDURE AUGMENTED($K, \delta$) {construct an $L_\delta$-augmented schedule $S^{L_\delta}[K]$}
    IF during the execution of PROCEDURE AUGMENTED($K, \delta$) a new kernel $K'$ arises
{update the current configuration according to the newly arisen kernel}
    THEN $K := K'$; repeat Iterative step
    IF the outcome of PROCEDURE AUGMENTED($K, \delta$) is failure THEN call Phase 2
{at Phase 2 new configuration is looked for such that there exist preschedule $PreS(K)$ respecting that
configuration, see Section 9}
    IF $L_\delta$-augmented schedule $S^{L_\delta}[K]$ is successfully created
{the outcome of PROCEDURE AUGMENTED($K, \delta$) and that of Phase 2 is successful, hence complete
secondary block $\mathcal{B}_K$ by ED-heuristics if there are available jobs which were not included in any of the
constructed augmented schedules, i.e., $\mathcal{B}_K \setminus \cup_{M \in \mathcal{B}_K}\{S^{L_\delta}[M]\} \neq \emptyset$}
    THEN update block $\mathcal{B}_K$ and schedule $\Sigma_{\mathcal{C}(\delta,K)}$ by merging it with partial ED-schedule
        $ED(\mathcal{B}_K \setminus \cup_{M \in \mathcal{B}_K}\{S^{L_\delta}[M]\})$
        (leave in the updated schedule $\Sigma_{\mathcal{C}(\delta,K)}$ the rest of the secondary blocks as they are)
    IF (the so updated) schedule $\Sigma_{\mathcal{C}(\delta,K)}$ is an $L_\delta$-schedule
    {continue the binary search with the next trial $\delta$}
    THEN $\delta$ :=the next trial value and repeat Iterative step; return the generated $L_\delta$-schedule with the
        minimum $\delta$ and halt if all the trial $\delta$s were already considered
    ELSE {there is a kernel with the delaying emerging job in schedule $\Sigma_{\mathcal{C}(\delta,K)}$}
        $K := K(\Sigma_{\mathcal{C}(\delta,K)})$; repeat Iterative step
    IF $L_\delta$-augmented schedule $S^{L_\delta}[K]$ could not been created
{the outcome of Phase 2 is failure and hence there exists no $L_\delta$-schedule; continue the binary search
with the next trial $\delta$}
    THEN $\delta$ := the next trial value and repeat Iterative step; return the generated $L_\delta$-schedule with the
        minimum $\delta$ and halt if all the trial $\delta$s were already considered.
```

7. Construction of Kernel Preschedules at Phase 1

At Phase 1, we distinguish two basic types of the available (yet unscheduled) jobs which can feasibly be included in bin $B^-(K)$, for every $K \in \mathcal{K}$. Given a current configuration, we call jobs that can only be scheduled within bin $B^-(K)$ y-jobs; we call jobs which can also be scheduled within some succeeding bin(s) the *x-jobs* for bin $B^-(K)$ or for kernel K. In this context, y-jobs have higher priority.

We have two different types of the y-jobs for bin $B^-(K)$. The set of the *Type (a)* y-jobs is formed by the jobs in set $K \setminus K^*$ and yet unscheduled jobs not from kernel K released within the interval of bin $B^-(K)$. The rest of the y-jobs are ones released before the interval of bin $B^-(K)$, and they are referred to as the *Type (b)* y-jobs.

Recall that the interval of bin $B^-(K)$ begins right after the atomic kernel of the preceding bin (or at $\min_i r_i$ if K is the earliest kernel in $\mathcal{K}$) and ends with the interval of schedule $S^*[K]$. The following proposition immediately follows:

Proposition 11. *Every x-job for bin $B^-(K)$ is an external job for kernel K, and there may also exist the external y-jobs for that kernel. A Type (a) y-job can feasibly be scheduled only within bin $B^-(K)$, whereas Type (b) y-jobs can potentially be scheduled within a preceding bin (as they are released before the interval of bin $B^-(K)$).*

Phase 1 for the construction of preschedule $PreS(K)$ of kernel K consists of two passes. In Pass 1 y-jobs of bin $B^-(K)$ are scheduled. In Pass 2, x-jobs of bin $B^-(K)$ are distributed within that bin. We know that all Type (a) y-jobs can be feasibly scheduled within bin $B^-(K)$ without surpassing the L-boundary (since they were so scheduled in that bin), and these jobs may only be feasibly scheduled within that bin. Note that, respecting the current configuration with the already created augmented schedules for the kernels in set $\mathcal{K}$), we are forced to include, besides Type (a) y-jobs, also all the Type (b) y-jobs into bin $B^-(K)$. If this does not work at Phase 1 in the current configuration, we try to reschedule some Type (b) y-jobs to the earlier bins in Phase 2 by changing the configuration.

7.1. Pass 1

Pass 1 consists of two steps. In Step 1, ED-heuristics is merely applied to all the y-jobs for scheduling bin $B^-(K)$.

If the resultant ED-schedule $PreS(K,y)$ is a feasible L-schedule (i.e., no job in it surpasses the current L-boundary and/or finishes after time $\psi_L(K)$), Step 1 completes with the successful outcome and Pass 1 outputs $PreS(K,y)$ (in this case, there is no need in Step 2), and Phase 1 continues with Pass 2 that augments $PreS(K,y)$ with x-jobs, as described in the next subsection.

If schedule $PreS(K,y)$ is not an L-schedule (there is a y-job in that schedule surpassing the L-boundary), *Pass 1* continues with Step 2.

Proposition 12 specifies two possible cases when preschedule $PreS(K,y)$ does not contain all the y-jobs for bin $B^-(K)$, and Step 1 fails to create an L-preschedule for kernel K at the current configuration.

Proposition 12. *Suppose $PreS(K,y)$ is not a feasible L-schedule, i.e., there arises a y-job surpassing the current L-boundary and/or completing after time $\psi_L(K)$.*

(1) If there is a Type (b) y-job surpassing the L-boundary, then there exists no feasible partial L-preschedule for kernel K containing all the Type (b) y-jobs for this kernel (hence there is no complete feasible L-schedule respecting the current configuration).

(2) If there is a Type (a) y-job y surpassing the L-boundary and there exists a feasible partial L-preschedule for kernel K containing all the y-jobs, it contains a new kernel consisting of some Type (a) y-jobs including job y.

Proof. We first show Case (2). As already mentioned, all Type (a) y-jobs may potentially be included in bin $B^-(K)$ without surpassing the L-boundary and be completed by time $\psi_L(K)$ (recall Equation (12)). Hence, since y is a Type (a) y-job, it should have been pushed by at least one y-job i with $d_i > d_y$ in preschedule $PreS(K,y)$. Then, there exists the corresponding kernel with the delaying emerging y-job (containing job y and possibly other Type (a) y-jobs).

Now, we prove Case (1). Let y be a Type (b) y-job that was forced to surpass the L-boundary and/or could not be completed by time moment $\psi_L(K)$. In the latter case, ED-heuristics could create no gap in preschedule $PreS(K,y)$ as all the Type (b) y-jobs were released from the beginning of the construction, and Case (1) obviously follows. In the former case, job y is clearly pushed by either another Type (b) y-job or a Type (a) y-job. Let k be a job pushing job y. Independently of whether k is a Type (a) or Type (b) y-job, since job y is released from the beginning of the construction and job k was included ahead job y, by ED-heuristics, $d_k \leq d_y$. Then, no emerging job for job y may exist in preschedule $PreS(K,y)$ and Case (1) again follows as all the Type (a) y-jobs must be included before time $\psi_L(K)$. □

For convenience, we refer to Case (1) in Proposition 12 as an *instance of Alternative (b2)* (IA(b2) for short) with Type (b) y-job y (we let y be the latest Type (b) y-job surpassing the L-boundary and/or completing after time $\psi_L(K)$). (The behavior alternatives were introduced in a wider context earlier in [13].) If an IA(b2) in bin $B^-(K)$ arises and there exists a complete L-schedule, then, in that schedule, some Type (b) y-job(s) from bin $B^-(K)$ is (are) included within the interval of some bin(s) preceding bin $B^-(K)$ in the current secondary block $\mathcal{B}_K$ (we prove this in Proposition 16 in Section 7).

In *Step 2*, Cases (1) and (2) are dealt with as follows. For Case (1) (an IA(b2)), Step 2 invokes PROCEDURE sl-SUBSTITUTION(K) of Phase 2. PROCEDURE sl-SUBSTITUTION(K) creates one or more new (temporary) configurations, as described in Section 7. For every created configuration, it reconstructs some bins, preceding bin $B^-(K)$ in the secondary block $\mathcal{B}_K$ incorporating some Type (b) y-jobs for bin $B^{(}K)$ into the reconstructed preschedules. The purpose of this is to find out if there exists an L-preschedule $PreS(K)$ respecting the current configuration and construct it if it exists.

For Case (2) in Proposition 12, Step 2 returns the newly arisen kernel K' and PROCEDURE MAIN is invoked with that kernel, which updates the current configuration respectively. PROCEDURE MAIN then returns the call to PROCEDURE AUGMENTED(K', δ) (see the description of Section 4) (note that, since PROCEDURE AUGMENTED(K', δ) invokes *Phase 1* now, for kernel K', Case (2) yields recursive calls of Phase 1).

7.2. Pass 2: DEF-Heuristics

If Pass 1 successfully completes, i.e., creates a feasible L-preschedule $PreS(\bar{K}, y)$, Pass 2, described in this subsection, is invoked (otherwise, IA(b2) with a Type (b) y-job from bin $B^-(K)$ arises and Phase 2 is invoked). Throughout this section, $PreS(K, y)$ stands for the output of Pass 1 containing all the y-jobs for bin $B^-(K)$. At Pass 2, the x-jobs released within the remaining available room in preschedule $PreS(K, y)$ are included by a variation of the *Next Fit Decreasing* heuristics, adopted for our scheduling problem with job release times. We call this variation *Decreasing Earliest Fit* heuristics, DEF-heuristics for short. It works with a *list* of x-jobs for kernel K sorted in non-increasing order of their processing times, the ties being broken by sorting jobs with the same processing time in the non-decreasing order of their due-dates.

DEF-heuristics, iteratively, selects next job x from the list and initially appends this job to the current schedule $PreS(K, y)$ by scheduling it at the earliest idle-time moment t' before time $\psi_L(K)$ (any unoccupied time interval in bin $B^-(K)$ before time $\psi_L(K)$ is an idle-time interval in that bin). Let $PreS(K, y, +x)$ be the resultant partial schedule, that is obtained by the application of ED-heuristics from time moment t' to job x and to the following y-jobs from schedule $PreS(K, y)$ which may possibly right-shifted in schedule $PreS(K, y, +x)$) (compared to their positions in schedule $PreS(K, y)$). In the description below, the assignment $PreS(K, y) := PreS(K, y, +x)$ updates the current partial schedule $PreS(K, y)$ according to the rearrangement in schedule $PreS(K, y, +x)$, removes job x from the list and assigns to variable x the next x-job from the list.

```
    PROCEDURE DEF($PreS(K, y), x$)
IF job $x$ completes before or at time $\psi_L(K)$ in schedule $PreS(K, y, +x)$ {i.e., $t' + p_x$ falls within the current bin}
THEN GO TO Step (A) {verify the conditions in Steps (A) and (B)}
ELSE remove job $x$ from the list {job $x$ is ignored for bin $B^-(K)$}; set $x$ to the next job from the list;
        CALL PROCEDURE DEF($PreS(K, y), x$)
(A) IF job $x$ does not push any y-job in schedule $PreS(K, y, +x)$ {$x$ can be scheduled at time moment $t'$ without the interference with any y-job, i.e., $t' + p_x$ is no greater than the starting time of the next y-job in preschedule $PreS(K, y)$} and it completes by time moment $\psi_L(K)$ in schedule $PreS(K, y, +x)$
    THEN $PreS(K, y) := PreS(K, y, +x)$; CALL PROCEDURE DEF($PreS(K, y), x$)
(B) IF job $x$ pushes some y-job in schedule $PreS(K, y, +x)$
    THEN {verify the conditions in Steps (B.1)–(B.3)}
    (B.1) IF in schedule $PreS(K, y, +x)$ no (right-shifted) y-job surpasses $L$-boundary and
            all the jobs are completed by time moment $\psi_L(K)$
          THEN $PreS(K, y) := PreS(K, y, +x)$; CALL PROCEDURE DEF($PreS(K, y), x$)
    (B.2) IF in schedule $PreS(K, y, +x)$ some y-job completes after time moment $\psi_L(K)$
    THEN set $x$ to the next x-job from the list and CALL PROCEDURE DEF($PreS(K, y), x$).
```

We need the following auxiliary lemma before we describe Step (B.3):

Lemma 5. *If a (right-shifted) y-job surpasses L-boundary in schedule $PreS(K, y, +x)$, then there arises a new kernel in that schedule (in bin $B^-(K)$) consisting of solely Type (a) y-jobs, and x is the delaying emerging job of that kernel.*

Proof. Obviously, by the condition in the lemma, there arises a new kernel in schedule $PreS(K, y, +x)$, call it K', and it consists of y-jobs following job x in schedule $PreS(K, y, +x)$. Clearly, x is the delaying emerging job of kernel K'. Such a right-shifted job y cannot be of Type (b) as otherwise it would have been included within the idle-time interval (occupied by job x) at Pass 1. Hence, kernel K' consists of only Type (a) y-jobs. □

Due to the above lemma, PROCEDURE DEF continues as follows:

(B.3) IF in schedule $PreS(K, y, +x)$ the lateness of some (right-shifted) y-job exceeds L
THEN return the newly arisen kernel K' and invoke PROCEDURE MAIN with kernel K' {this updates the current configuration respectively and makes a recursive call of Phase 1 now for kernel K'}

IF the list is empty THEN OUTPUT($PreS(K, y)$) and halt.

This completes the description of Pass 2 and that of Phase 1.

From here on, we let $PreS(K) = PreS(K, y, x)$ be the output of Phase 1 (a feasible L-preschedule for kernel K containing all the y-jobs for bin $B^-(K)$). An easily seen property of PROCEDURE DEF and preschedule $PreS(K, y, x)$ is summarized in the following proposition.

Proposition 13. *An L-preschedule cannot be obtained by replacing any x-job $x \in PreS(K, y, x)$ with a longer available x-job in preschedule $PreS(K, y, x)$. Hence, the omission of job x from preschedule $PreS(K, y, x)$ will create a new gap which may only be filled in by including job(s) with the same or smaller processing time.*

Let v and χ be the number of y-jobs and x-jobs of bin $B^-(K)$, respectively, $\nu = v + \chi$ is the total number of jobs in that bin, and let v_1 be the number of Type (b) y-jobs. The next theorem gives a valid upper bound on the cost of a call of PROCEDURE AUGMENTED(K, δ) at Phase 1 (including all the recursive calls that the initial call may yield).

Theorem 2. *The total cost of a call of Phase 1 for a kernel K is $O(\nu^2 \log \nu)$. Hence, the cost of a call of PROCEDURE AUGMENTED(K, δ) is the same.*

Proof. At Step 1 of Pass 1, during the construction of preschedule $PreS(K)$ ED-heuristics with an upper bound on its running time $O(v \log v)$ for scheduling up to v y-jobs is used, whereas at less than v_1 scheduling times a new kernel may arise (as the delaying emerging job may only be a Type (b) y-job). Phase 1 invokes PROCEDURE MAIN which, in turn, calls the decomposition procedure for each of these kernels. By Lemma 2, the total cost of all the calls of the decomposition procedure can be estimated as $O(\kappa_1^2 \log \kappa_1 + \kappa_2^2 \log \kappa_2 + \cdots + \kappa_{v_1}^2 \log \kappa_{v_1})$, where $\kappa_1, \ldots, \kappa_{v_1}$ is the number of jobs in each of the v_1 arisen kernels, correspondingly. Let m be the mean arithmetic of all these κs. Since any newly arisen kernel may contain only y-jobs for bin $B^-(K)$ and no two kernels may have a common job, $v_1 m \le v$. The maximum in the sum is reached when all the κs are equal to m, and from the above sum another no-smaller magnitude $O(v_1 m^2 \log m) \le O(v_1 (v/v_1)^2 \log(v/v_1)) \le O(v^2 \log v)$ is obtained (in the first and second inequalities, $v_1 m \le v$ and $v_1 \ge 1$, respectively, are applied).

Then, the total cost of Pass 1 for kernel K (including that of Step 2, Case (2)) is $O(v_1 v \log v + v^2 \log v) = O(v^2 \log v)$. The cost of Steps (A), (B.1) and (B.2) of Pass 2 is that of ED-heuristics, i.e., $O(\chi \log \chi)$. At Step (B.3), since the delaying emerging job for every newly arisen kernel is a distinct x-job for bin $B^-(K)$, the number of the calls of PROCEDURE MAIN for all the newly arisen kernels after the initial call of PROCEDURE AUGMENTED(K, δ), and hence the number of the recursive calls of Phase 1 for kernel K, is bounded by χ. Similar to what is done for Pass 1, we let $\kappa_1, \ldots, \kappa_{v_1}$ be the number of jobs in each of the χ arisen kernels, respectively. Again, by Lemma 2, the total cost of all the calls of PROCEDURE MAIN to the decomposition procedure is

$O(\kappa_1^2 \log \kappa_1 + \kappa_2^2 \log \kappa_2 + \cdots + \kappa_\chi^2 \log \kappa_\chi)$. We let again m be the mean arithmetic of all these κs, $\chi m \leq v$ and obtain an upper bound $O(\chi^2 \log \chi + \chi m^2 \log m) \leq O(\chi^2 \log \chi + \chi(v/\chi)^2 \log(v/\chi)) \leq O(\nu^2 \log \nu)$ on the cost of Pass 2 and hence the total cost of Phase 1 is $O(\nu^2 \log \nu)$.

The second claim in theorem follows as the cost of the generation of the second part of an augmented L_δ-schedule is absorbed by that of the first part. Indeed, recall that for a call of PROCEDURE AUGMENTED(K, δ), the second part of schedule $S^{L_\delta}[K]$ consisting of the jobs of the atomic kernel K^*, is constructed by ED-heuristics in time $O(\kappa' \log \kappa')$, where $\kappa' \leq \kappa$ is the total number of jobs in atomic kernel K^*, and κ is the number of jobs in kernel K (Proposition 10). Similar to above in this proof, we can show that the construction of the second part of the augmented schedules for the calls of PROCEDURE AUGMENTED for all the arisen kernels (for the same δ) is $O(n \log n)$. □

At this stage, we can give an optimality (sufficient) condition for problem $1|r_j|L_{\max}$, that is helpful also in that it exhibits where the complex nature of the problem is "hidden". Dealing with an IA(b2) is a complicated task as it implies the solution of NP-hard set/numerical problems such as 3-PARTITION yet with additional restrictions that impose job release times. As to the solution provided by PROCEDURE MAIN, as we have seen above, the recurrences at Step 2, Case (2) in Pass 1, and at Step (B.3) at Pass 2 do not, in fact, cause an exponential behavior.

Theorem 3. *PROCEDURE MAIN finds an optimal solution to problem* $1|r_j|L_{\max}$ *in time* $O(n^2 \log n \log p_{\max})$ *if no IA(b2) at Phase 1 arises.*

Proof. The proof is quite straightforward, we give a scratch. The initial step takes time $O(n \log n)$ (the cost of ED-heuristics). At iterative step, the cost of updates of $L^*_{\max}$ and $\delta(M^*)$, $M \in \mathcal{K}$ and that of the detection of every newly arisen kernel is bounded by the same magnitude. It is easy to see that an L-preschedule for every kernel in $\mathcal{K}$ will be generated at Phase 1 if no Type (b) y-job is forced to surpasses the L-boundary, or, equivalently, no IA(b2) arises (only a y-job may be forced to surpass the L-boundary, whereas, if a Type (a) y-job surpasses it, PROCEDURE MAIN proceeds with the newly arisen kernel). Hence, PROCEDURE AUGMENTED(K, δ) will create a feasible L-augmented schedule for every kernel (since no IA(b2) at Pass 1 arises). Then, it remains to estimate the calls of PROCEDURE AUGMENTED(K, δ) in the iterative step. The cost of a call of PROCEDURE AUGMENTED(K, δ) for a given kernel K including all the embedded recursive calls is $O(\nu^2 \log \nu)$ (Theorem 2). These recursive calls include the calls for all the kernels which may arise within bin $B^-(K)$. Hence, for the purpose of our estimation, it suffices to distinguish the calls PROCEDURE AUGMENTED(K, δ) and PROCEDURE AUGMENTED(M, δ) for two distinct kernels K and M such that bins $B^-(K)$ and $B^-(M)$ have no jobs in common. Then, similar to what is done to estimate the cost of Pass 1 in the proof of Theorem 2, we easily get an overall (amortized) cost of $O(n^2 \log n)$ for PROCEDURE MAIN for a given trial δ. Then, we obtain the overall cost of $O(n^2 \log n \log p_{\max})$ for PROCEDURE MAIN taking into account that there are no more than $\log p_{\max}$ trial δs. □

8. Construction of Compact Preschedules for Problem $1|p_j : divisible, r_j|L_{\max}$

This section starts Part 2, in which our basic task is to develop an auxiliary algorithm that deals with an IA(b2) occurred at Phase 1 (recall that if no IA(b2) occurs, PROCEDURE MAIN with PROCEDURE AUGMENTED(K, δ) using Phase 1 already solves problem $1|r_j|L_{\max}$). A compact feasible schedule, one without any redundant gap, has properties that are helpful for the establishment of the existence or the non-existence of a complete L-schedule whenever during the construction of a kernel preschedule at Phase 1 an instance of Alternative (b2) arises. In this section, we study the compactness properties for instances of problem $1|p_j : divisible, r_j|L_{\max}$.

Since the basic construction components of a complete feasible schedule are the secondary blocks, it suffices to deal with compact secondary blocks. A secondary block $\mathcal{B}$ is *compact* if there is no feasible L-schedule containing all the jobs of that block with the total length of all the gaps in it no-less than that in block $\mathcal{B}$.

We can keep the secondary blocks compact if the processing times of some non-kernel jobs are mutually divisible. For the commodity and without loss of generality, we assume that the processing times of the non-kernel jobs are powers of 2 (precisely, we identify the specific non-kernel jobs for which mutual divisibility is required on the fly). Below, we give a basic property of a set of divisible numbers and then we give another useful property of a kernel preschedule with divisible job processing times, which are used afterwards.

Lemma 6. *For a given job x, let $J^-(x)$ be the set of jobs $J^-(x) = \{i|p_i < p_x\}$ such that $p(J^-(x)) > p_x$ and the processing times of jobs in set $J^-(x) \cup \{x\}$ are mutually divisible. Then, there exists a proper subset J' of set $J^-(x)$ with $p(J') = p_x$ (that can be found in an almost liner time).*

Proof. The following simple procedure finds subset J'. Sort the jobs in set $J^-(x)$ in non-increasing order of their processing times, say $\{x_1, \ldots, x_k\}$. It is straightforward to see that, because of the divisibility of the processing times of the jobs in set $J^-(x) \cup \{x\}$, there exists integer $l < k$ such that $\sum_{\iota=1}^{l} x_\iota = p_x$, i.e., $J' = \{x_1, \ldots, x_k\}$. □

Lemma 7. *Preschedule $PreS(K, y, x)$, constructed at Pass 2 of Phase 1 for an instance of $1|p_j$: $divisible, r_j|L_{\max}$, contains no gap except one that may possibly arise immediately before time moment $\psi_L(K)$.*

Proof. By the way of contradiction, suppose I is an internal gap in schedule $PreS(K, y)$ of Pass 1. Note that initially, gap I was completely occupied in bin $B^-(K)$ in schedule σ, and that it is succeeded by at least one y-job in preschedule $PreS(K, y)$. That is, the x-jobs with the total length of at least $|I|$ should have been available while scheduling the interval of gap I in PROCEDURE DEF at Pass 2. Then, an idle time interval within the interval of gap I in preschedule $PreS(K, y, x)$ of Pass 2 may potentially occur only at the end of that interval, say at time moment τ, due to the non-permitted interference in schedule $PreS(K, y, +x)$ of an available (and not yet discarded) x-job with a succeeding y-job, say y (Step (B)). Note that job y is a Type (a) y-job (if it were of Type (b), then it would have been included ahead any x-job in bin $B^-(K)$) and that the lateness of that job did not exceed L before kernel K was detected in schedule $\Sigma(\mathcal{C}(\delta, K))$. Let X be the set of the x-jobs preceding job y in the interval of gap I in the latter schedule, and X' be the corresponding set of the x-jobs in preschedule $PreS(K, y, x)$ (by our construction, $P(X) > P(X')$). In PROCEDURE DEF, during the construction of schedule $PreS(K, y, x)$, at time moment τ there must have been no job with processing time $p(X) - p(X')$ or less available. However, this is not possible since, because of the divisibility of job processing times, set X must contain such a job (and that job must have been available and yet unscheduled). The existence of a gap from time moment τ in the interval of gap I in schedule $PreS(K, y, x)$ has led to a contradiction and hence it cannot exist. □

In the rest of this section, we assume that preschedule $PreS(K)$ contains a gap; i.e., it ends with a gap (Lemma 7). Our objective is to verify if that gap can be reduced. To this end, we define two kinds of jobs such that their interchange may possibly be beneficial.

The first type of jobs are formed from set $EP(K, L)$, the set of the passive emerging jobs for kernel K in the current configuration with threshold $L = L_\delta$. Recall that a job from set $EP(K, L)$ is included after kernel K in schedule $\Sigma_{\mathcal{C}(\delta,K)}$ but it may feasibly be included (as an x-job) in a preschedule of kernel K (in bin $B^-(K)$).

Recall at the same time, that a job from preschedule $PreS(K)$ which may be rescheduled after all jobs of kernel K without surpassing the L-boundary is one from set $E(K, L)$, the set of emerging jobs for kernel K at the current configuration (such a job was included as an x-job in preschedule $PreS(K)$).

A vulnerable component of a secondary block is a preschedule in it, in the sense that we can maintain a secondary block compact if every *preschedule* that it contains is also *compact*, i.e., there exists no other preschedule (for the same kernel) with the total length of the gaps less than that in the former preschedule (see Corollary 4 at the end of this section). A key informal observation here is that, if a preschedule for kernel K is not compact, then a compact one can only be obtained from the former preschedule by replacing some jobs from set $E(K, L)$ with some jobs from set $EP(K, L)$, whereas nothing is to be gained by substituting any jobs from a compact preschedule by any jobs from set $EP(K, L)$ (Proposition 14 below).

Let $A \subseteq E(K, L)$ and $B \subseteq EP(K, L)$. Consider A and B as potential "swap" subsets and denote by $PreS(K, -A, +B)$ the preschedule for kernel K obtained by interchanging the roles of jobs from sets A and B while reconstructing the current preschedule $PreS(K)$ by the procedure of Phase 1. Technically, preschedule $PreS(K, -A, +B)$ can be constructed at Phase 1 for the restricted problem instance PI$(PreS(K), -A, +B)$ that contains all jobs from preschedule $PreS(K)$ and set B but does not contain ones in set A (so jobs from set A are activated for kernel K). Note that a job from set A belongs to $PreS(K, -A, +B)$, and, along with the remaining jobs from preschedule $PreS(K)$, some job(s) from set B may also be included in $PreS(K, -A, +B)$.

Proposition 14. *If an L-preschedule $PreS(K)$ is not compact then there exist sets A and B such that an L-preschedule $PreS(K, -A, +B)$ is compact.*

Proof. Among the jobs included in schedule $\Sigma_{\mathcal{C}(\delta,K)}$ after preschedule $PreS(K)$, the available room (the gap) from preschedule $PreS(K)$ may only potentially be used by job(s) from set $EP(K, L)$. By the construction of Phase 1, this will not be possible unless some emerging job(s) from preschedule $PreS(K)$ is (are) rescheduled after kernel K. Then, this kind of the interchange of the jobs from set $E(K, L)$ with the jobs from set $EP(K, L)$ yields the only potentially improving rearrangement of the jobs in preschedule $PreS(K)$, and the proposition follows. □

Let us say that set A *covers* set B if preschedule $PreS(K, -A, +B)$ includes all jobs from problem instance PI$(PreS(K), -A, +B)$. Since we wish to reduce the total gap length in preschedule $PreS(K)$, $p(A) < p(B)$ must hold, which is our assumption from now on (we use $p(A)$ for the total processing time in job-set A; below, we use $p_{min}\{A\}$ for the minimum job processing time in A).

Let $\gamma(K)$ the total gap length in preschedule $PreS(K) \in \Sigma_{\mathcal{C}(\delta,K)}$. We call

$$ST(K) = \gamma(K) + \beta_L(K^*) \tag{13}$$

the *store* of kernel K in the current configuration $\mathcal{C}(\delta, K)$. It is easily seen that $ST(K)$ is the maximum available vacant room in preschedule $PreS(K) \in \Sigma_{\mathcal{C}(\delta,K)}$:

Proposition 15. *The total length of the jobs (the gaps, respectively) in preschedule $PreS(K) \in \Sigma_{\mathcal{C}(\delta,K)}$ might be increased (decreased, respectively) by at most $ST(K)$ time units in any L-preschedule for kernel K. If set A covers set B, then the store of kernel K in an updated configuration with preschedule $PreS(K, -A, +B)$ is*

$$ST(K) - (P(B) - P(A)).$$

Lemma 8. *If $ST(K) < p_{min}\{E(K, L)\}$, then preschedule $PreS(K)$ is compact. If preschedule $PreS(K)$ is not compact, then $ST(K) \geq p_{min}\{A\}$, for any $A \subseteq E(K, L)$.*

Proof. By the condition in lemma, the gap in preschedule $PreS(K)$ (see Lemma 7) can potentially be occupied only by a job j with $p_j \leq p_{min}\{E(K, L)\}/2$ (see Proposition 15). There may exist no such job in set $EP(K, L)$ as otherwise it would have been included in preschedule $PreS(K)$ as an x-job at Pass 2. Now, it can be straightforwardly seen that no interchange of jobs in set $E(K, L)$ from preschedule $PreS(K)$ with jobs from set $EP(K, L)$ may reduce the gap, because of the divisibility of the processing times of the jobs in sets $E(K, L)$ and $EP(K, L)$, and the first claim in lemma follows from Proposition 14.

Now, we show the second claim. Suppose preschedule $PreS(K)$ is not compact. Then, there exist sets A and B such that A covers B and preschedule $PreS(K, -A, +B)$ results in the reduction of the store of kernel K by $p(B) - p(A)$ (see Equation (13) and Propositions 14 and 15). Because of the divisibility of job processing times in sets A and B, $p(B) - p(A)$ is a multiple of $p_{min}\{A \cup B\}$. Hence, if $ST(K) < p_{min}\{A \cup B\}$, then preschedule $PreS(K)$ is compact; $ST(K) \geq p_{min}\{A \cup B\}$ must hold if $PreS(K)$ is not compact. $ST(K) \geq p_{min}\{B\}$ is not possible, as otherwise a job from set B with processing time $p_{min}\{B\}$ would have been included in preschedule $PreS(K)$ at Pass 2 of Phase 1. It follows that $ST(K) \geq p_{min}\{A\}$. □

Due to Lemma 8, from here on, assume that $ST(K) \geq p_{min}\{A\}$. It is not difficult to see that not all $ST(K)$ time units may potentially be useful. In particular, let $\nu \geq 1$ be the maximum integer such that $ST(K) \geq \nu p_{min}\{A\}$, and let $p' = \nu p_{min}\{A\}$.

Lemma 9. *A feasible L-preschedule $PreS(K, -A, +B)$ contains gap(s) with the total length of at least $ST(K) - p'$; hence, $p(B) \leq p(A) + p'$ when set A covers set B. Furthermore, $p_{min}(EP(K, L)) = 2^{\kappa} p_{min}\{A\}$, for some integer $\kappa \geq 1$, and $p' \leq 2^{\kappa} p_{min}\{A\}$.*

Proof. The first claim easily follows from the definitions and the mutual divisibility of the processing times of jobs in sets A and B, and inequality $p(B) \leq p(A) + p'$ immediately follows. As to the second claim, first we note that, for any $\pi \in EP(K, L)$, $p_{\pi} > ST(K)$, as otherwise job π would have been included in preschedule $PreS(K)$ at Pass 2 of Phase 1. Then, $p_{min}(EP(K, L)) > p'$, whereas $p' \geq p_{min}\{A\}$. Hence, $p_{min}(EP(K, L)) > p_{min}\{A\}$. Now, the second claim follows from the fact that the processing times of jobs in sets $EP(K, L)$ and A are powers of 2. □

Example 3. *Suppose $p_{min}\{A\} = 4$ and $ST(K) = 23$. Then, $p' = 5p_{min}\{A\} = 20$, hence a gap of length 3 is unavoidable. Let $p_{min}(EP(K, L)) = 2^3 p_{min}\{A\} = 32$. Since the shortest job that set B may contain has processing time 32, the most we may expect is to form set A of three jobs of (the minimal) length 4, set B being formed by a single job with the length 32. Then, after swapping sets A and B, we have a residue $32 - 3 \times 4 = 20$. Because of these extra 20 units, the available idle space of length 23 is reduced to 3 in schedule $S^L(K, -A, +B)$ in which set A covers set B. In that schedule, a gap of (the minimal possible) length $23 - 20 = 3$ occurs.*

We may restrict our attention to sets A and B which do not contain equal-length jobs, as otherwise we may simply discount the corresponding jobs from both sets. In particular, for given A and B with $i \in A$ and $j \in B$ with $p_i = p_j$, we obtain sets $A(-i)$ and $B(-j)$ by eliminating job i and job j, respectively, from sets A and B, respectively. Let $A(-all\_equal, B)$ and $B(-all\_equal, A)$ be the reduced sets A and B, respectively, obtained by the repeated application of the above operation for all equal-length jobs. Sets $A(-all\_equal, B)$ and $B(-all\_equal, A)$ contain no equal-length jobs. We have proved the following lemma.

Lemma 10. *If set A covers set B, then set $A(-all\_equal, B)$ covers set $B(-all\_equal, A)$, where $p(B) - p(A) = p(B(-all\_equal, A)) - p(A(-all\_equal, B))$.*

Theorem 4. *If set A covers set B, then there are also (reduced) sets $A' \subseteq A$ and $B' \subseteq B$, where set B' contains a single element $\pi \in EP(K, L)$ with the minimum processing time in set B and with $P(B') - p' \leq P(A') < P(B')$ such that set A' covers set B' and $P(B') - P(A') = P(B) - P(A)$.*

Proof. Let A and B be the reduced sets that contain no equal-length jobs and such that A covers B (see Lemma 10). We can further reduce sets A and B by discounting, similarly, for each job $j \in B$, jobs from set A, for which processing times sum up to p_j. In particular, take a longest job $j \in B$ and longest jobs from set A that sum up to p_j. Due to the divisibility of job processing times and the inequalities $p(B) > p(A)$ and $p_{min}(EP(K, L)) = 2^{\kappa} p_{min}\{A\}$ (see Lemma 9), this will be possible as long as the total processing time in A is no smaller than p_j. The sets A and B are reduced respectively, and the same operation for these reduced sets is repeated until the total processing time of the remaining jobs in the reduced set A is less than p_j. Then, we are left with a single job $j \in B$ (one with the minimum processing time in B) and the jobs in set A with the total processing time less than p_j, and such that $p_j - p(A) \leq p'$ (see Lemma 9).

Let A' and B' be the reduced sets obtained from sets A and B, respectively. By the construction of set A' and B' and the fact that set A covers set B, it immediately follows that $P(B') - P(A') = P(B) - P(A)$ and that set A' covers set B'. □

Now, we show that the current secondary block $\mathcal{B}_K$ will be kept compact if we merely unify the compact preschedules in schedule $\Sigma_{\mathcal{C}(\delta,K)}$.

Theorem 5. *A secondary block $\mathcal{B}$ consisting of compact L-preschedules is compact.*

Proof. If the time interval of every preschedule $PreS(K)$ from block $\mathcal{B}$ extends up to time $\psi_L(K)$ and it contains no gap then the secondary block $\mathcal{B}$ is clearly compact. Suppose there is preschedule $PreS(K)$ from block $\mathcal{B}$ that contains a gap and/or completes before time $\psi_L(K)$. First, we observe that no extra job can be included within preschedule $PreS(K)$ to obtain another L-preschedule with an extended time interval and/or with less total gap length. Indeed, let x', $p_{x'} < p_x$, be a shortest available x-job from set $\in J^-(x)$. By PROCEDURE DEF, schedule $PreS(K, +x')$ is not a feasible L-preschedule for kernel K (as otherwise PROCEDURE DEF would include job x' in preschedule $PreS(K)$ at Pass 2). Thus, job x' may only feasibly be included in preschedule $PreS(K)$ by removing a longer job x from that preschedule. However, such a rearrangement may, at most, fill in the former execution interval of job x due to the above made observation and Lemma 6.

To prove the lemma, now it clearly suffices to show that nothing is to be gained by a job rearrangement in preschedule $PreS(K)$ that involves, besides the jobs from sets $E(K, L)$ and $EP(K, L)$, the jobs from a preschedule preceding preschedule $PreS(K)$.

Let $PreS'(K)$ be an arbitrary L-preschedule for kernel K (one respecting the current threshold L_δ). Without loss of generality, assume preschedules $PreS(K)$ and $PreS'(K)$ start at the same time, whereas none of them may complete after time $\psi_L(K)$ (Equation (12)). Let W and Z, respectively, be the sets of integer numbers, the processing times of jobs in the current preschedule $PreS(K) \in \Sigma_{\mathcal{C}(\delta,K)}$ and in preschedule $PreS'(K)$, respectively (here, we assume that sets W and Z consist of mutually divisible integer numbers, possibly with some repetitions).

Similar to what is done in Lemma 10 and Theorem 4, we discount the same numbers from sets W and Z and the numbers from one set which sum up to another number from the other set (taking a combination with the longest possible jobs). Note that both sets are reduced by the same amount (a sum of powers of 2). Denote by W' and Z' the resultant sets.

If $p(W') \geq p(Z')$ then the total gap length in preschedule $PreS(K)$ cannot be more than that in preschedule $PreS'(K)$, and the theorem follows if the condition holds for all preschedules in block $\mathcal{B}$.

Otherwise, suppose $p(W') < p(Z')$. By the definition of the sets W' and Z' and the store of kernel K (Equation (13)), $p(Z') - p(W') = p(Z) - p(W) \leq ST(K)$ (see Theorem 4) and the preschedule for kernel K consisting of the jobs associated with the set of processing times $\{W \setminus W'\} \cup Z'$ will have the same total gap length as preschedule $PreS'(K)$ (the substitution of the jobs corresponding to set W' by those from set Z' would result in a preschedule with the same total gap length as that in preschedule $PreS'(K)$). By the construction of preschedule $PreS(K)$ at Phase 1, no job x with processing time from set Z' which could have been feasibly included within preschedule $PreS(K)$ was available during the construction of that preschedule. Hence, every such job x should have been already scheduled in a preschedule $PreS(K')$ preceding preschedule $PreS(K)$ in block $\mathcal{B}$. By rescheduling job x from preschedule $PreS(K')$ to preschedule $PreS(K)$, the total gap length in the newly created preschedule of kernel K will be reduced by p_x, but a new gap of the same length will occur in the resultant new preschedule of kernel K' as there is no other suitable job available (otherwise, it would have been included in preschedule $PreS(K)$). Hence, the total gap length in block $\mathcal{B}$ will remain the same. Thus, no matter how the jobs are redistributed among the preschedules from block $\mathcal{B}$, the total length of the remaining gaps in that block will remain the same. The lemma is proved. □

Corollary 4. *If a secondary block $\mathcal{B}$ is constituted by the preschedules created at Phase 1, then it is compact.*

Proof. For every kernel $K \in \mathcal{B}$, if an L-preschedule $PreS(K, x, y)$ of Phase 1 is not compact then there exist sets $A \subseteq E(K, L)$ and $B \subseteq EP(K, L)$ such that an L-preschedule $PreS(K, -A, +B)$ is compact (Proposition 14). By Theorem 4, $B = \{\pi\}$, for some job $\pi \in EP(K, L)$. However, since for every job $j \in A$, $p_j < p_\pi$ (see Lemma 9), set A cannot cover set B in preschedule $PreS(K, -A, +B)$, as otherwise job π would have been included in preschedule $PreS(K, x, y)$ at Pass 2 instead of the shorter jobs from set A. It follows that every preschedule from block $\mathcal{B}$ is compact, and the corollary follows from Theorem 5. □

9. Phase 2: Search for an L-preschedule When an IA(b2) at Phase 1 Arises

Throughout this section, we consider the scenario when a compact preschedule for a newly arisen kernel K cannot be constructed at Phase 1, i.e., an IA(b2) with a Type (b) y-job y at Pass 1 arises. Recall that this happens when Pass 1 is unable to include job y in preschedule $PreS(K, y)$ in the current configuration (see Proposition 12). Phase 2, invoked from Phase 1, generates one or more new problem instances and calls back Phase 1 to create the corresponding new configurations. Thus, Phase 2 has no proper algorithmic features except that it generates new problem instances.

We refer to the earliest occurrence of IA(b2) in secondary block $\mathcal{B}_K$ at Phase 1 as the basic case. In the inductive case (abbreviated IA(b2-I)), IA(b2) repeatedly arises in the current secondary block (roughly, we "stay" in the current secondary block for IA(b2-I) in the inductive case, whereas we are brought to a new secondary block with every newly occurred IA(b2) in the basic case). In general, different occurrences of an IA(b2-I) in the inductive case may occur for different kernels, where all of them pertain to the current secondary block $\mathcal{B}$.

Throughout this section, let K^- be the kernel immediately preceding kernel K in block $\mathcal{B}_K$. We let y be an incoming job in bin $B^-(K) = B^+(K^-)$ at Phase 1; y is an incoming job in the first bin of block $\mathcal{B}_K$ if there exists no K^-. Note that r_y is no smaller than the starting time of block $\mathcal{B}_K$, and, since it can feasibly be scheduled within every bin that initiates at or after time r_y up to (and including) bin $B^-(K)$, y is a former x-job for any such a bin (except that it is a Type (b) job for bin $B^-(K)$), i.e., it may potentially be included in any of these bins. We explore such possibility and seek for a suitable distribution of all the x-jobs and Type (b) y-jobs into these bins at Phase 2.

Proposition 16. *Suppose during the construction of preschedule $PreS(K, y)$ an IA(b2)/IA(b2-I) with job y occurs and there exists schedule S^L. Then, job y or a Type (b) y-job included between kernel K^- and job y in bin $B^-(K)$ is scheduled before kernel K^- in schedule S^L.*

Proof. Note that the critical block in schedule $\Sigma_{C(\delta,K)}$ coincides with the secondary block $\mathcal{B}_{K^-}$, and it is compact when the above IA(b2)/IA(b2-I) occurs by Corollary 4. Then, job y cannot be restarted earlier in any feasible L-schedule in which the same jobs (which were included in preschedule $PreS(K, y)$ at Pass 1) are left scheduled before job y. The lemma obviously follows if y is the earliest considered job to be scheduled in bin $B^-(K)$. Otherwise, job y may potentially be started earlier either by scheduling it before kernel K^- or by decreasing (left-shifting) its current early starting time. The latter will only be possible if some job included in bin $B^-(K)$ ahead of job y is rescheduled behind job y. By the construction at Phase 1, any job included in bin $B^-(K)$ ahead of job y is a no less urgent than job y y-job and it cannot be rescheduled after job y without surpassing the L-boundary. Then, job y may be left-shifted only if one of the latter jobs is rescheduled before kernel K^-. However, this is not possible for a Type (a) y-job and the lemma is proved. □

By the above proposition, either job y or a Type (b) y-job included between kernel K^- and job y in bin $B^-(K)$ is to be rescheduled before kernel K^-. In particular, the following observations are evident:

- (1) If job y, is the first scheduled job in bin $B^-(K)$ or is preceded only by Type (a) y-jobs in that bin, then job y is to be entirely rescheduled before kernel K^-.

- (2) If job y is preceded by some Type (b) y-job(s), then either job y or some of these Type (b) y-job(s) is (are) to be rescheduled before kernel K^-. Since in any L-schedule job y needs to be left-shifted by at least λ_y amount of time (the L-delay of job y (see Equation (5))), the total processing time of these Type (b) y-jobs to be rescheduled before kernel K^- must be no-less than λ_y.

Let us denote by Λ_y the set of the y-jobs to be rescheduled before kernel K^- as defined in Cases (1) and (2) above. Set Λ_y will not be explicitly defined; it will be formed implicitly during the activation procedure that we describe in this section. In Case (1) above, set Λ_y will contain a single job y, hence $p_s \geq p_y$ must clearly hold, whereas, in Case (2), p_s must clearly be no-less than the minimum processing time of a y-job in set Λ_y. Let $\bar{p}_{\min\{y\}}$ be the minimum processing time among these y-jobs. The next proposition follows:

Proposition 17. $p_s \geq \bar{p}_{\min\{y\}}$.

9.1. The Activation of a Substitution Job

Given that an IA(b2)/IA(b2-I) with job y after kernel K^- arises, $s \in \mathcal{B}_K$ is called a *substitution* job if $d_s > d_y$. Intuitively, job s is an emerging job for job y (the latter job surpasses the current L-boundary, and in this sense, it is a potential overflow job).

PROCEDURE ACTIVATE(s) that activates substitution job s has some additional features compared to the basic definition of Section 2, as we describe in this subsection (in the next subsection, we complete the description of Phase 2 by a subroutine that tries different substitution jobs to determine a "right" one).

Let $B\{(s)\}$ be the bin from secondary block $\mathcal{B}_K$ containing substitution job s (it follows that s was included as an x-job in bin $B\{(s)\}$). PROCEDURE ACTIVATE(s) reconstructs preschedules for the kernels in the current schedule $\Sigma_{\mathcal{C}(\delta,K)}$ between the kernel K' with $B^-(K') = B\{(s)\}$ (the kernel with its first surrounding bin $B\{(s)\}$) and kernel K^-, including these two kernels, calling Phase 1 for each of these kernels (the kernel preschedules are reconstructed in their precedence order). This reconstruction leads to a new temporal configuration. PROCEDURE ACTIVATE(s) aims to verify if there exists a feasible L-preschedule for kernel K respecting this configuration. If it does not exist, PROCEDURE sl-SUBSTITUTION(K), described in the next subsection, tries another substitution job for kernel K, calling again Phase 1 for kernel K; each call creates a new temporary configuration and is carried out for a specially derived problem instance that depends on the selected substitution job.

For notational simplicity, we denote every newly constructed preschedule of kernel K by $PreS(K)$; we distinguish preschedules constructed at different calls of Phase 1 just by referring to the call with the corresponding substitution job, and will normally use $PreS(K)$ for the latest so far created preschedule for kernel K.

In the inductive case, the activation procedure for a substitution job s calls Phase 1 with a *non-empty* set $\mathcal{S}_\mathcal{B}$ of the substitution jobs, ones in the state of activation in the secondary block $\mathcal{B}$ by the corresponding call of Phase 1 (note that $s \notin \mathcal{S}_\mathcal{B}$). As already noted, the activation procedure may be called for different kernels which belong to the current secondary block, so that this block may contain a preschedule, already reconstructed by an earlier call of the activation procedure for another kernel from that block (set $\mathcal{S}_\mathcal{B}$ contains all the corresponding substitution jobs).

Problem instances for the basic and inductive cases. The problem instances for the basic and inductive cases are different, as we specify now. The problem instance PI$(current, +y, [s])$ of the *basic* case contains the jobs in schedule $\Sigma_{\mathcal{C}(\delta,K)}$ from all the bins between bin $B\{(s)\}$ and bin $B^-(K^-)$, including the jobs of bins $B^-(K^-)$ and $B\{(s)\}$ except job s, job y and all the y-jobs included before job y in preschedule $PreS(K,y)$ of Pass 1 (the latter y-jobs are ones which were already included in bin $B^-(K)$ at Pass 1 when the IA(b2) with job y has occurred; note that no x-job for bin $B^-(K)$ is included in instance PI$(current, +y, [s])$).

The problem instance of the *inductive* case contains the same set of jobs as that in the basic case, and it also contains the substitution jobs from set $\mathcal{S}_{\mathcal{B}}$. For the sake of simplicity, we denote that problem instance also by $\mathrm{PI}(current, +y, [s])$.

Successful and failure outcomes. As already specified, the activation of job s consists of the rescheduling of preschedules of bins $B\{(s)\}, \dots, B^-(K^-)$ by a call of Phase 1 for instance $\mathrm{PI}(current, +y, [s])$ in this precedence order (note that while rescheduling these bins only the jobs from that instance are considered). As we show at the end of this subsection in Lemma 11, all these bins will be successfully reconstructed at Phase 1.

PROCEDURE ACTIVATE(s) halts either with the successful outcome or with the failure outcome. For every successful outcome, the current call of Phase 2 (invoked for the IA(b2) with job y) completes and Phase 1 is repeatedly invoked from PROCEDURE MAIN for the construction of a new preschedule $PreS(K)$ for kernel K. Intuitively, the difference between the configurations after this new and the previous calls of Phase 1 for kernel K is that, as a result of the new call, no job from problem instance $\mathrm{PI}(current, +y, [s])$ may again surpass the L-boundary, and job y is already included in current secondary block in the new configuration. We omit a straightforward proof of the next proposition.

Proposition 18. *If there is a job from instance $PI(current, +y, [s])$ that the activation procedure could not include in any of the reconstructed bins $B\{(s)\}, \dots, B^-(K^-)$, this job is a y-job for bin $B^-(K)$ (or it is a job from set $\mathcal{S}_{\mathcal{B}}$ in the inductive case). If a former y-job is of Type (a), then all such Type (a) y-jobs can be included in bin $B^-(K)$ during the construction of a new preschedule $PreS(K)$ for kernel K at Phase 1.*

Note that, independently of the outcome, the activation procedure cannot include job s before any of the Type (b) y-jobs for bin $B^-(K)$ from instance $\mathrm{PI}(current, +y, [s])$ in the basic case. However, as shown below, job s may be included ahead some of these Type (b) y-jobs at a later call of the activation procedure for a substitution job, different from job s, in the inductive case.

Extension of Phase 1 for a call from the inductive case. The activation procedure for the inductive case takes a special care on the jobs from set $\mathcal{S}_{\mathcal{B}}$ while invoking Phase 1 for instance $\mathrm{PI}(current, +y, [s])$ (or instance $\mathrm{PI}(current, +y, [\emptyset])$ which we define below). In particular, when Phase 1 is called from the inductive case, two types of the x-jobs are distinguished during the (re)construction of a preschedule $PreS(\bar{K})$, $\bar{K} \neq K$ (one of the bins $B\{(s)\}, \dots, B^-(K^-)$). The *Type (b)* x-jobs are ones which are also x-jobs for bin $B^-(K)$, and the rest of the x-jobs are *Type (a)* x-jobs. We observe that a Type (a) x-job for bin $B^-(\bar{K})$ will transform to a Type (b) y-job for bin $B^-(K)$ unless it is included in one of the preceding reconstructed bins $B^-(\bar{K})$, and that a substitution job from set $\mathcal{S}_{\mathcal{B}}$ is a Type (b) x-job for any bin $B^-(\bar{K})$.

Phase 1, when invoked from the inductive case, is extended with an additional, Pass 3, designed for scheduling the substitution jobs from set $\mathcal{S}_{\mathcal{B}}$. Pass 3 uses the algorithm of Pass 2, DEF-heuristics, but with a different input, restricted solely to Type (b) x-jobs (hence, a former substitution job from $\mathcal{S}_{\mathcal{B}}$ may potentially be included at Pass 3). There is a respective modification in the input of Pass 2, which consists now of only Type (a) x-jobs (hence no substitution job from set $\mathcal{S}_{\mathcal{B}}$ will be included at Pass 2). Pass 3 is invoked after Pass 2, and Pass 2 is invoked after Pass 1, which remains unmodified while rescheduling each of the bins $B\{(s)\}, \dots, B^-(K^-)$.

Once (in both basic and inductive cases) preschedules of bins $B\{(s)\}, \dots, B^-(K^-)$ are reconstructed (Lemma 11), Phase 1 continues with the reconstruction of preschedule $PreS(K)$ as follows.

- (A) If there remains no unscheduled job from instance $\mathrm{PI}(current, +y, [s])$ (except possibly jobs from set $\mathcal{S}_{\mathcal{B}}$ in the inductive case), i.e., all these jobs are included in one of the reconstructed bins $B\{(s)\}, \dots, B^-(K^-)$, the activation procedure halts with the successful outcome.

  If there is a job from instance $\mathrm{PI}(current, +y, [s])$ that could not have been included in any of the reconstructed bins $B\{(s)\}, \dots, B^-(K^-)$ (excluding jobs from set $\mathcal{S}_{\mathcal{B}}$ in the inductive case), then it is a y-job for bin $B^-(K)$ (and it might also be a job from set $\mathcal{S}_\mathcal{B}$ in the inductive case). PROCEDURE ACTIVATE(s) proceeds as described below.

- (B) If every job from instance $\text{PI}(current, +y, [s])$ that could not have been included in any of the reconstructed bins $B\{(s)\}, \ldots, B^-(K^-)$ is a Type (a) y-job for bin $B^-(K)$ (or a job from set $\mathcal{S}_\mathcal{B}$ in the inductive case), the outcome of the activation of job s is again successful (see Proposition 18). {all the Type (a) y-jobs for bin $B^-(K)$ will fit in that bin}.
- If there is a Type (b) y-job for bin $B^-(K)$ from instance $\text{PI}(current, +y, [s])$ that could not have been included in any of the reconstructed bins $B\{(s)\}, \ldots, B^-(K^-)$, the outcome of the activation procedure depends on whether Phase 1 will succeed to construct L-preschedule $PreS(K)$ including all such Type (b) y-jobs.

  (C1) If during the construction of preschedule $PreS(K)$ at Pass 1 an iteration is reached at which all the Type (b) y-jobs from instance $\text{PI}(current, +y, [s])$ are included, then the outcome of the activation of job s is again successful and Phase 1 continues with the construction of (a new) preschedule $PreS(K)$ for kernel K by considering all the available jobs (including job s) without any further restriction.

  (C2) If the above iteration during the construction of preschedule $PreS(K)$ does not occur, then either (C2.1) a new kernel K' including the corresponding type (a) y-job(s) arises or (C2.2) an IA(b2) with a Type (b) y-job occurs (see Proposition 12).

  In Case (C2.1), Step 2 of Pass 1 returns kernel K' and calls PROCEDURE MAIN to update the current configuration (see the description of Pass 1 in Section 7.1).

  In Case (C2.2), PROCEDURE ACTIVATE(s) completes with the failure outcome (then PROCEDURE sl-SUBSTITUTION(K), described in the next subsection, looks for another substitution job s' and calls repeatedly PROCEDURE ACTIVATE(s')).

This completes the description of PROCEDURE ACTIVATE(s). In the next subsection, we describe how we select a substitution job in the basic and inductive cases completing the description of Phase 2.

Lemma 11. *PROCEDURE ACTIVATE(s) creates an L-preschedule, for every reconstructed bin $B\{(s)\}, \ldots, B^-(K^-)$ with the cost of Phase 1.*

Proof. In this proof, we refer to a call of PROCEDURE ACTIVATE(s) from the condition of the lemma as the current call of that procedure; note that, for the inductive case, there should have been performed earlier calls of the same procedure within the current secondary block. In particular, prior to the current call of PROCEDURE ACTIVATE(s), every bin $B^-(\bar{K}) \in \{B\{(s)\}, \ldots, B^-(K^-)\}$ was (re)constructed directly at Phase 1 (one or more times). The current call reconstructs bin $B^-(\bar{K})$ (preschedule $PreS(\bar{K})$) once again. Recall also that problem instance $\text{PI}(current, +y, [s])$ contains additional job y and the Type (b) y-jobs preceding that job by the construction of the preschedule for kernel K at Pass 1 (these jobs were included prior to the occurrence of an IA(b2) with job y). All these Type (b) y-jobs for bin $B^-(K)$ become x-jobs for a bin $B^-(\bar{K})$ after the current call of PROCEDURE ACTIVATE(s).

Again, the activation procedure calls Phase 1, and by the construction of Phase 1, it will suffice to show that during the reconstruction of any of the bins $B^-(\bar{K})$, there will occur no Type (b) y-job that cannot be included in the newly created preschedule $PreS(\bar{K})$ (note that no such Type (a) y-job may arise). Let us now distinguish two kinds of Type (b) y-jobs for bin $B^-(\bar{K})$: a Type (b) y-job that was also a Type (b) y-job during the previous (re)construction of preschedule $PreS(\bar{K})$, and a newly arisen Type (b) y-job for bin $B^-(\bar{K})$, i.e., one that was earlier included as an x-job in a preceding preschedule $PreS(K')$ but which turned out to be a Type (b) y-job during the current construction of preschedule $PreS(\bar{K})$.

The lemma is obviously true if there exists no latter kind of a y-job for bin $B^-(\bar{K})$. To the contrary, suppose job x was scheduled in bin $B^-(K')$ (preceding bin $B^-(\bar{K})$) as an x-job, but it was forced to be rescheduled to (a later) bin $B^-(\bar{K})$ as an y-job during the current call of PROCEDURE

ACTIVATE(s). Then, during the current construction of preschedule $PreS(K')$ (the last call of PROCEDURE ACTIVATE(s) that has invoked Phase 1) a new x-job z was included before job x was considered at Pass 2 of Phase 1. By DEF-heuristics (Pass 2), this may only occur if a job scheduled in bin $B^-(K')$ at the previous call is not considered at the current call during the construction of that bin (the preschedule $PreS(K')$). Let N be the set consisting of all such jobs. By the definition of instance PI($current, +y, [s]$) and the activation procedure, a job in set N may be job s or a job which was left-shifted within the time intervals liberated by job s or by other left-shifted job(s).

Thus, job z has now occupied the time intervals within which job x and job(s) in set N were scheduled. $p_z \geq 2p_x$, as otherwise job x would have been considered and included in bin $B^-(K')$ ahead of job z by DEF-heuristics (recall that the smallest job processing time, larger than p_x is $2p_x$). Then, $p(N) < p_x$ is not possible, since otherwise $p(N) + p_x < 2p_x \leq p_z$ and the length of the released time intervals would not be sufficient to include job z in bin $B^-(K')$ (hence, job z would not push out job x). If $p(N) = p_x$, because of the divisibility of job processing times and by DEF-heuristics, job z may only push out job x if $p_z = 2p_x = 2p(N)$. Then, p_z is greater than the processing time of any job in set N. However, in this case, job z would have been included at the previous call in bin $B^-(\bar{K})$ ahead of job x and the jobs in set N since it is longer than any of these jobs, a contradiction.

If now at the current call $p(N) > p_x$, a job can be included ahead of job z in preschedule $PreS(K')$ within the time intervals earlier occupied by the jobs in set N. Let p', $p' \leq p(N)$, be the length of the remaining total idle-time intervals. If $p_z \leq p'$, then job z cannot push out job z since it fits within the remaining idle-time interval. If $p_z > p'$, then p_z must be no smaller than the smallest power of 2 greater than $p_x + p'$. Hence, job z cannot fit within the intervals of the total length of $p_x + p'$, and, again, it cannot pull out job x.

We showed that job z cannot exist, hence job x does not exist and PROCEDURE ACTIVATE(s) creates an L-preschedule for the bins $B\{(s)\}, \ldots, B^-(K^-)$. The cost of the procedure is the same as that of Phase 1 since the cost of the creation of problem instance PI($current, +y, [s]$) is obviously absorbed by the cost of Phase 1. □

9.2. Selecting a Substitution Job

Now, we describe PROCEDURE sl-SUBSTITUTION(K) that repeatedly activates different substitution jobs for an IA(b2) occurred at Phase 1 (using PROCEDURE ACTIVATE(s)) to determine one for which PROCEDURE ACTIVATE(s) completes with the successful outcome (whenever there exists such a substitution job). From here on, we refer to the original precedence order of the substitution jobs in the current secondary block $\mathcal{B}_K$ (their precedence order corresponding to the last configuration in which none of them were activated).

Lemma 12. *Suppose an IA(b2)/IA(b2-I) with job y arises and s' and s'' are the substitution jobs such that job s'' preceded job s'. Then, if the outcome of activation of job s' is the failure then outcome of activation of job s'' will also be the failure.*

Proof. Let j be any candidate job to be rescheduled before kernel K^-, i.e., $j = y$ or j is any of the Type (b) y-jobs included after kernel K^- before the above IA(b2)/IA(b2-I) with job y has occurred (see Proposition 16). Job j is released either: (1) before the (former) execution interval of job s'; or (2) within or after that interval. In Case (1), job j can immediately be included in bin $B\{(s')\}$. Moreover, as $p_{s'} > p_j$, if j cannot be included in bin $B\{(s')\}$, it can also not be included in any other bin before kernel K^- (one preceding bin $B\{(s')\}$). In Case (2), job j cannot be included before kernel K^- unless some jobs from bin $B\{(s')\}$ and the following bins are left-shifted within the idle-time interval released by job s' (releasing, in turn, the idle-time within which job j may be included). Again, since $p_{s'} > p_j$, job j will fit within the idle-time interval released by job s', given that all the intermediate jobs are "sufficiently" left-shifted. Since job s' succeeds job s'', the activation of job s' will left-shift these jobs no-less than the activation of job s'' (being a substitution job, s' is "long enough"). The lemma now obviously follows. □

Determining the sl-substitution job. We use the above lemma for the selection of a right substitution job. Let us call the shortest latest scheduled substitution job such that the outcome of its activation is successful, the *sl-substitution* job for job y. We show in Lemma 16 that, if there exists no sl-substitution job, there exists no L-schedule.

Our procedure for determining the sl-substitution job is easy to describe. PROCEDURE sl-SUBSTITUTION(K) (invoked for an IA(b2) with a Type (b) y-job from Phase 1 during the construction of preschedule $PreS(K)$) finds the sl-substitution job or otherwise returns the failure outcome. Iteratively, it calls PROCEDURE ACTIVATE(s) for the next substitution job s (a candidate for the sl-substitution job) until PROCEDURE ACTIVATE(s) delivers a successful outcome or all the candidate jobs (which may potentially be the sl-substitution job) are considered.

The order in which the candidate substitution jobs are considered is dictated by Lemma 12. Recall from Proposition 17 that a substitution job is at least as long as $\bar{p}_{\min\{y\}}$. Let $\bar{p} \geq \bar{p}_{\min\{y\}}$ be the minimum processing time no smaller than $\bar{p}_{\min\{y\}}$ of any yet unconsidered substitution job. PROCEDURE sl-SUBSTITUTION(K), iteratively, among all yet unconsidered substitution jobs with processing time $\bar{p}$ determines the latest scheduled substitution job s and calls PROCEDURE ACTIVATE(s) (see Lemma 12). If the outcome of PROCEDURE ACTIVATE(s) is successful, the outcome of PROCEDURE sl-SUBSTITUTION(K) is also successful and it returns job s (s is the sl-substitution job). Otherwise, if there exits the sl-substitution job, it is longer than job s. $\bar{p}$ is set to the next smallest processing time larger than the current $\bar{p}$, s becomes the latest scheduled substitution job with the processing time $\bar{p}$ and PROCEDURE ACTIVATE(s) is called again. The procedure continues in this fashion as long as the latest outcome is the failure and $\bar{p}$ can be increased (i.e., a substitution job with the processing time greater than that of the latest considered one exists). Otherwise, PROCEDURE sl-SUBSTITUTION(K) halts with the failure outcome.

Let μ be the number of non-kernel jobs in the current secondary block $\mathcal{B}_K$.

Lemma 13. *PROCEDURE sl-SUBSTITUTION finds the sl-substitution job or establishes that it does not exist by verifying at most* $\log p_{\max}$ *substitution jobs in time* $O(\log p_{\max}\mu^2 \log \mu)$.

Proof. The preprocessing step of PROCEDURE sl-SUBSTITUTION creates a list in which the substitution jobs are sorted in non-decreasing order of their processing times, whereas the jobs of the same processing time are included into the inverse precedence order of these jobs in that list. The preprocessing step takes time $O(\mu \log \mu)$.

Since the processing time of every next tried substitution job is larger than that of the previous one, the procedure works on $\log p_{\max}$ iterations (assuming that the processing times of the substitution jobs are powers of 2). By Lemma 12, among all the candidate substitution jobs with the same processing time, it suffices to consider only the latest scheduled one. For the failure outcome, by the same lemma, it suffices to consider the latest scheduled substitution job with the next smallest processing time (given that the procedure starts with the latest scheduled substitution job with the smallest processing time).

At every iteration, the corresponding bins from the current secondary block $\mathcal{B}_K$ are rebuilt at Phase 1. Applying Theorem 2 and the fact that different bins have no common jobs, we easily obtain that the cost of the reconstruction of all the bins at that iteration is $O(\mu^2 \log \mu)$ and hence the total cost is $O(\mu \log \mu + \log p_{\max}\mu^2 \log \mu) = O(\log p_{\max}\mu^2 \log \mu)$. □

10. More Examples

Before we prove the correctness of our algorithm for problem $1|p_j : \mathit{divisible}, r_j|L_{\max}$, we give final illustrations using the problem instances of Examples 1 and 2 and one additional problem instance, for which an IA(b2) arises. Recall that Figures 5 and 9 represent optimal solutions for the former two problem instances.

For the problem instance of Example 1, in the schedule of Figure 6 the collapsing of kernel K is complete and the decomposition procedure identifies the atomic kernel K^*; hence, the corresponding two bins are determined. The atomic kernel K^* consists of Job 1 with the lateness $-1 = L^*_{\max}$. The binary search is carried out within the interval $[0, 13)$ ($\Delta = 16 - 3 = 13$). For $\delta = 7$, the L_δ-boundary is $-1 + 7 = 6$. At Phase 1, bins B_1 and B_2 are scheduled as depicted in the schedule of Figure 5 (in bin 1 only a single x-Job 2 at Pass 2 can be included, whereas bin B_2 is packed at Pass 1 with two y-Jobs 3 and l). Hence, the L-schedule of Figure 5 for $L = 6$ is successfully created. For the next $\delta = 4$, the L_δ-boundary is $-1 + 4 = 3$. Bin B_1 is scheduled similarly at the iteration with $\delta = 7$; while scheduling bin B_2 at Phase 1, an IA(b2) with y-Job 3 occurs (since its lateness results to be greater than 3), but there exists no substitution job. Hence, there exists no L_δ-schedule for $\delta = 4$, $L = 3$. Phase 1 will complete with the similar outcome for the iteration in the binary search with $\delta = 6$, and the algorithm halts with the earlier obtained feasible solution for $\delta = 7$.

For the problem instance of Example 2, the schedule of Figure 8 represents the result of the decomposition of both arisen kernels K^1 and K^2 (kernel K^2 arises once the decomposition of kernel K^1 is complete and bin B_1 gets scheduled). We have $L_{\max}(K^{1^*}) = L_3(\sigma_{l,2}) = 19 - 20 = -1$, whereas $\Delta = 32 - 3 = 29$. For $\delta = 0$, bin B_1 may contain only Job 1. Once bin B_1 is scheduled, the second kernel K^2 arises. The result of its collapsing is reflected in Figure 8. We have $L^*_{\max} = L_{\max}(K^{2^*}) = L_6(\sigma_{l,2,4}) = 62 - 58 = 4$. Then, $\delta(K^{1^*}) = 5$ (while $\delta(K^{2^*}) = 0$), and an extra delay of 5 is now allowed for kernel K^1. Note that the current secondary block $\mathcal{B}_{K^2}$ includes all the three bins. For $\delta = 0$, bin B_1 is newly rescheduled and at Pass 2 of Phase 1 an x-Job 7 is now included in that bin (due to the allowable extra delay for kernel K^{1^*}). No other job besides Job l can be included in bin B_2, and the last bin B_3 is formed by Job 4. A complete L-schedule (with $L_\delta = L^*_{\max} + \delta = 4 + 0 = 4$) with the objective value equal to a lower bound 4 is successfully generated (see Figure 9).

Example 4. *In this example, we modify the problem instance of Example 2. The set of jobs is augmented with one additional Job 8, and the parameters of Jobs 4 and 7 are modified as follows:*
$r_4 = 0,\ p_4 = 8,\ d_4 = 66,$
$r_7 = 0,\ p_7 = 4,\ d_7 = 60,$
$r_8 = 0, p_8 = 4, d_8 = 63.$

Figure 10 represents the last step in the decomposition of kernel K^1, which is the same as for the problem instance of Example 2 (the schedules represented in Figures 10–13 have different scaling due to the differences in their lengths). This decomposition defines the two bins surrounding atomic kernel K^{1^*}. The binary search is invoked for $\delta = 0$; since K^{1^*} is the only detected kernel so far, $\delta(K^{1^*}) = 0$ and $L_\delta = L_3(S^*[K^1]) = -1$. The first bin is successfully packed with an additional external x-Job 7 at Pass 2 of Phase 1 (since there exists no y-job, Pass 1 is not invoked). PROCEDURE MAIN proceeds by applying ED-heuristics from Time 21 during which the second kernel K^2 arises. Figure 11 represents the resultant partial schedule with the first packing of bin B^1 and kernel K^2. Figure 12 represents the result of the full decomposition of kernel K^2 (which is again the same as for the problem instance of Example 2). Now, $\delta(K^{2^*}) = 0$ and $\delta(K^{1^*}) = 5$. Bin B_1 is repacked, in which a longer x-Job 4 can now be included, and bin B_2 at Phase 1 is packed (at Pass 1 an y-Job 2, and at Pass 2 an x-Job l is included in that bin, see Figure 12). PROCEDURE MAIN is resumed to expand the current partial schedule, but now an IA(b2) with the earliest included Job 7 arises (as its resultant lateness is $66 - 60 = 6 > 4$). Job 4 from bin B_1 is the sl-substitution job. The result of its activation is reflected in Figure 13: bin B_1 is repacked now with x-Jobs 7 and 8, bin B_2 remains the same, and the last Job 4 is included in bin B_3 at Phase 0, yielding a complete S^{L_0}-schedule with the optimal objective value 4 (both Jobs 6 and 4 realize this optimal value).

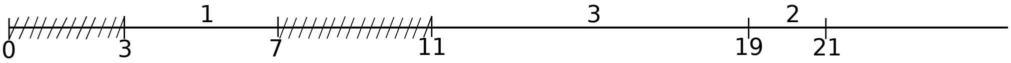

Figure 10. Full decomposition of kernel K^1.

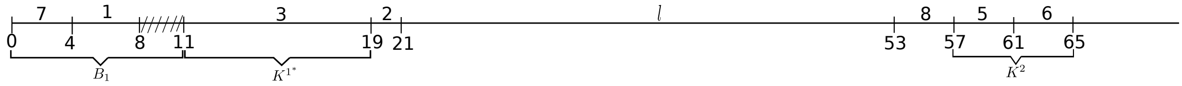

Figure 11. Extended schedule $S^L(current, K^1)$ with kernel K^2.

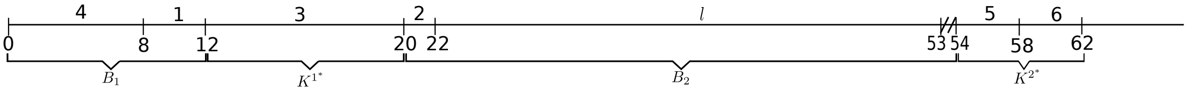

Figure 12. In schedule $S^L(current, K^2)$ Job 7 cannot be included.

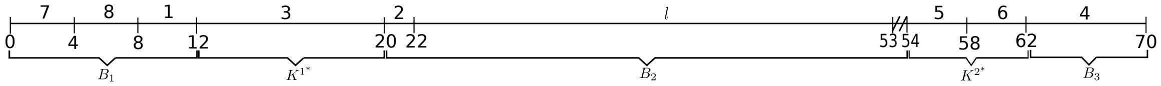

Figure 13. An optimal schedule S^L in which bins B_1, B_2 and B_3 are successfully repacked.

11. Correctness of the Framework for Jobs with Divisible Processing Times

In Section 8, we show that, for an instance of $1|p_j : divisible, r_j|L_{\max}$, the current secondary block is kept active at Phase 1 (Corollary 4). Now, we generalize this result proving a similar statement for a secondary block that gets reconstructed at Phase 2 (we cannot affirm a similar property for an instance of $1|r_j|L_{\max}$, a reason PROCEDURE MAIN may not provide an optimal solution to the general problem). For the commodity in the proof that we present here, we introduce a few new definitions.

First, we observe that a call of PROCEDURE ACTIVATE(s) may create the new, so-called *critical* gap(s) in a reconstructed preschedule in the current secondary block $\mathcal{B}_K$. To every critical gap in secondary block $\mathcal{B}_K$, the substitution job from set $\mathcal{S}_{\mathcal{B}}$ which activation has yielded that gap, corresponds. Denote by $CG(s)$ the set of all the currently remaining (yet unused at that configuration) critical gaps yielded by the activation of a substitution job s; let $|CG(s)|$ be the total length of these gaps.

A substitution job $s \in \mathcal{S}_{\mathcal{B}}$ is *stable* if $|CG(s)| = 0$. When a substitution job s is activated, the total length of the critical gaps arisen after its activation depends, in particular, on p_s. For example, in the basic case, or in the inductive case if substitution jobs in $\mathcal{S}_{\mathcal{B}}$ are stable, the new critical gaps with the total length $p_s - p_y$ will arise, where y is the y-job for which s was activated.

If an activated substitution job s is non-stable and during a later call of PROCEDURE ACTIVATE(s'), $s' \neq s$, some y-job within the interval of the gaps in $CG(s)$ is included, $|CG(s)|$ will be reduced. In this way, job s may eventually become stable.

For a substitution job $s' \in \mathcal{S}_{\mathcal{B}}$, we let $Y(s')$ be the set of all the newly included y-jobs in the reconstructed bins after a call of PROCEDURE ACTIVATE(s') (see Lemma 11).

Suppose a call of PROCEDURE ACTIVATE(s) (succeeding an earlier call of PROCEDURE ACTIVATE(s')) includes job s' before all the y-jobs from set $Y(s')$. Then, job s' is said to be *inactivated*. The intuition behind this definition is that job s' will not necessity remain in the state of activation for all jobs from set $Y(s')$ in the case that the activation of a new substitution job s gives a sufficient room for a proper accommodation of the jobs in set $Y(s')$ (this is rectified in more details in the proof below). At the same time, we may note that job s' may not be included in any of the newly reconstructed bins and neither in bin $B^-(K)$ (then it eventually will be included within a succeeding bin of the secondary block $\mathcal{B}_K$).

Lemma 14. *At Phase 2, the current secondary block is kept compact given that for every occurrence of an IA(b2-I) the corresponding sl-substitution job exists.*

Proof. For the basic case, before the activation of the corresponding sl-substitution job, say s_1, the critical block $\mathcal{B}_K$ is compact by Corollary 4. Since s_1 is the sl-substitution job, block $\mathcal{B}_K$ will remain compact after the above activation. We are brought to the inductive case if an IA(b2) repeatedly arises in the above block, the first occurrence of an IA(b2-I) with the number of the activated substitution jobs $k = 1$.

We proceed with the proof using the induction on the number of the activated substitution jobs. We now prove our claim for $k = 2$, in the case that the second substitution job s_2 is activated in the current secondary block. Consider the following possibilities. Originally, job s_2 either: (i) succeeded job s_1; or (ii) preceded job s_1.

In Case (i), if $p_{s_2} \geq 2p_{s_1}$, all the y-jobs already included within $CG(s_1)$ together with jobs in set Λ_{y_2} can be feasibly scheduled within $CG(s_2)$ as $p_{s_2} \geq 2p(\Lambda_{y_2})$. Hence, after a call of PROCEDURE ACTIVATE(s_2) job s_1 will be inactivated at Pass 3 (see Lemma 11). Hence, job s_1 becomes stable and we are left with a single substitution job s_2 in the state of activation.

In Case (ii), note that no job from set Λ_{y_2} was included within $CG(y_1)$ after a call of PROCEDURE ACTIVATE(s_1). Hence, $|CG(s_1)| < p(\Lambda_{y_2})$. If job s_2 is long enough and all jobs in $Y(s_1)$ are released early enough and can fit within the newly released space by job s_2 after a call of PROCEDURE ACTIVATE(s_2), once Pass 3 of the activation procedure completes, job s_1 will again become stable and we are again left with a single substitution job s_2 in the state of activation.

Since in the above considered cases, the only non-stable substitution job is s_2, our claim follows from case $k = 1$ and the fact that s_2 is the sl-substitution job. It only remains to consider the cases when job s_1 remains in the state of the activation after a call of PROCEDURE ACTIVATE(s_2), i.e., both substitution jobs s_1 and s_2 remain in the state of activation. This happens in Case (i) if $p_{s_2} \leq p_{s_1}$ (note that in this case $p_{s_2} \leq p(\Lambda_{y_2})$ also holds as otherwise job s_2, instead of job s_1, would have been selected as the sl-substitution job for job y_1). Either jobs in set Λ_{y_2} are not released early enough to be included within $CG(s_1)$ or $|CG(s_1)|$ is not large enough. Hence, another substitution job needs to be activated to include jobs in Λ_{y_2} (see Lemma 15 below). Since s_2 is the sl-substitution job, $|CG(s_2)|$ is the minimal possible. The lemma holds if job s_1 again becomes stable. Otherwise, note that, since both s_1 and s_2 are the sl-substitution jobs, the only remaining possibility to be considered is when a single substitution job s with $p_s < p_{s_1} + p_{s_2}$ (instead of jobs s_1 and s_2) is activated.

Consider the following sub-cases: (1) $|CG(s_1)| \geq p(\Lambda_{y_2})$; and (2) $|CG(s_1)| < p(\Lambda_{y_2})$. In Case (1). jobs in Λ_{y_2} are not released early enough to be included within $CG(s_1)$ as otherwise they would have been included by an earlier call of PROCEDURE ACTIVATE(s_1). Hence, no job preceding originally job s_1 can be beneficially activated. At the same time, any substitution job succeeded originally job s_1 is longer than s_1 (by the definition job s_1 and PROCEDURE sl-SUBSTITUTION). Then, $p_s \geq 2p_{s_1}$ because of the divisibility of job processing times. In Case (2), $p_s \geq 2p_{s_1}$ must also hold as otherwise all jobs in set $Y(s_1)$ together with jobs in set Λ_{y_2} would not fit within the time intervals that potentially might be liberated by a call of PROCEDURE ACTIVATE(s).

Hence, in both Cases (1) and (2) above, $p_s < p_{s_1} + p_{s_2}$ is not possible and hence the activation of job s will yield the critical gaps with a total length no less than our procedure yields, and the lemma follows. The proof for the Case (ii) when the jobs in $Y(s_1)$ do not fit within $CG(s_2)$ or they are not released early enough is quite similar to case (i) above (the roles of jobs s_1 and s_2 being interchanged).

For the inductive pass with $k \geq 3$, let s_k be the next activated sl-substitution job and let $\mathcal{S}_\mathcal{B} = \{s_1, \ldots, s_{k-1}\}$ be the substitution jobs in the state of activation in the current critical block $\mathcal{B}$). By the inductive assumption, block $\mathcal{B}$ was compact before job s_k is activated. Now, we show that the block remains compact once job s_k is activated. This follows if s_k, as before, remains the only (non-stable) substitution job in the state of activation after a call of PROCEDURE ACTIVATE(s_k). Otherwise, originally, job s_k: (i) succeeded all the jobs $\{s_1, \ldots, s_{k-1}\}$; (ii) preceded these jobs; or (iii) was scheduled in between their original positions. We use similar arguments as for $k = 2$. We give a scratch.

In Case (ii), note that the time intervals released by a call of PROCEDURE ACTIVATE(s_k) will be available for the jobs from set $Y(s_1) \cup \cdots \cup Y(s_{k-1})$ during the execution of the procedure at Pass 2 of Phase 1, and they may potentially be left-shifted to these intervals. Because of the mutual divisibility of processing times of these jobs and by the construction of Pass 2, the total length of the remaining idle-time intervals, if any, will be the minimal possible (this can be straightforwardly seen). It follows that, at Pass 3, the corresponding jobs from $\mathcal{S}_{\mathcal{B}}$ will become inactivated and hence stable, whereas the rest of them are to stay in the state of activation, and our claim follows from the inductive assumption.

In Case (i), all jobs from $Y(s_1) \cup \cdots \cup Y(s_{k-1})$ are released early enough to be included within the intervals newly released by a call of PROCEDURE ACTIVATE(s_k). Again, because of the mutual divisibility of processing times of these jobs and by the construction of Pass 2, the remaining idle-time intervals, if any, will be the minimal possible, and at Pass 3 the corresponding substitution jobs will be inactivated.

The proof of Case (iii) merely combines those for Cases (i) and (ii): at Pass 2, the intervals released by a call of PROCEDURE ACTIVATE(s_k) might be used by jobs from $Y(s_1) \cup \cdots \cup Y(s_{k-1})$ preceding and also succeeding these intervals, and the corresponding jobs from $\{s_1, \ldots, s_{k-1}\}$ will again become stable. □

Lemma 15. *Suppose an IA(b2)/IA(b2-I) with job y during the construction of preschedule $PreS(K)$ arises and there exists an L-schedule S^L. Then, a substitution job is scheduled after kernel K^- in schedule S^L. That is, there exists no L-schedule if there exists no substitution job.*

Proof. The lemma is a kind of reformulation of Proposition 16. For the basic case, before the activation of the sl-substitution job s_1, the secondary block $\mathcal{B}_K$ is compact by Corollary 4. Similar to in the proof of Proposition 16, we can see that the current starting time of job y cannot be reduced by any job rearrangement that leaves the same set of jobs scheduled before job y. Hence, some emerging x-job s from one of the bins from the secondary block $\mathcal{B}_K$ pushing job y is included behind job y in schedule S^L (recall that $d_s > d_y$ must hold as, otherwise, once rescheduled after kernel K^-, job s will surpass the L-boundary or will force another y-job to surpass it). Job s cannot be from bin $B^-(K)$ since no x-job can be included ahead of job y during the construction of $PreS(K)$ as job y is released from the beginning of that construction (and it would have been included at Pass 1 of Phase 1 before any x-job is considered at Pass 2). Therefore, job s belongs to one of the bins preceding bin $B^-(K)$ in block $\mathcal{B}_K$. The proof for the inductive case is similar except that it uses Lemma 14 instead of Corollary 4. □

Lemma 16. *If there exists no sl-substitution job, then no L-schedule exists.*

Proof. If there exists no substitution job at all, then the statement follows from Lemma 15. Otherwise, the outcome of the activation of every tried substitution is the failure. We claim that there exists no L-preschedule that contains the jobs from problem instance PI$(current, +y, [s])$ together with all the jobs from all the (intermediate) kernels between the bins $B\{(s)\}$ and $B^-(K^-)$. Let s be the earliest tried substitution job by PROCEDURE sl-SUBSTITUTION(K). If job s becomes non-stable after a call of PROCEDURE ACTIVATE(s), then, due to the failure outcome, it must be the case that the corresponding y-job(s) (see Proposition 16) cannot be left-shifted within the time intervals liberated by job s (because of their release times). Hence, neither they can be left-shifted by activation of any substitution job preceding job s (Lemma 12). Otherwise, it must have been stable once activated, but the interval released by job s is not long enough (again, due to the failure outcome). Hence, only another longer substitution job may be of a potential benefit, whereas the latest scheduled one, again, provides the maximum potential left-shift for the above y-job(s). We continue applying this argument to every next tried substitution job. Our claim and hence the lemma follow due to the failure outcome for the latest tried (the longest) substitution job. □

Now, we immediately obtain the following corollary that already shows the correctness of PROCEDURE MAIN for divisible processing times:

Corollary 5. *For every trial δ, PROCEDURE MAIN generates an L_δ-schedule if the outcome of every call of PROCEDURE sl-SUBSTITUTION(K) for an IA(b2) is successful (or no IA(b2) arises at all); otherwise (there exists no sl-substitution job for some IA(b2)), no L_δ-schedule exists.*

Theorem 6. *PROCEDURE MAIN optimally solves problem* $1|p_j : divisible, r_j|L_{\max}$ *in time* $O(n^3 \log n \log^2 p_{\max})$.

Proof. The soundness part immediately follows from Corollary 5 and the definition of the binary search in Section 5 (see Proposition 8). We show the time complexity. Due to Theorem 3, it remains to estimate an additional cost yielded by Phase 2 for instances of alternative (b2). Recall from Theorem 2 that, for every arisen kernel K, the cost of the generation of L_δ-augmented schedule $S^{L_\delta}[K]$ for a given δ is $O(\nu^2 \log \nu)$, where ν is the total number of jobs in bin $B^-(K)$. Recall also that this cost includes the cost of all the embedded recursive calls for all the kernels which may arise within bin $B^-(K)$. Similar to in the proof of Theorem 3, it suffices to distinguish the calls of PROCEDURE AUGMENTED(K, δ) and PROCEDURE AUGMENTED(M, δ) for two distinct kernels K and M such that bins $B^-(K)$ and $B^-(M)$ have no jobs in common. Now, we count the number of such calls of PROCEDURE AUGMENTED(K, δ) from Phase 2 by PROCEDURE sl-SUBSTITUTION(K). The number of times, an IA(b2) at Phase 1 may arise is bounded by υ_1, the number of Type (b) y-jobs (note that any Type (b) y-job may yield at most one IA(b2)). Hence, for any bin $B^-(K)$, PROCEDURE AUGMENTED(K, δ) may be called less than υ_1 times for different instances of Alternative (b2), whereas for the same IA(b2) no more than $p_{\max}$ different substitution jobs might be tried (Lemma 13). Hence, the total number of calls of PROCEDURE AUGMENTED(K, δ) is bounded above by $O(\upsilon_1 + p_{\max})$, which easily yields the overall bound $O(n^3 \log n \log^2 p_{\max})$. □

12. Possible Extensions and Applications

We describe our framework for the single-machine environment and with a due-date oriented objective function $L_{\max}$. It might be a subject of a future research to adopt and extend the proposed framework for other machine environments with this or another due-date oriented objective function. Both the recurrence substructure properties and the schedule partitioning into kernel and bin intervals can be extended for the identical machine environment and shop scheduling problems with job due-dates. Less straightforward would be its adaptation for the uniform machine environment, and, unlikely, the approach can be extended to the unrelated machine environment.

The framework can obviously be converted to a powerful heuristic algorithm, as well as to an exact implicit enumeration scheme for a general setting with arbitrary job processing times. For both heuristic and enumerative approaches, it will clearly suffice to augment the framework with an additional search procedure invoked for the case when the condition of Theorem 3 is not satisfied.

Based on the constructed framework, we have obtained an exact polynomial-time algorithm for problem $1|p_j : divisible, r_j|L_{\max}$. A natural question is whether, besides the scheduling and bin packing problems ([4]), there are other NP-hard combinatorial optimization problems for which restrictions with divisible item sizes are polynomially solvable (the properties of mutually divisible numbers exploited in reference [4] and here could obviously be helpful).

Finally, we argue that scheduling problems with divisible job processing times may naturally arise in practice. As an example, consider the problem of distribution of the CPU time and the computer memory, the basic functions of the operating systems. In Linux operating systems buddy memory allocation is used, in which memory blocks of sizes of powers of 2 are allocated. To a request for memory of size K, the system allocates a block of size 2^k where $2^{k-1} < K \leq 2^k$ (if currently there is no available block of size 2^k, it splits the shortest available block of size 2^{k+1} or more). In buddy systems, memory allocation and deallocation operations are naturally simplified, as an $O(n)$ time search is reduced to $O(\log n)$ time using binary tree representation for blocks.

A similar "buddy" approach for the CPU time sharing in operating systems would assume the "rounding" of the arriving requests with arbitrary processing times within the allowable patterns of processing times—the powers of 2. In the CPU time sharing, the system must decide which of the arriving requests to assign to the processor and when. The request may arrive over time or, in the case of the scheduled maintenance and other scheduled computer services (for example, operating system updates), the arrival time of the requests and their processing times are known in advance. The latter scenario fits into our model. One may think on the rounding of a processing time of a request up or down to a closer power of 2. Alternatively, to avoid unnecessary waste of the processor time, one may always round down and process the remaining small part in a parallel or sequential manner immediately upon the completion of the main part or later on. Possible efficient and practical strategies for "completing" the solution with divisible processing times in a single-processor or multiprocessor environment deserves an independent study.

The "buddy" approach for the CPU time sharing in operating systems is justified by our results, as we show that the scheduling problems with mutually divisible processing times can be solved essentially more efficiently than with arbitrary job processing times. The degree of the "waste" during the rounding of the memory blocks and processing requirements is somewhat similar and comparable in both the memory allocation and the CPU time sharing methods. In the case of the memory allocation we may waste an extra memory, and in the case of the time sharing we waste an extra time (which would influence on the quality of the solution of course). It is important and not trivial how an input with arbitrary job processing times can be converted to an input with divisible processing times, and how close the obtained optimal solution for the instance with divisible times will be from an optimal solution for the original instance. This interesting topic can be a subject of a future independent study.

Funding: This research received no external funding.

Conflicts of Interest: The author declares no conflict of interest.

References

1. Garey, M.R.; Johnson, D.S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*; Freeman: San Francisco, CA, USA, 1979.
2. Vakhania, N.; Werner, F. Minimizing maximum lateness of jobs with naturally bounded job data on a single machine in polynomial time. *Theor. Comput. Sci.* **2013**, *501*, 72–81, doi:10.1016/j.tcs.2013.07.001. [CrossRef]
3. Graham, R.L.; Lawler, E.L.; Lenstra, J.K.; Kan, A.H.G.R. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Ann. Discret. Math.* **1979**, *5*, 287–328.
4. Coffman, E.G., Jr.; Garey, M.R.; Johnson, D.S. Bin packing with divisible item sizes. *J. Complex.* **1987**, *3*, 406–428. [CrossRef]
5. McMahon, G.; Florian, M. On scheduling with ready times and due-dates to minimize maximum lateness. *Oper. Res.* **1975**, *23*, 475–482. [CrossRef]
6. Carlier, J. The one-machine sequencing problem. *Eur. J. Oper. Res.* **1982**, *11*, 42–47. [CrossRef]
7. Jackson, J.R. Scheduling a production line to minimize the maximum lateness. In *Management Science Research Report 43*; University of California: Los Angeles, CA, USA, 1955.
8. Horn, W.A. Some simple scheduling algorithms. *Naval Res. Logist. Q.* **1974**, *21*, 177–185. [CrossRef]
9. Garey, M.R.; Johnson, D.S.; Simons, B.B.; Tarjan, R.E. Scheduling unit-time tasks with arbitrary release times and deadlines. *SIAM J. Comput.* **1981**, *10*, 256–269. [CrossRef]
10. Vakhania, N. Single-Machine Scheduling with Release Times and Tails. *Ann. Oper. Res.* **2004**, *129*, 253–271. [CrossRef]
11. Lazarev, A.A.; Arkhipov, D.I. Minimization of the Maximal Lateness for a Single Machine. *Autom. Remote Control.* **2016**, *77*, 656–671. [CrossRef]
12. Vakhania, N. Fast solution of single-machine scheduling problem with embedded jobs. *Theor. Comput. Sci.* **2019**. [CrossRef]

13. Vakhania, N. A better algorithm for sequencing with release and delivery times on identical processors. *J. Algorithms* **2003**, *48*, 273–293. [CrossRef]
14. Schrage, L. Obtaining optimal solutions to resource constrained network scheduling problems, March 1971. Unpublished manuscript.