Fast Approximation for Scheduling One Machine

Federico Alonso-Pecina ¹ , José Alberto Hernández ¹ , José Maria Sigarreta ²
and Nodari Vakhania ^{3,*}

¹ Faculty of Account, Administration and Informatics, Universidad Autónoma del Estado de Morelos, Cuernavaca 62209, Mexico; federico.alonso@uaem.mx (F.A.-P.); jose_hernandez@uaem.mx (J.A.H.)

² Facultad de Matemáticas UAGro, Universidad Autónoma de Guerrero, Acapulco 39650, Mexico; josemariasigarretaalmira@hotmail.com

³ Centro de Investigación en Ciencias, UAEMor, On sabbatical leave at Facultad de Matemáticas UAGro, Universidad Autónoma de Guerrero, Acapulco 39650, Mexico

* Correspondence: nodari@uaem.mx

Received: 27 July 2020; Accepted: 4 September 2020; Published: 7 September 2020

Abstract: We propose an approximation algorithm for scheduling jobs with release and delivery times on a single machine with the objective to minimize the makespan. The algorithm is based on an implicit enumeration of the set of complete solutions in a search tree. By analyzing specific structural properties of the solutions created in each branch of the solution tree, a certain approximation factor of each solution from that branch is calculated. Our algorithm guarantees approximation factor $1 + 1/\kappa$ for a generated solution if there are κ jobs with a specified property in that solution (typically, the longer the length of the path from the root to the node representing that solution in the solution tree, the larger the κ value). We have carried out an extensive computational study to verify the practical performance of our algorithm and the effectiveness of the approximation factor provided by us. While our problem instances are generated randomly, we have discarded a considerable number of the instances, ones which were already solved optimally by the earlier known dominance rules. For the vast majority of the tested problem instances, within a short running time of our algorithm parameter κ becomes sufficiently large so that the approximation factor which the algorithm guarantees becomes better than that provided by the earlier known approximation algorithms.

Keywords: scheduling algorithm; enumeration; approximation; release time; due-date; delivery time; single-machine

1. Introduction

The one-machine scheduling problem that we consider in this paper can be formulated as follows: n jobs have to be scheduled on a single machine. Job j becomes available at its *release time* r_j , can be scheduled on the machine at time moment r_j or later, and needs to be processed continuously during p_j time units by the machine. The machine can handle at most one job at a time. If job j starts at time s_j on the machine, then its *completion time* is $c_j = s_j + p_j$. The *due-date* d_j of job j is the time moment at which the completion of that job is desired. A *feasible schedule* S is a mapping that assigns to each job j a starting time $s_j(S)$ on the machine, such that $s_j(S) \geq r_j$ and $s_j(S) \geq s_i(S) + p_i$, for any job i assigned to the machine before time $s_j(S)$ in schedule S . The first inequality represents the restriction that a job cannot be started before its release time, and the second one represents the resource restriction that the machine can handle only one job at any time moment. The *lateness* of job j in schedule S is $L_j(S) = c_j(S) - d_j$. The objective is to find a feasible schedule that minimizes the maximum job lateness $L_{\max}$.

The problem has an equivalent setting in which job due-dates are replaced by job *delivery times*. In this setting, every j completed on the machine requires an additional (constant) *delivery time* q_j for

its *full completion* (the finished orders need to be delivered by an independent agent, which needs no machine time). Thus, the *full completion time* of job j is $C_j = c_j + q_j$. A feasible schedule is defined as for the first setting. The objective is to find an *optimal* schedule, one minimizing the maximum full job completion time (the so-called *makespan*).

Note that the smaller the due-date of a job, the more urgent the job is; equivalently, the larger the delivery time of a job, the more urgent it is. The equivalence between the two settings is established by interchanging the roles of job delivery times and job due-dates. Given the setting with delivery times, for every job j , take a suitably large constant K (any magnitude no less than the maximum job delivery time) and define due-date of that job $d_j = K - q_j$, and vice-versa (Bratley et al. [1]).

Since both settings describe the same problem, we will refer to them interchangeably. According to the Graham's three-field notation, the settings with job due-dates and delivery times are abbreviated as $1|r_j|L_{\max}$ and $1|r_j, q_j|C_{\max}$, respectively (in the first, the second and the third fields, respectively, the single-machine environment, the job parameters, and the objective criteria, respectively are specified). The problem is well-known to be strongly NP-hard Garey and Johnson [2], and it remains (weakly) NP-hard with only two allowable job release times [3].

Overview of some related work. We first give a short overview of some work related to our scheduling problem and then we mention about recent work on some related single-machine scheduling problems.

The first exponential-time implicit enumeration (branch-and-bound) algorithms for problem $1|r_j|L_{\max}$ were proposed in late 70s and a few more implicit enumeration algorithms were suggested in 80s (see, for example, McMahon and Florian [4], Carlier [5] and Grabowski et al [6]).

As to basic polynomially solvable special cases, remarkably, if all jobs are released simultaneously or all jobs have the same delivery time (or the same due-date), the greedy algorithm proposed by Jackson in the late 50s [7] finds an optimal solution. Jackson's greedy algorithm was adopted for the general case $1|r_j|L_{\max}$ by Schrage [8]. The extended Jackson's algorithm, iteratively, determines the current scheduling time, which is either job release time of job completion time, and among the jobs which are released by this time schedules a job with the smallest due-date (or the largest delivery time). For further references, we abbreviate the heuristics as ED (Earliest Due-date) and LDT (Largest Delivery Time) heuristics, respectively.

For the preemptive version of our problem, $1|r_j, pmtn|L_{\max}$ (any running job can be interrupted in favor of another job and its execution can later be resumed), Jackson's extended heuristic is optimal. Hence, the non-preemptive version of this problem is essentially more complex than the preemptive one. With non-simultaneously released jobs, if the processing time of all jobs is the same, the (non-preemptive) problem is essentially harder to solve than the version without job release times, but it still admits polynomial-time solution Garey et al. [9], and remains polynomially solvable with only two allowable job processing times [10]. If the set of job processing times consists of mutually divisible numbers (e.g., powers of 2), then the problem $1|r_j|L_{\max}$ can still be solved in polynomial time [11] (apparently, this is a most general setting with non-arbitrary job processing times that can be solved in polynomial time, since the setting in which job processing times are from the set $\{p, 2p, 3p, \dots\}$, for any integer p , is strongly NP-hard [12]).

In general setting, since our scheduling problem $1|r_j|L_{\max}$ is NP-hard, a reasonable alternative to an implicit enumeration of the feasible solutions is an approximation solution method. A λ -approximation algorithm obtains a solution with the objective value at most λ times the optimal objective value, for any instance of a given problem (λ is commonly referred to as the *approximation ratio*). Jackson's extended heuristic gives a 2-approximation solution for the general setting $1|r_j, q_j|C_{\max}$. Approximation algorithms with approximation ratios $3/2$ and $4/3$ respectively, were suggested by Potts [13] and Hall and Shmoys [14] (both are based on the Jackson's greedy algorithm). To the best of our knowledge, there is no polynomial-time algorithm with an approximation ratio better than $4/3$.

We refer the reader to Lawler et al. [15] for an extensive overview on machine scheduling problems. Below we address a few recent contributions on some related one-machine scheduling problems in which job parameters or/and objective functions are different from those in our problem.

As to different objectives, in some applications it is desirable to complete the jobs “just-in-time”, i.e., it is undesirable to complete them not only far behind but also far before their due-dates (for instance, because of the warehousing and transportation issues). A job is said to be *early* if it is completed before its due-date, and is *tardy* if it is completed after its due-date. Clearly, the early completion of all jobs favors the minimization of the maximum lateness (note that in a feasible schedule the lateness of some or even of all jobs can be negative). The *tardiness* of a job is defined similarly as its lateness with the difference that it cannot be negative, i.e., if the lateness of j is negative then its tardiness is set to 0. There are possible different objective functions that favor neither early nor tardy jobs. One such a function is the sum of the maximum job earliness and the maximum job tardiness. With non-equal job release times and due-dates, the setting with this objective function is known to be strongly NP-hard. Mahnam et al. [16] proposed a branch-and-bound method for the problem. Yazdani et al. [17] consider a model with the objective function without job release times (i.e., all jobs are simultaneously released) but with an additional restriction that there are multiple machine unavailability periods, time intervals in which no job can be scheduled on the machine (such a restriction might be motivated by the machine maintenance, repair, etc.). The authors present a variable neighborhood search meta-heuristic for this model (which is also known to be NP-hard).

In some industries the production process is not heterogeneous with respect to the set of the jobs, i.e., different jobs have different priorities. For such environments, the (positive) weights for each job are introduced (the jobs with larger weight have higher priority). An overview on the weighted single-machine scheduling problems with the objective to minimize the weighted number of the tardy jobs can be found in Adamu and Adewumi [18].

Scheduling models with learning and deterioration effects were recently considered. Roughly, by the deterioration effect, the processing time of a job depends on its starting time—the later it starts, the more processor time it needs. This might be caused by the fact that the machine may deteriorate during a long period of the load time or due to the fact that the raw materials that need to be processed deteriorate over time. By the learning effect, the job processing time depends also on the position of this job among the other jobs—the higher this position is, the smaller the amount of time that this order will need to complete: the workers and the machines can improve the production efficiency with more processing experiences. These effects are reflected in job processing times with the help of some auxiliary parameters. In the papers listed below all jobs are simultaneously released.

Yin and Xu [19] consider one-machine scheduling problems with learning and deterioration effects with the objective to minimize the makespan, sum of the i th power of completion times of jobs, total lateness and sum of earliness penalties. They show that the proposed models are polynomially solvable. The authors also give some conditions under which the settings with the objective to minimize the total weighted completion time, total weighted completion time, maximum lateness, maximum tardiness, total tardiness and total weighted earliness penalties are polynomially solvable. Lu et al. [20] study other related one-machine scheduling problems with learning effects. They consider settings with different objective functions including again the makespan and the total completion time of all jobs and propose polynomial-time solution methods. Hou et al. [21] propose another one-machine scheduling model with a single machine maintenance period, after which job processing times change according to a given deterioration formula (the machine deteriorates over time and it can be fully or partially restored after the maintenance period). Recently Cheng et al. [22] consider other single-machine scheduling models with multiple machine maintenance periods. The jobs between the maintenance periods are partitioned into the batches. Each maintenance period depends on the total processing time of the jobs from the batch processed before that period, and the job processing times are deteriorated according to given formulas. Polynomial time algorithms are proposed for the objectives to minimize the makespan and the total job completion time. Park and Choi [23] consider a more complex situation

with a single in-house machine and the outsourcing costs. If a job is not scheduled on the (in-house) machine then it causes the outsourcing cost. The authors consider uncertainty setting in which the job processing times and the outsourcing costs are dependent on a given scenario. The objective is to minimize a specially defined weighted sum of the completion times of the jobs assigned to the in-house machine plus all the outsourcing costs. Since the considered scheduling problems are NP-hard even in the deterministic settings, the authors present some optimality conditions yielding polynomial time solutions.

Our contribution. Due to the complexity status of our scheduling problem, implicit enumeration algorithms cannot guarantee the optimal solution for “large enough” problem instances as their worst-case running time is exponential in the length of the input. Although the algorithm that we propose here is based on the enumeration of the feasible solutions, it uses an approximation condition that provides a certain approximation factor for each created feasible solution in the search tree. Regarded as an exact solution method, our algorithm carries out implicit enumeration of complete feasible solutions in a search tree, in which each node represents a complete feasible solution. The reduction of the search space is accomplished on the basis of the established pruning and halting conditions. Based on these conditions, unpromising feasible solutions are discarded in the search tree. At the same time, our algorithm can be regarded as an approximation algorithm since it calculates an approximation factor λ for each enumerated feasible solution and hence, in practice, the algorithm can be stopped if a desired approximation is already guaranteed in the next created feasible solution. The approximation factor of each solution from a branch of the solution tree is calculated by constructing that solution according to specific rules and then by analyzing specific structural properties of the earlier created solutions in that branch. In particular, the approximation factor $\lambda = 1 + 1/\kappa$ for a created solution is guaranteed if there are detected κ jobs with a specific property in the branch of the search tree to which that solution belongs (typically, the larger the length of the path from the root to the node representing a solution in the search tree, the larger the value of κ). In this way, a complete enumeration can be avoided whenever a desired approximation factor is already attained in the next generated feasible solution. Since the number of feasible solutions grows exponentially with the length of the input, the worst-case time and space complexities of our basic enumeration scheme are exponential. Although we cannot give a reasonable theoretical estimation for the parameter κ , in practice, it becomes large enough within a short running time of our implicit enumeration algorithm.

To verify the practical performance of our algorithm and the effectiveness of the approximation factor calculated for each generated solution, we have carried out extensive computational experiments. As it turned out, the basic enumeration scheme gives an optimal solution for a moderate problem instances. At the same time, since our main goal is to find an approximation solution in a short period of time, we were mainly interested in the guaranteed approximation factor of the best generated solution within a short interval of time. As the experimental results have shown, the approximation factor that our algorithm attains within a small execution time is, on average, better than the approximation factors provided by the earlier known approximation algorithms (these are traditional heuristic algorithms in the sense that they are not based on the enumeration of the feasible solution set).

We have created and tested over thousand problem instances randomly, 50 problem instances with different number of jobs. We stress that a very considerable number of problem instances were solved optimally already by the standard earlier known dominance rules (in particular, ones used in the earlier mentioned enumeration algorithms). These instances (which formed over 80% of all the generated instances) are not present in the experimental data that we report here. The vast majority of the tested “difficult” instances up to 50 jobs were solved optimally almost instantaneously, whereas the guaranteed average approximation factor λ from 100 to 600 jobs was about 1.3 and from 700 to 1000 jobs it was about 1.2. About 10% of larger problem instances with 2000 and 5000 jobs were solved optimally within one minute of the running time of our algorithm and the average approximation factor λ was about 1.3. Remarkably, the average approximation factor for the largest tested instances with 10,000 has decreased to about 1.25.

To test the behavior of our enumeration framework in the worst possible scenarios, we have generated pseudo-random artificial problem instances as well. This second class of instances were created as inconvenient ones for our algorithm so that it would be forced to perform an almost complete enumeration of the candidate ED-schedules. Our aim was to verify the approximation that our algorithm would still guarantee in a reasonable time. As intended, our algorithm has failed to create an optimal solution for already moderately sized artificial instances. However, extremely good approximation factors very close to 1 for these instances were attained.

In the following Section 2, we give preliminaries including some earlier known definitions and properties. In Section 3 we describe the basic framework of our implicit enumeration algorithm. In Section 4 we give our optimality, halting and approximability conditions that we incorporate into the basic framework. We analyze our experimental study in Section 5 and we give our final remarks in Section 6.

2. Preliminaries

This section contains the earlier known preliminary notions which are used here (see, e.g., [10,24]). First, we give a more detailed description of ED-heuristic (and LDT-heuristic, respectively), a tool for the generation of our schedules. The algorithm works on n scheduling times, at which a job is assigned to the machine. At the first iteration the initial scheduling time is defined as the minimum job release time, and a most urgent job, one with the minimum due-date (the maximum delivery time, respectively) is scheduled on the machine (ties can be broken arbitrarily). Iteratively, the current scheduling time is determined as the maximum between the completion time of the latest scheduled job and the release time of an earliest released yet unscheduled job (recall that no job can be started before its release time and no job can be started while the machine remains busy). Again, among all jobs released by this scheduling time, a job with the minimum due-date (the maximum delivery time, respectively) is scheduled on the machine. The heuristic may create no avoidable gap, since whenever the machine becomes idle and there is a released job, it schedules it on the machine. At the same time, among the yet unscheduled jobs released by each scheduling time it gives priority to the most *urgent* job (one with the minimum due-date or the maximum delivery time).

The two above heuristics give equivalent results for the two equivalent scheduling problems. We shall refer to a schedule created by ED-heuristic (LDT-heuristic, respectively) an ED (LDT, respectively) schedule. If we apply the heuristics to the originally given problem instance, we obtain the initial ED (LDT) schedule that we denote by σ . As we will see a bit later, by slightly modifying the original problem instance, alternative ED (LDT) schedules with desired characteristics can be created.

We will refer to a maximal consecutive time interval in a schedule within which the machine is idle as a *gap*. For convenience, we will assume that there occurs a 0-length gap (c_j, t_i) if job i starts at time $t_i = r_i$ immediately after the completion time c_j of job j .

An ED-schedule S can naturally be divided into independent portions that will be referred to as *blocks*: A block is a consecutive part in schedule S consisting of the successively scheduled jobs without any gap in between any two neighboring jobs; a block is preceded and succeeded by a (possibly 0-length) gap.

The kernels. The whole set of jobs can be partitioned, roughly, into two kinds of jobs: non-critical and critical ones. Intuitively, the non-critical jobs are flexible in the sense that they might be moved within a feasible schedule without affecting the objective value, unlike the critical jobs which attain the maximum value of the objective function. Below we define our critical jobs formally.

Consider a maximal consecutive job sequence K in a block of an ED-schedule S ending with say, job o , such that

$$L_o(S) = \max_j \{L_j(S)\}$$

and no job from this sequence has a due-date more than d_o , i.e.,

$$d_j \leq d_o,$$

for all $j \in K$. In terms of the delivery times, for the equivalent setting we have

$$C_o(S) = \max_j \{C_j(S)\}$$

and

$$q_j \geq q_o,$$

for all $j \in K$ (so any job of the sequence is no less urgent than job o).

We call such sequence K in ED (LDT) schedule S a *kernel*, and we call job o the corresponding *overflow job* (abusing the notation, we use K also for the corresponding job-set). We let

$$r(K) = \min_{j \in K} \{r_j\}.$$

We stress that there may exist no gap within a kernel.

If a kernel K is immediately preceded by a gap, then it starts a new block; otherwise, it is immediately preceded and delayed by job l with $d_l > d_o$ ($q_l < q_o$, respectively). In general, we may have more than one job e with $d_e > d_o$ ($q_e < q_o$) scheduled before kernel K within the block containing that kernel (job e is pushing the jobs of kernel K in the sense that the removal of that job would restart the first job of kernel K earlier).

We call such a job e an *emerging job*, and job l above the *delaying* emerging job for the kernel K in schedule S .

As we will see in detail later, by rescheduling an emerging job to a later time moment and restarting kernel jobs earlier the current maximum objective value can be reduced in a newly created ED (LDT) schedule. The following halting condition is from [11] (see Proposition 1 in [11]):

Proposition 1. *The initial ED (LDT) schedule σ is optimal if it contains a kernel K with its earliest scheduled job starting at time $r(K)$, equivalently, if there exists no (delaying) emerging job for that kernel.*

If the above condition does not hold, our enumeration procedure is initiated by creating alternative branches, one for each of the emerging jobs. Before we describe our branching scheme in more detail, we give some basic properties of the ED schedules that we enumerate.

First, we observe that an ED schedule S may contain more than one kernel, and that the overflow job o of a kernel K is succeeded by a gap or it is succeeded by a job j with $L_j(S) < L_o(S)$ (if S is an LDT schedule, then $C_j(S) < C_o(S)$ (hence $d_j > d_o$ and $q_j < q_o$). We will denote the earliest arisen kernel in schedule S by $K = K(S)$ and will refer to it as *the kernel* in that schedule; respectively, we will refer to the corresponding overflow job as *the overflow job* of schedule S .

Given an ED-schedule S with the delaying emerging job $l = l(S)$ for the kernel $K(S)$, we denote by

$$\delta(K) = c_l(S) - r(K)$$

the delay of kernel $K = K(S)$ in schedule S , which is a forced right-shift imposed by the delaying job l for the jobs of kernel K . The following known fact easily follows from Proposition 1 and an easily seen observation that no job of kernel K is released by the time when job l is started in schedule S , as otherwise ED (LDT) heuristic would have included the former job instead of job l :

Property 1. $\delta(K) < p_l$.

Let $|S|$ be the objective value of an ED (LDT) schedule S and let OPT be that of an optimal schedule S^{opt} . Property 1 easily implies the next well-known corollary.

Corollary 1. $|S| - \text{OPT} < p_l$.

Given a (non-optimal) ED-schedule S , now we specify how an alternative ED (LDT) schedule can be created from that schedule. We will say that an emerging job e is *activated* for the kernel $K(S)$ in schedule S if it is rescheduled after that kernel. We activate job e so that the resultant schedule S_e , called a *complementary* (to S) schedule, is also an LDT (ED) schedule. We do this by merely increasing artificially the release time of job e to the maximum release time of a job in kernel $K(S)$ so that the heuristic, once applied to the modified (in this way) problem instance, will reschedule job e after all the jobs of kernel $K(S)$. Indeed, since job e becomes released no earlier than any job of kernel $K(S)$, the heuristic will include any kernel job before job e .

The jobs of kernel $K(S)$ can be left-shifted in the complementary schedule S_e , i.e., they may be restarted earlier than in schedule S . In particular, this will be the case if no new emerging job (one, included after kernel $K(S)$ in schedule S) gets included before kernel $K(S)$ in schedule S_e ; otherwise, a newly included emerging job may similarly be activated.

Example 1. In Tables 1 and 2 we give a small randomly generated problem instance with 10 jobs, one of the instances that we have tested during our experiments. The initial ED-schedule σ is represented in Table 2 and Figure 1, in the form of a table and graphically, respectively. It is easy to see that kernel $K(\sigma)$ consists of jobs 3 and 8 (8 is the overflow job), jobs 0 and 9 are emerging jobs. Table 3 represents a modified problem instance, in which the release time of the delaying emerging job 9 is artificially increased. The result of the application of ED-heuristics to that modified instance is a complementary schedule σ_9 represented in Table 4 and Figure 2. The optimal schedule for the instance is depicted in Figure 3. The kernel of that schedule consists of a single job 5, which is also the overflow job; there exists no emerging job in that schedule.

Table 1. Data of the instance N_12_10.

j	0	1	2	3	4	5	6	7	8	9
r_j	32	86	86	191	133	8	276	137	238	100
p_j	67	20	23	11	32	82	23	32	18	100
d_j	391	422	420	277	252	107	485	426	289	295

Table 2. Initial solution σ of the instance N_12_10.

j	5	0	4	9	3	8	2	1	7	6
s_j	8	90	157	189	289	300	318	341	361	393
c_j	90	157	189	289	300	318	341	361	393	416
l_j	−17	−234	−63	−6	23	29	−79	−61	−33	−69
Belong to κ	N	N	N	N	Y	Y	N	N	N	N

Figure 1. Graphics of the initial Earliest Due-date (ED)-schedule σ ; kernel jobs are in red, emerging job are in green and the rest of the jobs are in blue.

Table 3. Data of a modified instance.

j	0	1	2	3	4	5	6	7	8	9
r_j	32	86	86	191	133	8	276	137	238	238
p_j	67	20	23	11	32	82	23	32	18	100
d_j	391	422	420	277	252	107	485	426	289	295

Table 4. Table for the complementary schedule σ_9 .

j	5	0	4	2	3	9	8	1	7	6
s_j	8	90	157	189	212	223	323	341	361	393
c_j	90	157	189	212	223	323	341	361	393	416
l_j	−17	−234	−63	−208	−54	28	52	−61	−33	−69
Belong to κ	N	N	N	N	N	N	Y	N	N	N

Figure 2. Graphics of the complementary schedule σ_9 . **Figure 3.** The optimal solution for instance Dif_12_10.

3. Basic Enumeration Scheme

In this section we describe how we enumerate the feasible solutions. Our basic branching scheme that relies on Propositions 1 and 2 is similar to that used in [4,5]. In the next section we complete our enumeration framework with further pruning and halting conditions straightforwardly incorporated into this section's basic scheme.

We associate with every node h in our search tree $\mathcal{T}$ a complete ED schedule S^h . For simplicity, we will refer to node $h \in \mathcal{T}$ and stage h interchangeably, and denote by $\mathcal{T}^h$ the enumeration tree generated by stage h ; initially, $S^0 = \sigma$ (where 0 is the root). A *closed* node in tree $\mathcal{T}^h$ is one without successors which may not have any successor, whereas an *open* node has no successor but may have it.

At stage h we determine kernel $K^h = K(S^h)$, the overflow job $o^h = o(S^h)$ from that kernel and the set of the emerging jobs $E^h = E(S^h)$ in schedule S^h in time $O(n)$. By the definition, a kernel K^h may contain no emerging job, but the block containing kernel K may include one or more emerging jobs for that kernel, where the latest scheduled one is the delaying emerging job l (the one that pushes the earliest scheduled job of kernel K). It is a known fact that if an ED (LDT) schedule S is not optimal, then in an optimal schedule S^{opt} at least one of the emerging jobs from that block is scheduled after kernel K ; in general, the objective value in schedule S^h cannot be reduced in any descendant of node h if $E^h = \emptyset$, otherwise it suffices to generate complementary schedule S_e^h , for each emerging job e :

Proposition 2. *If either $E^h = \emptyset$ or the first scheduled job of kernel K^h starts at its release time, then node h can be closed. Otherwise, let $e_1, \dots, e_k$, $k = |E^h|$, be the emerging jobs in set E^h . Then it suffices to create k immediate successors of node h , $S_{e_1}^h, \dots, S_{e_k}^h$.*

Proof. First, it is easy to see that if the equality $E^h = \emptyset$ holds, then either kernel K^h starts schedule S^h or the earliest scheduled job of that kernel starts at its release time. Our claim is obvious for the former case. Assume that the first scheduled job of kernel K^h starts at its release time. It is easy to see that no rearrangement of the jobs of that kernel may decrease the objective value of the overflow job $o(K^h)$. At the same time, since the first scheduled job of kernel K^h starts at its release time, no job rearrangement involving the jobs scheduled before kernel K^h may restart the earliest scheduled job of kernel K^h earlier than it is scheduled in schedule S^h . It follows that the full completion time of job $o(K^h)$ cannot be decreased by any job rearrangement and hence node h can be closed.

In case $E^h \neq \emptyset$, let $e_1, \dots, e_k$, $k = |E^h|$, be the enumeration of the emerging jobs in set E^h in the order, reverse as they appear in schedule S^h (so e_1 is the delaying emerging job). The k immediate successors of node h are created (for every emerging job in the set E^h in this order, the i th successor representing the complementary schedule $S_{e_i}^h$). $\square$

A formal description of our enumeration procedure that creates a complete search tree $\mathcal{T}$ is in Algorithm 1.

Algorithm 1: PROCEDURE *Enumerate()* {Generates tree $\mathcal{T}$ }

```

1 Initial settings
2  $h := 0; s^0 := \sigma$ 
3 IF there is no emerging job in schedule  $\sigma$  THEN stop { by Proposition 1 }
4 { Iterative step }
5 IF  $E^h = \emptyset$ 
6   THEN
7     close node  $h$ ; { by Proposition 2 }
8     IF there is an open node in tree  $\mathcal{T}$ 
9       THEN { backtrack }
10       $h :=$  the (leftmost) closest open node and repeat Iterative step
11    ELSE stop
12  ELSE
13     $\{ E^h = \{e_1, \dots, e_k\} \neq \emptyset, k = |E^h| \}$ 
14    { construct complementary schedule  $(S^h)_e$  for each emerging job  $e \in E^h$  }
15    create  $k = |E^h|$  successors of node  $h$ ,  $S^{h+i} := (S^h)_{e_i}$ , for  $i = 1, \dots, k$ 

```

4. Optimality and Approximability Conditions

While enumerating different complementary schedules in tree $\mathcal{T}$, an interaction of kernel K^h of the current complementary schedule S^h with the kernel of some earlier generated complementary schedule(s) may occur. In particular, the former kernel may coincide with the latter one, or the jobs from two or more earlier detected kernels may be joined into a newly formed kernel K^h . A detection and a proper analysis of such an interaction gives more insight into the problem and is also beneficial for the reduction of the search space. Below we describe briefly kernel interactions and give a few relevant definitions (the reader is referred to [11] for a more detailed presentation).

Recall that given that the kernel of schedule $S^h \in \mathcal{T}$ possesses an emerging job, such an emerging job e is activated in the complementary schedule S^h resulting in a new complementary schedule S_e^h , an immediate successor of schedule S^h . In schedule S_e^h the processing order of jobs of kernel K^h may coincide or may not coincide with the processing order of these jobs in schedule S^h . If the processing orders are different, then it is the case that a job $j \in K^h$ becomes rescheduled earlier in schedule S_e^h compared to its (former) position in schedule S^h and it becomes an emerging job immediately in schedule S_e^h or in a descendant S^g of schedule S^h for the kernel $K(S^g)$. Then PROCEDURE *Enumerate()* will similarly generate another complementary schedule S_j^g , an immediate successor of schedule S^g in solution tree $\mathcal{T}$. Similar scenario can be repeated as long as, as a result of the order change in the next created complementary schedule, a former kernel job becomes the delaying emerging job for the newly arisen kernel (a sub-kernel of kernel K^h), i.e., kernel K^h *collapses*; kernel K^h *fully collapses* in a complementary schedule S^f (a descendant of schedules S^h and S^g) if the last activation of a job of kernel K^h in a predecessor of schedule S^f yielded no further order change whereas all jobs of kernel K^f are ones from kernel K^h and the latter kernel possesses no delaying emerging job (we refer the reader to Section 4 of [11] for complete formal definitions and more details). Thus in each above newly created complementary schedule except schedule S^f a new emerging job, a former kernel job is activated (note that since during the collapsing of kernel K^h at most $|K^h| - 1$ emerging jobs from kernel K^h may arise, the total number of the created complementary schedules in which that kernel collapses is bounded by $|K^h| - 1$).

Example 2. We illustrate the kernel collapsing using a small instance of Tables 5 and 6. It is our third randomly generated problem instance with 10 jobs that we abbreviate by N_3_10. Figures 4 and 5 illustrate the initial

ED-schedule σ and ED-schedule σ_1 . As we can observe of Tables 7 and 8, in schedule σ_1 kernel $K(\sigma)$, consisting of jobs 0 and 6, is collapsed.

Table 5. A problem instance N_3_10 for which kernel $K(\sigma)$ collapses.

j	0	1	2	3	4	5	6	7	8	9
r_j	204	240	167	292	245	322	215	415	32	57
p_j	22	78	33	18	58	10	40	17	72	97
d_j	265	497	356	437	326	407	258	494	242	496

Table 6. Initial solution σ of the instance N_3_10.

j	8	9	2	6	0	4	5	3	1	7
s_j	32	104	201	234	274	296	354	364	382	460
c_j	104	201	234	274	296	354	364	382	460	477
l_j	−138	−295	−122	16	31	28	−43	−55	−37	−17
Belong to κ	N	N	N	Y	Y	N	N	N	N	N

Figure 4. Initial ED-schedule σ .

Table 7. A problem instance modified N_3_10 for which kernel $K(\sigma)$ collapses.

j	0	1	2	3	4	5	6	7	8	9
r_j	204	240	204	292	245	322	215	415	32	57
p_j	22	78	33	18	58	10	40	17	72	97
d_j	265	497	356	437	326	407	258	494	242	496

Table 8. Initial solution σ of the instance N_12_10.

j	8	9	0	6	4	2	5	3	1	7
s_j	32	104	204	226	266	324	357	367	385	463
c_j	104	201	226	266	324	357	367	385	463	480
l_j	−138	−295	−39	8	−2	1	−40	−52	−34	−14
Belong to κ	N	N	N	Y	N	N	N	N	N	N

Figure 5. ED-schedule σ_1 in which kernel $K(\sigma)$ is collapsed.

So far, we have basically relied on earlier known facts in the branching and the pruning rules used in the enumeration procedure of Section 3. In this section we give a few additional properties which are beneficial for the further reduction of the size of tree $\mathcal{T}$.

Lemma 1. Let g be a successor-node of a node h in the tree $\mathcal{T}$. If the kernel K^h does not collapse in schedule S^g and kernels K^g and K^h have a job in common, then $K^g = K^h$. Furthermore, if the first job of the kernel K^g starts at its release time in schedule S^g , then that schedule is optimal.

Proof. Since kernel K^h does not collapse in schedule S^g the order in both sequences K^h and K^g in both schedules S^h and S^g is the same. Furthermore, since kernel K^g contains a job of kernel K^h , the last job

in both kernels is the overflow job in both schedules and therefore $K^g = K^h$. The second claim follows from Proposition 1. $\square$

Kernels K^h and K^g are said to be *independent* if $K^h \cap K^g = \emptyset$, i.e., the two kernels have no job in common (h and g being defined as above).

Suppose kernels K^h and K^g are not independent and $K^g \neq K^h$. Then it is easy to see that if all jobs of kernel K^g also belong to kernel K^h , then kernel K^g is obtained as a result of the collapsing of kernel K^h . Otherwise (all jobs of kernel K^h belong to kernel K^g), kernel K^g is said to be an *extension* of kernel K^h .

Lemma 2. *If kernel K^g is an extension of kernel K^h , then kernel K^g includes the emerging job(s) activated for the kernel K^h in the corresponding branch of tree $\mathcal{T}$.*

Proof. Note that at least one emerging job e is activated for the kernel K^h in tree $\mathcal{T}$. Since kernels K^h and K^g belong to the same block (otherwise they would have been independent), job e belongs also to this block together with the jobs of both kernels. Moreover, job e is not an emerging job for the kernel K^g , as otherwise the two kernels would be independent. Since job e is not an emerging job for the kernel K^g and belongs to the same block as the latter kernel, it forms part of it. $\square$

Now we present our approximability condition that guarantees a certain approximation factor for each complementary schedule created in tree $\mathcal{T}$.

Theorem 1. *Suppose a branch of tree $\mathcal{T}$ contains κ complementary schedules possessing κ different independent kernels with the κ different delaying emerging jobs. Then the κ th complementary schedule $\sigma[\kappa]$ is an $(1 + 1/\kappa)$ -approximation one.*

Proof. Let $i_1, \dots, i_\kappa$ be the stages with the κ independent kernels and the corresponding κ distinct delaying emerging jobs $e_{i_1}, \dots, e_{i_\kappa}$. Let $\sigma[1], \dots, \sigma[\kappa]$ be the corresponding LDT-schedules. By Corollary 1, $|\sigma[l]| - \text{OPT} < p_{e_{i_l}}$, for $l = 1, \dots, \kappa$. Since these are κ distinct jobs, $\sum_{l=1, \dots, \kappa} p_{e_{i_l}} < \text{OPT}$. Furthermore, for the purpose of this estimation, we may assume that all κ delaying emerging jobs that have arisen have the same processing time, as otherwise the minimum of $|\sigma[l]| - \text{OPT}$ will be achieved at the earliest stage with the shortest e_{i_l} (notice that for the estimation of our approximation, every $|\sigma[l]| - \text{OPT}$ is a valid expression).

Thus $p_{e_{i_l}} < \text{OPT}/\kappa$ and the delay of none of the kernels is more than OPT/κ . Then

$$\frac{|\sigma[\kappa]|}{\text{OPT}} < \frac{\text{OPT} + \text{OPT}/\kappa}{\text{OPT}} = 1 + \frac{1}{\kappa}.$$

$\square$

Theorem 1 gives a guaranteed approximation factor for each next complementary schedule created in tree $\mathcal{T}$. Of course, our algorithm can be stopped at any moment when a desired approximation is attained in the next enumerated schedule. In fact, within a short running time of the procedure parameter κ becomes large enough to guarantee a good approximation (see Section 5).

Theorem 1 will not guarantee a desired approximation for complementary schedule S^h if the branch of tree $\mathcal{T}$ ending with node h does not contain “enough” complementary schedules with independent kernels; i.e., this branch contains the complementary schedules in which either the kernel of a predecessor schedule was extended or it is collapsed. These cases are dealt with in the following two lemmas.

Lemma 3. *Suppose that in a complementary schedule $S^g \in \mathcal{T}$ kernel K^h fully collapses and the overflow job in kernel K^g is a job from kernel K^h (where h is a predecessor node of node g in tree $\mathcal{T}$). Then schedule S^g is optimal.*

Proof. The full completion time of any job from a fully collapsed kernel is a lower bound on the optimal schedule makespan (see again Section 4 of [11]). Then the full completion time of the last job of kernel K^g is a lower bound on the optimum since this job is the overflow job in schedule S^g . Hence, schedule S^g is optimal. $\square$

Lemma 4. *Let kernel K^g be an extension of kernel K^h . Then an emerging job for the kernel K^g in schedule S^g , if any, is a (former) emerging job for the kernel K^h and is included before the jobs of that kernel in schedule S^g . If there exists no such emerging job, then node g can be closed.*

Proof. Since kernel K^g is an extension of kernel K^h , there may exist no emerging job for the kernel K^g included after the jobs of kernel K^h in schedule S^g (observe that stage g is a successor of stage h in the corresponding branch of solution tree $\mathcal{T}$). This shows the first claim. As to the second claim, suppose there exists no emerging job for the kernel K^g . In particular, the delaying emerging job l of kernel K^h in schedule S^h is either in the state of activation in schedule S^g or/and it is not an emerging job for the kernel K^g . In the first case, there occurs a gap before the first job of kernel K^h in schedule S^g , and hence the activation of no job included before that gap may left-shift any job scheduled after this gap in schedule S^g . In particular, the completion time of the overflow job o^g is the minimal possible for the jobs of kernel K^g and the current branch of computation can be abandoned. In the second case above, as a result of the activation of emerging job(s) scheduled before job l , either there again arises a gap (now before the delaying emerging job l), or there is no such gap. In the former case we similarly apply the above reasoning. In the latter case, any emerging job activated for the kernel K^h belongs to kernel K^g , none of them being an emerging job for the kernel K^g . It follows that there may exist no job whose activation may potentially decrease the current completion time of the jobs from kernel K^h and the second claim in the lemma is proved. $\square$

5. Discussion and Experimental Results

We have implemented our algorithm in C++ using Visual Studio 2012 on a personal computer with 16 GB, Windows 10 operating system and a 3.4 GHZ Intel i64470 processor (the code together with the generated problem instances can be found at [25]). We have generated our instances randomly applying standard rules commonly used for scheduling problems. In each generated instance with n jobs, the release time r_i , the processing time p_i and the due-date d_i (the delivery time q_i) of job i were obtained as follows: $r_i = \text{rand}() \bmod \text{range}$, $p_i = \text{rand}() \bmod 100$ and $d_i = \text{rand}() \bmod \text{range}$, where we let $\text{range} = 50n$.

As we have mentioned earlier, in our statistics, we have omitted the instances which were solved optimally already by a standard optimality condition from Proposition 1 (these instances formed the majority of the generated instances). As to the publicly available instances for the problem, we have only found ones from an earlier mentioned reference [16]. Although the input (i.e., the job parameters) from that reference coincide with those for our problem, the objective function dealt with in this reference is different from ours. Hence, our results cannot be directly compared to ones reported in [16]. However, we have tested our algorithm for a considerable number of the largest instances (ones with 1000 jobs) from [16], in particular, ones that were generated with parameters (ALPHA, BETA) = (0.25, 0.1), (0.25, 0.25), (0.25, 0.5), sixty instances in total. All these instances were solved optimally instantaneously at the first stage of our algorithm by ED-heuristic.

We have tested 50 difficult problem instances (non-discarded by Proposition 1 ones) with 10, 20, 50, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 2000, 5000 and 10,000 jobs. For large instances, we have imposed an upper limit on the running time of our procedure (for 1000 and 2000 jobs 10, 30 and 60 s, and for 5000 and 10,000 jobs 300, 900 and 3600 s, respectively) since our goal was to verify the approximation guaranteed by Theorem 1 within a short execution time of the procedure. Moderate-sized instances were solved optimally. In particular, most of the instances with up to 50 jobs were solved optimally. A good approximation was attained for the instances with larger number of

jobs which were not solved optimally. In general, we have observed that for larger instances better approximation was attained, in practice. From 50 up to 1000 jobs the guaranteed approximation has monotonically improved, reaching the factor of 1.2 for 1000 jobs. About 10% of larger problem instances with 2000 and 5000 jobs were solved optimally within one minute of the running time of our algorithm and the average approximation factor was about 1.3. Remarkably, the average approximation factor for the largest tested instances with 10,000 jobs has decreased to about 1.25.

Recall from Theorem 1 that we estimate the approximation attained by an enumerated feasible solution by the parameter κ , the number of the detected independent kernels with different delaying emerging jobs in the branch of tree $\mathcal{T}$ containing that solution (the more such kernels arise, the better the ensured approximation is). In the summary table below column “ $|E|$ ” (“ κ ”, respectively) specifies the number of the emerging jobs (the maximal number of the arisen independent kernels with different delaying emerging jobs, respectively) in a branch of search tree $\mathcal{T}$, and λ is the corresponding (average) approximation factor.

Table 9 summarizes the average performance of our algorithm for the tested difficult problem instances, and Figure 6 plots an average dependence of the approximation factor on the total number of jobs.

Table 9. The average number of kernels and the guaranteed approximation factor for each group of the difficult instances.

Number of Jobs	Average κ	Average λ	Average $ E $	Number of Instances	Solved Optimally
10	1.22	1.82	1.98	50	50
20	1.84	1.54	4.56	50	50
30	1.82	1.55	6.22	50	30
40	2.38	1.42	10.76	50	44
50	2.5	1.40	13.74	50	40
100	3.16	1.32	32.2	50	11
200	3.08	1.32	58.5	50	13
300	3.08	1.32	80.82	50	9
400	3.42	1.29	118.96	50	9
500	3.1	1.32	156.58	50	5
600	3	1.33	149.18	50	9
700	3.52	1.28	149.64	50	5
800	3.78	1.26	166.76	50	5
900	3.7	1.27	175.32	50	4
1000	4.18	1.24	202.02	50	2
2000	3.2	1.31	172.56	50	1
5000	2.74	1.36	314.72	50	7
10,000	3.26	1.31	885.36	50	0

Figure 6. Approximation factor λ with maximum 1000 jobs.

The reader may also have a look at the detailed tables presented in [25] from where we may observe that the approximation factor that we guarantee for each created solution is not tight. For example, an instance might be solved optimally by our enumeration procedure, where the approximation factor that we can guarantee for any created solution cannot be 1 (see the value of κ for the optimally solved instances in the detailed tables from [25]).

We have also tested the behavior of our enumeration framework in the worst possible scenarios that we could create. For that, we have generated pseudo-random artificial problem instances from the second class of instances. They were created as the most inconvenient ones for *our algorithm* so that it would be forced to perform an almost complete enumeration of the candidate ED-schedules. Our aim was to verify the approximation that our algorithm could still guarantee in a reasonable time. As intended, it has failed to create an optimal solution for already quite moderate sized artificial instances. At the same time, extremely good approximation factor very close to 1 for these instances were guaranteed. Before presenting the results of the computational experiments, we describe how we have created these instances. Each artificial instance contains three different types of jobs, the same number of jobs of each type. We have generated 50 instances with $3 \times 4 = 12$, $3 \times 7 = 21$, $3 \times 10 = 30$, $3 \times 14 = 42$, $3 \times 17 = 51$, $3 \times 34 = 102$, $3 \times 134 = 402$, $3 \times 167 = 501$, $3 \times 200 = 600$ and $3 \times 334 = 1002$ jobs. We have three types of jobs, to which we refer as the type α , the type β and the type γ jobs. Type α jobs are “tight” jobs, i.e., for an α -type job j , $d_j - r_j = p_j$. Every type α job may potentially form a kernel consisting of that single job. We have left enough space between the different jobs of type α so that for any neighboring pair j, i of type α jobs with $r_j < r_i$, $r_j + p_j < r_i$. The release times of type α jobs were generated pseudo-randomly subject to this restriction. The type β and type γ jobs are designed to be included in between and after the (urgent) type α jobs. All the type β and type γ jobs are released at time 0. At the same time, the type β and type γ jobs are paired in a special way; in particular, their due-dates and the processing times are determined as follows. For each pair (b, c) of a type β job b and the corresponding type γ job c , the due-date of job b is slightly larger than that job c job, so that, whenever two or more jobs of both types are available, the ED-heuristic would take a wrong choice giving the priority to a type γ job. In particular, $d_j = \sum_{i \in J} p_i$ if job j is of type γ , and $d_j = \sum_{i \in J} p_i + 1$ if job j is a type β job. We have one type β job k for each pair of two neighboring type α jobs j, i ($r_j < r_i$) so that it would exactly fill in the space between the jobs j and i in the optimal solution; i.e., $p_k = r_i - (r_j + p_j)$ (in addition, there is a type β job with processing time r^1 that can fit before the earliest released at time r^1 type α job). Type γ jobs are, on average, longer than type β jobs according

to the way how the trial interval for the random derivation of the processing times of type γ jobs was determined.

In Table 10 below we illustrate an artificial instance with 12 jobs and we represent an optimal solution for that instance in Table 11 and Figure 7.

Table 10. Data of the instance E_1_12.

j	0	1	2	3	4	5	6	7	8	9	10	11
r_j	12	0	0	170	0	0	287	0	0	382	0	0
p_j	95	12	148	30	63	205	36	87	102	91	59	159
d_j	107	1088	1087	200	1088	1087	323	1088	1087	473	1088	1087
type	α	β	γ	α	β	γ	α	β	γ	α	β	γ

Table 11. Optimal solution for instance E_1_12.

j	1	0	4	3	7	6	10	9	2	5	8	11
s_j	0	12	107	170	200	287	323	382	473	621	826	928
c_j	12	107	170	200	287	323	382	473	621	826	928	1087
l_j	−1076	0	−918	0	−801	0	−706	0	−466	−261	−159	0
type	β	α	β	α	β	α	β	α	γ	γ	γ	γ

Figure 7. Optimal solution for E_1_12, the red jobs are of type α , the green jobs are of type β and the blue jobs are of type γ .

As we can easily see, in the optimal schedule all the type α jobs are started at their release times and the corresponding type β job is scheduled in between each pair j, i of the neighboring type α jobs, whereas all the type γ jobs are included behind all type α and type β jobs at the end of the schedule (see Figure 7). The optimal objective value is easy to calculate for any such instance: clearly, the maximum job lateness in the optimal schedule is 0 (it cannot be less because of the tight type α jobs). This optimal schedule, however, is difficult to create by our implicit enumeration of ED-schedules. Indeed, since type γ jobs are more urgent than type β jobs, ED-heuristic repeatedly includes a type γ job instead of the corresponding type β job between each two neighboring type α jobs j and i . So there is a type γ job included between each pair of the neighboring type α jobs in the initial ED-schedule σ , whereas more urgent type β jobs are included after the type γ jobs. Since type γ jobs are longer than type β jobs, every type γ emerging job, included in between two neighboring type α jobs j and i yields a forced delay of a tight type α job i . As a result, job i either forms a kernel or becomes part of it, whereas the corresponding type γ job becomes the delaying emerging job. Once activated, that type γ job repeatedly gets included before another tight type α job, again becomes an emerging job and is newly activated. In each so created ED-schedule some type α job is the overflow job, whereas all the type γ jobs, included before that type α job, are emerging jobs. Once all type γ emerging jobs become activated, in the interval before a type α job, a “wrong” type β job might be included, e.g., one which is longer than the correct type β job corresponding to the later type α job. Then such a type β job also becomes an emerging job and gets activated. Thus both, type γ and type β jobs are the potential emerging jobs, which causes the creation of an excessive number of ED-schedules in the search tree $\mathcal{T}$.

In Table 12 we summarize the average performance of our algorithm for some of the artificial instances for which we have imposed no prior restriction on the execution time of the algorithm: the execution was stopped due to the memory overflow as indicated in the column labeled by “Time”. The results for the rest of the instances of the second class can be found in Tables from [25]. In Table 12, the column “Instance” indicates the name of the corresponding instance, in column “Jobs” the number of the jobs is specified, columns “Width” and “Depth” indicate the width and the depth, respectively, of

the solution tree $\mathcal{T}$ constructed for the corresponding instance, “BS Level” stands for the level in search tree $\mathcal{T}$ of the best obtained solution for the corresponding instance, the execution (processor) time is specified in seconds in column “Time”, the column σ (“Best”, respectively) specifies the maximum job lateness in the initial ED-schedule σ (in the best obtained solution, respectively); column “Completed” indicates if the best obtained solution is optimal (“Y”) or if its optimality is not guaranteed (“N”):

Table 12. Results for some artificial instances.

Instance	Jobs	Nodes	Width	Depth	BS Level	E	Time	κ	σ	Best	Completed
E_1_21	21	95951	11,531	45	39	13	28	7	168	0	Y
E_2_21	21	427159	49,443	42	35	14	2475	7	161	0	Y
E_3_21	21	1008765	135,137	43	32	13	10,789	7	118	0	Y
E_4_21	21	1500000	183,214	40	32	14	40,101	7	88	3	N
E_5_21	21	2000000	267,814	43	36	14	38,624	7	114	4	N
E_1_30	30	9400000	1,734,469	66	52	20	511,470	10	166	14	N
E_1_42	42	2400000	348,845	100	90	27	82,412	14	158	10	N
E_1_51	51	2500000	402,956	129	105	32	84,663	16	185	54	N
E_1_102	102	4300000	746,828	386	326	59	82,166	34	199	48	N
E_1_201	201	1700000	307,756	1234	1011	110	78,360	67	172	46	N
E_1_501	501	1900000	236,220	6617	6548	249	196,255	166	192	38	N
E_1_600	600	5800000	1,020,177	9148	8924	353	770,493	198	197	41	N

As we can see, the maximal number of the activated emerging jobs in a branch of tree $\mathcal{T}$ is close to the $2/3$ of the total number of jobs (see the column labeled by “E”), whereas the number of the created ED-schedules is growing very fast with the number of jobs (see column “Nodes”). As it was intended, the number of enumerated solutions for the instances of the second class turned out essentially more than that for the first class of instances. However, the approximation factor provided by our algorithm for the smallest instances with 12 jobs from the second class already reached $9/8$ (although these instances were solved optimally), whereas for the larger instances, the average approximation ratio was sharply improving from $1 + 1/7$ for 21 jobs, $1 + 1/10$ for 30 jobs, $1 + 1/195$ for 600 jobs. Within the first 10 seconds of the execution time of our algorithm the guaranteed approximation is almost 1 for the instances with 1002 and 2001 jobs. Thus, although our algorithm was forced to enumerate a very large number of feasible ED-schedules, it has provided an extremely good approximation for the artificial instances of the second class within a very short execution time.

6. Conclusions

We have described an implicit enumeration algorithm for scheduling one machine, which we have combined with a tool for the calculation of the approximation factor of each enumerated solution. Since the approximation factor is calculated for each created solution, in practice, the search can be detained in case a desired approximation is already reached or if the execution time becomes unacceptable. Unlike the earlier conducted work in the literature, we have tested the performance of our algorithm and the effectiveness of the guaranteed approximation factor on the specially created difficult problem instances. Indeed, we have discarded all the randomly generated problem instances which were solved by the earlier known dominance relation or already by ED-heuristic. For the remaining over one thousand difficult problem instances, either the algorithm has found an optimal solution or it has guaranteed a good approximation to the optimum after a short running time. Importantly, for larger problem instances better approximation factors were obtained, hence the method seems to work effectively for larger instances. Moreover, for the artificially created “extremely” difficult problem instances the algorithm has provided almost optimal solutions within a short running time.

The proposed approach might be extended for the multiprocessor versions of the studied single-machine scheduling problem. This will require a similar study of the structural properties of the corresponding ED-schedules. The generalization of the pruning, optimality and approximation conditions derived here for the multiprocessor ED-schedules may require an independent study.

Author Contributions: Conceptualization, N.V., F.A.-P. and J.M.S.; Methodology, F.A.-P. and N.V.; Validation, N.V.; Formal Analysis, J.M.S. and N.V. Resources, UAEMor administrated by F.A.-P. and J.A.H.; Writing N.V.; Supervision J.M.S. and N.V.; Project administration, N.V.; All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by Programa Fortalecimiento a la Excelencia Educativa (PROFEXCE) publication grant, PRODEP 511/6 grant, CONACyT 2020-000019-01NACV-00008 grant and Agencia Estatal de Investigación (PID2019-106433GB-I00/AEI/10.13039/501100011033) grant (Spain).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Bratley, P.; Florian, M.; Robillard, P. On sequencing with earliest start times and due-dates with application to computing bounds for $(n/m/G/F_{max})$ problem. *Nav. Res. Logist. Quart.* **1973**, *20*, 57–67. [\[CrossRef\]](#)
2. Garey, M.R.; Johnson, D.S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*; Freeman: San Francisco, CA, UUSA, 1979.
3. Chinos, E.; Vakhania, N. Adjusting scheduling model with release and due dates in production planning. *Cogent Eng.* **2017**, *4*. [\[CrossRef\]](#)
4. McMahon, G.; Florian, M. On scheduling with ready times and due dates to minimize maximum lateness. *Oper. Res.* **1975**, *23*, 475–482. [\[CrossRef\]](#)
5. Carlier, J. The one-machine sequencing problem. *Eur. J. Oper. Res.* **1982**, *11*, 42–47. [\[CrossRef\]](#)
6. Grabowski, J.; Nowicki, E.; Zdrzalka, S. A block approach for single-machine scheduling with release dates and due dates. *Eur. J. Oper. Res.* **1986**, *26*, 278–285. [\[CrossRef\]](#)
7. Jackson, J.R. Scheduling a production line to minimize the maximum tardiness. In *Management Science Research Project*; University of California: Los Angeles, CA, USA, 1955.
8. Schrage, L. Obtaining Optimal Solutions to Resource Constrained Network Scheduling Problems. 1971, Unpublished Manuscript.
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. Vakhania, N. Dynamic Restructuring Framework for Scheduling with Release Times and Due-Dates. *Mathematics* **2019**, *7*, 1104. [\[CrossRef\]](#)
12. 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. [\[CrossRef\]](#)
13. Potts, C.N. Analysis of a heuristic for one machine sequencing with release dates and delivery times. *Oper. Res.* **1980**, *28*, 1436–1441. [\[CrossRef\]](#)
14. Hall, L.A.; Shmoys, D.B. Jackson's rule for single-machine scheduling: Making a good heuristic better. *Math. Oper. Res.* **1992**, *17*, 22–35. [\[CrossRef\]](#)
15. Lawler, E.L.; Lenstra, J.K.; Kan, A.R. Recent developments in deterministic sequencing and scheduling: A survey. In *Deterministic and Stochastic Scheduling*; Springer: Berlin/Heidelberg, Germany, 1982. pp. 35–73.
16. Mahnam, M.; Moslehi, G.; Ghomi, S.M.T.F. Single machine scheduling with unequal release times and idle insert for minimizing the sum of maximum earliness and tardiness. *Math. Comput. Model.* **2013**, *57*, 2549–2563. [\[CrossRef\]](#)
17. Yazdani, M.; Khalili, S.M.; Babagolzadeh, M.; Jolai, F. A single-machine scheduling problem with multiple unavailability constraints: A mathematical model and an enhanced variable neighborhood search approach. *J. Comput. Des. Eng.* **2017**, *4*, 46–59. [\[CrossRef\]](#)
18. Adamu, M.O.; Adewumi, A.O. A survey of single machine scheduling to minimize weighted number of tardy jobs. *J. Ind. Manag. Optim.* **2014**, *10*, 219. [\[CrossRef\]](#)
19. Yin, Y.; Xu, D. Some single-machine scheduling problems with general effects of learning and deterioration. *Comput. Math. Appl.* **2011**, *61*, 100–108. [\[CrossRef\]](#)
20. Lu, Y.Y.; Wei, C.M.; Wang, J.B. Several single-machine scheduling problems with general learning effects. *Appl. Math. Model.* **2012**, *36*, 5650–5656. [\[CrossRef\]](#)
21. Hou, Y.T.; Yang, D.L.; Kuo, W.H.; Wu, L.S. A single-machine scheduling problem with a deterioration model and partial maintenance. *J. Stat. Manag. Syst.* **2018**, *21*, 1501–1511. [\[CrossRef\]](#)

22. Cheng, M.; Xiao, S.; Luo, R.; Lian, Z. Single-machine scheduling problems with a batch-dependent aging effect and variable maintenance activities. *Int. J. Prod. Res.* **2018**, *56*, 7051–7063. [[CrossRef](#)]
23. Park, M.J.; Choi, B.C. A Single-Machine Scheduling Problem with Uncertainty in Processing Times and Outsourcing Costs. *Math. Probl. Eng.* **2017**, *2017*, 5791796. [[CrossRef](#)]
24. Vakhania, N. A better algorithm for sequencing with release and delivery times on identical processors. *J. Algorithms* **2003**, *48*, 273–293. [[CrossRef](#)]
25. Alonso-Pecina, F. A Code and Supplementary Data for a Hybrid Implicit Enumeration and Approximation Algorithm for Scheduling One Machine with Job Release Times and Due Dates (A Complement to the Manuscript “Fast Approximation for Scheduling One Machine”). Available online: <https://github.com/FedericoAlonsoPecina/Scheduling> (accessed on 21 August 2020).

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).