A segment partitioning heuristic for scheduling jobs with release times and due-dates

Nodari Vakhania\textsuperscript{a},
Federico Alonso-Pecina\textsuperscript{b}, Crispin Zavala\textsuperscript{b}
(a) Centro de Investigación en Ciencias, UAMor, Mexico.
(b) Facultad de Contaduría, Administración e Informática UAMor, Mexico.

Abstract—In a standard strongly NP-hard single-machine scheduling problem the jobs are characterized by release times and due-dates and the objective to minimize the maximum job lateness. We develop a heuristic method for solving this problem based on the partition of the schedule horizon into two types of time intervals containing urgent and non-urgent jobs, respectively. We report the results of the preliminary computational experiments testing the practical performance of the proposed algorithm.

Key words: scheduling single-machine; release-time; due-date; lateness; bin packing; polynomial-time algorithm

I. INTRODUCTION

In scheduling problems have a finite set of resources (machines or processors) that may perform orders (jobs or tasks) from another finite set. The objective is to arrange the assignment of the orders to the resources to minimize some overall (usually time) criteria.

In this paper we address a single-machine scheduling problem when every job \( j \) is characterized by its release time \( r_j \) and due-date \( d_j \); \( r_j \) is the time moment when job \( j \) arrives to the system hence becomes available for processing on the machine, and \( d_j \) is the desired completion time for job \( j \). The problem of scheduling jobs with release times and due-dates on a single machine with the objective to minimize the maximum job lateness, with the common abbreviation \( 1/r_j/L_{\text{max}} \) (Graham et al. [4]), can be stated as follows.

We are given \( n \) jobs in \( \{1, 2, \ldots, n\} \). Each job \( j \) has (non-interruptible) processing time \( p_j \), release time \( r_j \) and due-date \( d_j \). The \( n \) jobs are to be scheduled on a single machine that can process at most one job at a time. A feasible schedule \( S \) is a mapping that assigns to each job \( j \) a starting time \( t_j(S) \), such that \( t_j(S) \geq r_j \) and \( t_j(S) \geq t_k(S) + p_k \) for any job \( k \) included earlier in \( S \) (for notational simplicity, we use \( S \) also for the corresponding job-set); the first inequality says that a job cannot be started before its release time, and the second one reflects the restriction that the machine can handle only one job at any time. \( c_j(S) = t_j(S) + p_j \) is the completion time of job \( j \). We aim to find an optimal schedule, i.e., one minimizing the maximum job lateness \( L_{\text{max}} = \max\{j | c_j - d_j\} \). We denote by \( L(S) \) (\( L_j(S) \), respectively) the maximum lateness in \( S \) (the lateness of job \( j \) in \( S \), respectively).

The problem is known to be strongly NP-hard (Garey & Johnson [2]). Hence, the development of efficient heuristics with a good practical behavior is of a primary interest. The earliest proposed and the most widely used heuristics for an approximate solution of problem \( 1/r_j/L_{\text{max}} \) is the ED (Earliest Due-date) heuristic, suggested by Jackson [6]. This heuristic is iterative; at each scheduling time \( t \) (given by job release or completion time), among the jobs released by time \( t \) schedules one with the largest delivery time or the smallest due-date (breaking ties by selecting a longest one).

In the worst-case, Jackson’s heuristic delivers a solution which is twice worse than an optimal one, i.e., ED-heuristic is a 2-approximation algorithm. Potts [8] has proposed an alternative approximation algorithm with an improved approximation ratio of 3/2, in which Jackson’s heuristic is repeatedly applied \( O(n) \) times. Hall and Shmoys [5] have proposed polynomial approximation schemes for the same problem, and also an 4/3-approximation an algorithm for its version with the precedence relations with the same time complexity of \( O(n^2 \log n) \) as the above algorithm from [8].

Implicit enumerative algorithms have also been developed for problem \( 1/r_j/L_{\text{max}} \). Among the most efficient such algorithms are ones proposed by McMahon & Florian [7] and Carlier [1].

The problem can naturally be simplified by imposing some restrictions on job processing times. Two such versions are known to be polynomially solvable. Garey et al. [3] have developed a sophisticated \( O(n \log n) \) algorithm for the case when all jobs have equal integer length \( p \) (abbreviated \( 1/p_j = p, r_j/L_{\text{max}} \)). Later in [10] was proposed an \( O(n^2 \log n \log p) \) algorithm solving a more general setting when a job processing time can be either \( p \) or \( 2p \) (abbreviated \( 1/p_j \in \{p, 2p\}, r_j/L_{\text{max}} \)).

Recently in [11] certain conditions which satisfaction guarantees the obtainment of an optimal solution to problem \( 1/r_j/L_{\text{max}} \) were presented. These conditions take an advantage of a close relationship between the scheduling problem and a version of the bin packing problem with different bin capacities. The heuristic method that we build here also takes an advantage of this relationship. The schedules that we create are partitioned into two types of intervals, containing, roughly classifying, urgent and non-urgent jobs. We call the intervals containing urgent jobs kernel intervals, and the intervals containing non-urgent jobs bin intervals. In every optimal schedule, kernel jobs form a tight sequence in the sense that...
the delay of its earliest scheduled job (i.e., the difference between the starting and release times of that job) cannot exceed some precalculable magnitude \( \delta \in [0, p_{\text{max}}] \), where \( p_{\text{max}} \) is the maximal job processing time.

Because of a little degree of the flexibility, it is easier to arrange kernel intervals. Our heuristic method uses ED-heuristic to schedule these intervals. The rest of the scheduling horizon consists of the bin intervals, within which all the non-urgent jobs are to be distributed. Our task is then to find a proper such job distribution. We use a variation of LPT (Longest Processing Time) heuristic to find such distribution of the non-urgent jobs. The LPT-heuristic, iteratively, at each scheduling time \( t \) (given by job release or completion time), among the jobs released by time \( t \) schedules one with the largest processing time (breaking ties by selecting a most urgent one).

The practical behavior of our algorithm was tested for a number of randomly generated problem instances, described in the concluding section.

II. PRELIMINARY CONCEPTS AND NOTIONS

From here on, let \( S \) be an ED-schedule, one created by ED-heuristic.

Schedule \( S \) may contain a gap, that is its maximal consecutive time interval in which the machine is idle. We assume that there occurs a 0-length gap \((c_j, t_i)\) whenever job \( i \) starts at its release time immediately after the completion of job \( j \).

A block in \( S \) is its consecutive part consisting of the successively scheduled jobs in without any gap in between, which is preceded and succeeded by a (possibly a 0-length) gap.

Given schedule \( S \), let \( i \) be a job that realizes the maximum job lateness in \( S \), i.e., \( L_i(S) = \max_j \{L_j(S)\} \). Let, further, \( B \) be the block in \( S \) that contains job \( i \). Among all the jobs in \( B \) with this property, the latest scheduled one is called an overflow job in \( S \) (not necessarily it ends block \( B \)).

Note that if schedule \( S \) contains two or more overflow jobs then they belong to different blocks in \( S \).

A kernel in \( S \) is a maximal (consecutive) job sequence ending with an overflow job \( o \) such that no job from this sequence has a due-date more than \( d_o \). For a kernel \( K \), we let \( r(K) = \min_{i \in K} \{r_i\} \), and will denote by \( L(K) \) the maximum lateness of a job in \( K \).

It follows that every kernel is contained in some block in \( S \), and the number of kernels in \( S \) equals to the number of the overflow jobs in it. Furthermore, since any kernel belongs to a single block, it may contain no gap.

In schedule \( S \), the delay of kernel \( K \) is the difference between the starting time of its earliest scheduled job and \( r(K) \).

Observation 1: The maximum job lateness in a kernel \( K \) cannot be reduced if it has no delay (i.e., the earliest scheduled job in \( K \) starts at time \( r(K) \)). Hence, if an ED-schedule \( S \) contains a kernel with this property, then it is optimal.

Proof. Recall that all jobs in \( K \) are no less urgent than the overflow job \( o \), and that jobs in \( K \) form a tight sequence (i.e., without any gap). Then since the earliest job in \( K \) starts at its release time, no reordering of jobs in \( K \) can reduce the current maximum lateness, which is \( L_o(S) \). Hence, there is no feasible schedule \( S' \) with \( L(S') < L_o(S) \), i.e., \( S \) is optimal

Due to the above observation, assume, without loss of generality, that the condition in Observation 1 does not hold. Then there exists a job, less urgent than \( o \), scheduled before all jobs in \( K \) that delays the starting of jobs in \( K \). By rescheduling such a job to a later time moment behind \( K \), the jobs in \( K \) can be restarted earlier. We define now this operation formally.

Suppose \( i \) precedes \( j \) in \( S \). We will say that \( i \) pushes \( j \) in \( S \) if ED-heuristic will reschedule \( j \) earlier if \( i \) is discarded.

It follows that the earliest scheduled job of every kernel is immediately preceded and pushed by a job \( e \) with \( d_e > d_o \). In general, we may have more than one such a job scheduled before kernel \( K \) in block \( B \) (one containing \( K \)). We call such a job an emerging job for \( K \), and we call the latest scheduled one (job \( e \) above) the delaying emerging job.

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

III. THE HEURISTIC

As we have mentioned in the introduction, our heuristic is based on the idea of partitioning the scheduling horizon into the urgent (kernel) and non-urgent (bin) intervals. It consists of the two basic stages. First, at the partitioning stage, all the kernel and bin intervals are determined. At the construction stage, kernel and bin intervals are filled in by urgent and the non-urgent jobs, respectively.

A. The partitioning stage

We have implemented two versions for extracting the kernel intervals at the partitioning stage. In both of these versions, the initial ED-schedule \( \sigma \) is created; \( \sigma \) is obtained by ED-heuristic, which is applied to the originally given problem instance. In schedule \( \sigma \), one or more kernels in different blocks (with the same value of the maximum job lateness, may arise). Note that the corresponding overflow jobs have the same lateness and they pertain to different block in \( \sigma \). This set of kernels in schedule \( \sigma \) form the initial set of kernels.

If we will have a deeper look into the structure of the ED-schedules we may see that extra potential kernels may be “hidden” within schedule \( \sigma \). Consider a simple instance with three jobs with the parameters: (job 1) \( r_1 = 0, p_1 = \)
10, $d_1 = 100$, (job 2) $r_2 = 1$, $p_2 = 3$, $d_2 = 4$ and (job 3) $r_3 = 5$, $p_3 = 3$, $d_3 = 9$.

Since at time 0 only job 1 is released, the initial schedule $\sigma$ assigns job 1 at time 0, then it assigns job 2 right at the completion time 10 of job 1, and finally it assigns job 3 at time $10 + 3 = 13$. There is a single kernel in $\sigma$ consisting of job 2, which is the overflow job with the lateness $10 + 3 - 4 = 9$, whereas job 1 is the delaying emerging job. Note that the lateness of job 3 in $\sigma$ is $13 + 3 - 9 = 7$.

If we activate the delaying emerging job 1 for the above kernel, we obtain another ED-schedule $\sigma_1$, in which job 1 starts at time 1 and completes at time 4 with 0 lateness; at that completion time, only job 1 is released, hence it is assigned at time 4 and is completed at time 14; at that time, job 3 is assigned. The lateness of job 3 in schedule $\sigma_1$ is $14 + 3 - 9 = 8$. Hence, there arises a new kernel consisting of a single job 3 in schedule $\sigma_1$ (the former kernel of schedule $\sigma$ consisting of job 2 disappears in $\sigma_1$).

During the partitioning stage of our heuristic, the augmentation of the initial set of kernels in schedule $\sigma$ by the above kind of the “hidden” kernels yields an improved, more accurate, performance results.

The kernel augmentation procedure has two versions. In the first one, whenever a new kernel arises, the delaying emerging job is temporally omitted, the corresponding kernel is rescheduled by ED-heuristic (being correspondingly left-shifted), and the construction proceeds similarly by ED-heuristic (with the rescheduled kernel though) until another kernel is encountered or schedule $\sigma^*$, consisting of all the jobs except the omitted delaying emerging jobs, is constructed. Note that the earliest scheduled job of every arisen during the procedure kernel $K$ will start at its release time $r(K)$ in $\sigma^*$.

Let $L^*_i$ be the (reduced) lateness of a kernel job $i$ in $\sigma^*$, and let $\delta(K) = L^*_i - L^*(K)$. Since every kernel $K$ is restarted at time $r(K)$ in $\sigma^*$, $L^*(K) = \max_{i \in K} \{L^*_i\}$, and hence $L^* = \max_{i} \{L^*(K^*_i)\}$ are lower bounds on the objective value:

**Observation 2:** The maximum lateness in schedule $\sigma^*$ obtained on the partitioning stage is a lower bound on the optimal objective value.

Proof. By the definition of schedule $\sigma^*$, every kernel $K$ arisen during the partitioning stage starts at its earliest possible starting time $r(K)$ in $\sigma^*$. Then our claim immediately follows from Observation 1.

The kernels intervals can be defined with some degree of the flexibility, due to the observation.

**Observation 3:** Every kernel $K$ can be delayed by $\delta(K)$ without increasing the maximum lateness.

Proof. Let $K'$ be a kernel that realizes $\max_{i} \{L^*(K'^*_i)\}$. By definition of $\delta(K)$, the completion time of every job in $K \neq K'$ can be increased by $\delta(K)$ so that none of the jobs in $K$ will be completed later than a job realizing $\max_{i \in K'} \{L^*_i\}$. This clearly proves the observation.

From Observation 3, we may assert that in an optimal schedule $S_{opt}$ every kernel $K$ starts either no later than at time $r(K) + \delta(K)$ or it is delayed by some $\delta \geq 0$ (the latter delay, as we will see later, may be unavoidable for a proper accommodation of the non-kernel jobs). Let $\Delta = L_o(\sigma) - L^*$, where $o$ is an overflow job in $\sigma$. Then note that the maximum lateness in any feasible ED-schedule in which the delay of some kernel is more than $\Delta$ is no less than that in $\sigma$, i.e. Hence, no such schedule will be created by our heuristic.

We shall refer to the magnitude $L^* + \delta (0 \leq \delta \leq \Delta)$ as the $\delta$-boundary.

Recall that the first version of the kernel augmentation procedure, in schedule $\sigma^*$, each delaying job is omitted. In the second version of the procedure, every delaying emerging job is activated for the corresponding kernel. Thus the second version of the kernel augmentation procedure is similar to the first one, with the difference that, for every arisen kernel, the corresponding delaying emerging job is activated for that kernel (instead of being omitted).

### B. The construction stage

At the construction stage, the heuristic schedules kernel jobs so that none of them surpasses $\delta$-boundary, for any given choice of $\delta$. The kernel intervals are given some degree of flexibility, depending on the value of $\delta$ according to Observation 3. The value of $\delta$ can be taken arbitrarily from the interval $[0, \Delta]$. 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 (Observation 3) the starting and completion times of the corresponding kernel and bin intervals are defined with the allowable flexibility, determined by the current value of the parameter $\delta$ (note that, since there may exist no gap within any kernel segment, the length of every kernel interval and hence the corresponding bin intervals are fixed).

Bin intervals are scheduled by LPT-heuristic so that the bin interval before every kernel $K$ is extended up to the time moment $r(K) + \delta(K) + \delta$. If the next job selected by LPT-heuristic completes by time $r(K) + \delta(K) + \delta$, it is scheduled the next; otherwise, among the available jobs, the next shortest job is similarly selected, until none of the released jobs fits into the bin (within still available interval before time moment $r(K) + \delta(K) + \delta$). Then the next bin is similarly scheduled until all bins are scheduled.

### IV. PRELIMINARY COMPUTATIONAL EXPERIMENTS AND FINAL REMARKS

We have implemented our heuristic (with both versions of the kernel augmentation procedure) in Java using the development environment Eclipse IDE for Java Developers (version Luna Service Release 1 (4.4.1)) under Windows 8.1 operative system for 64 bits, and have used a laptop with Intel Core i7 (2.4 GHz) and 8GB of RAM DDR3 to run the code. The inputs in our main program are plain texts with job data that we have generated randomly, as we briefly describe below. The program for the generation of our instances was

constructed under the same development environment as our main program.

The computational experiments are at an early stage of development, still an ongoing research. So far, job release times and due dates were generated with the \( rdn() \) function in Java, with an open range \((0, 50n)\), where \( n \) is the number of jobs in a corresponding instance. The processing times were generated from the interval \([1, 50]\) and also from the interval \([1, 100]\).

For a majority of the created problem instances the heuristic with the second version of the kernel augmentation procedure gave a solution with the objective value equal to the corresponding lower bound (as in Observation 2), whereas about 60% of the solutions with the first version of the kernel augmentation procedure achieved this lower bound. Since the heuristic runs in time \( n \log n \), all the instances were solved instantly.

In the instances that were not solved optimally, the activated delaying emerging jobs have converted to the overflow jobs, hence the objective value could have been improved. We intend to extend the heuristic with an additional subroutine dealing with that kind of scenario. This, we believe, will improve its performance. Besides, we plan to test the heuristic for larger amount of problem instances, also generated randomly but in several different ways. For instance, the set of jobs can be divided into two or more subsets and job parameters for each subset can be derived independently, from different time intervals.

REFERENCES